Yale University
Department of Computer Science

A Temporal-Logic Approach to Programming with Dependent Types and Higher-Order Encodings

Adam Poswolsky Carsten Schürmann

YALEU/DCS/TR-1355
April 2006

A Temporal-Logic Approach to Programming with Dependent Types and Higher-Order Encodings

Adam Poswolsky* Carsten Schürmann†

Abstract

In this work, we propose a temporal logic with a past-time operator, show its soundness, and establish a Curry-Howard correspondence to a λ-calculus. Furthermore, we illustrate how it serves as a logical foundation for programming with dependently typed data, supports programming with higher-order abstract syntax and hypothetical judgments. Sample programs that we discuss in this paper include bracket abstraction.

Important Note: There is an updated/simplified presentation of this system available as Yale Technical Report TR-1364

1 Update

Important Note: There is an updated/simplified presentation of this system available as Yale Technical Report TR-1364. This version presents past-time as a modal operator in a meta-logic for LF. The version in TR-1364 separates it into a three-tiered system where past-time is on the meta-meta level for LF. Reference:

"A Temporal-Logic Approach to Functional Calculi for Dependent Types and Higher-Order Encodings"
Yale University Tech Report TR-1364, 2006.
http://www.cs.yale.edu/publications/techreports/tr1364.pdf

*Department of Computer Science, Yale University, CT
†IT University of Copenhagen, Copenhagen, Denmark

2 Introduction

Temporal extensions of logics have proved useful for binding time analysis [Dav96] and meta-programming [Tah04]. In this paper we show how temporal logic can be used to develop a functional programming language utilizing the representational power of logical frameworks [Pfe99] for programming with dependent datatypes and higher-order representation techniques.

Dependent datatypes allow for type systems that are more expressive than their simply-typed counterparts as types may be indexed by expressions. For example, a list type could be indexed by its length, or a certificate type could be indexed by the formula it is witnessing.

Both higher-order encodings, or higher-order abstract syntax (HOAS), and hypothetical judgments employ functional abstraction of the logical framework to model variable binding. This permits programmers to program efficiently with complex data-structures without having to worry about the representation of variables, binding constructs, or substitutions that are prevalent in logic derivations, typing derivations, operational semantics, and intermediate languages.

The computational model involving higher-order encodings differs from the usual one in that computation may have to traverse λ-binders (of the logical framework) during computation. In order to go under a λ-binder we need a method to introduce new variables, or parameters, which can be applied to the function. In this paper we identify parameters by the points in time they were introduced and we use a temporal logic with past-time described in Section 3 to reason about the most important of their properties, namely when they can be accessed and when they can not. Its semantics is given in form of a sequent calculus which is shown to be sound.

Augmented with proof terms in Section 4, the logic is turned into the $\lambda^{\ominus}$-calculus, which identifies provability with computation via a Curry-Howard isomorphism. Since the $\lambda^{\ominus}$-calculus only provides base types of $\top$ (unit) and $\bot$ (void) we augment it further thus turning it into an interesting programming language. To this end, we connect it to two logical frameworks in Section 5: the simply-typed λ-calculus with user defined constants and types, and the logical framework LF [HHP93]. The Curry-Howard correspondence still holds for this calculus. When extended with dependent-types we have the $\lambda^{\ominus\Pi}$-calculus.

We first add an operator for programming with higher-order abstract syntax, in Section 6, which just corresponds to an admissible rule in $\lambda^{\ominus\Pi}$. We then shift our focus from logic to programming in Section 7 and give the

corresponding natural deduction version of $\lambda^{\ominus\Pi}$, called $\lambda^{\mathcal{D}}$. In Section 8 we add a mechanism to conduct case analysis, and finally in section 9 we add recursion.

We illustrate the resulting $\lambda^{\mathcal{D}}$ calculus with examples of bracket abstraction and theorem proving in Section 10. We describe related work in Section 11 before we conclude and assess results in Section 12.

3 Past-Time Temporal Logic

We derive our temporal logic with past-time operator from the sequent calculus for the implicational fragment of propositional logic. We write $\Omega \vdash \tau$ for the central derivability judgment and we cannot permit reordering of Ω since we discuss the dependently typed case below. In slight abuse of notation, we write Ω_1, Ω_2 for the concatenation of two contexts.

We consider only the implicational fragment of temporal logic with a past-time operator $\ominus\tau$.

$$\tau, \sigma, \varrho ::= \tau \supset \sigma \mid \ominus\tau \mid \top \mid \bot$$

We write $\ominus\tau$ to indicate that τ makes sense in the past. In addition we will write $\ominus^k\tau$ as shorthand for $k \geq 0$ leading $\ominus$s to τ. We ascribe a linear time semantics to this logic which allows the past to influence the present but does *not* allow the present to influence the past. Anything in the past can be used in the present but not vice-versa.

The intrinsic behavior behind our logic is that we desire that $\ominus\tau$ means that τ can be shown in the past with just the tools of the past. It should be no different concluding $\ominus\tau$ in the present or going to the past and proving τ. For instance if τ is a proposition ascertaining culpability of a past crime using DNA technology of today, then our logic will disallow proving $\ominus\tau$. This inherent strengthening property is the essential necessary component to our logic.

We give a proof-theoretical account by the following sequent calculus that comprises an axiom rule as well as left and right rules for the different connectives.

First, we define a few preliminaries. Namely, $[\![\tau]\!]$ simply shifts τ to the present-time by stripping off any leading $\ominus$s. Additionally, $\Omega^{-\ominus}$ changes our point-of-reference one step to the past by removing anything in the present (without a leading $\ominus$) and removing a leading $\ominus$ from the remaining.

Definition 3.1 (Shifting)

$$[\![\tau]\!] = \begin{cases} \top & \text{if } \tau = \top \\ \bot & \text{if } \tau = \bot \\ \sigma_1 \supset \sigma_2 & \text{if } \tau = \sigma_1 \supset \sigma_2 \\ [\![\sigma]\!] & \text{if } \tau = \ominus\sigma \end{cases}$$

Definition 3.2 (MovePast)

$$\Omega^{-\ominus} = \begin{cases} \cdot & \text{if } \Omega = \cdot \\ \Omega'^{-\ominus}, \tau & \text{if } \Omega = \Omega', \ominus\tau \\ \Omega'^{-\ominus} & \text{if } \Omega = \Omega', \tau \text{ and } \tau \neq \ominus\tau' \end{cases}$$

$$\frac{\tau \text{ in } \Omega}{\Omega \vdash [\![\tau]\!]}\,\mathsf{ax}$$

$$\frac{}{\Omega \vdash \top}\,\top\mathsf{R} \qquad \text{no rule } \top\mathsf{L}$$

$$\text{no rule } \bot\mathsf{R} \qquad \frac{}{\Omega_1, \ominus^k\bot, \Omega_2 \vdash [\![\sigma]\!]}\,\bot\mathsf{L}$$

$$\frac{\Omega, \tau \vdash \sigma}{\Omega \vdash \tau \supset \sigma}\,\supset\mathsf{R}$$

$$\frac{\Omega_1, \ominus^k(\tau \supset \sigma), \Omega_2 \vdash \tau \quad \Omega_1, \ominus^k(\tau \supset \sigma), \Omega_2, [\![\sigma]\!] \vdash \varrho}{\Omega_1, \ominus^k(\tau \supset \sigma), \Omega_2 \vdash \varrho}\,\supset\mathsf{L}$$

$$\frac{\Omega^{-\ominus} \vdash \tau}{\Omega \vdash \ominus\tau}\,\ominus\mathsf{R}$$

The ax rule lets us conclude any assumptions that we have. However, since we are reasoning in the present-time, the assumption is first shifted to the present. In addition, $\top$ is always valid. Falsity can conclude anything in the present but not the past. The right rule for implication is standard and $\tau \supset \sigma$ is valid if one can derive σ from τ, when τ is assumed valid. The left rule will receive special attention. When utilizing an implication, $\tau \supset \sigma$, the consequent is shifted to the present when we continue in order to enforce that the reasoning is in the present. In order to reason about something

in the past, we must use $\ominus\mathsf{R}$ to go into the past hence yielding our desired semantics that the present does not influence the past.

A *weakened* context Ω' of Ω refers to an extension of Ω by new assumptions. Formally, we write $\Omega \leq \Omega'$.

We can easily convince ourselves by inductive reasoning that the following three lemmas hold. In addition it has been verified in Twelf.

Lemma 3.3 (Weakening) *Let* $\Omega \leq \Omega'$. *If* $\Omega \vdash \sigma$ *then* $\Omega' \vdash \sigma$.

Lemma 3.4 (Shifting)

- *If* $\Omega \vdash \ominus\tau$ *then* $\Omega \vdash \tau$.
- *If* $\Omega \vdash \tau$ *then* $\Omega \vdash [\![\tau]\!]$.

Lemma 3.5 (DeShifting)

- *If* $\Omega_1, \tau, \Omega_2 \vdash \sigma$ *then* $\Omega_1, \ominus\tau, \Omega_2 \vdash \sigma$
- *If* $\Omega_1, [\![\tau]\!], \Omega_2 \vdash \sigma$ *then* $\Omega_1, \tau, \Omega_2 \vdash \sigma$

The final is the most complex to prove as it relies on a rather large case analysis, and follows the outline of Pfenning's proof [Pfe95].

Lemma 3.6 (Admissibility of cut) *If* $\mathcal{D} :: \Omega_1 \vdash \tau$ *and* $\mathcal{E} :: \Omega_1, \tau, \Omega_2 \vdash \sigma$ *then* $\Omega_1, \Omega_2 \vdash \sigma$.

Proof: By induction on the cut formula τ, and simultaneously over derivations $\mathcal{D}$ and $\mathcal{E}$. This proof has been verified in Twelf and is available upon request and is also available in the Appendix.

Case: $\overset{\mathcal{D}}{\Omega_1 \vdash \tau} \quad \mathcal{E} = \dfrac{\tau \text{ in } (\Omega_1, \tau, \Omega_2)}{\Omega_1, \tau, \Omega_2 \vdash [\![\tau]\!]}\,\mathsf{ax}$

$\Omega_1, \leq \Omega_1, \Omega_2$	by Definition
$\Omega_1, \Omega_2 \vdash \tau$	by Weakening (Lemma 3.3)
$\Omega_1, \Omega_2 \vdash [\![\tau]\!]$	by Shifting (Lemma 3.4)

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 :: \Omega_1, \tau \vdash \sigma}{\Omega_1 \vdash \tau \supset \sigma}\supset\mathsf{R}$,

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1 :: \Omega_1, \tau \supset \sigma, \Omega_2 \vdash \tau \\ \mathcal{E}_2 :: \Omega_1, \tau \supset \sigma, \Omega_2, [\![\sigma]\!] \vdash \varrho\end{array}}{\Omega_1, \tau \supset \sigma, \Omega_2 \vdash \varrho}\supset\mathsf{L}$$

$\mathcal{E}_1' :: \Omega_1, \Omega_2 \vdash \tau$ by induction hypothesis on $\mathcal{D}$ and $\mathcal{E}_1$
$\mathcal{E}_2' :: \Omega_1, \Omega_2, [\![\sigma]\!] \vdash \varrho$ by induction hypothesis on $\mathcal{D}$ and $\mathcal{E}_2$
$\Omega_1, \tau \leq \Omega_1, \Omega_2, \tau$ by Definition
$\mathcal{D}_1' :: \Omega_1, \Omega_2, \tau \vdash \sigma$ by Weakening (Lemma 3.3) on $\mathcal{D}_1$
$\mathcal{D}_1'' :: \Omega_1, \Omega_2 \vdash \sigma$ by induction hypothesis on $\mathcal{E}_1'$ and $\mathcal{D}_1'$
$\mathcal{E}_2'' :: \Omega_1, \Omega_2, \sigma \vdash \varrho$ by DeShifting (Lemma 3.5) on $\mathcal{E}_2'$
$\mathcal{F} :: \Omega_1, \Omega_2 \vdash \varrho$ by induction hypothesis on $\mathcal{D}_1''$ and $\mathcal{E}_2''$

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 :: {\Omega_1}^{-\ominus} \vdash \tau}{\Omega_1 \vdash \ominus\tau}\,\ominus\mathsf{R}$

$\mathcal{E} = \dfrac{\mathcal{E}_1 :: (\Omega_1, \ominus\tau, \Omega_2)^{-\ominus} \vdash \sigma}{\Omega_1, \ominus\tau, \Omega_2 \vdash \ominus\sigma}\,\ominus\mathsf{R}$

$\mathcal{E}_1' :: {\Omega_1}^{-\ominus}, \tau, {\Omega_2}^{-\ominus} \vdash \sigma$ by Property of $(\_)^{-\ominus}$ on $\mathcal{E}_1$
$\mathcal{F} :: {\Omega_1}^{-\ominus}, {\Omega_2}^{-\ominus} \vdash \sigma$ by induction hypothesis on $\mathcal{D}_1$ and $\mathcal{E}_1'$
$\mathcal{F}' :: (\Omega_1, \Omega_2)^{-\ominus} \vdash \sigma$ by Property of $(\_)^{-\ominus}$ on $\mathcal{F}$
$\mathcal{F}'' :: \Omega_1, \Omega_2 \vdash \ominus\sigma$ by $\ominus\mathsf{R}$

All non-essential cases are omitted but available in the Appendix. The entire proof has been verified in Twelfand is also available in the Appendix. □

Hence, we add the admissible cut rule to the proof theory of $\mathcal{L}^{\ominus}$ and obtain a logic, we refer to as $\mathcal{L}^{\ominus\mathsf{cut}}$.

$$\frac{\Omega_1 \vdash \tau \quad \Omega_1, \tau, \Omega_2 \vdash \sigma}{\Omega_1, \Omega_2 \vdash \sigma}\,\mathsf{cut}$$

Theorem 3.7 (Cut-Elimination) *Let $\Omega \vdash \tau$ be a derivation in $\mathcal{L}^{\ominus\mathsf{cut}}$. Then there exists a derivation of $\Omega \vdash \tau$ in $\mathcal{L}^{\ominus}$.*

It is easy to see that $\mathcal{D} :: \cdot \vdash \bot$ cannot be a valid derivation in $\mathcal{L}^{\ominus\mathsf{cut}}$. If it were, there is a valid derivation $\mathcal{E} :: \cdot \vdash \bot$ in $\mathcal{L}^{\ominus}$, which is impossible by inspection of the right rules.

Corollary 3.8 (Soundness) *The logic $\mathcal{L}^{\ominus\mathsf{cut}}$ is sound.*

4 Curry-Howard Isomorphism

The sequent-calculus for first-order logic lends itself for an operational interpretation where the cut-rule is the one that triggers evaluation. Hence cut-free derivations play the role of normal forms, or values. Similarly, cut-free derivations will play the role of values in our system.

Proof terms convey the idea that sequent derivations are programs, and thus we generalize the derivability judgment to $\Omega \vdash e \in \tau$ where each assumption in Ω is endowed with a unique name u or w and e is the proof term, also often called expression throughout this paper.

$$\begin{array}{rcl} e, f & ::= & u \mid \text{unit} \mid \text{void } u \\ & & \mid \lambda u \in \tau\,.e \mid \text{let } w = u \cdot e \text{ in} f \\ & & \mid \text{prev } e \mid [e/u]f \end{array}$$

The proof term algebra almost resembles the λ-calculus extended by primitives handling past-time. Instead of application, we use the more awkward looking, yet cleanly motivated let $w = u \cdot e$ in$f$. prev e is a proof-term that prompts us to execute e in the past. We call the resulting calculus $\lambda^{\ominus}$ as it is clearly a λ-calculus that captures computation. The type system of the $\lambda^{\ominus}$-calculus endows the proof theory of $\mathcal{L}^{\ominus}$ from Section 3 with proof terms.

Definition 4.1 (MovePast)

$$\Omega^{-\ominus} = \begin{cases} \cdot & \text{if } \Omega = \cdot \\ \Omega'^{-\ominus}, u \in \tau & \text{if } \Omega = \Omega', u \in \ominus\tau \\ \Omega'^{-\ominus} & \text{if } \Omega = \Omega', u \in \tau \text{ and } \tau \neq \ominus\tau' \end{cases}$$

$$\frac{(u \in \tau) \text{ in } \Omega}{\Omega \vdash u \in [\![\tau]\!]}\,\mathsf{ax}$$

$$\frac{}{\Omega \vdash \text{unit} \in \top}\,\top\mathsf{R} \qquad \text{no rule } \top\mathsf{L}$$

$$\text{no rule } \bot\mathsf{R} \qquad \frac{}{\Omega_1, u \in \ominus^k \bot, \Omega_2 \vdash \text{void } u \in [\![\sigma]\!]}\,\bot\mathsf{L}$$

$$\frac{\Omega, u \in \tau \vdash e \in \sigma}{\Omega \vdash \lambda u \in \tau\, . e \in \tau \supset \sigma}\,\supset \mathsf{R}$$

$$\frac{\begin{array}{c}\Omega_1, u \in \ominus^k(\tau \supset \sigma), \Omega_2 \vdash e \in \tau \\ \Omega_1, u \in \ominus^k(\tau \supset \sigma), \Omega_2, w \in [\![\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1, u \in \ominus^k(\tau \supset \sigma), \Omega_2 \vdash \text{let } w = u \cdot e \text{ in} f \in \varrho}\,\supset \mathsf{L}$$

$$\frac{\Omega^{-\ominus} \vdash e \in \tau}{\Omega \vdash \text{prev } e \ominus \tau}\,\ominus\mathsf{R}$$

and finally, the proof term for the cut-rule resembles that of an (explicit) substitution.

$$\frac{\Omega_1 \vdash e \in \tau \quad \Omega_1, u \in \tau, \Omega_2 \vdash f \in \sigma}{\Omega_1, \Omega_2 \vdash [e/u]f \in \sigma}\,\mathsf{cut}$$

Substitutions cannot be executed in place, but must remain lazy, as the proof term associated with $\supset\,\mathsf{L}$ does not act upon an arbitrary term, but rather destruct only a variable from the context. To be consistent with our notation, we will call the system of $\lambda^{\ominus}$ extended with cut as $\lambda^{\ominus\mathsf{cut}}$.

It is also important to note the new form of the Weakening, Shifting, and DeShifting lemmas when extended with proof-terms.

Lemma 4.2 (Weakening (with proof-terms))
Let $\Omega \leq \Omega'$. *If* $\Omega \vdash e \in \sigma$ *then* $\Omega' \vdash e \in \sigma$.

First we define $[\![e]\!]_\diamond$ as an operation which takes an e of type $\ominus\tau$ and returns an e' of type τ.

$$\begin{array}{rcl}[\![\text{prev } e]\!]_\diamond & = & e \\ [\![\text{let } w = u \cdot\ e \text{ in } f]\!]_\diamond & = & \text{let } w = u \cdot\ e \in [\![f]\!]_\diamond\end{array}$$

In addition, we define $[\![e]\!]^*_\diamond$ which takes an e of type τ and returns an e' of type $[\![\tau]\!]$.

$$\begin{array}{rcl} [\![\text{prev } e]\!]^*_\diamond & = & [\![e]\!]^*_\diamond \\ [\![\text{let } w = u \cdot\ e \text{ in } f]\!]^*_\diamond & = & \text{let } w = u \cdot\ e \in [\![f]\!]^*_\diamond \\ \text{Above doesn't match, } [\![e]\!]^*_\diamond & = & e \end{array}$$

Lemma 4.3 (Shifting (with proof-terms))

- *If* $\Omega \vdash e \in \ominus\sigma$ *then* $\Omega \vdash [\![e]\!]_\diamond \in \sigma$.
- *If* $\Omega \vdash e \in \sigma$ *then* $\Omega \vdash [\![e]\!]^*_\diamond \in [\![\sigma]\!]$

Lemma 4.4 (DeShifting (with proof-terms))

- *If* $\Omega_1, u \in \tau, \Omega_2 \vdash e \in \sigma$ *then* $\Omega_1, u \in \ominus\tau, \Omega_2 \vdash e \in \sigma$
- *If* $\Omega_1, u \in [\![\tau]\!], \Omega_2 \vdash e \in \sigma$ *then* $\Omega_1, u \in \tau, \Omega_2 \vdash e \in \sigma$

The operational behavior of the $\lambda^\ominus$-calculus is footed on how substitutions are pushed into the proof terms, which corresponds directly to the process of cut-elimination. We therefore interpret the proof of the admissibility of cut (Theorem 5.12) as equivalences on proof terms. The proof can be found in the appendix and in the interest of space, we only show some of the interesting equivalences below.

$$\begin{array}{rcl} [e/u]w & \equiv & \begin{cases} [\![e]\!]^*_\diamond & \text{if } u = w \\ w & \text{otherwise} \end{cases} \\ [(\lambda u' \in \tau\,.e')/u] & & \\ (\text{let } w = u \cdot e \text{ in } f) & \equiv & [\dfrac{[\frac{\lambda u' \in \tau\,.e'}{u}]e}{u'}]e' ([\dfrac{\lambda u' \in \tau\,.e'}{u}]f) \\ [(\text{prev } e)/u](\text{prev } f) & \equiv & \text{prev } ([e/u]f) \\ [\dfrac{(\text{let } w = u' \cdot e' \text{ in } f)}{u}]e & \equiv & \text{let } w = u' \cdot e' \text{ in } [f/u]e \end{array}$$

Definition 4.5 (Values) *The following syntactic category denotes all values of the* $\lambda^\ominus$*-calculus:* $v ::= \text{prev } v \mid \lambda u \in \tau\,.e \mid \text{unit}$[1].

[1] does void belong in values?

Theorem 4.6 (Totality) *If $\cdot \vdash e \in \tau$ then e is a total function that computes a value.*

Proof: Direct consequence of the cut-elimination Theorem 3.7. □

Therefore, by construction, we have an isomorphism between $\mathcal{L}^{\ominus}$ derivations $\cdot \vdash \tau$ and well-typed programs in the $\lambda^{\ominus}$-calculus, which is also called the Curry-Howard isomorphism.

Example 4.7 (Distributivity of $\ominus$) The distributivity law is a defining property of past-time logic:

$$
\begin{array}{rl}
\mathcal{D}_1 :: [\![\tau]\!] \supset [\![\sigma]\!], [\![\tau]\!] \vdash [\![\tau]\!] & \text{by } \mathsf{ax} \\
\mathcal{D}_2 :: [\![\tau]\!] \supset [\![\sigma]\!], [\![\tau]\!], [\![\sigma]\!] \vdash [\![\sigma]\!] & \text{by } \mathsf{ax} \\
\mathcal{D}_3 :: [\![\tau]\!] \supset [\![\sigma]\!], [\![\tau]\!] \vdash [\![\sigma]\!] & \text{by } \supset \mathsf{L} \text{ on } \mathcal{D}_1 \text{ and } \mathcal{D}_2 \\
\mathcal{D}_4 :: \ominus([\![\tau]\!] \supset [\![\sigma]\!]), \ominus([\![\tau]\!]) \vdash \ominus[\![\sigma]\!] & \text{by } \ominus\mathsf{R} \text{ on } \mathcal{D}_3 \\
\mathcal{D}_5 :: \ominus([\![\tau]\!] \supset [\![\sigma]\!]) \vdash \ominus[\![\tau]\!] \supset \ominus[\![\sigma]\!] & \text{by } \supset \mathsf{R} \text{ on } \mathcal{D}_4 \\
\mathcal{D}_6 :: \cdot \vdash \ominus([\![\tau]\!] \supset [\![\sigma]\!]) \supset \ominus[\![\tau]\!] \supset \ominus[\![\sigma]\!] & \text{by } \supset \mathsf{R} \text{ on } \mathcal{D}_5
\end{array}
$$

The entire derivation of $\mathcal{D}_6$ is isomorphically captured by the proof term

$$
\begin{array}{c}
\lambda u \in \ominus([\![\tau]\!] \supset [\![\sigma]\!]) . \lambda u' \in \ominus[\![\tau]\!] . \\
\text{prev } (\text{let } w = u \cdot u' \text{ in } w)
\end{array}
$$

in the $\lambda^{\ominus}$-calculus.

5 Logical Frameworks

So far, $\mathcal{L}^{\ominus}$ only provides two propositional constants $\top$ and $\bot$. As $\mathcal{L}^{\ominus}$ is propositional, the standard Church encoding of inductive datatypes does not apply. To remedy this shortcoming, we study the connection between $\mathcal{L}^{\ominus}$ and logical frameworks, where two frameworks are of particular concern: the simply-typed λ-calculus that goes back to Church [Chu40] and the Edinburgh logical framework [HHP93]. Both frameworks define syntactic categories of objects M and types A to which we assign the usual logic meaning. We say $\langle A \rangle$ is true if and only if the type A is inhabited. If M is the witness of inhabitation, we interpret M as a proof of $\langle A \rangle$. Both logical frameworks illustrate interesting facts of $\mathcal{L}^{\ominus}$, and we consider both in turn.

5.1 The Simply-Typed Logical Framework

The usual definition of the simply-typed λ-calculus provides a notion of types $A, B ::= a \mid A \to B$ and objects $M, N ::= x \mid c \mid M\ N \mid \lambda x : A.N$. The functional $\to$ is not to be confused with the $\supset$-connective of the previous section. x stands for variables while a and c are type and object constants, respectively. These constants are provided a priori in a collection $\Sigma ::= \cdot \mid \Sigma, a : \text{type} \mid \Sigma, c : A$. Σ should be seen as datatype declaration, keeping in mind that our notion of datatype is non-standard as we permit higher-order functions to be passed to constants without the usual restriction that variables of the datatype that is being defined cannot occur in negative positions.

We write $\Gamma \vdash M : A$ for the typing judgment that M has type A, in context $\Gamma ::= \cdot \mid \Gamma, x : A$ which assigns types to variables. The definition of the typing rules are standard. We take $\beta\eta$ as the underlying notion of equivalence between λ-terms. Terms in β-normal η-long form are also called canonical forms.

Theorem 5.1 (Canonical forms) *Every term in the simply-typed λ-calculus possesses a unique canonical form.*

The existence of canonical forms is instrumental for encodings to be *adequate*, which means that there exists a bijection between expressions of the source language and their representations (as canonical forms) in the logical framework.

Example 5.2 (Expressions) As a sample signature, we choose the standard language of untyped λ terms $t ::= x \mid \mathbf{lam}\ x.\, t \mid t_1 @ t_2$. In the simply typed logical framework an expression t can be encoded as $\ulcorner t \urcorner$, which gives rise to the following signature.

$$
\begin{array}{lll}
 & & \text{exp} : \text{type}, \\
\ulcorner x \urcorner & = x & \\
\ulcorner \mathbf{lam}\ x.\, t \urcorner & = \text{lam}\ (\lambda x : \text{exp}\, .\ulcorner t \urcorner) & \text{lam} : (\text{exp} \to \text{exp}) \to \text{exp}, \\
\ulcorner t_1 @ t_2 \urcorner & = \text{app}\ \ulcorner t_1 \urcorner\ \ulcorner t_2 \urcorner & \text{app} : \text{exp} \to \text{exp} \to \text{exp}
\end{array}
$$

Theorem 5.3 (Adequacy) *There exists a bijection between untyped λ-terms t with free variables among $x_1 \ldots x_n$ and canonical derivations in the simply typed logical framework of $x_1 : \text{exp} \ldots x_n : \text{exp} \vdash \ulcorner t \urcorner : \text{exp}$.*

Proof: By induction over the structure of t in one direction and the structure of the β-normal η-long form of $\ulcorner t \urcorner$ in the other. $\square$

We now discuss the bridge between the logical (computation) level and the logical framework (representation) level.

$$[\![\langle A \rangle]\!] = \langle A \rangle$$
$$[\![M]\!]^*_\diamond = M$$

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \langle A \rangle \\ e, f & ::= & \ldots \mid M \end{array}$$

$$\frac{[\Omega]_{\mathrm{LF}} \vdash M : A}{\Omega \vdash M \in \langle A \rangle}\, \langle\rangle\mathsf{R}$$

We define $[\Omega]_{\mathrm{LF}}$ as an operation which takes our meta-context, Ω, and converts it into a context, Γ, suitable for the logical framework by simply filtering out any meta-information. The resulting context is of the form:

$$\Gamma ::= \cdot \mid \Gamma, x : A$$

We call this operation thinning and define it as follows.

Definition 5.4 (Thinning)

$$[\Omega]_{LF} = \begin{cases} \cdot & \text{if } \Omega = \cdot \\ [\Omega']_{LF}, x : A & \text{if } \Omega = \Omega', x \in \ominus^k \langle A \rangle \\ [\Omega]_{LF} & \text{if } \Omega = \Omega', u \in \tau \text{ and not match Above} \end{cases}$$

This resulting logic and λ-calculus are called $\mathcal{L}^{\ominus\langle\rangle}$ and $\lambda^{\ominus\langle\rangle}$ respectively. We define $[M/u]_{\mathrm{LF}}N$ as standard substitution of the logical-framework.

Lemma 5.5 (Admissibility of cut) *The cut rule is still admissible for this logic, $\mathcal{L}^{\ominus\langle\rangle}$.*

Proof: By structural induction as Theorem 3.6.

Case: $\mathcal{D} = \dfrac{[\Omega_1]_{\mathrm{LF}} \vdash M : A}{\Omega_1 \vdash M \in \langle A \rangle}\, \langle\rangle\mathsf{R}$

$$\mathcal{E} = \frac{\mathcal{E}_1 :: [(\Omega_1, u \in \langle A \rangle, \Omega_2)]_{\mathrm{LF}} \vdash N : B}{\Omega_1, u \in \langle A \rangle, \Omega_2 \vdash N \in \langle B \rangle}\, \langle\rangle\mathsf{R}$$

$\mathcal{E}_1' :: [\Omega_1]_{\mathrm{LF}}, u : A, [\Omega_2]_{\mathrm{LF}} \vdash N : B$ by Property of $[\_]_{\mathrm{LF}}$ on $\mathcal{E}_1$
$\mathcal{F} :: [\Omega_1]_{\mathrm{LF}}, [\Omega_2]_{\mathrm{LF}} \vdash [M/u]_{\mathrm{LF}} N \in B$ by LF Substitution
$\mathcal{F}' :: [(\Omega_1, \Omega_2)]_{\mathrm{LF}} \vdash [M/u]_{\mathrm{LF}} N \in B$
by Property of $[\_]_{\mathrm{LF}}$ on $\mathcal{F}$
$\Omega_1, \Omega_2 \vdash [M/u]_{\mathrm{LF}} N \in \langle B \rangle$ by $\langle\rangle\mathsf{R}$

We omit the rest of the cases, but the proof is available in the Appendix and has been verified in Twelf. □

We extend $\mathcal{L}^{\ominus\langle\rangle}$ with the cut rule to yield $\mathcal{L}^{\ominus\langle\rangle\mathsf{cut}}$. Analogously we also extend $\lambda^{\ominus\langle\rangle}$ with its cut rule to yield $\lambda^{\ominus\langle\rangle\mathsf{cut}}$.

Theorem 5.6 (Cut-Elimination) *Let $\Omega \vdash \tau$ a derivation in $\mathcal{L}^{\ominus\langle\rangle\mathsf{cut}}$. Then there exists a derivation of $\Omega \vdash \tau$ in $\mathcal{L}^{\ominus\langle\rangle}$.*

Thus, the resulting logic is still consistent.

The proof of the admissibility of cut extends our notion of equivalences on proof-terms. Most importantly:

$$[M/u]N \quad \equiv \quad [M/u]_{\mathrm{LF}} N$$

Definition 5.7 (Values) *In the presence of a new right rule, we need to extend Definition 4.5:* $v ::= \mathrm{prev}\ v \mid \lambda u \in \tau.e \mid \mathrm{unit} \mid M$.

The simply typed λ-calculus has been successfully used in systems like Isabelle [Pau94]. Other systems, such as Twelf [SP98], rely on a logical framework with dependent types.

5.2 The Dependently-Typed Logical Framework

By adding dependent types to this formulation of the simply-typed λ-calculus we arrive at the dependently typed logical framework λ^P, or often called LF [HHP93]. In this framework, types may be indexed by objects, and function types assign names to their arguments: $A, B ::= a \mid A\ M \mid \Pi x : A.B$. The syntactic category of objects remains unchanged, but keep in mind that λ-binders are dependently typed as well.

All types are valid in the simply-typed case, but we must distinguish between valid types and invalid ones in the dependently typed case. The *kind* system of LF acts as a type system for types. Valid kinds are defined a follows: $K ::= \mathrm{type} \mid \Pi x : A.K$. The typing rules and kinding rules of

LF may not be as well-known as those of the simply typed λ-calculus, but they are standard [HHP93]. We write $\Gamma \vdash M : A$ for valid objects and $\Gamma \vdash A : K$ for valid types. In addition, for both kinds and types, when we have $\Pi x : A$ and the x does not occur free in the body we use the standard $\rightarrow$ as shorthand.

Theorem 5.8 (Canonical forms) *Every well-typed object in the dependently-typed λ-calculus possesses a unique canonical form.*

Example 5.9 (Natural deduction calculus) Let $A, B ::= A \Rightarrow B \mid p$ be the language of formulas.

$$\begin{array}{ll} & \mathrm{o} : \mathrm{type}, \\ \ulcorner A \Rightarrow B \urcorner = \ulcorner A \urcorner \Rightarrow \ulcorner B \urcorner & \Rightarrow : \mathrm{o} \rightarrow \mathrm{o} \rightarrow \mathrm{o} \end{array}$$

We write $\mathcal{D} ::\vdash A$ if $\mathcal{D}$ is a derivation in the natural deduction calculus. $\vdash A$ is a hypothetical judgment as impi shows below.

$$\frac{\begin{array}{c} \overline{\vdash A}\, u \\ \vdots \\ \vdash B \end{array}}{\vdash A \Rightarrow B}\,\mathsf{impi} \qquad \frac{\vdash A \quad \vdash A \Rightarrow B}{\vdash B}\,\mathsf{impe}$$

Natural deduction derivation $\mathcal{D} ::\vdash A$ are encoded in LF as $\ulcorner \mathcal{D} \urcorner : \mathrm{nd}\ \ulcorner A \urcorner$, which gives rise to the following signature.

$$\begin{array}{l} \mathrm{nd} : \mathrm{o} \rightarrow \mathrm{type}, \\ \mathrm{impi} : (\mathrm{nd}\ A \rightarrow \mathrm{nd}\ B) \rightarrow \mathrm{nd}\ (A \Rightarrow B), \\ \mathrm{impe} : \mathrm{nd}\ (A \Rightarrow B) \rightarrow \mathrm{nd}\ A \rightarrow \mathrm{nd}\ B. \end{array}$$

Theorem 5.10 (Substitution [HHP93]) *If $\Gamma, x : A \vdash B : K$ and $\Gamma \vdash M : A$ then $\Gamma \vdash [M/x]B : [M/x]K$.*

As in the simply typed case, the existence of canonical forms is instrumental for encodings to be adequate. The dependently typed logical framework is significantly more expressive than the simply typed one as it permits adequate encodings of proof systems, type systems, and operational semantics that arise in logic design and programming languages theory.

Due to our formulation, we can very naturally extend $\mathcal{L}^{\ominus\langle\rangle}$ with dependencies. For types, we replace $\tau \supset \sigma$ with $\Pi u \in \tau\, .\sigma$ (not to be confused with LF's dependent function type $\Pi x : A\, .B$). For completeness reasons, we add

$\Sigma u \in \tau\,.\sigma$ for dependent products. Expressions remain unchanged except that we add proof-terms for products, $(e_1,\ e_2)$ and let $(w_1, w_2) = u$ in e. In addition we save the $\supset$ notation to refer to an instance of Π which is not dependent. And we update our shifting operations as:

$$[\![\Sigma u \in \tau\,.\sigma]\!] = \Sigma u \in \tau\,.\sigma$$

$$[\![(e_1, e_2)]\!]^*_\diamond = (e_1, e_2)$$

$$[\![\text{let } (w_1, w_2) = u \text{ in } e]\!]_\diamond = \text{let } (w_1, w_2) = u \text{ in } [\![e]\!]_\diamond$$

$$[\![\text{let } (w_1, w_2) = u \text{ in } e]\!]^*_\diamond = \text{let } (w_1, w_2) = u \text{ in } [\![e]\!]^*_\diamond$$

We write $[e/u]\tau$ as notation for an explicit substitution on the now dependent types. In addition, we define $[e/u]\Omega = \Omega'$, where Ω' is formed by transforming every $(w \in \sigma)$ in Ω into $(w \in ([e/u]\sigma))$ in Ω'

Notice that substitution on types is *not* defined everywhere. Namely, we do not allow dependencies on meta-level computation.

$$\begin{array}{rcl}
[e/u]\top & = & \top \\
[e/u]\bot & = & \bot \\
[e/u](\Pi u' \in \tau\,.\sigma) & = & \Pi u' \in [e/u]\tau\,.[e/u]\sigma \\
[e/u](\Sigma u' \in \tau\,.\sigma) & = & \Sigma u' \in [e/u]\tau\,.[e/u]\sigma \\
[(\text{prev } e)/u](\ominus\tau) & = & \ominus([e/u]\tau) \\
[(\text{prev}^k\ M)/u]\langle A\rangle & = & \langle [M/u]_{\text{LF}} A\rangle \\
[e/u]\tau, u \text{ not free in } \tau & = & \tau \\
\text{None of the above match, } [e/u]\tau & = & \text{undefined}
\end{array}$$

With the introduction of dependencies all judgments and inference rules get significantly more complicated. For example, we now need to distinguish between well-formed and ill-formed formulas. We write $\Omega \vdash \tau$ wff for well-

formed formulas.

$$\frac{}{\Omega \vdash \top \text{ wff}}\,\top\mathsf{wff} \qquad \frac{}{\Omega \vdash \bot \text{ wff}}\,\bot\mathsf{wff}$$

$$\frac{[\Omega]_{\mathrm{LF}} \vdash A : \text{type}}{\Omega \vdash \langle A \rangle \text{ wff}}\,\langle\rangle\mathsf{wffR}$$

$$\frac{\Omega \vdash \tau \text{ wff} \quad \Omega, u \in \tau \vdash \sigma \text{ wff}}{\Omega \vdash \Pi u \in \tau\,.\sigma \text{ wff}}\,\Pi\mathsf{wffR}$$

$$\frac{\Omega \vdash \tau \text{ wff} \quad \Omega, u \in \tau \vdash \sigma \text{ wff}}{\Omega \vdash \Sigma u \in \tau\,.\sigma \text{ wff}}\,\Sigma\mathsf{wffR}$$

$$\frac{\Omega^{-\ominus} \vdash \tau \text{ wff}}{\Omega \vdash \ominus\tau \text{ wff}}\,\ominus\mathsf{wffR}$$

Theorem 5.11 (Admissibility of cutT) *If* $\mathcal{D} :: \Omega_1 \vdash e \in \tau$ *and* $\mathcal{E} :: \Omega_1, u \in \tau, \Omega_2 \vdash \varrho$ wff *and* $[e/u]\Omega_2$ *is not undefined and* $[e/u]\varrho$ *is not undefined, then* $\Omega_1, [e/u]\Omega_2 \vdash [e/u]\varrho$ wff.

Proof: This proof simply goes by induction over derivation $\mathcal{E}$ utilizing the definition of $[e/u]\varrho$. □

With this refined notion of formulas and a notion of formula level reduction, we now revisit the inference system of $\mathcal{L}^{\ominus\langle\rangle}$ and extend it by dependencies and in place substitutions on types and contexts. Note that an important *implicit* condition on all the rules is that all elements of Ω are well-formed, as well as the resulting type.

$$\frac{(u \in \tau) \text{ in } \Omega}{\Omega \vdash u \in [\![\tau]\!]}\,\mathsf{ax}$$

$$\frac{}{\Omega \vdash \text{unit} \in \top}\,\top\mathsf{R} \qquad \text{no rule } \top\mathsf{L}$$

$$\text{no rule } \bot\mathsf{R} \qquad \frac{}{\Omega_1, u \in \ominus^k \bot, \Omega_2 \vdash \text{void } u \in [\![\sigma]\!]}\,\bot\mathsf{L}$$

$$\frac{\Omega, u \in \tau \vdash e \in \sigma}{\Omega \vdash \lambda u \in \tau\,.e \in \Pi u \in \tau\,.\sigma}\,\Pi\mathsf{R}$$

$$\frac{\begin{array}{c}\Omega_1, u \in \ominus^k(\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash e \in \tau \\ \Omega_1, u \in \ominus^k(\Pi u' \in \tau\,.\sigma), \Omega_2, w \in [\![[e/u']\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash \text{let } w = u \cdot e \text{ in } f \in \varrho}\,\Pi\mathsf{L}$$

$$\frac{\Omega^{-\ominus} \vdash e \in \tau}{\Omega \vdash \text{prev } e \ominus \tau}\,\ominus\mathsf{R} \qquad \frac{[\Omega]_{\mathrm{LF}} \vdash M : A}{\Omega \vdash M \in \langle A \rangle}\,\langle\rangle\mathsf{R}$$

$$\frac{\Omega \vdash e_1 \in \tau \quad \Omega \vdash e_2 \in [e_1/u]\sigma}{\Omega \vdash (e_1, e_2) \in \Sigma u \in \tau\,.\sigma}\,\Sigma\mathsf{R}$$

$$\frac{\Omega_1, u \in \ominus^k(\Sigma u' \in \tau\,.\sigma), \Omega_2, w_1 \in [\![\tau]\!], w_2 \in [w_1/u'][\![\sigma]\!] \vdash e \in \varrho}{\Omega_1, u \in \ominus^k(\Sigma u' \in \tau\,.\sigma), \Omega_2 \vdash \text{let } (w_1, w_2) = u \text{ in } e \in \varrho}\,\Sigma\mathsf{L}$$

This system of inference rules will be called $\mathcal{L}^{\ominus\Pi}$ if we want to draw attention to the logical character and the $\lambda^{\ominus\Pi}$-calculus if it is used to describe computation. The cut-rule is also admissible for this calculus.[2]

Theorem 5.12 (Admissibility of cut) *If $\mathcal{D} :: \Omega_1 \vdash e \in \tau$ and $\mathcal{E} :: \Omega_1, u \in \tau, \Omega_2 \vdash f : \sigma$ and $[e/u]\Omega_2$ is not undefined and $[e/u]\sigma$ is not undefined, then $\Omega_1, [e/u]\Omega_2 \vdash [e/u]f \in [e/u]\sigma$ for some proof term $[e/u]f$.*

Proof: By induction on the cut formula τ, and simultaneously over derivations $\mathcal{D}$ and $\mathcal{E}$. See Appendix for Proof. □

Thus we add the cut-rule to $\mathcal{L}^{\ominus\Pi}$ and obtain a logic with cut called

[2]Perhaps we should change the name to include an $\langle\rangle$

$\mathcal{L}^{\ominus\Pi\mathsf{cut}}$ (and also a calculus called $\lambda^{\ominus\Pi\mathsf{cut}}$).

$$\frac{\Omega_1 \vdash e \in \tau \quad \Omega_1, u \in \tau, \Omega_2 \vdash f \in \sigma}{\Omega_1, [e/u]\Omega_2 \vdash [e/u]f \in [e/u]\sigma}\,\mathsf{cut}$$

Definition 5.13 (Values) *The following are the values extending Definition 5.7.*
$v ::= \text{prev}\ v \mid \lambda u \in \tau\,.e \mid \text{unit} \mid M \mid (v_1, v_2)$.

Theorem 5.14 (Cut-Elimination) *Let $\Omega \vdash e \in \tau$ be a derivation in $\mathcal{L}^{\ominus\Pi\mathsf{cut}}$. Then there exists a derivation of $\Omega \vdash v \in \tau$ in $\mathcal{L}^{\ominus\Pi}$.*

Theorem 5.15 (Completeness of Substitution on Values)
$[v/u]\tau$ is not undefined.

Proof: *By analysis of the definition of substitution.* □

Example 5.16 (Distributivity of $\ominus$) The distributivity law depicted in Example 4.7 can be carried over into the dependently setting of $\mathcal{L}^{\ominus\Pi\mathsf{cut}}$ as

$$\ominus(\Pi u \in [\![\tau]\!]\,.[\![\sigma]\!]) \supset \Pi w \in [\![\ominus\tau]\!]\,.\ominus[\![[w/u]\sigma]\!]$$

whose entire derivation is isomorphically captured by the proof term

$$\lambda u \in \ominus(\Pi u \in [\![\tau]\!]\,.[\![\sigma]\!])\,.\lambda w \in \ominus[\![\tau]\!]\,.\\ \text{prev}\ (\text{let}\ u' = u \cdot w\ \text{in}\ u')$$

in the $\lambda^{\ominus}$-calculus.

This concludes the logical motivation for this paper. The remainder is dedicated to interpreting the corresponding $\lambda^{\ominus\Pi}$ calculus as a functional programming language. We first add an operator for programming with higher-order encodings, which is just an admissible rule in $\lambda^{\ominus\Pi}$. We will then convert it to its natural deduction form, $\lambda^{\mathcal{D}}$, and then proceed to add operators that facilitate case analysis and recursion. It is beyond the scope of this paper to give a completely logical account for these new concepts. For example, in order to still guarantee cut-elimination, we would have to show that all cases are covered, and that the recursion always terminates.

6 Higher-Order Abstract Syntax

The temporal features of the $\lambda^{\ominus\Pi}$-calculus are instrumental when programming with higher-order abstract syntax and hypothetical judgments. When programming with higher-order abstract syntax, one always runs into the problem that computation somehow has to progress under a representation-level λ-binder. There are many different views on how this can best be achieved. For example, one may use a custom tailored iteration construct [SDP01] or one may use an explicit ν-operator that introduces new parameters and abstracts the result [Tah04, Sch05]. The main difficulty, particularly with the latter technique, is to express what to do with the new parameters upon return. To always abstract it may be too rigid because it is possible that the result does not depend on the parameter. However, to allow the programmer to express when to abstract it is dangerously general because now parameters are permitted to escape their scope.

The temporal properties of the $\lambda^{\ominus\Pi}$-calculus give the programmer the ability to express which parameters should be abstracted and when, while simultaneously enforcing that parameters cannot escape their scope. We can observe that if an expression evaluates to a value v of type $\ominus\langle B\rangle$, then anything that lies in Ω without a $\ominus$ is invisible and can be removed without destroying the derivability that v is a value of type $\langle B\rangle$. This was the critical notion behind the design of this system. This operation is also called strengthening and it is a property of the temporal part of the calculus.

Theorem 6.1 (Strengthen)
If $\Omega, u \in [\![\tau]\!] \vdash e \in \ominus\sigma$, *then* $\Omega \vdash e \in \ominus\sigma$

Proof: *By induction over the typing derivation.* □

Thus, the following rule is admissible in temporal logic with past-time

$$\frac{\Omega, u \in [\![\tau]\!] \vdash \ominus\sigma}{\Omega \vdash \ominus\sigma}\,\mathsf{new}$$

which can be endowed with a proof term $\nu u \in \tau\,.e$, to form a rule for the $\lambda^{\ominus\Pi}$-calculus.

$$\frac{\Omega, u \in [\![\tau]\!] \vdash e \in \ominus\sigma}{\Omega \vdash \nu u \in \tau\,.e \in \ominus\sigma}\,\mathsf{new}$$

Notice that new will be used to introduce parameters into the context which we can use to go under λ-binders.

Now Finally, our system allows us to reason about the past, but we now need a way to move our point-of-reference into the future. We define the dual of $\Omega^{-\ominus}$ as $\Omega^{+\ominus}$ which simply adds a $\ominus$ in front of everything in Ω.

Theorem 6.2 (Future)
If $\Omega^{+\ominus} \vdash e \in \ominus\tau$*, then* $\Omega \vdash \mathsf{next}(e) \in \tau$*, where:*

$$\begin{array}{rcl} \mathsf{next}(\text{prev } e) & = & e \\ \mathsf{next}(\text{let } w = u \cdot\ e \text{ in } f) & = & \mathsf{next}(f) \\ \mathsf{next}(\text{let } (w_1, w_2) = u \text{ in } e) & = & \mathsf{next}(e) \end{array}$$

Proof: *By induction over the typing derivation.* □

Thus, the following rule is admissible in temporal logic with past-time

$$\frac{\Omega^{+\ominus} \vdash \ominus\tau}{\Omega \vdash \tau}\,\mathsf{future}$$

which can be endowed with a proof term (next e), to form a rule for the $\lambda^{\ominus\Pi}$-calculus.

$$\frac{\Omega^{+\ominus} \vdash e \in \ominus\tau}{\Omega \vdash \text{next } e \in \tau}\,\mathsf{future}$$

7 Natural Deduction

Since we are now shifting our focus from a logical motivation to functional programming, we define the corresponding natural deduction version of the system, $\lambda^{\mathcal{D}}$, which we will now focus on.

Types remain unchanged (except that we remove $\bot$) and are:

$$\tau, \sigma ::= \top \mid \Pi u \in \tau\,.\sigma \mid \ominus\tau \mid, \Sigma u \in \tau\,.\sigma \mid \langle A \rangle$$

Expressions are now:

$$\begin{array}{rcl} e, f & ::= & u \mid \text{unit} \mid \lambda u \in \tau\,.e \mid e_1\ e_2 \mid \text{prev } e \mid \text{next } e \\ & & \mid (e_1, e_2) \mid e\,.\mathsf{fst} \mid e\,.\mathsf{snd} \mid \nu u \in \tau\,.e \mid M \end{array}$$

$$\frac{(u \in \tau) \text{ in } \Omega}{\Omega \vdash u \in [\![\tau]\!]}\,\mathsf{ax} \qquad \frac{}{\Omega \vdash \text{unit} \in \top}\,\mathsf{top}$$

$$\frac{\Omega, u \in \tau \vdash e \in \sigma}{\Omega \vdash \lambda u \in \tau\,.e \in \Pi u \in \tau\,.\sigma}\,\mathsf{lam}$$

$$\frac{\Omega \vdash e_1 \in (\Pi u' \in \tau\,.\sigma) \quad \Omega \vdash e_2 \in \tau}{\Omega \vdash e_1\ e_2 \in [\![[e_1/u']\sigma]\!]}\,\mathsf{app}$$

$$\frac{\Omega^{-\ominus} \vdash e \in \tau}{\Omega \vdash \text{prev}\ e \ominus \tau}\,\mathsf{past} \qquad \frac{\Omega^{+\ominus} \vdash e \in \ominus\tau}{\Omega \vdash \text{next}\ e \in \tau}\,\mathsf{future}$$

$$\frac{\Omega \vdash e_1 \in \tau \quad \Omega \vdash e_2 \in [e_1/u]\sigma}{\Omega \vdash (e_1, e_2) \in \Sigma u \in \tau\,.\sigma}\,\mathsf{pair}$$

$$\frac{\Omega \vdash e \in (\Sigma u \in \tau\,.\sigma)}{\Omega \vdash e\,.\mathsf{fst} \in [\![\tau]\!]}\,\mathsf{fst} \qquad \frac{\Omega \vdash e \in (\Sigma u \in \tau\,.\sigma)}{\Omega \vdash e\,.\mathsf{snd} \in [e\,.\mathsf{fst}/u][\![\sigma]\!]}\,\mathsf{snd}$$

$$\frac{\Omega, u \in [\![\tau]\!] \vdash e \in \ominus\sigma}{\Omega \vdash \nu u \in \tau\,.e \in \ominus\sigma}\,\mathsf{new}$$

$$\frac{[\Omega]_{\mathrm{LF}} \vdash M : A}{\Omega \vdash M \in \langle A \rangle}\,\langle\rangle\mathsf{I}$$

Since we extend our term language to include $e\,.\mathsf{fst}$ and $e\,.\mathsf{snd}$, we need to revisit its interaction with the LF level. But we must be careful since we do not supports pairs on the LF-level. It would be quite natural to do if we had pairs on the LF-level, so this technique can be viewed as a way to circumvent the necessity of adding pairs to LF. By analysis of the sequent calculus version ($\Sigma\mathsf{L}$) we see that we can open a pair and allow dependencies. To be consistent, we need a way to access $e\,.\mathsf{fst}$ and $e\,.\mathsf{snd}$ on the LF-level. Therefore, we need to extend our Thinning operation as follows.

Definition 7.1 (Thinning for Natural Deduction)

$$[\Omega]_{LF} = \begin{cases} \cdot & \text{if } \Omega = \cdot \\ [\Omega']_{LF}, x : A & \text{if } \Omega = \Omega', x \in \ominus^k \langle A \rangle \\ \left[\begin{array}{l} \Omega', e\,.\mathsf{fst} \in \tau, \\ e\,.\mathsf{snd} \in \\ \quad [e\,.\mathsf{fst}/u][\![\sigma]\!] \end{array} \right]_{LF} & \text{if } \Omega = \begin{array}{l} \Omega', \\ e \in \ominus^k(\Sigma u \in \tau\,.\sigma) \end{array} \\ [\Omega]_{LF} & \begin{array}{l} \text{if } \Omega = \Omega', u \in \tau \\ \text{and not match Above} \end{array} \end{cases}$$

So now the Thinning operation is actually doing some work. Namely, on the meta-level, $e\,.\mathsf{fst}$ is an operation which takes the first element of e, whereas when we cast e to the LF-level, it creates $e\,.\mathsf{fst}$ which is just a variable name. This explains why we chose the syntax $e\,.\mathsf{fst}$ instead of $(\mathsf{fst}\ e)$ on the meta-level – because the former can be treated as a variable name for LF. And it is imperative to state that in our rule snd, when we see the substitution $[e\,.\mathsf{fst}/u]$, this $e\,.\mathsf{fst}$ refers to an LF-variable name. We can do this because u must have type $\langle A \rangle$ or u must not occur free in σ.

Notice that $\bot\mathsf{L}$ does not serve us in a programming paradigm, so we omit it from the system. However, it is straightforward to show an equivalence between our sequent-calculus version and this natural deduction version extended with the following corresponding $\bot\mathsf{L}$ rule:

$$\frac{\Omega \vdash e \in \bot}{\Omega \vdash \text{void } e \in [\![\sigma]\!]}\ \mathsf{bot}$$

It is important to note that our $\mathsf{openPair}$ rule which corresponds to $\Sigma\mathsf{L}$ has kept its sequent flavor in lieu of the typical fst and snd deconstructors. We could have chosen the latter approach but that would needlessly complicate type-level substitutions. Currently, types can only depend on LF terms and are not allowed to depend on meta-level computation. However, if we introduce meta-level constructs of fst and snd and would like to keep them as dependent, then $\mathsf{snd}\ e$ would have a type of the form $[(\mathsf{fst}\ e)/u]\sigma$, and we would have to allow types to handle these meta-level constructs. The only thing we lose with this approach is that we lose correspondences between multiple openings of the same pair which is inconsequential and can be viewed as poor programming practice.

When looking at term-level substitution in $\lambda^{\ominus\Pi}$ we notice that the ones corresponding to left-commutative cases of cut propagate what is being substituted which is counter to what we expect from substitution. For instance,

in order to substitute $[(\text{let } w = u' \cdot e' \text{ in } f)/u]$ into an M, where u occurs free, the following left-commutative case is the only one that applies:

$$[\frac{(\text{let } w = u' \cdot e' \text{ in } f)}{u}]M \equiv (\text{let } w = u' \cdot e' \text{ in } [f/u]M)$$

and in our natural deduction version, $\lambda^{\mathcal{D}}$, this would correspond to:

$$[\frac{(e_1\ e_2)}{u}]M \equiv (\lambda u \in \tau\,.M)\ (e_1\ e_2), \quad \text{for some } \tau$$

Therefore, we see that if we use a *lazy* operational semantics, function application may simply do nothing and result in itself! However, if we adopt an eager operational semantics, we get a much more natural definition of substitution and typical behavior.
Values in $\lambda^{\mathcal{D}}$ remain the same as in Definition 5.13, and are:

$$v ::= \text{prev } v \mid \lambda u \in \tau\,.e \mid \text{unit} \mid M \mid (v_1, v_2)$$

By inspection of the cases we see that type-level substitution is only defined when we are substituting a value (in particular a value of the form $\text{prev}^k\ M$), or when the substitution is redundant (variable does not occur free). Thus we can write our substitution on types in $\lambda^{\mathcal{D}}$ as:

$$\begin{aligned}
[v/u](\Pi u' \in \tau\,.\sigma) &= \Pi u' \in [v/u]\tau\,.[v/u]\sigma \\
[v/u](\Sigma u' \in \tau\,.\sigma) &= \Sigma u' \in [v/u]\tau\,.[v/u]\sigma \\
[(\text{prev } v)/u](\ominus\tau) &= \ominus([v/u]\tau) \\
[(\text{prev}^k\ M)/u]\langle A\rangle &= \langle [M/u]_{\text{LF}} A\rangle \\
[e/u]\tau, u \text{ not free in } \tau &= \tau \\
\text{None of the above match, } [e/u]\tau &= \text{undefined}
\end{aligned}$$

Since we are restricted to values, $[\![v]\!]^*_\diamond$ simply refers to removing all leading prev's from v. And finally if we restrict our term-level substitutions to only substitute values, we get a much more simplistic definition of term-level substitution in $\lambda^{\mathcal{D}}$:

Definition 7.2 *Substitution for Delphin Values*

$$
\begin{array}{rcl}
[v/u]u' & \equiv & \begin{cases} [\![v]\!]^*_\diamond & \textit{if } u = u' \\ u' & \textit{otherwise} \end{cases} \\
[v/u]\text{unit} & \equiv & \text{unit} \\
[v/u](\lambda u' \in \tau\,.f) & \equiv & \lambda u' \in [v/u]\tau\,.[v/u]f \\
[v/u](e_1\ e_2) & \equiv & ([v/u]e_1)\ ([v/u]e_2) \\
[v/u](e\,.\mathsf{fst}) & \equiv & ([v/u]e)\,.\mathsf{fst} \\
[v/u](e\,.\mathsf{snd}) & \equiv & ([v/u]e)\,.\mathsf{snd} \\
[v/u](\text{prev } f) & \equiv & \begin{cases} \text{prev } ([v'/u]f) & \textit{if } v = \text{prev } v' \\ \text{prev } f & \textit{otherwise} \end{cases} \\
[v/u]N & \equiv & \begin{cases} [M/u]_{LF}N & \textit{if } v = M \\ N & \textit{otherwise} \end{cases}
\end{array}
$$

Therefore, we will define Delphin to utilize an eager operational semantics.

8 Case Analysis

The idea of case analysis is conceptually simple, but bares many interesting facts, primarily due to the presence of dependent types. In the simply typed scenario, given a variable $u \in \langle \text{exp} \rangle$, where expressions are defined in Example 5.2, u will be instantiated during run-time by a canonical object $\Gamma \vdash M : \text{exp}$. As our logical frameworks possesses the canonical form properties, a straightforward analysis yields that $M = \text{lam}\ (\lambda x : \text{exp}\,.N\ x)$, or $M = \text{app}\ N_1\ N_2$ or $M = x$, where $x : \text{exp}$ is in Γ. In the first case we may assume that we can compute $N : \text{exp} \to \text{exp}$, a variable of functional type, from M. In the second case we can compute $N_1 : \text{exp}$ and $N_2 : \text{exp}$. In the third case, we can extract *parameter* x out of M. It is this last case which we need to pay careful attention to. When considering cases over a datatype we need to consider all the ways to construct it and in addition the possibility for parameters of that type to exist (which is created using new).

Theorem 8.1 (Case Analysis)
If $\mathcal{D} :: [\Omega_1]_{LF} \vdash M : A$ *in canonical form*
and $\Omega_1, u \in \langle A \rangle, \Omega_2$ *is a valid context.*
and $\Omega_1, [M/u]\Omega_2 \vdash e \in [M/u]\tau$
and $[\Omega_1]_{LF} \vdash N[M_1/u_1]...[M_n/u_n] \equiv M : A$
where $[\Omega_1]_{LF} \vdash M_i : A_i$ *or* $M_i = w_i : A_i$ *in* $[\Omega_1]_{LF}$.

then $\Omega_1, u_1 \in \langle A_1 \rangle \ldots u_n \in \langle A_n \rangle, [N/u]\Omega_2 \vdash e \in [N/x]\tau$

Proof: By induction on the structure of $\mathcal{D}$. □

In the dependently typed case, we prefer to conduct case analysis over one or more assumptions simultaneously. The reason being that variables declared in Ω may occur in other types as index arguments to LF type families, and only by considering cases over several variables simultaneously can we reason whether all cases are covered or some are forgotten. Thus we add a switching statement, g, that comprises several cases to our expression language. The new syntactic category g is defined as follows.

$$\begin{array}{rcl} e, f & ::= & \ldots \mid \text{switch } (g_1 \mid \ldots \mid g_n) \\ g & ::= & \text{case } u = p \mid e \\ p & ::= & \epsilon u \in \tau\,.p \mid \nabla u \in \tau\,.p \mid M \text{ in } g \end{array}$$

The ϵ and ∇-quantified variables represent the pattern variables that can occur in M. We use ∇ to refer to parameters (introduced by new) and ϵ stands for typical pattern-variables. Notice that ϵ and ∇ are treated identically in typing but are operationally distinguished where the former can be instantiated with anything but the latter can only be instantiated with parameters (variables in the context).

We write $\Omega \vdash_g g \in \tau$ and $\Omega_1, u \in \sigma, \Omega_2 \vdash_p^u p \in \tau$ for the typing judgment for cases, which is defined below.

$$\frac{\Omega \vdash e \in \tau}{\Omega \vdash_g e \in \tau}\ \mathsf{switchBase}$$

$$\frac{\Omega_1, u \in \sigma, \Omega_2 \vdash_p^u p \in \tau}{\Omega_1, u \in \sigma, \Omega_2 \vdash_g \text{case } u = p \in \tau}\ \mathsf{switchCase}$$

$$\frac{\Omega_1, w \in \tau, u \in \sigma, \Omega_2 \vdash_p^u p \in \tau}{\Omega_1, u \in \sigma, \Omega_2 \vdash_p^u (\epsilon w \in \tau\,.p) \in \tau}\ \mathsf{patternVar}$$

$$\frac{\Omega_1, w \in \tau, u \in \sigma, \Omega_2 \vdash_p^u p \in \tau}{\Omega_1, u \in \sigma, \Omega_2 \vdash_p^u (\nabla w \in \tau\,.p) \in \tau}\ \mathsf{patternParam}$$

$$\frac{\Omega_1 \vdash M \in \sigma \quad \Omega_1, [M/u]\Omega_2 \vdash_g g \in [M/u]\tau}{\Omega_1, u \in \sigma, \Omega_2 \vdash_p^u (M \text{ in } g) \in \tau}\ \mathsf{patternBase}$$

And finally, we give the typing rule in $\lambda^{\mathcal{D}}$ for cases:

$$\frac{\begin{array}{c}\text{for all } 1 \leq i \leq m \\ \Omega \vdash g_i \in \tau\end{array}}{\Omega \vdash \text{switch } (g_1 \mid \ldots \mid g_m) \in \tau}\ \mathsf{case}, m \geq 0$$

The interaction between the cut rule and the switch rule is subtle. The cut-elimination procedure is designed to drive a cut deeper and deeper into a term until it disappears. If we were to adopt the same strategy, the cut would need to be pushed into each case of a switch statement. This is operationally not only horrendously inefficient, but may even render the cut-elimination procedure unsound, i.e. non-normalizing, in the case of general recursion (as discussed in Section 9).

Instead, we allow cuts and substitutions to accumulate outside a switch until all variables that are considered in a particular case are known. Subsequently, all ϵ-quantified variables are computed by matching the patterns against the LF objects contained in the substitutions, forming new substitutions, which are then pushed along with all other delayed cuts deeper into a particular case. Due to the higher-order nature of our logical framework, pattern-matching is in general not decidable. In our experience, a decidable fragment of higher-order pattern matching (which also works for the dependently type LF) is that of Miller patterns [Mil91], into which all of our examples fall (especially those in Section 10). In addition, a strictness analysis [PS98] on ϵ bound variables can ensure that their respective instantiations are uniquely determined during program execution.

Delaying cuts is not a straightforward application as they need to commute with substitutions, and substitutions need to commute with each other to transport information to the particular case. We can show the following properties.

Lemma 8.2 (Commutativity) *The following commutativity*[3] *property holds. Notice that we are using the substitution for $\lambda^{\mathcal{D}}$, given in Definition 7.2.*

1. $[v_1/u]([v_2/w]f) = [\frac{[v_1/u][v_2]}{w}]([v_1/u]f)$

Proof: By inspection of the rules. □

The following equivalence rules require that ϵ and ∇-bound variables are only instantiated by well-typed objects. Let the equation be valid in context Ω.

[3] Isn't this distributivity, I called it that in appendix???

$$\begin{aligned}
\text{switch } (g_1 \mid \ldots \mid g_n) &\equiv g_i \text{ for some } i && (1)\\
[N/u]\text{case } u = N \text{ in } e &\equiv e && (2)\\
\nabla w \in \ominus^k\langle B\rangle\,.p &\equiv [w'/w]p && (3)\\
\epsilon w \in \ominus^k\langle B\rangle\,.p &\equiv [N/w]p && (4)
\end{aligned}$$

In the above equivalences, both z and N must be appropriately well-typed. We give some examples of case analysis in Section 10.

9 General Recursion

Next, we extend the $\lambda^{\mathcal{D}}$-calculus by an explicit recursion operator. In general, recursion immediately breaks the logical meaning of the calculus, as non-terminating programs cannot be interpreted as proofs unless it is guaranteed to be well-founded. However, in this paper we place the logical interpretation aside and focus on the computational aspects. Thus we add a general recursion principle.

$$e, f ::= \ldots \mid \mu u \in \tau\,.e$$

which satisfies the usual specification.

$$\frac{\Omega, u \in \tau \vdash e \in \tau}{\Omega \vdash (\mu u \in \tau\,.e) \in \tau}\,\mathsf{rec}$$

and is interpreted operationally as

$$\Gamma \vdash (\mu u \in \tau\,.e) \equiv [\mu u \in \tau\,.e/u]e. \qquad (5)$$

Unrolling this fix-point means to drive it deeper and deeper into the expression until no occurrences are left. We give some examples of the recursion operator in Section 10.

10 Examples

Next, we give a few illustrative toy examples of $\lambda^{\mathcal{D}}$-programs. We first introduce some extra syntactic sugar so we can typeset our programs a little

nicer.

$$\begin{array}{rcl}
\text{let } u = e \text{ in } f & \equiv & (\lambda u \in \tau\,.f)e \\
 & & \quad \text{when } \tau \text{ is inferable.} \\
\text{let } (u, v) = e \text{ in } f & \equiv & [e\,.\mathsf{fst}/u][e\,.\mathsf{snd}/v]f \\
\textbf{fun } f\ (u_1 \in \tau_1)\ (u_2 \in \tau_2) & \equiv & \mu f \in (\Pi u_1 \in \tau_1\,.\Pi u_2 \in \tau_2\,. \\
\ldots (u_n \in \tau_n) : \sigma & & \ldots \Pi u_n \in \tau_n\,.\sigma)\,. \\
= e & & \lambda u_1 \in \tau_1 \ldots . \lambda u_n \in \tau_n\,.e
\end{array}$$

Example 10.1 (Structural Identity) Recall the encoding of the untyped λ-calculus from Example 5.2. We use the hypothetical judgment $E = F$ to define when two λ-expressions are equal.

$$\frac{E_1 = F_1 \quad E_2 = F_2}{E_1 @ E_2 = F_1 @ E_2}\,\mathsf{cp\_app} \qquad \frac{\begin{array}{c}\dfrac{}{x = x}\,u \\ \vdots \\ E = F\end{array}}{\mathbf{lam}\ x.\, E = \mathbf{lam}\ x.\, F}\,\mathsf{cp\_lam}^{x,u}$$

The judgment can be adequately represented in LF as type family cp and the two rules as two constants:

$$\begin{array}{lcl}
\text{cp} & : & \text{exp} \to \text{exp} \to \text{type}, \\
\text{cp_app} & : & \text{cp } E_1\ F_1 \to \text{cp } E_2\ F_2 \\
 & & \quad \to \text{cp } (\text{app } E_1 E_2)\ (\text{app } F_1\ F_2), \\
\text{cp_lam} & : & (\Pi x : \text{exp}\,.\text{cp } x\ x \to \text{cp } (E\ x)\ (F\ x)) \\
 & & \quad \to \text{cp } (\text{lam } E)\ (\text{lam } F).
\end{array}$$

As it is common practice in the logical framework LF we omit the leading Πs from the types as they can be easily inferred from the free variables that occur in the types. Next we define the function $\mathsf{cpfun} \in \Pi e \in \langle \text{exp} \rangle\,.\langle \text{cp } e\ e \rangle$.

```
fun cpfun (e ∈ ⟨exp⟩) : ⟨cp e e⟩
= switch
    case e = εE1 ∈ ⟨exp⟩ .εE2 ∈ ⟨exp⟩ .(app E1 E2) in
      let D1 = cpfun · E1 in
      let D2 = cpfun · E2 in
      cp_app D1 D2
  | case e = εE' ∈ ⟨exp → exp⟩ .(lam E') in
      next (ν(x ∈ ⟨exp⟩) .ν(u ∈ ⟨cp x x⟩) .
            let D' = cpfun · (E' x) in
            switch
              case D' = εD ∈ ⊖⟨Πx : exp .cp x x
                                → cp (E' x) (E' x)⟩ .
                             (D x u) in
              prev (cp_lam D)
  | case e = ∇x ∈ ⟨exp⟩ .∇u ∈ ⟨cp x x⟩ .x in u
```

We explain each of the three cases in turn. The $e = \text{app } E_1\ E_2$ case is straightforward. The result of the recursive calls is combined into the desired proof by cp_app.

The $e = \text{lam } E'$ case illustrates our ability to go under representation-level λ's using next and ν. next brings us one step into the future, its type $\ominus\tau$ ensures that the result can be interpreted in the present. Next, we create a new parameter x and recurse on $E'\ x$. The result is valid in the future. To ensure that the type checker knows that we eventually return to the present, we stipulate D to exist in the past, which is sufficient to guarantee that "prev cp_lam D" is valid now. Thus neither x nor u can escape their scope.

The final case is the parameter case $e = x$ for $x \in \langle\text{exp}\rangle$. There will be only one $u \in \langle\#\text{cp } x\ x\rangle$ in the context during runtime, which is promptly returned. This illustrates the difference between ∇ and ϵ.

Example 10.2 (Combinators) Recall the definition of the natural deduction calculus from Example 5.9. We will give an algorithmic procedure that

converts natural deduction derivations into the Hilbert calculus.

$$\frac{}{\vdash o \supset p \supset o}\,\mathsf{K} \qquad \frac{\vdash o \supset p \quad \vdash o}{\vdash p}\,\mathsf{MP}$$

$$\frac{}{\vdash (o \supset p \supset q) \supset (o \supset p) \supset (o \supset q)}\,\mathsf{S}$$

Following the standard Judgments-as-types paradigm, our encoding is:

$$\text{comb} : \text{o} \rightarrow \text{type},$$

$$\begin{array}{lcll} \ulcorner K \urcorner & = & \text{K} & : \text{comb}\ (A \Rightarrow B \Rightarrow A) \\ \ulcorner S \urcorner & = & \text{S} & : \text{comb}\ ((A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \\ & & & \qquad \Rightarrow A \Rightarrow C) \\ \ulcorner MP \urcorner & = & \text{MP} & : \text{comb}(A \Rightarrow B) \rightarrow \text{comb}\ A \rightarrow \text{comb}\ B \end{array}$$

Any of our simply-typed λ-expressions, exp can be converted into a combinator in a two-step algorithm. The first step is called bracket abstraction, or ba, which converts a parametric combinator (a representation-level function of type comb A $\rightarrow$ comb B) into a combinator with one less parameter (of type comb (A $\Rightarrow$ B)). Formally, ba is written as.

```
fun ba (A ∈ ⟨o⟩) (B ∈ ⟨o⟩) (F ∈ ⟨comb A → comb B⟩)
          : ⟨comb A ⇒ B⟩
=  switch
       case F = ∇z ∈ ⟨comb B⟩ .(λx : comb A .z) in (MP K z)
       | case B = A in
            case F = (λx : comb A .x) in
               (MP (MP S K) K)
       | case B = ϵC ∈ ⟨o⟩ .ϵD ∈ ⟨o⟩ .(C ⇒ D ⇒ C) in
            case F = (λx : comb A .K) in
             (MP K K)
       | case B = ϵC ∈ ⟨o⟩ .ϵD ∈ ⟨o⟩ .ϵE ∈ ⟨o⟩ .
            (C ⇒ D ⇒ E) ⇒ (C ⇒ D) ⇒ C ⇒ E in
            case F = (λx : comb A .S) in
             (MP K S)
       | case F = ϵC ∈ ⟨o⟩ .
            ϵD1 ∈ ⟨comb A → comb (C ⇒ B)⟩ .
            ϵD2 ∈ ⟨comb A → comb C⟩ .
            (λx : comb A .MP (D1 x) (D2 x)) in
               let D1' = ba · D1 in
               let D2' = ba · D2 in
                  (MP (MP S D1')) D2'
```

The first two cases of ba illustrate how to distinguish x, which is to be abstracted, from parameters that are introduced in the function convert, which we discuss next. The function convert traverses natural deduction derivation and uses ba to convert them into Hilbert style combinators.

```
fun convert (A ∈ ⟨o⟩) (D ∈ ⟨nd A⟩) : ⟨comb A⟩
= switch
    case D = ∇u ∈ ⟨comb A → ndA⟩ .∇c ∈ ⟨comb A⟩ .(u c) in c
  | case D = εB ∈ ⟨o⟩ .εC ∈ ⟨o⟩ .
             εD1 ∈ ⟨nd (B ⇒ C)⟩ .εD2 ∈ ⟨nd B⟩ .(impi D1 D2) in
               let C1 = convert (B ⇒ C) D1 in
               let C2 = convert B D2 in
                 MP C1 C2
  | case D = εB ∈ ⟨o⟩ .εC ∈ ⟨o⟩ .εD' ∈ ⟨nd (B ⇒ C)⟩ .(lam D') in
        next (ν(u ∈ ⟨(comb B) → (nd B)⟩) .ν(c ∈ ⟨comb B⟩) .
          let D'' = convert C (E (u c)) in
          switch case D'' = εD''' ∈ ⊖⟨comb B → comb C⟩ .(D''' c) in
          prev (ba B C D''')
```

The last case illustrates how a parameter of functional type may introduce information to be used when the parameter is matched. Rather than introduce a parameter x of type (exp B), we introduce a parameter of type (comb B) $\to$ (exp B) that carries a combinator as "payload." In our example, the payload is another parameter c : (comb B), the image of x under convert. This technique is known as payload-carrying parameters and is applicable to a wide range of examples.

11 Related Work

Both research with higher-order and abstract syntax and research with programming with dependent types are related to the $\lambda^{\ominus\Pi}$-calculus.

Higher-order abstract syntax. The $\lambda^{\ominus\Pi}$-calculus is the result of many years of design, originally inspired by an extension to ML proposed by Dale Miller [Mil90], the type theory $\mathcal{T}_\omega^+$ [Sch01], and Hofmann's work on higher-order abstract syntax [Hof99]. It extends the ∇-calculus [Sch05], by dependent types, but more importantly, it gives a clean and logical account of programming with higher-order encodings without being confined by a rigidly defined iteration construct as proposed in [SDP01] and [Lel98], which separates representation-level from computation-level functions via a modality within one single λ-calculus.

Closely related to our work are programming languages with freshness [GP99],

which provide a built-in α-equivalence relation for first-order encodings but provide neither $\beta\eta$ nor any support for higher-order encodings. Also closely related are meta-programming languages, such as MetaML [TS00], which provide hierarchies of computation levels but do not single out a particular level for representation. Many other attempts have been made to combine higher-order encodings and functional programming, in particular Honsell, Miculan, and Scagnetto's embedding of the π-calculus in Coq[HMS01], and Momigliano, Amber, and Crole's Hybrid system [MAC03].

Dependent types The $\lambda^{\ominus\Pi}$-calculus is not the first to attempt to combine dependent-types with functional programming. DML [XP99] used to have index datatypes, whose index domains were recently generalized to LF objects [CX05] to form the ATS/LF system. Both Cayenne [Aug98] and Epigram [MM04] are dependently typed functional languages. Westbrook's et al. system [WaIW05] is designed for verified imperative programming with dependent types. All but the ATS/LF system support dependent types but lack support for higher-order encodings and are motivated by quite different goals. ATS/LF can explicitly manipulate contexts, and may be the closest to our calculus. Cayenne combines dependent-types and first class types, thus making more programs typeable. These languages differ radically from the $\lambda^{\ominus\Pi}$-calculus in their structural design. Our calculus is a two-tiered language. Its upper layer, a recursive function space used for computation, is entirely separate from its lower representation (LF) layer, which is used for data representation. By contrast, DML and Cayenne introduce dependent types directly into the type system of the host language. DML only uses restricted dependent types; type index objects are drawn from a constraint domain which is much less powerful than LF's λ^{Π} type system. DML and Cayenne also differ with respect to the data structures that can be easily supported by the language. Because dependent types in DML and Cayenne are introduced for typing purposes only, their data structures are the same as those provided by the respective host languages. Thus it is still very cumbersome to program with complex data structures such as those which represent proofs or typing derivations. The $\lambda^{\ominus\Pi}$-calculus is specifically designed to support programs that can easily represent and operate upon such complex data structures.

Epigram [MM04] is a novel system based on Lego, which provides a functional calculus of total functions, it stops short, however of providing higher-order encodings.

12 Conclusion

In this paper we have given a formulation of temporal logic with past-time, and have shown how it lends itself as a programming paradigm via the Curry-Howard isomorphism to programming with dependent types and higher-order abstract syntax. The applications of the resulting $\lambda^{\ominus\Pi}$-calculus are manifold. It can likely be turned into a novel programming language, which is called Delphin and currently under development in our group. The temporal operator and LF could be added to existing programming languages to provide a system for programming with higher-order encodings. And finally, with the appropriate syntactical enforceable side conditions of coverage and termination, $\lambda^{\ominus\Pi}$ is a meta-logic for the logical framework LF.

We have shown examples of programming with proofs including combinator transformations and theorem proving. In future work we will study meta-theoretic properties of the system including coverage and termination which will allow us to guarantee that our functions are complete proofs and concentrate on the implementation efforts. We are also interested in automated program synthesis from type information.

A Appendix: Weakening and Shifting Extended

We extend Weakening (Lemma 3.3) and Shifting (Lemma 3.4 and Lemma 3.5) for all the systems discussed in this paper ($\mathcal{L}^{\ominus}$, $\mathcal{L}^{\ominus\mathsf{cut}}$, $\mathcal{L}^{\ominus\langle\rangle}$, $\mathcal{L}^{\ominus\langle\rangle\mathsf{cut}}$, $\mathcal{L}^{\ominus\Pi}$, $\mathcal{L}^{\ominus\Pi\mathsf{cut}}$, $\lambda^{\ominus}$, $\lambda^{\ominus\mathsf{cut}}$, $\lambda^{\ominus\langle\rangle}$, $\lambda^{\ominus\langle\rangle\mathsf{cut}}$, $\lambda^{\ominus\Pi}$, $\lambda^{\ominus\Pi\mathsf{cut}}$)

Lemma A.1 (Weakening) *Let* $\Omega \leq \Omega'$.

- *If* $\Omega \vdash \sigma$ *then* $\Omega' \vdash \sigma$.
- *If* $\Omega \vdash e \in \sigma$ *then* $\Omega' \vdash e \in \sigma$.

Proof: *Straight-forward induction.* □

Definition A.2 (Shifting)

$$[\![\tau]\!] = \begin{cases} \top & \text{if } \tau = \top \\ \bot & \text{if } \tau = \bot \\ \sigma_1 \supset \sigma_2 & \text{if } \tau = \sigma_1 \supset \sigma_2 \\ \Pi u \in \tau\,.\sigma & \text{if } \tau = \Pi u \in \tau\,.\sigma \\ \Sigma u \in \tau\,.\sigma & \text{if } \tau = \Sigma u \in \tau\,.\sigma \\ \langle A \rangle & \text{if } \tau = \langle A \rangle \\ [\![\sigma]\!] & \text{if } \tau = \ominus\sigma \end{cases}$$

Definition A.3 (MovePast)

$$\Omega^{-\ominus} = \begin{cases} \cdot & \text{if } \Omega = \cdot \\ \Omega'^{-\ominus}, u \in \tau & \text{if } \Omega = \Omega', u \in \ominus\tau \\ \Omega'^{-\ominus} & \text{if } \Omega = \Omega', u \in \tau \text{ and } \tau \neq \ominus\tau' \end{cases}$$

We define $[\![e]\!]_\diamond$ as an operation which takes an e of type $\ominus\tau$ and returns an e' of type τ.

$$\begin{array}{rcl} [\![\text{prev } e]\!]_\diamond & = & e \\ [\![\text{let } w = u \cdot\ e \text{ in } f]\!]_\diamond & = & \text{let } w = u \cdot\ e \in [\![f]\!]_\diamond \\ [\![\text{let } (w_1, w_2) = u \text{ in } e]\!]_\diamond & = & \text{let } (w_1, w_2) = u \text{ in } [\![e]\!]_\diamond \end{array}$$

In addition, we define $[\![e]\!]^*_\diamond$ which takes an e of type τ and returns an e' of type $[\![\tau]\!]$.

$$\begin{array}{rcl}
[\![\text{prev } e]\!]^*_\diamond & = & [\![e]\!]^*_\diamond \\
[\![\text{let } w = u \cdot\ e \text{ in } f]\!]^*_\diamond & = & \text{let } w = u \cdot\ e \in [\![f]\!]^*_\diamond \\
[\![\text{let } (w_1, w_2) = u \text{ in } e]\!]^*_\diamond & = & \text{let } (w_1, w_2) = u \text{ in } [\![e]\!]^*_\diamond \\
[\![u]\!]^*_\diamond & = & u \\
[\![\text{unit}]\!]^*_\diamond & = & \text{unit} \\
[\![\text{void } u]\!]^*_\diamond & = & \text{void } u \\
[\![\lambda u \in \tau\, .e]\!]^*_\diamond & = & \lambda u \in \tau\, .e \\
[\![M]\!]^*_\diamond & = & M \\
[\![(e_1, e_2)]\!]^*_\diamond & = & (e_1, e_2)
\end{array}$$

Lemma A.4 (Shifting)

- *If* $\Omega \vdash \ominus\sigma$ *then* $\Omega \vdash \sigma$.
- *If* $\Omega \vdash \sigma$ *then* $\Omega \vdash [\![\sigma]\!]$
- *If* $\Omega \vdash e \in \ominus\sigma$ *then* $\Omega \vdash [\![e]\!]_\diamond \in \sigma$.
- *If* $\Omega \vdash e \in \sigma$ *then* $\Omega \vdash [\![e]\!]^*_\diamond \in [\![\sigma]\!]$

Proof: *Straight-forward induction.* □

Lemma A.5 (DeShifting)

- *If* $\Omega_1, \tau, \Omega_2 \vdash \sigma$ *then* $\Omega_1, \ominus\tau, \Omega_2 \vdash \sigma$
- *If* $\Omega_1, [\![\tau]\!], \Omega_2 \vdash \sigma$ *then* $\Omega_1, \tau, \Omega_2 \vdash \sigma$
- *If* $\Omega_1, u \in \tau, \Omega_2 \vdash e \in \sigma$ *then* $\Omega_1, u \in \ominus\tau, \Omega_2 \vdash e \in \sigma$
- *If* $\Omega_1, u \in [\![\tau]\!], \Omega_2 \vdash e \in \sigma$ *then* $\Omega_1, u \in \tau, \Omega_2 \vdash e \in \sigma$

Proof: *Straight-forward induction. The instrumental characteristic of this lemma is that the proof-term does* not *change when moving an assumption into the past.* □

B Appendix: Proof of cut in $\lambda^{\ominus\langle\rangle}$ (or $\mathcal{L}^{\ominus\langle\rangle}$)

$$\frac{(u \in \tau) \text{ in } \Omega}{\Omega \vdash u \in [\![\tau]\!]}\,\mathsf{ax}$$

$$\frac{}{\Omega \vdash \text{unit} \in \top}\,\top\mathsf{R} \qquad \text{no rule } \top\mathsf{L}$$

$$\text{no rule } \bot\mathsf{R} \qquad \frac{}{\Omega_1, u \in \ominus^k \bot, \Omega_2 \vdash \text{void } u \in [\![\sigma]\!]}\,\bot\mathsf{L}$$

$$\frac{\Omega, u \in \tau \vdash e \in \sigma}{\Omega \vdash \lambda u \in \tau\, . e \in \tau \supset \sigma}\,\supset\mathsf{R}$$

$$\frac{\begin{array}{c}\Omega_1, u \in \ominus^k(\tau \supset \sigma), \Omega_2 \vdash e \in \tau \\ \Omega_1, u \in \ominus^k(\tau \supset \sigma), \Omega_2, w \in [\![\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1, u \in \ominus^k(\tau \supset \sigma), \Omega_2 \vdash \text{let } w = u \cdot e \text{ in } f \in \varrho}\,\supset\mathsf{L}$$

$$\frac{\Omega^{-\ominus} \vdash e \in \tau}{\Omega \vdash \text{prev } e \ominus \tau}\,\ominus\mathsf{R} \qquad \frac{[\Omega]_{\mathrm{LF}} \vdash M : A}{\Omega \vdash M \in \langle A \rangle}\,\langle\rangle\mathsf{R}$$

Lemma B.1 (Admissibility of cut) *If* $\mathcal{D} :: \Omega_1 \vdash e \in \tau$ *and* $\mathcal{E} :: \Omega_1, u \in \tau, \Omega_2 \vdash f \in \sigma$ *then* $\Omega_1, \Omega_2 \vdash [e/u]f \in \sigma$. *This proof yields the definition of* $[e/u]f$ *which is summarized first.*

$$
\begin{array}{rcl}
[e/u]u & \equiv & [\![e]\!]^*_\diamond \\
[(\lambda u' \in \tau\,.e')/u] \\ (\text{let } w = u \cdot e \text{ in } f) & \equiv & [\frac{[\frac{\lambda u' \in \tau\,.e'}{u}]e}{u'}]e' \over w](\frac{\lambda u' \in \tau\,.e'}{u}]f) \\
[(\text{prev } e)/u](\text{prev } f) & \equiv & \text{prev } ([e/u]f) \\
[M/u]N & \equiv & [M/u]_{\text{LF}}N \\
\hline
[(\text{prev } e)/u](\text{void } u) & \equiv & [e/u](\text{void } u) \\
[(\text{prev } e)/u] \\ (\text{let } w = u \cdot e' \text{ in } f) & \equiv & [e/u](\text{let } w = u \cdot [(\text{prev } e)/u]e' \\ & & \text{in } [(\text{prev } e)/u]f) \\
[(\text{prev } e)/u]M & \equiv & [e/u]M \\
\hline
[w/u]e & \equiv & \text{rename } u \text{ to } w \text{ in } e \\
[\frac{(\text{let } w = u' \cdot e' \text{ in } f)}{u}]e & \equiv & \text{let } w = u' \cdot e' \text{ in } [f/u]e \\
[(\text{void } w')/u] \\ (\text{let } w = u \cdot\ e \text{ in } f) & \equiv & [(\text{void } w')/w] \\ & & ([(\text{void } w')/u]f) \\
[(\text{void } w)/u](\text{void } u) & \equiv & \text{void } w \\
[(\text{void } w)/u]M & \equiv & \text{void } w \\
\hline
[(\lambda u' \in \tau'\,.e')/u](\text{prev } f) & \equiv & \text{prev } f \\
[\text{unit}/u](\text{prev } f) & \equiv & \text{prev } f \\
[\text{void } w/u](\text{prev } f) & \equiv & \text{prev } f \\
[M/u](\text{prev } f) & \equiv & \text{prev } f \\
[(\lambda u' \in \tau'\,.e')/u]M & \equiv & M \\
[\text{unit}/u]M & = & M \\
\hline
u \neq w, [e/u]w & \equiv & w \\
[e/u]\text{unit} & \equiv & \text{unit} \\
u \neq w, [e/u](\text{void } w) & \equiv & \text{void } w \\
[e/u](\lambda u' \in \tau'\,.e') & \equiv & \lambda u' \in \tau'\,.([e/u]e') \\
u \neq u', [e/u] \\ (\text{let } w = u' \cdot\ e' \text{ in } f) & \equiv & \text{let } w = u' \cdot\ ([e/u]e') \\ & & \text{in } ([e/u]f)
\end{array}
$$

Proof: By induction on the cut formula τ, and simultaneously over derivations $\mathcal{D}$ and $\mathcal{E}$. This proof has been verified in Twelf and is also in the appendix.

First we handle the essential conversions.

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{(u \in \tau) \text{ in } (\Omega_1, u \in \tau, \Omega_2)}{\Omega_1, u \in \tau, \Omega_2 \vdash u \in [\![\tau]\!]}\,\mathsf{ax}$$

$\Omega_1, \leq \Omega_1, \Omega_2$ by Definition
$\Omega_1, \Omega_2 \vdash e \in \tau$ by Weakening (Lemma A.1)
$\Omega_1, \Omega_2 \vdash [\![e]\!]^*_\diamond \in [\![\tau]\!]$ by Shifting (Lemma A.4)

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 :: \Omega_1, u' \in \tau \vdash e' \in \sigma}{\Omega_1 \vdash \lambda u' \in \tau\, .e' \in \tau \supset \sigma} \supset \mathsf{R}$,

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1 :: \Omega_1, u \in (\tau \supset \sigma), \Omega_2 \vdash e \in \tau \\ \mathcal{E}_2 :: \Omega_1, u \in (\tau \supset \sigma), \Omega_2, w \in [\![\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1, u \in (\tau \supset \sigma), \Omega_2 \vdash \text{let } w = u \cdot\ e \text{ in } f \in \varrho} \supset \mathsf{L}$$

$\mathcal{E}'_1 :: \Omega_1, \Omega_2 \vdash [\frac{\lambda u' \in \tau\, .e'}{u}]e \in \tau$ by IH on $\mathcal{D}$ and $\mathcal{E}_1$
$\mathcal{E}'_2 :: \Omega_1, \Omega_2, w \in [\![\sigma]\!] \vdash [\frac{\lambda u' \in \tau\, .e'}{u}]f \in \varrho$ by IH on $\mathcal{D}$ and $\mathcal{E}_2$
$\Omega_1, u' \in \tau \leq \Omega_1, \Omega_2, u' \in \tau$ by Definition
$\mathcal{D}'_1 :: \Omega_1, \Omega_2, u' \in \tau \vdash e' \in \sigma$
by Weakening (Lemma 3.3) on $\mathcal{D}_1$
$\mathcal{D}''_1 :: \Omega_1, \Omega_2 \vdash [\frac{[\frac{\lambda u' \in \tau\, .e'}{u}]e}{u'}]e' \in \sigma$ by IH on $\mathcal{E}'_1$ and $\mathcal{D}'_1$
$\mathcal{E}''_2 :: \Omega_1, \Omega_2, w \in \sigma \vdash [\frac{\lambda u' \in \tau\, .e'}{u}]f \in \varrho$
by DeShifting (Lemma 3.5) on $\mathcal{E}'_2$
$\mathcal{F} :: \Omega_1, \Omega_2 \vdash [\frac{[\frac{[\frac{\lambda u' \in \tau\, .e'}{u}]e}{u'}]e'}{w}]([\frac{\lambda u' \in \tau\, .e'}{u}]f) \in \varrho$
by IH on $\mathcal{D}''_1$ and $\mathcal{E}''_2$

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 :: {\Omega_1}^{-\ominus} \vdash e \in \tau}{\Omega_1 \vdash \text{prev } e \in \ominus\tau} \ominus\mathsf{R}$

$$\mathcal{E} = \frac{\mathcal{E}_1 :: (\Omega_1, u \in \ominus\tau, \Omega_2)^{-\ominus} \vdash f \in \sigma}{\Omega_1, u \in \ominus\tau, \Omega_2 \vdash \text{prev } f \in \ominus\sigma}\ \ominus\mathsf{R}$$

$\mathcal{E}_1' :: \Omega_1{}^{-\ominus}, u \in \tau, \Omega_2{}^{-\ominus} \vdash f \in \sigma$ by Property of $(\_)^{-\ominus}$ on $\mathcal{E}_1$
$\mathcal{F} :: \Omega_1{}^{-\ominus}, \Omega_2{}^{-\ominus} \vdash [e/u]f \in \sigma$ by IH on $\mathcal{D}_1$ and $\mathcal{E}_1'$
$\mathcal{F}' :: (\Omega_1, \Omega_2)^{-\ominus} \vdash [e/u]f \in \sigma$ by Property of $(\_)^{-\ominus}$ on $\mathcal{F}$
$\mathcal{F}'' :: \Omega_1, \Omega_2 \vdash \text{prev } ([e/u]f) \in \ominus\sigma$ by $\ominus\mathsf{R}$

Case: $\mathcal{D} = \dfrac{[\Omega_1]_{\text{LF}} \vdash M : A}{\Omega_1 \vdash M \in \langle A\rangle}\ \langle\rangle\mathsf{R}$

$$\mathcal{E} = \frac{\mathcal{E}_1 :: [(\Omega_1, u \in \langle A\rangle, \Omega_2)]_{\text{LF}} \vdash N : B}{\Omega_1, u \in \langle A\rangle, \Omega_2 \vdash N \in \langle B\rangle}\ \langle\rangle\mathsf{R}$$

$\mathcal{E}_1' :: [\Omega_1]_{\text{LF}}, u : A, [\Omega_2]_{\text{LF}} \vdash N : B$ by Property of $[\_]_{\text{LF}}$ on $\mathcal{E}_1$
$\mathcal{F} :: [\Omega_1]_{\text{LF}}, [\Omega_2]_{\text{LF}} \vdash [M/u]_{\text{LF}}N \in B$ by LF Substitution
$\mathcal{F}' :: [(\Omega_1, \Omega_2)]_{\text{LF}} \vdash [M/u]_{\text{LF}}N \in B$ by Property of $[\_]_{\text{LF}}$ on $\mathcal{F}$
$\Omega_1, \Omega_2 \vdash [M/u]_{\text{LF}}N \in \langle B\rangle$ by $\langle\rangle\mathsf{R}$

Now we handle when we have $\ominus\mathsf{R}$ on the left and a non-commutative, non-$\ominus\mathsf{R}$ on the right.

Case: $\mathcal{D} = \dfrac{\Omega_1{}^{-\ominus} \vdash e \in \ominus^k\bot}{\Omega_1 \vdash \text{prev } e \in \ominus^{k+1}\bot}\ \ominus\mathsf{R}$

$$\mathcal{E} = \frac{}{\Omega_1, u \in \ominus^{k+1}\bot, \Omega_2 \vdash \text{void } u \in [\![\sigma]\!]}\ \bot\mathsf{L}$$

$\mathcal{D}' :: \Omega_1 \vdash [\![\text{prev } e]\!]_\diamond \in \ominus^k\bot$ by Shifting (Lemma A.4) on $\mathcal{D}$
$\mathcal{D}'' :: \Omega_1 \vdash e \in \ominus^k\bot$ by Definition of $[\![\_]\!]_\diamond$
$\mathcal{E}' :: \Omega_1, u \in \ominus^k\bot, \Omega_2 \vdash \text{void } u \in [\![\sigma]\!]$ by $\bot\mathsf{L}$
$\mathcal{F} :: \Omega_1, \Omega_2 \vdash [e/u](\text{void } u) \in [\![\sigma]\!]$ by IH on $\mathcal{D}''$ and $\mathcal{E}'$

Case: $\mathcal{D} = \dfrac{\Omega_1{}^{-\ominus} \vdash e \in \ominus^k(\tau \supset \sigma)}{\Omega_1 \vdash \text{prev } e \in \ominus^{k+1}(\tau \supset \sigma)}\ \ominus\mathsf{R}$

$$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1 :: \Omega_1, u \in \ominus^{k+1}(\tau \supset \sigma), \Omega_2 \vdash e' \in \tau \\ \mathcal{E}_2 :: \Omega_1, u \in \ominus^{k+1}(\tau \supset \sigma), \Omega_2, w \in [\![\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1, u \in \ominus^{k+1}(\tau \supset \sigma), \Omega_2 \vdash \text{let } w = u \cdot\ e' \text{ in } f \in \varrho} \supset \mathsf{L}$$

$\mathcal{D}' :: \Omega_1 \vdash [\![\text{prev } e]\!]_\diamond \in \ominus^k(\tau \supset \sigma)$
by Shifting (Lemma A.4) on $\mathcal{D}$

$\mathcal{D}'' :: \Omega_1 \vdash e \in \ominus^k(\tau \supset \sigma)$ by Definition of $[\![\_]\!]_\diamond$

$\mathcal{E}_1' :: \Omega_1, \Omega_2 \vdash [(\text{prev } e)/u]e' \in \tau$ by IH on $\mathcal{D}$ and $\mathcal{E}_1$

$\mathcal{E}_2' :: \Omega_1, \Omega_2, w \in [\![\sigma]\!] \vdash [(\text{prev } e)/u]f \in \varrho$ by IH on $\mathcal{D}$ and $\mathcal{E}_2$

$\mathcal{F}_1 :: \Omega_1, \Omega_2, u \in \ominus^k(\tau \supset \sigma)$
$\quad \vdash \text{let } w = u \cdot\ [(\text{prev } e)/u]e' \text{ in } [(\text{prev } e)/u]f \in \varrho$
by Weakening (Lemma A.1) and $\supset \mathsf{L}$

Ω_1, Ω_2
$\quad \vdash [e/u](\text{let } w = u \cdot\ [(\text{prev } e)/u]e' \text{ in } [(\text{prev } e)/u]f) \in \varrho$
by IH on $\mathcal{D}''$ and $\mathcal{F}_1$

Case: $\mathcal{D} = \dfrac{{\Omega_1}^{-\ominus} \vdash e \in \ominus^k\langle A\rangle}{\Omega_1 \vdash \text{prev } e \in \ominus^{k+1}\langle A\rangle} \ominus\mathsf{R}$

$$\mathcal{E} = \dfrac{\mathcal{E}_1 :: [(\Omega_1, u \in \ominus^{k+1}\langle A\rangle, \Omega_2)]_{\text{LF}} \vdash M : B}{\Omega_1, u \in \ominus^{k+1}\langle A\rangle, \Omega_2 \vdash M \in \langle B\rangle} \langle\rangle\mathsf{R}$$

$\mathcal{D}' :: \Omega_1 \vdash [\![\text{prev } e]\!]_\diamond \in \ominus^k\langle A\rangle$ by Shifting (Lemma A.4) on $\mathcal{D}$

$\mathcal{D}'' :: \Omega_1 \vdash e \in \ominus^k\langle A\rangle$ by Definition of $[\![\_]\!]_\diamond$

$\mathcal{E}_1' :: [(\Omega_1, u \in \ominus^k\langle A\rangle, \Omega_2)]_{\text{LF}} \vdash M : B$
by Property of $[\_]_{\text{LF}}$ on $\mathcal{E}_1$

$\mathcal{F}_1 :: \Omega_1, u \in \ominus^k\langle A\rangle, \Omega_2 \vdash M \in \langle B\rangle$
by $\langle\rangle\mathsf{R}$

$\Omega_1, \Omega_2 \vdash [e/u]M \in \langle B\rangle$ by IH on $\mathcal{D}''$ and $\mathcal{F}_1$

Now we handle commutative conversion on the left.

Case: $\mathcal{D} = \dfrac{(w \in \tau) \in \Omega_1}{\Omega_1 \vdash w \in [\![\tau]\!]} \mathsf{ax}$

$\mathcal{E} = \Omega_1, u \in [\![\tau]\!], \Omega_2 \vdash e \in \sigma$

$\Omega_1, \Omega_2 \vdash \text{rename } u \text{ to } w \text{ in } e \in \sigma$ by variable renaming

Case:

$$\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_0 :: (u' \in (\tau \supset \sigma)) \in \Omega_1 \\ \mathcal{D}_1 :: \Omega_1 \vdash e' \in \tau \\ \mathcal{D}_2 :: \Omega_1, w \in [\![\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1 \vdash \text{let } w = u' \cdot\ e' \text{ in } f \in \varrho} \supset \mathsf{L}$$

$\mathcal{E} = \Omega_1, u \in \varrho, \Omega_2 \vdash e \in \varrho'$

$\mathcal{F}_0 :: (u' \in (\tau \supset \sigma)) \in \Omega_1, \Omega_2$ by $\mathcal{D}_0$
$\mathcal{F}_1 :: \Omega_1, \Omega_2 \vdash e' \in \tau$ by Weakening (Lemma A.1) on $\mathcal{D}_1$
$\mathcal{E}' :: \Omega_1, w \in [\![\sigma]\!], u \in \varrho, \Omega_2 \vdash e \in \varrho'$ by Weakening (Lemma A.1) on $\mathcal{E}$
$\mathcal{F}_2' :: \Omega_1, w \in [\![\sigma]\!], \Omega_2 \vdash [f/u]e \in \varrho'$ by IH on $\mathcal{D}_2$ and $\mathcal{E}'$
$\mathcal{F}_2 :: \Omega_1, \Omega_2, w \in [\![\sigma]\!] \vdash [f/u]e \in \varrho'$ by Context Reordering
$\Omega_1, \Omega_2 \vdash \text{let } w = u' \cdot\ e' \text{ in } [f/u]e \in \varrho'$ by $\supset \mathsf{L}$

Case:

$$\mathcal{D} = \dfrac{(w' \in \ominus^k \bot) \in \Omega_1}{\Omega_1 \vdash \text{void } w' \in (\tau \supset \sigma)} \bot\mathsf{L}$$

$$\mathcal{E} = \dfrac{\begin{array}{l}\mathcal{E}_1 :: \Omega_1, u \in (\tau \supset \sigma), \Omega_2 \vdash e \in \tau \\ \mathcal{E}_2 :: \Omega_1, u \in (\tau \supset \sigma), \Omega_2, w \in [\![\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1, u \in (\tau \supset \sigma), \Omega_2 \vdash \text{let } w = u \cdot\ e \text{ in } f \in \varrho} \supset \mathsf{L}$$

$\mathcal{F}_1 :: \Omega_1, \Omega_2, w \in [\![\sigma]\!] \vdash [(\text{void } w')/u]f \in \varrho$ by IH on $\mathcal{D}$ and $\mathcal{E}_2$
$\mathcal{F}_2 :: \Omega_1, \Omega_2 \vdash \text{void } w' \in [\![\sigma]\!]$ by $\bot\mathsf{L}$
$\Omega_1, \Omega_2 \vdash [(\text{void } w')/w]([(\text{void } w')/u]f) \in \varrho$ by IH on $\mathcal{F}_2$ and $\mathcal{F}_1$

Case:

$$\mathcal{D} = \dfrac{(w \in \ominus^k \bot) \in \Omega_1}{\Omega_1 \vdash \text{void } w \in \bot} \bot\mathsf{L}$$

$\mathcal{E} = \Omega_1, u \in \bot, \Omega_2 \vdash (\text{void } u) \in [\![\sigma]\!]$

$\Omega_1, \Omega_2 \vdash \text{void } w \in [\![\sigma]\!]$ by $\bot\mathsf{L}$

Case:

$$\mathcal{D} = \frac{(w \in \ominus^k \bot) \in \Omega_1}{\Omega_1 \vdash \text{void } w \in [\![\tau]\!]}\, \bot\mathsf{L}$$

$$\mathcal{E} = \Omega_1, u \in [\![\tau]\!], \Omega_2 \vdash M \in \langle A \rangle$$

$\Omega_1, \Omega_2 \vdash \text{void } w \in \langle A \rangle$ by $\bot\mathsf{L}$

Now we handle strengthening conditions when the left is in the present and the right is in the past.

Case: In this case, we consider when e is
either $\lambda u' \in \tau' .e'$, unit, (void w), or M

$$\mathcal{D} = \Omega_1 \vdash e \in [\![\tau]\!]$$

$$\mathcal{E} = \frac{\mathcal{E}_1 :: (\Omega_1, u \in [\![\tau]\!], \Omega_2)^{-\ominus} \vdash f \in \sigma}{\Omega_1, u \in [\![\tau]\!], \Omega_2 \vdash \text{prev } f \in \ominus\sigma}\, \ominus\mathsf{R}$$

$\mathcal{F}_1 :: (\Omega_1, \Omega_2)^{-\ominus} \vdash f \in \sigma$ by Property of $(\_)^{-\ominus}$ on $\mathcal{E}_1$
$\Omega_1, \Omega_2 \vdash \text{prev } f \in \ominus\sigma$ by $\ominus\mathsf{R}$

Case: In this case, e is either $\lambda u' \in \tau' .e'$ or unit
so τ is either $(\tau' \supset \sigma')$ or $\top$

$$\mathcal{D} = \Omega_1 \vdash e \in \tau$$

$$\mathcal{E} = \frac{\mathcal{E}_1 :: [(\Omega_1, u \in \tau, \Omega_2)]_{\mathrm{LF}} \vdash M : A}{\Omega_1, u \in \tau, \Omega_2 \vdash M \in \langle A \rangle}\, \langle\rangle\mathsf{R}$$

$\mathcal{F}_1 :: [\Omega_1, \Omega_2]_{\mathrm{LF}} \vdash M : A$
by Property of $[\_]_{\mathrm{LF}}$ on $\mathcal{E}_1$ and restriction on τ
$\Omega_1, \Omega_2 \vdash M \in \langle A \rangle$ by $\langle\rangle\mathsf{R}$

Now we handle commutative conversion cases on the right.

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{(w \in \sigma) \text{ in } (\Omega_1, u \in \tau, \Omega_2)}{\Omega_1, u \in \tau, \Omega_2 \vdash w \in [\![\sigma]\!]}\, \mathsf{ax}, \text{ where } u \neq w$$

$(w \in \sigma)$ in (Ω_1, Ω_2) since $u \neq w$
$\Omega_1, \Omega_2 \vdash w \in [\![\sigma]\!]$ by ax

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{}{\Omega_1, u \in \tau, \Omega_2 \vdash \text{unit} \in \top}\, \top\mathsf{R}$$

$\Omega_1, \Omega_2 \vdash \text{unit} \in \top$ by $\top\mathsf{R}$

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{(w \in \ominus^k \bot) \in (\Omega_1, u \in \tau, \Omega_2)}{\Omega_1, u \in \tau, \Omega_2 \vdash \text{void}\ w \in [\![\sigma]\!]}\, \bot\mathsf{L}, \text{ where } u \neq w$$

$(w \in \ominus^k \bot) \in (\Omega_1, \Omega_2)$ since $u \neq w$
$\Omega_1, \Omega_2 \vdash \text{void}\ w \in [\![\sigma]\!]$ by $\bot\mathsf{L}$

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{\mathcal{E}_1 :: \Omega_1, u \in \tau, \Omega_2, u' \in \tau' \vdash e' \in \sigma}{\Omega_1, u \in \tau, \Omega_2 \vdash \lambda u' \in \tau' . e' \in \tau' \supset \sigma}\, \supset\mathsf{R}$$

$\Omega_1, \Omega_2, u' \in \tau' \vdash [e/u]e' \in \sigma$ by IH on $\mathcal{D}$ and $\mathcal{E}_1$
$\Omega_1, \Omega_2 \vdash \lambda u' \in \tau' . ([e/u]e') \in \tau' \supset \sigma$ by $\supset\mathsf{R}$

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{\begin{array}{l} u \neq u' \\ \mathcal{E}_0 :: (u' \in (\tau' \supset \sigma')) \in (\Omega_1, u \in \tau, \Omega_2) \\ \mathcal{E}_1 :: \Omega_1, u \in \tau, \Omega_2 \vdash e' \in \tau' \\ \mathcal{E}_2 :: \Omega_1, u \in \tau, \Omega_2, w \in [\![\sigma']\!] \vdash f \in \varrho \end{array}}{\Omega_1, u \in \tau, \Omega_2 \vdash \text{let}\ w = u' \cdot\ e' \ \text{in}\ f \in \varrho}\, \supset\mathsf{L}$$

$\mathcal{F}_0 :: (u' \in (\tau' \supset \sigma')) \in (\Omega_1, \Omega_2)$ since $u \neq u'$
$\mathcal{F}_1 :: \Omega_1, \Omega_2 \vdash [e/u]e' \in \tau'$ by IH on $\mathcal{D}$ and $\mathcal{E}_1$

$\mathcal{F}_2 :: \Omega_1, \Omega_2, w \in [\![\sigma']\!] \vdash [e/u]f \in \varrho$ by IH on $\mathcal{D}$ and $\mathcal{E}_2$

$\Omega_1, \Omega_2 \vdash \text{let } w = u' \cdot\ ([e/u]e') \text{ in } ([e/u]f) \in \varrho$ by $\supset \mathsf{L}$

□

C Appendix: Proof of cut in $\lambda^{\ominus\Pi}$ (or $\mathcal{L}^{\ominus\Pi}$)

Note that an important *implicit* condition on all the rules is that all elements of Ω are well-formed, as well as the resulting type.

$$\frac{}{\Omega \vdash \top \text{ wff}}\,\top\mathsf{wff} \qquad \frac{}{\Omega \vdash \bot \text{ wff}}\,\bot\mathsf{wff}$$

$$\frac{[\Omega]_{\mathrm{LF}} \vdash A : \text{type}}{\Omega \vdash \langle A\rangle \text{ wff}}\,\langle\rangle\mathsf{wffR} \qquad \frac{[\Omega]_{\mathrm{LF}} \vdash A : \text{type}}{\Omega \vdash \langle \#A\rangle \text{ wff}}\,\langle\rangle\#\mathsf{wffR}$$

$$\frac{\Omega \vdash \tau \text{ wff} \quad \Omega, u \in \tau \vdash \sigma \text{ wff}}{\Omega \vdash \Pi u \in \tau\,.\sigma \text{ wff}}\,\Pi\mathsf{wffR}$$

$$\frac{\Omega \vdash \tau \text{ wff} \quad \Omega, u \in \tau \vdash \sigma \text{ wff}}{\Omega \vdash \Sigma u \in \tau\,.\sigma \text{ wff}}\,\Sigma\mathsf{wffR}$$

$$\frac{\Omega^{-\ominus} \vdash \tau \text{ wff}}{\Omega \vdash \ominus\tau \text{ wff}}\,\ominus\mathsf{wffR}$$

$$\frac{(u \in \tau) \text{ in } \Omega}{\Omega \vdash u \in [\![\tau]\!]}\,\mathsf{ax}$$

$$\frac{}{\Omega \vdash \text{unit} \in \top}\,\top\mathsf{R} \qquad \text{no rule } \top\mathsf{L}$$

$$\text{no rule } \bot\mathsf{R} \qquad \frac{}{\Omega_1, u \in \ominus^k\bot, \Omega_2 \vdash \text{void } u \in [\![\sigma]\!]}\,\bot\mathsf{L}$$

$$\frac{\Omega, u \in \tau \vdash e \in \sigma}{\Omega \vdash \lambda u \in \tau\,.e \in \Pi u \in \tau\,.\sigma}\,\Pi\mathsf{R}$$

$$\frac{\begin{array}{c}\Omega_1, u \in \ominus^k(\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash e \in \tau \\ \Omega_1, u \in \ominus^k(\Pi u' \in \tau\,.\sigma), \Omega_2, w \in [\![[e/u']\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash \text{let } w = u \cdot e \text{ in } f \in \varrho}\,\Pi\mathsf{L}$$

$$\frac{\Omega^{-\ominus} \vdash e \in \tau}{\Omega \vdash \text{prev } e \ominus \tau}\,\ominus\mathsf{R} \qquad \frac{[\Omega]_{\mathrm{LF}} \vdash M : A}{\Omega \vdash M \in \langle A\rangle}\,\langle\rangle\mathsf{R}$$

$$\frac{\Omega \vdash e_1 \in \tau \quad \Omega \vdash e_2 \in [e_1/u]\sigma}{\Omega \vdash (e_1, e_2) \in \Sigma u \in \tau\,.\sigma}\,\Sigma\mathsf{R}$$

$$\frac{\Omega_1, u \in \ominus^k(\Sigma u' \in \tau\,.\sigma), \Omega_2, w_1 \in [\![\tau]\!], w_2 \in [w_1/u'][\![\sigma]\!] \vdash e \in \varrho}{\Omega_1, u \in \ominus^k(\Sigma u' \in \tau\,.\sigma), \Omega_2 \vdash \text{let } (w_1, w_2) = u \text{ in } e \in \varrho}\,\Sigma\mathsf{L}$$

Now that we have substitution on the type-level, we define it below. However, notice that this definition is *not* defined everywhere. We do not allow the type to depend on meta-level computation. In addition, we will write $\text{prev}^k\ e$ as shorthand for $k \geq 0$ leading prevs to e.

$$\begin{array}{rcl}
[e/u]\top & = & \top \\
[e/u]\bot & = & \bot \\
[e/u](\Pi u' \in \tau\,.\sigma) & = & \Pi u' \in [e/u]\tau\,.[e/u]\sigma \\
[e/u](\Sigma u' \in \tau\,.\sigma) & = & \Sigma u' \in [e/u]\tau\,.[e/u]\sigma \\
[(\text{prev } e)/u](\ominus\tau) & = & \ominus([e/u]\tau) \\
[(\text{prev}^k\ M)/u]\langle A\rangle & = & \langle [M/u]_{\text{LF}} A\rangle \\
[e/u]\tau, u \text{ not free in } \tau & = & \tau \\
\text{None of the above match, } [e/u]\tau & = & \text{undefined}
\end{array}$$

In Addition, we define $[e/u]\Omega = \Omega'$, where Ω' is formed by transforming every $(w \in \sigma)$ in Ω into $(w \in ([e/u]\sigma))$ in Ω'

Lemma C.1 (Admissibility of cutT) *If $\mathcal{D} :: \Omega_1 \vdash e \in \tau$ and $\mathcal{E} :: \Omega_1, u \in \tau, \Omega_2 \vdash \varrho$ wff and $[e/u]\Omega_2$ is not undefined and $[e/u]\varrho$ is not undefined, then $\Omega_1, [e/u]\Omega_2 \vdash [e/u]\varrho$ wff.*

Proof: *This proof simply goes by induction over derivation $\mathcal{E}$ utilizing the definition of $[e/u]\varrho$.* □

Lemma C.2 (Substitution Shift Property)

- *If $[e/u][\![\tau]\!]$ is not undefined, then there exists a τ' such that $[e/u][\![\tau]\!] = [\![\tau']\!]$*
- *If $[e/u]\sigma$ is not undefined, then $[\![[e/u]\sigma]\!] = [[\![e]\!]^*_\diamond/u][\![\sigma]\!]$*

Proof: *Trivial.* □

Lemma C.3 (Substitution Equal Property)
If $[e/u]\sigma$ is not undefined and $[e/u][\![\sigma]\!]$ is also not undefined, then $[e/u]\sigma = [e/u][\![\sigma]\!]$

Proof: *Trivial.* □

Lemma C.4 (Substitution DeShift Property)

- *If* $[e/u]\sigma$ *is not undefined, then* $[e/u]\sigma = [(\text{prev } e)/u]\sigma$
- *If* $[[\![e]\!]^*_\diamond/u]\sigma$ *is not undefined, then* $[[\![e]\!]^*_\diamond/u]\sigma = [e/u]\sigma$
- *If* $[\![[e/u]\sigma]\!]$ *is not undefined, then* $[\![[e/u]\sigma]\!] = [e/u][\![\sigma]\!]$

Proof: *Trivial.* □

Lemma C.5 (Redundant Substitution)

- *If* u *is not free in* τ*, then* $[e/u]\tau = \tau$
- *If* $\Omega \vdash \tau$ wff *and* $(u \in \tau) \in \Omega$ *and* $\tau \neq \ominus^k\langle A\rangle$*, then* $[e/u]\tau = \tau$

Proof: *Trivial. The second case appeals to the first since well-formed types cannot have variables of non-*$\ominus^k\langle A\rangle$ *type occur free.* □

Lemma C.6 (Substitution Distributive Property)
$[e/u]([e'/u']\tau) = [\frac{[e/u]e'}{u'}]([e/u]\tau)$

Proof: *(Note that the definition of substitution on expressions is given in Lemma C.12).*

Therefore, If u' is not free in τ, then it follows from Redundant Sub Lemma (Lemma C.5). Therefore, we now assume that u' occurs free in τ. which also means that e' must be of the form $\text{prev}^k\ M$ *by our definition of substitution.*
We first assert that it also holds that if u is not free in f, then $[e/u]f = f$
We now proceed by induction on τ

Case: $[e/u]([e'/u'](\Pi w \in \tau\,.\sigma)) = [\frac{[e/u]e'}{u'}]([e/u](\Pi w \in \tau\,.\sigma))$

$$
\begin{aligned}
&[e/u]([e'/u'](\Pi w \in \tau\,.\sigma)) \\
&\quad = \Pi w \in ([e/u]([e'/u']\tau))\,.(([e/u]([e'/u']\sigma))) && \textit{by Def.} \\
&\quad = \Pi w \in ([\tfrac{[e/u]e'}{u'}]([e/u]\tau))\,.([\tfrac{[e/u]e'}{u'}]([e/u]\sigma)) && \textit{by IH} \\
&\quad = [\tfrac{[e/u]e'}{u'}]([e/u](\Pi w \in \tau\,.\sigma)) && \textit{by Def.}
\end{aligned}
$$

Case: $[e/u]([e'/u'](\Sigma w \in \tau\,.\sigma)) = [\frac{[e/u]e'}{u'}]([e/u](\Sigma w \in \tau\,.\sigma))$

Same As Previous.

Case: $[e/u]([e'/u']\top) = [\frac{[e/u]e'}{u'}]([e/u]\top)$

Both sides equal to $\top$ by definition

Case: $[e/u]([e'/u']\bot) = [\frac{[e/u]e'}{u'}]([e/u]\bot)$

Both sides equal to $\bot$ by definition

Case: $[e/u]([e'/u']\langle A\rangle) = [\frac{[e/u]e'}{u'}]([e/u]\langle A\rangle)$

$e' = M$ *since assuming* u' *free in* $\langle A \rangle$

$[e/u]([M/u']\langle A \rangle)$

$= [e/u]\langle [M/u']_{LF} A \rangle$ *by Def.*

Subcase: $[e/u]\langle [M/u']_{LF} A \rangle = \langle [M/u']_{LF} A \rangle$

So u *is neither free in* M *nor* A

$[\frac{[e/u]M}{u'}]([e/u]\langle A \rangle)$

$= [M/u']([e/u]\langle A \rangle)$ *since* u *free in* M

$= [M/u']\langle A \rangle$ *since* u *free in* A

$= \langle [M/u']_{LF} A \rangle$ *by Definition*

Subcase: $e = N$,

$[e/u]\langle [M/u']_{LF} A \rangle = \langle [N/u]_{LF}([M/u']_{LF} A) \rangle$

$[\frac{[N/u]M}{u'}]([N/u]\langle A \rangle)$

$= [([N/u]_{LF} M)/u']([N/u]\langle A \rangle)$ *by Def.*

$= [([N/u]_{LF} M)/u']\langle [N/u]_{LF} A \rangle$ *by Def.*

$= \langle [([N/u]_{LF} M)/u']_{LF}([N/u]_{LF} A) \rangle$ *by Def.*

$= \langle [N/u]_{LF}([M/u']_{LF} A) \rangle$ *by Distributivity of LF*

Case: $[e/u]([e'/u'](\ominus\tau)) = [\frac{[e/u]e'}{u'}]([e/u](\ominus\tau))$

$e' = \text{prev } f$ *since assuming* u' *free in* $\ominus\tau$

$[e/u]([(\text{prev } f)/u'](\ominus\tau))$

$= [e/u](\ominus([f/u']\tau))$ *by Def.*

Subcase: $[e/u](\ominus([f/u']\tau)) = \ominus([f/u']\tau)$

So u *is neither free in* f *nor* τ

$[\frac{[e/u](\text{prev } f)}{u'}]([e/u](\ominus\tau))$

$= [(\text{prev } f)/u']([e/u](\ominus\tau))$ *since* u *free in* $(\text{prev } f)$

$= [(\text{prev } f)/u'](\ominus\tau)$ *since* u *free in* $(\ominus\tau)$

$= \ominus([f/u']\tau)$ *by Definition*

Subcase: $e = (\text{prev } f')$,

$[e/u](\ominus([f/u']\tau)) = \ominus([f'/u]([f/u']\tau))$

$[\frac{[(\text{prev } f')/u](\text{prev } f)}{u'}]([(\text{prev } f')/u](\ominus\tau))$

$= [\frac{\text{prev } ([f'/u]f)}{u'}]([(\text{prev } f')/u](\ominus\tau))$ *by Def.*

$= [\frac{\text{prev } ([f'/u]f)}{u'}](\ominus([f'/u]\tau))$ *by Def.*

$$
\begin{array}{rl}
= \ominus([\frac{[f'/u]f}{u'}]([f'/u]\tau)) & \textit{by Def.} \\
= \ominus([f'/u]([f/u']\tau)) & \textit{by IH on } \tau
\end{array}
$$

□

Lemma C.7 (Not Undefined Property)
If $[e/u]\sigma$ *is not undefined and* $[f/w]\sigma$ *is also not undefined, then* $[e/u]([f/w]\sigma)$ *is also not undefined.*

Proof: *Trivial.* □

Lemma C.8 (Strengthening)
Let Ω' *be a transformation on* Ω *where we remove every element* $(u \in \sigma)$ *from* Ω *when* $\sigma \neq \ominus\sigma'$

If $\Omega \vdash e \in \ominus\tau$ *then* $\Omega' \vdash e \in \tau$

Proof: *By Induction.* □

$$
\begin{array}{rcl}
\perp\!\!\!\top \text{prev } e \top\!\!\!\perp_{w'} & = & \text{prev } e \\
\perp\!\!\!\top \text{let } w = u \cdot\ e \text{ in } f \top\!\!\!\perp_{w'} & = & \perp\!\!\!\top f \top\!\!\!\perp_{w'} \\
\perp\!\!\!\top \text{let } (w_1, w_2) = u \text{ in } e \top\!\!\!\perp_{w'} & = & \perp\!\!\!\top e \top\!\!\!\perp_{w'} \\
\perp\!\!\!\top e \top\!\!\!\perp_{w'} & = & \text{void } w', \text{ when } e = u, \text{unit}, (\text{void } u), \\
 & & \quad (\lambda u \in \tau\, .e'), M, \text{ or } (e_1, e_2)
\end{array}
$$

Lemma C.9 (Commute $\perp$)
Let Ω_2' *be a transformation on* Ω_2 *where we* may *remove elements of* Ω_2 *of the form* $(u \in \Pi u \in \tau\, .\sigma)$ *or* $(u \in \Sigma u \in \tau\, .\sigma)$

If $\Omega_1, w \in \ominus^k \perp, \Omega_2 \vdash e \in \varrho$*, then* $\Omega_1, w \in \perp, \Omega_2' \vdash \perp\!\!\!\top e \top\!\!\!\perp_w \in \varrho$

Proof: *By Induction and Strengthening (Lemma C.8)*
□

Lemma C.10 (Redundant Let)

- *If* $\Omega \vdash \text{let } w = u \cdot\ e \text{ in } f \in \varrho$ *and neither* u *nor* w *occur free in* f*, then* $\Omega \vdash f \in \varrho$

- *If* $\Omega \vdash \text{let } (w_1, w_2) = u \text{ in } f \in \varrho$ *and neither* u *nor* w *occur free in* f*, then* $\Omega \vdash f \in \varrho$

Proof: *By Induction.* □

Lemma C.11 (Redundant Term)
If u *is neither free in* e *nor* Ω_2*, and* $\Omega_1, u \in \tau, \Omega_2 \vdash e \in \sigma$*, then* $\Omega_1, \Omega_2 \vdash e \in \sigma$

Proof: *By Induction.* □

Lemma C.12 (Admissibility of cut) *If $\mathcal{D} :: \Omega_1 \vdash e \in \tau$ and $\mathcal{E} :: \Omega_1, u \in \tau, \Omega_2 \vdash f \in \sigma$ and $[e/u]\sigma$ is not undefined and $[e/u]\Omega_2$ is not undefined, then $\Omega_1, [e/u]\Omega_2 \vdash [e/u]f \in [e/u]\sigma$. The definition of $[e/u]f$ is the result of this lemma, which is summarized first.*

$$
\begin{array}{rcl}
[e/u]u & \equiv & [\![e]\!]^*_\diamond \\
[(\lambda u' \in \tau\,.e')/u] \\ (\text{let } w = u \cdot e \text{ in } f) & \equiv & [\dfrac{[\frac{\lambda u' \in \tau\,.e'}{u}]e}{u'}]e' \Big/ [\dfrac{\lambda u' \in \tau\,.e'}{u}]f) \\
[(\text{prev } e)/u](\text{prev } f) & \equiv & \text{prev } ([e/u]f) \\
[M/u]N & \equiv & [M/u]_{\text{LF}}N \\
[(e_1, e_2)/u] \\ (\text{let } (w_1, w_2) = u \text{ in } f) & \equiv & [[\![e_2]\!]^*_\diamond/w_2]([[\![e_1]\!]^*_\diamond/w_1] \\ & & ([(e_1, e_2)/u]f)) \\
\hline
[(\text{prev } e)/u](\text{void } u) & \equiv & [e/u](\text{void } u) \\
[(\text{prev } e)/u] \\ (\text{let } w = u \cdot e' \text{ in } f) & \equiv & [e/u] \\ & & (\text{let } w = u \cdot [(\text{prev } e)/u]e' \\ & & \text{in } [(\text{prev } e)/u]f) \\
[(\text{prev } e)/u]M & \equiv & [e/u]M \\
[(\text{prev } e)/u] \\ (\text{let } (w_1, w_2) = u \text{ in } f) & \equiv & [e/u] \\ & & (\text{let } (w_1, w_2) = u \\ & & \text{in } [(\text{prev } e)/u]f) \\
\hline
[w/u]e & \equiv & \text{rename } u \text{ to } w \text{ in } e \\
[\dfrac{(\text{let } w = u' \cdot e' \text{ in } f)}{u}]e & \equiv & \begin{cases} [f/u]e & \\ \quad \text{if } u \text{ not free in } e & \\ \text{let } w = u' \cdot e' \text{ in } [f/u]e & \\ \quad \text{otherwise} & \end{cases} \\
[\dfrac{(\text{let } (w_1, w_2) = u' \text{ in } f)}{u}]e & \equiv & \begin{cases} [f/u]e & \\ \quad \text{if } u \text{ not free in } e & \\ \text{let } (w_1, w_2) = u' \text{ in } [f/u]e & \\ \quad \text{otherwise} & \end{cases} \\
[(\text{void } w')/u] \\ (\text{let } w = u \cdot\ e \text{ in } f) & \equiv & \llbracket f \rrbracket_{w'} \\
[(\text{void } w')/u] \\ (\text{let } (w_1, w_2) = u \text{ in } f) & \equiv & \llbracket f \rrbracket_{w'} \\
[(\text{void } w)/u](\text{void } u) & \equiv & \text{void } w \\
[(\text{void } w)/u]M & \equiv & \begin{cases} M & \text{if } u \text{ not free in } M \\ \text{void } w & \text{otherwise} \end{cases} \\
\hline
\text{If } e = (\lambda u' \in \tau'\,.e'), \\ \text{unit}, (\text{void } w), M, \\ \text{or } (e_1, e_2), \text{ then} & & \\
[e/u](\text{prev } f) & \equiv & \text{prev } f \\
\text{If } e = (\lambda u' \in \tau'\,.e') \\ \text{unit, or } (e_1, e_2), \text{ then} & & \\
[e/u]M & \equiv & M \\
\hline
u \neq w, [e/u]w & \equiv & w \\
[e/u]\text{unit} & \equiv & \text{unit} \\
u \neq w, [e/u](\text{void } w) & \equiv & \text{void } w \\
[e/u](\lambda u' \in \tau'\,.e') & \equiv & \lambda u' \in [e/u]\tau'\,.([e/u]e') \\
u \neq u', [e/u] \\ (\text{let } w = u' \cdot\ e' \text{ in } f) & \equiv & \text{let } w = u' \cdot\ ([e/u]e') \\ & & \text{in } ([e/u]f) \\
[e/u](e_1, e_2) & \equiv & ([e/u]e_1, [e/u]e_2) \\
u \neq u', [e/u] \\ (\text{let } (w_1, w_2) = u' \text{ in } f) & \equiv & \text{let } (w_1, w_2) = u' \\ & & \text{in } ([e/u]f)
\end{array}
$$

First we define the *size* of a formula τ as:

$$\begin{array}{rcl} size(\ominus\tau) & = & 1 + size(\tau) \\ size(\Pi u \in \tau\,.\sigma) & = & 1 + max\{size(\tau), size(\sigma)\} \\ size(\Sigma u \in \tau\,.\sigma) & = & 1 + max\{size(\tau), size(\sigma)\} \\ \text{all other } \tau, size(\tau) & = & 0 \end{array}$$

Notice that $size([e/u]\tau) = size(\tau)$

Proof: By induction on the *size* of the cut formula τ, and simultaneously over derivations $\mathcal{D}$ and $\mathcal{E}$.

First we handle the essential conversions.

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{(u \in \tau) \text{ in } (\Omega_1, u \in \tau, \Omega_2)}{\Omega_1, u \in \tau, \Omega_2 \vdash u \in [\![\tau]\!]}\,\mathsf{ax}$$

$[e/u]\Omega_2$ is not undefined.	by Assumption
$[e/u][\![\tau]\!]$ is not undefined.	by Assumption
$\Omega_1, \leq \Omega_1, [e/u]\Omega_2$	by Definition
$\Omega_1, [e/u]\Omega_2 \vdash e \in \tau$	by Weakening (Lemma A.1)
$\Omega_1, [e/u]\Omega_2 \vdash [\![e]\!]^*_\diamond \in [\![\tau]\!]$	by Shifting (Lemma A.4)
$\Omega_1, [e/u]\Omega_2 \vdash [\![e]\!]^*_\diamond \in [e/u][\![\tau]\!]$	By Redundant Sub., Since u not free in $[\![\tau]\!]$ (Lemma C.5)

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 :: \Omega_1, u' \in \tau \vdash e' \in \sigma}{\Omega_1 \vdash \lambda u' \in \tau\,.e' \in \Pi u' \in \tau\,.\sigma}\,\Pi\mathsf{R},$

$$\mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: \Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash e \in \tau \\ \mathcal{E}_2 :: \Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2, w \in [\![[e/u']\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash \text{let } w = u \cdot\ e \text{ in } f \in \varrho}\,\Pi\mathsf{L}$$

$[\frac{\lambda u' \in \tau \,.\, e'}{u}]\Omega_2$ is not undefined. by Assumption

$[\frac{\lambda u' \in \tau \,.\, e'}{u}]\varrho$ is not undefined. by Assumption

$\mathcal{E}_1' :: \Omega_1, [\frac{\lambda u' \in \tau \,.\, e'}{u}]\Omega_2 \vdash [\frac{\lambda u' \in \tau \,.\, e'}{u}]e \in \tau$
by IH on $\mathcal{D}$ and $\mathcal{E}_1$ and Redundant Sub. Prop. (Lemma C.5)

$\mathcal{E}_2' :: \Omega_1, [\frac{\lambda u' \in \tau \,.\, e'}{u}]\Omega_2, w \in [\![[e/u']\sigma]\!] \vdash [\frac{\lambda u' \in \tau \,.\, e'}{u}]f \in \varrho$
by IH on $\mathcal{D}$ and $\mathcal{E}_2$ and Redundant Sub. Prop. (Lemma C.5)

$\Omega_1, u' \in \tau \leq \Omega_1, [\frac{\lambda u' \in \tau \,.\, e'}{u}]\Omega_2, u' \in \tau$ by Definition

$\mathcal{D}_1' :: \Omega_1, [\frac{\lambda u' \in \tau \,.\, e'}{u}]\Omega_2, u' \in \tau \vdash e' \in \sigma$
by Weakening (Lemma 3.3) on $\mathcal{D}_1$

$[\frac{[\frac{\lambda u' \in \tau \,.\, e'}{u}]e}{u'}]\sigma = [\frac{\lambda u' \in \tau \,.\, e'}{u}]([e/u']\sigma) = [e/u']\sigma$
by Sub. Prop. Distribute and Redundant (Lemmas C.6 and C.5)

$\mathcal{D}_1'' :: \Omega_1, [\frac{\lambda u' \in \tau \,.\, e'}{u}]\Omega_2 \vdash [\frac{[\frac{\lambda u' \in \tau \,.\, e'}{u}]e}{u'}]e' \in [e/u']\sigma$
by IH on $\mathcal{E}_1'$ and $\mathcal{D}_1'$

$\mathcal{E}_2'' :: \Omega_1, [\frac{\lambda u' \in \tau \,.\, e'}{u}]\Omega_2, w \in [e/u']\sigma \vdash [\frac{\lambda u' \in \tau \,.\, e'}{u}]f \in \varrho$
by DeShifting (Lemma 3.5) on $\mathcal{E}_2'$

$[\frac{[\frac{[\frac{\lambda u' \in \tau \,.\, e'}{u}]e}{u'}]e'}{w}]\varrho = \varrho$
by Redundant Sub., since w not free in ϱ (Lemma C.5)

$\mathcal{F} :: \Omega_1, [\frac{\lambda u' \in \tau \,.\, e'}{u}]\Omega_2 \vdash [\frac{[\frac{[\frac{\lambda u' \in \tau \,.\, e'}{u}]e}{u'}]e'}{w}]([\frac{\lambda u' \in \tau \,.\, e'}{u}]f) \in \varrho$
by IH on $\mathcal{D}_1''$ and $\mathcal{E}_2''$

$\Omega_1, [\frac{\lambda u' \in \tau \,.\, e'}{u}]\Omega_2 \vdash [\frac{[\frac{[\frac{\lambda u' \in \tau \,.\, e'}{u}]e}{u'}]e'}{w}]([\frac{\lambda u' \in \tau \,.\, e'}{u}]f) \in [\frac{\lambda u' \in \tau \,.\, e'}{u}]\varrho$
by Redundant Sub. (Lemma C.5)

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 :: {\Omega_1}^{-\ominus} \vdash e \in \tau}{\Omega_1 \vdash \text{prev } e \in \ominus\tau}\ominus\mathsf{R}$

$\mathcal{E} = \dfrac{\mathcal{E}_1 :: (\Omega_1, u \in \ominus\tau, \Omega_2)^{-\ominus} \vdash f \in \sigma}{\Omega_1, u \in \ominus\tau, \Omega_2 \vdash \text{prev } f \in \ominus\sigma}\ominus\mathsf{R}$

$[(\text{prev } e)/u]\Omega_2$ is not undefined. by Assumption

$[(\text{prev } e)/u](\ominus\sigma)$ is not undefined. by Assumption

$[e/u]\sigma$ is not undefined. by Above and Def. of Sub.

$\mathcal{E}_1' :: {\Omega_1}^{-\ominus}, u \in \tau, {\Omega_2}^{-\ominus} \vdash f \in \sigma$

by Property of $(\_)^{-\ominus}$ on $\mathcal{E}_1$

$[e/u]({\Omega_2}^{-\ominus})$ is not undefined.

by Above and Property of $(\_)^{-\ominus}$

$\mathcal{F} :: {\Omega_1}^{-\ominus}, [e/u]({\Omega_2}^{-\ominus}) \vdash [e/u]f \in [e/u]\sigma$

by IH on $\mathcal{D}_1$ and $\mathcal{E}_1'$

$\mathcal{F}' :: (\Omega_1, [(\text{prev } e)/u]\Omega_2)^{-\ominus} \vdash [e/u]f \in [e/u]\sigma$

by Property of $(\_)^{-\ominus}$ and Weakening (Lemma A.1) on $\mathcal{F}$

$\mathcal{F}'' :: \Omega_1, [(\text{prev } e)/u]\Omega_2 \vdash \text{prev } ([e/u]f) \in \ominus([e/u]\sigma)$

by $\ominus\mathsf{R}$

$\mathcal{F}''' :: \Omega_1, [(\text{prev } e)/u]\Omega_2 \vdash \text{prev } ([e/u]f) \in [(\text{prev } e)/u](\ominus\sigma)$

by Definition of Sub.

Case: $\mathcal{D} = \dfrac{[\Omega_1]_{\text{LF}} \vdash M : A}{\Omega_1 \vdash M \in \langle A\rangle}\,\langle\rangle\mathsf{R}$

$\mathcal{E} = \dfrac{\mathcal{E}_1 :: [(\Omega_1, u \in \langle A\rangle, \Omega_2)]_{\text{LF}} \vdash N : B}{\Omega_1, u \in \langle A\rangle, \Omega_2 \vdash N \in \langle B\rangle}\,\langle\rangle\mathsf{R}$

$[M/u]\Omega_2$ is not undefined. by Assumption

$[M/u]\langle B\rangle$ is not undefined. by Assumption

$\mathcal{E}_1' :: [\Omega_1]_{\text{LF}}, u : A, [\Omega_2]_{\text{LF}} \vdash N : B$ by Property of $[\_]_{\text{LF}}$ on $\mathcal{E}_1$

$\mathcal{F} :: [\Omega_1]_{\text{LF}}, [M/u]_{\text{LF}}[\Omega_2]_{\text{LF}} \vdash [M/u]_{\text{LF}}N \in [M/u]_{\text{LF}}B$

by LF Substitution

$\mathcal{F}' :: [(\Omega_1, [M/u]\Omega_2)]_{\text{LF}} \vdash [M/u]_{\text{LF}}N \in [M/u]_{\text{LF}}B$

by Property of $[\_]_{\text{LF}}$ and Substitution on $\mathcal{F}$

$\mathcal{F}'' :: \Omega_1, [M/u]\Omega_2 \vdash [M/u]_{\text{LF}}N \in \langle[M/u]_{\text{LF}}B\rangle$ by $\langle\rangle\mathsf{R}$

$\Omega_1, [M/u]\Omega_2 \vdash [M/u]_{\text{LF}}N \in [M/u]\langle B\rangle$ by Definition of Sub.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \Omega_1 \vdash e_1 \in \tau & \Omega_1 \vdash e_2 \in [e_1/u']\sigma\end{array}}{\Omega \vdash (e_1, e_2) \in \Sigma u' \in \tau\,.\sigma}\,\Sigma\mathsf{R}$

$\mathcal{E} = \dfrac{\begin{array}{c}\Omega_1, u \in (\Sigma u' \in \tau\,.\sigma), \Omega_2,\\ \mathcal{E}_1 :: \quad w_1 \in [\![\tau]\!], w_2 \in [w_1/u'][\![\sigma]\!] \quad \vdash f \in \varrho\end{array}}{\Omega_1, u \in (\Sigma u' \in \tau\,.\sigma), \Omega_2 \vdash \text{let } (w_1, w_2) = u \text{ in } f \in \varrho}\,\Sigma\mathsf{L}$

$[(e_1, e_2)/u]\Omega_2$ is not undefined. by Assumption
$[(e_1, e_2)/u]\varrho$ is not undefined. by Assumption
$\mathcal{F}_1 :: \Omega_1, [(e_1, e_2)/u]\Omega_2, w_1 \in [\![\tau]\!], w_2 \in [w_1/u'][\![\sigma]\!]$
$\quad \vdash [(e_1, e_2)/u]f \in \varrho$
by IH on $\mathcal{D}$ and $\mathcal{E}_1$ and Redundant Sub. Prop. (Lemma C.5)
$\mathcal{D}'_1 :: \Omega_1 \vdash [\![e_1]\!]^*_\diamond \in [\![\tau]\!]$ by Shifting (Lemma A.4)
$[\![[e_1/u']\sigma]\!] = [[\![e_1]\!]^*_\diamond/u'][\![\sigma]\!]$
by Sub. Shift Property (Lemma C.2)
$\mathcal{D}'_2 :: \Omega_1, [(e_1, e_2)/u]\Omega_2 \vdash [\![e_2]\!]^*_\diamond \in [[\![e_1]\!]^*_\diamond/u'][\![\sigma]\!]$
by Weakening and Shifting (Lemma A.1 and A.4)
$[[\![e_1]\!]^*_\diamond/w_1]\varrho = \varrho$
by Redundant Sub., since w_1 not free in ϱ (Lemma C.5)
$[[\![e_1]\!]^*_\diamond/w_1]([w_1/u'][\![\sigma]\!]) = [[\![e_1]\!]^*_\diamond/u'][\![\sigma]\!]$
By Prop. of Sub., Distributivity and Redundancy (on w_1) (Lemmas C.6 and C.5)
$\mathcal{F}_2 :: \Omega_1, [(e_1, e_2)/u]\Omega_2, w_2 \in [[\![e_1]\!]^*_\diamond/u'][\![\sigma]\!]$
$\quad \vdash [[\![e_1]\!]^*_\diamond/w_1]([(e_1, e_2)/u]f) \in \varrho$
by IH on $\mathcal{D}'_1$ and $\mathcal{F}_1$
$[[\![e_2]\!]^*_\diamond/w_2]\varrho = \varrho$
by Redundant Sub., since w_2 not free in ϱ (Lemma C.5)
$\Omega_1, [(e_1, e_2)/u]\Omega_2$
$\quad \vdash [[\![e_2]\!]^*_\diamond/w_2]([[\![e_1]\!]^*_\diamond/w_1]([(e_1, e_2)/u]f)) \in \varrho$
by IH on $\mathcal{D}'_2$ and $\mathcal{F}_2$
$\Omega_1, [(e_1, e_2)/u]\Omega_2$
$\quad \vdash [[\![e_2]\!]^*_\diamond/w_2]([[\![e_1]\!]^*_\diamond/w_1]([(e_1, e_2)/u]f))$
$\quad \in [(e_1, e_2)/u]\varrho$ by Redundant Sub (Lemma C.5)

Now we handle when we have $\ominus\mathsf{R}$ on the left and a non-commutative, non-$\ominus\mathsf{R}$ on the right.

Case: $\mathcal{D} = \dfrac{{\Omega_1}^{-\ominus} \vdash e \in \ominus^k \bot}{\Omega_1 \vdash \text{prev } e \in \ominus^{k+1} \bot}\ \ominus\mathsf{R}$

$\mathcal{E} = \dfrac{}{\Omega_1, u \in \ominus^{k+1}\bot, \Omega_2 \vdash \text{void } u \in [\![\sigma]\!]}\ \bot\mathsf{L}$

$[(\text{prev } e)/u]\Omega_2$ is not undefined. by Assumption
$[(\text{prev } e)/u][\![\sigma]\!]$ is not undefined. by Assumption

$[(\text{prev } e)/u][\![\sigma]\!] = [\![\sigma^*]\!]$
by Property of Sub. (Lemma C.2)
$\mathcal{E}' :: \Omega_1, u \in \ominus^k \bot \vdash \text{void } u \in [\![\sigma^*]\!]$ by $\bot\mathsf{L}$
$\mathcal{D}' :: \Omega_1 \vdash [\![\text{prev } e]\!]_\diamond \in \ominus^k \bot$ by Shifting (Lemma A.4) on $\mathcal{D}$
$\mathcal{D}'' :: \Omega_1 \vdash e \in \ominus^k \bot$ by Definition of $[\![\_]\!]_\diamond$
$[e/u][\![\sigma^*]\!] = [\![\sigma^*]\!]$
By Redundant Sub., since u not free in $[\![\sigma^*]\!]$ (Lemma C.5)
$\mathcal{F} :: \Omega_1 \vdash [e/u](\text{void } u) \in [\![\sigma^*]\!]$
by IH on $\mathcal{D}''$ and $\mathcal{E}'$
$\mathcal{F}' :: \Omega_1, [(\text{prev } e)/u]\Omega_2 \vdash [e/u](\text{void } u) \in [\![\sigma^*]\!]$
by Weakening (Lemma A.1) on $\mathcal{F}$
$\Omega_1, [(\text{prev } e)/u]\Omega_2 \vdash [e/u](\text{void } u) \in [(\text{prev } e)/u][\![\sigma]\!]$
by Above

Case: $\mathcal{D} = \dfrac{{\Omega_1}^{-\ominus} \vdash e \in \ominus^k(\Pi u' \in \tau\,.\sigma)}{\Omega_1 \vdash \text{prev } e \in \ominus^{k+1}(\Pi u' \in \tau\,.\sigma)}\ominus\mathsf{R}$

$$\mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: \Omega_1, u \in \ominus^{k+1}(\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash e' \in \tau \\ \mathcal{E}_2 :: \Omega_1, u \in \ominus^{k+1}(\Pi u' \in \tau\,.\sigma), \Omega_2, w \in [\![[e'/u'']\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1, u \in \ominus^{k+1}(\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash \text{let } w = u \cdot\ e' \text{ in } f \in \varrho}\,\Pi\mathsf{L}$$

$[(\text{prev } e)/u]\Omega_2$ is not undefined. by Assumption
$[(\text{prev } e)/u]\varrho$ is not undefined. by Assumption
$[(\text{prev } e)/u]\tau = [e/u]\tau = \tau$ By Redundant Sub., since u not free in τ (Lemma C.5)
$\mathcal{D}' :: \Omega_1 \vdash [\![\text{prev } e]\!]_\diamond \in \ominus^k(\Pi u' \in \tau\,.\sigma)$
by Shifting (Lemma A.4) on $\mathcal{D}$
$\mathcal{D}'' :: \Omega_1 \vdash e \in \ominus^k(\Pi u' \in \tau\,.\sigma)$ by Definition of $[\![\_]\!]_\diamond$
$\mathcal{E}_1' :: \Omega_1, [(\text{prev } e)/u]\Omega_2 \vdash [(\text{prev } e)/u]e' \in \tau$
by IH on $\mathcal{D}$ and $\mathcal{E}_1$
$[(\text{prev } e)/u][\![[e'/u']\sigma]\!] = [e/u][\![[e'/u']\sigma]\!] = [\![[e'/u']\sigma]\!]$
By Redundant Sub., since u not free in $[\![\sigma]\!]$ (Lemma C.5)
$\mathcal{E}_2' :: \Omega_1, [(\text{prev } e)/u]\Omega_2, w \in [\![[e'/u']\sigma]\!]$
$\quad \vdash [(\text{prev } e)/u]f \in [(\text{prev } e)/u]\varrho$ by IH on $\mathcal{D}$ and $\mathcal{E}_2$
$\mathcal{F}_1 :: \Omega_1, [(\text{prev } e)/u]\Omega_2, u \in \ominus^k(\Pi u' \in \tau\,.\sigma)$
$\quad \vdash \text{let } w = u \cdot\ [(\text{prev } e)/u]e' \text{ in } [(\text{prev } e)/u]f$

$\in [(\text{prev } e)/u]\varrho$

by Weakening (Lemma A.1) and $\Pi\mathsf{L}$

$[e/u]([(\text{prev } e)/u]\varrho) = [(\text{prev } e)/u]\varrho$

By Redundant Sub.,
since u not free in $[(\text{prev } e)/u]\varrho$ (Lemma C.5)

$\Omega_1, [(\text{prev } e)/u]\Omega_2$
$\vdash [e/u](\text{let } w = u \cdot\ [(\text{prev } e)/u]e' \text{ in } [(\text{prev } e)/u]f)$
$\in [(\text{prev } e)/u]\varrho$

by IH on $\mathcal{D}''$ and $\mathcal{F}_1$

Case: $\mathcal{D} = \dfrac{{\Omega_1}^{-\ominus} \vdash e \in \ominus^k\langle A\rangle}{\Omega_1 \vdash \text{prev } e \in \ominus^{k+1}\langle A\rangle}\ominus\mathsf{R}$

$\mathcal{E} = \dfrac{\mathcal{E}_1 :: [(\Omega_1, u \in \ominus^{k+1}\langle A\rangle, \Omega_2)]_{\text{LF}} \vdash M : B}{\Omega_1, u \in \ominus^{k+1}\langle A\rangle, \Omega_2 \vdash M \in \langle B\rangle}\langle\rangle\mathsf{R}$

$[(\text{prev } e)/u]\Omega_2$ is not undefined. by Assumption
$[(\text{prev } e)/u]\langle B\rangle$ is not undefined. by Assumption
$\mathcal{D}' :: \Omega_1 \vdash [\![\text{prev } e]\!]_\diamond \in \ominus^k\langle A\rangle$ by Shifting (Lemma A.4) on $\mathcal{D}$
$\mathcal{D}'' :: \Omega_1 \vdash e \in \ominus^k\langle A\rangle$ by Definition of $[\![\_]\!]_\diamond$
Transform Ω_2 into Ω_2^* such that for all $(w \in \tau)$ in Ω_2
If $\tau = \ominus\tau'$, then $(w \in \tau')$ in Ω_2^*
Else, $(w \in \tau)$ in Ω_2^*.
Note that this operation removes a past whenever possible.
And preserves the dependencies between terms.
And $[e/u]\Omega_2$ is defined everywhere.
$\mathcal{E}_1' :: [(\Omega_1, u \in \ominus^k\langle A\rangle, \Omega_2^*)]_{\text{LF}} \vdash M : B$

by Property of $[\_]_{\text{LF}}$ and Above on $\mathcal{E}_1$

$\mathcal{F}_1 :: \Omega_1, u \in \ominus^k\langle A\rangle, \Omega_2^* \vdash M \in \langle B\rangle$

by $\langle\rangle\mathsf{R}$

$[(\text{prev } e)/u]\langle B\rangle = [e/u]\langle B\rangle$ by Definition of Sub.
$\Omega_1, [e/u]\Omega_2^* \vdash [e/u]M \in [e/u]\langle B\rangle$ by IH on $\mathcal{D}''$ and $\mathcal{F}_1$
$\Omega_1, [e/u]\Omega_2^* \vdash [e/u]M \in [(\text{prev } e)/u]\langle B\rangle$ by Above
$\Omega_1, [(\text{prev } e)/u]\Omega_2 \vdash [e/u]M \in [(\text{prev } e)/u]\langle B\rangle$

by Property of Sub, where
DeShifting (Lemma 3.5)
is applied to every element that
was shifted in construction of Ω_2^*

Case: $\mathcal{D} = \dfrac{{\Omega_1}^{-\ominus} \vdash e \in \ominus^k(\Pi u' \in \tau\,.\sigma)}{\Omega_1 \vdash \text{prev } e \in \ominus^{k+1}(\Pi u' \in \tau\,.\sigma)}\ \ominus\mathsf{R}$

$$\mathcal{E} = \dfrac{\begin{array}{c}\Omega_1, u \in \ominus^{k+1}(\Sigma u' \in \tau\,.\sigma),\\ \mathcal{E}_1 :: \quad \Omega_2, w_1 \in [\![\tau]\!], w_2 \in [w_1/u'][\![\sigma]\!] \quad \vdash f \in \varrho\end{array}}{\Omega_1, u \in \ominus^{k+1}(\Sigma u' \in \tau\,.\sigma), \Omega_2 \vdash \text{let } (w_1, w_2) = u \text{ in } f \in \varrho}\ \Sigma\mathsf{L}$$

$[(\text{prev } e)/u]\Omega_2$ is not undefined. by Assumption

$[(\text{prev } e)/u]\varrho$ is not undefined. by Assumption

$[(\text{prev } e)/u]\tau = \tau$

By Redundant Sub., since u not free in τ (Lemma C.5)

$[(\text{prev } e)/u]([w_1/u'][\![\sigma]\!]) = [w_1/u'][\![\sigma]\!]$

By Redundant Sub.,
since u not free in $[w_1/u'][\![\sigma]\!]$ (Lemma C.5)

$\mathcal{D}' :: \Omega_1 \vdash [\![\text{prev } e]\!]_\diamond \in \ominus^k(\Pi u' \in \tau\,.\sigma)$

by Shifting (Lemma A.4) on $\mathcal{D}$

$\mathcal{D}'' :: \Omega_1 \vdash e \in \ominus^k(\Pi u' \in \tau\,.\sigma)$ by Definition of $[\![\_]\!]_\diamond$

$\mathcal{E}_1' :: \Omega_1, [(\text{prev } e)/u]\Omega_2, w_1 \in [\![\tau]\!], w_2 \in [w_1/u'][\![\sigma]\!]$
$\quad \vdash [(\text{prev } e)/u]f \in [(\text{prev } e)/u]\varrho$

by IH on $\mathcal{D}$ and $\mathcal{E}_1$

$\mathcal{F}_1 :: \Omega_1, [(\text{prev } e)/u]\Omega_2, u \in \ominus^k(\Sigma u' \in \tau\,.\sigma)$
$\quad \vdash \text{let } (w_1, w_2) = u \text{ in } [(\text{prev } e)/u]f$
$\quad \in [(\text{prev } e)/u]\varrho$

by Weakening (Lemma A.1) and $\Sigma\mathsf{L}$

$[e/u]([(\text{prev } e)/u]\varrho) = [(\text{prev } e)/u]\varrho$

By Redundant Sub.,
since u not free in $[(\text{prev } e)/u]\varrho$ (Lemma C.5)

$\Omega_1, [(\text{prev } e)/u]\Omega_2$
$\quad \vdash [e/u](\text{let } (w_1, w_2) = u \text{ in } [(\text{prev } e)/u]f)$
$\quad \in [(\text{prev } e)/u]\varrho$

by IH on $\mathcal{D}''$ and $\mathcal{F}_1$

Now we handle commutative conversion on the left.

Case: $\mathcal{D} = \dfrac{(w \in \tau) \in \Omega_1}{\Omega_1 \vdash w \in [\![\tau]\!]}\ \mathsf{ax}$

$\mathcal{E} = \Omega_1, u \in [\![\tau]\!], \Omega_2 \vdash e \in \sigma$

$[w/u]\Omega_2$ is not undefined. by Assumption
$[w/u]\sigma$ is not undefined. by Assumption
$\Omega_1, [w/u]\Omega_2 \vdash$ rename u to w in $e \in [w/u]\sigma$
by variable renaming

Case:

$$\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_0 :: (u' \in \ominus^k(\Pi u'' \in \tau\,.\sigma)) \in \Omega_1 \\ \mathcal{D}_1 :: \Omega_1 \vdash e' \in \tau \\ \mathcal{D}_2 :: \Omega_1, w \in [\![[e'/u'']\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1 \vdash \text{let } w = u' \cdot\ e' \text{ in } f \in \varrho}\ \Pi\mathsf{L}$$

$\mathcal{E} = \Omega_1, u \in \varrho, \Omega_2 \vdash e \in \varrho'$

$[(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\Omega_2$ is not undefined. by Assumption
$[(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\varrho'$ is not undefined. by Assumption
$\mathcal{F}_0 :: (u' \in \ominus^k(\Pi u'' \in \tau\,.\sigma))$
$\quad \in \Omega_1, [(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\Omega_2$ by $\mathcal{D}_0$
$\mathcal{F}_1 :: \Omega_1, [(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\Omega_2 \vdash e' \in \tau$
by Weakening (Lemma A.1) on $\mathcal{D}_1$
$\mathcal{E}' :: \Omega_1, w \in [\![[e'/u'']\sigma]\!], u \in \varrho, \Omega_2 \vdash e \in \varrho'$
by Weakening (Lemma A.1) on $\mathcal{E}$
$\Omega_1, w \in [\![[e'/u'']\sigma]\!], [(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\Omega_2$
$\quad \vdash [f/u]e$
$\quad \in [(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\varrho'$ by IH on $\mathcal{D}_2$ and $\mathcal{E}'$
$\Omega_1, [(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\Omega_2, w \in [\![[e'/u'']\sigma]\!]$
$\quad \vdash [f/u]e$ by Context Reordering
$\quad \in [(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\varrho'$ since w not free in Ω_2
$\Omega_1, [(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\Omega_2$
$\quad \vdash \text{let } w = u' \cdot\ e' \text{ in } [f/u]e$
$\quad \in [(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\varrho'$ by $\Pi\mathsf{L}$

If u does not occur free in e, then
$\Omega_1, [(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\Omega_2$
$\quad \vdash [f/u]e$
$\quad \in [(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\varrho'$
by Redundant Let Lemma C.10

Case:

$$\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_0 :: (u' \in \ominus^k(\Sigma u'' \in \tau\,.\sigma)) \in \Omega_1 \\ \mathcal{D}_1 :: \Omega_1, w_1 \in [\![\tau]\!], w_2 \in [w_1/u'][\![\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1 \vdash \text{let } (w_1, w_2) = u' \text{ in } f \in \varrho}\ \Sigma\mathsf{L}$$

$\mathcal{E} = \Omega_1, u \in \varrho, \Omega_2 \vdash e \in \varrho'$

$[(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\Omega_2$ is not undefined.
by Assumption

$[(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\varrho'$ is not undefined.
by Assumption

$\mathcal{F}_0 :: (u' \in \ominus^k(\Sigma u'' \in \tau\,.\sigma))$
$\quad \in \Omega_1, [(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\Omega_2$ by $\mathcal{D}_0$

$\mathcal{E}' :: \Omega_1, w_1 \in [\![\tau]\!], w_2 \in [w_1/u'][\![\sigma]\!], u \in \varrho, \Omega_2 \vdash e \in \varrho'$
by Weakening (Lemma A.1) on $\mathcal{E}$

$\Omega_1, w_1 \in [\![\tau]\!], w_2 \in [w_1/u'][\![\sigma]\!],$
$[(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\Omega_2$
$\quad \vdash [f/u]e$
$\quad \in [(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\varrho'$
by IH on $\mathcal{D}_1$ and $\mathcal{E}'$

$\Omega_1, [(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\Omega_2,$
$w_1 \in [\![\tau]\!], w_2 \in [w_1/u'][\![\sigma]\!]$
$\quad \vdash [f/u]e$
$\quad \in [(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\varrho'$
by Context Reordering
since w_1 and w_2 are not free in Ω_2

$\Omega_1, [(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\Omega_2$
$\quad \vdash \text{let } (w_1, w_2) = u' \text{ in } [f/u]e$
$\quad \in [(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\varrho'$ by $\Sigma\mathsf{L}$

If u does not occur free in e, then
$\Omega_1, [(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\Omega_2$
$\quad \vdash [f/u]e$
$\quad \in [(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\varrho'$
by Redundant Let Lemma C.10

Case:

$$\mathcal{D} = \dfrac{(w' \in \ominus^k \bot) \in \Omega_1}{\Omega_1 \vdash \text{void } w' \in (\Pi u' \in \tau\,.\sigma)}\ \bot\mathsf{L}$$

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1 :: \Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash e \in \tau \\ \mathcal{E}_2 :: \Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2, w \in [\![[e/u]\sigma]\!] \vdash f \in \varrho\end{array}}{\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash \text{let } w = u \cdot\ e \text{ in } f \in \varrho}\ \Pi\mathsf{L}$$

$[(\text{void } w')/u]\Omega_2$ is not undefined. by Assumption
$[(\text{void } w')/u]\varrho$ is not undefined. by Assumption
$[(\text{void } w')/u]\Omega_2 = \Omega_2$ by Above and Def. of Sub.
$[(\text{void } w')/u]\varrho = \varrho$ by Above and Def. of Sub.
$\Omega_1, \Omega_2 \vdash \mathbb{\Pi}\text{let } w = u \cdot\ e \text{ in } f \mathbb{\Pi}_{w'} \in \varrho$
by Commute $\perp$ (Lemma C.9)
$\Omega_1, \Omega_2 \vdash \mathbb{\Pi}\, f\, \mathbb{\Pi}_{w'} \in \varrho$
by Definition of $\mathbb{\Pi}\_\mathbb{\Pi}_{w'}$
$\Omega_1, [(\text{void } w')/u]\Omega_2 \vdash \mathbb{\Pi}\, f\, \mathbb{\Pi}_{w'} \in [(\text{void } w')/u]\varrho$
by Above

Case:

$$\mathcal{D} = \frac{(w' \in \ominus^k \perp) \in \Omega_1}{\Omega_1 \vdash \text{void } w' \in (\Sigma u' \in \tau\,.\sigma)}\ \perp\mathsf{L}$$

$$\mathcal{E} = \frac{\begin{array}{c}\Omega_1, u \in (\Sigma u' \in \tau\,.\sigma), \Omega_2, \\ \mathcal{E}_1 ::\ \ w_1 \in [\![\tau]\!], w_2 \in [w_1/u'][\![\sigma]\!]\ \ \vdash f \in \varrho\end{array}}{\Omega_1, u \in (\Sigma u' \in \tau\,.\sigma), \Omega_2 \vdash \text{let } (w_1, w_2) = u \cdot\ e \text{ in } f \in \varrho}\ \Sigma\mathsf{L}$$

$[(\text{void } w')/u]\Omega_2$ is not undefined. by Assumption
$[(\text{void } w')/u]\varrho$ is not undefined. by Assumption
$[(\text{void } w')/u]\Omega_2 = \Omega_2$ by Above and Def. of Sub.
$[(\text{void } w')/u]\varrho = \varrho$ by Above and Def. of Sub.
$\Omega_1, \Omega_2 \vdash \mathbb{\Pi}\text{let } (w_1, w_2) = u \text{ in } f \mathbb{\Pi}_{w'} \in \varrho$
by Commute $\perp$ (Lemma C.9)
$\Omega_1, \Omega_2 \vdash \mathbb{\Pi}\, f\, \mathbb{\Pi}_{w'} \in \varrho$
by Definition of $\mathbb{\Pi}\_\mathbb{\Pi}_{w'}$
$\Omega_1, [(\text{void } w')/u]\Omega_2 \vdash \mathbb{\Pi}\, f\, \mathbb{\Pi}_{w'} \in [(\text{void } w')/u]\varrho$
by Above

Case:

$$\mathcal{D} = \frac{(w \in \ominus^k \perp) \in \Omega_1}{\Omega_1 \vdash \text{void } w \in \perp}\ \perp\mathsf{L}$$

$\mathcal{E} = \Omega_1, u \in \perp, \Omega_2 \vdash (\text{void } u) \in [\![\sigma]\!]$

$[(\text{void } w)/u]\Omega_2$ is not undefined. by Assumption
$[(\text{void } w)/u][\![\sigma]\!]$ is not undefined. by Assumption
$[(\text{void } w)/u][\![\sigma]\!] = [\![\sigma^*]\!]$
by Property of Sub. (Lemma C.2)
$\Omega_1, [(\text{void } w)/u]\Omega_2 \vdash \text{void } w \in [\![\sigma^*]\!]$ by $\bot\mathsf{L}$
$\Omega_1, [(\text{void } w)/u]\Omega_2 \vdash \text{void } w \in [(\text{void } w)/u][\![\sigma]\!]$ by Above

Case:

$$\mathcal{D} = \frac{(w \in \ominus^k \bot) \in \Omega_1}{\Omega_1 \vdash \text{void } w \in [\![\tau]\!]}\ \bot\mathsf{L}$$

$\mathcal{E} = \Omega_1, u \in [\![\tau]\!], \Omega_2 \vdash M \in \langle A \rangle$

$[(\text{void } w)/u]\Omega_2$ is not undefined. by Assumption
$[(\text{void } w)/u]\langle A\rangle$ is not undefined. by Assumption
$[(\text{void } w)/u]\Omega_2 = \Omega_2$ by Inversion on Def. of Sub. and Above
u does not occur free in Ω_2 by Above
$[(\text{void } w)/u]\langle A\rangle = \langle A\rangle$ by Inversion on Def. of Sub. and Above
$\Omega_1, \Omega_2 \vdash \text{void } w \in \langle A\rangle$ by $\bot\mathsf{L}$
$\Omega_1, [(\text{void } w)/u]\Omega_2 \vdash \text{void } w \in [(\text{void } w)/u]\langle A\rangle$ by Above

If u does not occur free in M, then
$\Omega_1, \Omega_2 \vdash M \in \langle A\rangle$ by Redundant Term C.11
$\Omega_1, [(\text{void } w)/u]\Omega_2 \vdash M \in [(\text{void } w)/u]\langle A\rangle$ by Above

Now we handle strengthening conditions when the left is in the present and the right is in the past.

Case: In this case, we consider when e is
either $\lambda u' \in \tau'.e'$, unit, $(\text{void } w)$, M, or (e_1, e_2)

$\mathcal{D} = \Omega_1 \vdash e \in [\![\tau]\!]$

$$\mathcal{E} = \frac{\mathcal{E}_1 :: (\Omega_1, u \in [\![\tau]\!], \Omega_2)^{-\ominus} \vdash f \in \sigma}{\Omega_1, u \in [\![\tau]\!], \Omega_2 \vdash \text{prev } f \in \ominus\sigma}\ \ominus\mathsf{R}$$

$[e/u]\Omega_2$ is not undefined. by Assumption

$[e/u](\ominus\sigma)$ is not undefined. by Assumption
$[e/u](\ominus\sigma) = \ominus\sigma$ By Redundant Sub., since u not free in σ (Lemma C.5)
Transform Ω_2 into Ω_2^* such that for all $(u' \in \tau')$ in Ω_2
If $\tau' \neq \ominus\tau''$, then u' is not in Ω_2^*
Else, $(u' \in \tau')$ in Ω_2^*.
Note that this operation removes anything that is not past.
And preserves the dependencies between terms.
And $[e/u]\Omega_2^* \leq [e/u]\Omega_2$.

$\mathcal{F}_1 :: {\Omega_1}^{-\ominus}, {\Omega_2^*}^{-\ominus} \vdash f \in \sigma$
by Property of $(\_)^{-\ominus}$ and Above on $\mathcal{E}_1$
$[e/u]{\Omega_2^*}^{-\ominus} = {\Omega_2^*}^{-\ominus}$
by Property of $(\_)^{-\ominus}$ and restriction on e
$(\Omega_1, [e/u]\Omega_2^*)^{-\ominus} \vdash f \in \sigma$
by Property of $(\_)^{-\ominus}$ and Above
$\Omega_1, [e/u]\Omega_2^* \vdash \text{prev } f \in \ominus\sigma$ by $\ominus\mathsf{R}$
$\Omega_1, [e/u]\Omega_2^* \vdash \text{prev } f \in [e/u](\ominus\sigma)$ by Above
$\Omega_1, [e/u]\Omega_2 \vdash \text{prev } f \in [e/u](\ominus\sigma)$
by Weakening (Lemma A.1)

Case: In this case, e is either $\lambda u' \in \tau' .e'$, unit, or (e_1, e_2)
so τ is either $(\Pi u' \in \tau' .\sigma')$, $(\Sigma u' \in \tau' .\sigma')$, or $\top$

$\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{\mathcal{E}_1 :: [(\Omega_1, u \in \tau, \Omega_2)]_{\text{LF}} \vdash M : A}{\Omega_1, u \in \tau, \Omega_2 \vdash M \in \langle A \rangle}\, \langle\rangle\mathsf{R}$$

$[e/u]\Omega_2$ is not undefined. by Assumption
$[e/u]\langle A \rangle$ is not undefined. by Assumption
$[e/u]\langle A \rangle = \langle A \rangle$ By Redundant Sub. (Lemma C.5)
$\mathcal{F}_1 :: [\Omega_1]_{\text{LF}}, [\Omega_2]_{\text{LF}} \vdash M : A$ by Property of $[\_]_{\text{LF}}$ on $\mathcal{E}_1$
$[[e/u]\Omega_2]_{\text{LF}} = [\Omega_2]_{\text{LF}}$ by Property of $[\_]_{\text{LF}}$
$\mathcal{F}_2 :: [\Omega_1, [e/u]\Omega_2]_{\text{LF}} \vdash M : A$ by Property of $[\_]_{\text{LF}}$ and Above
$\mathcal{F}_3 :: \Omega_1, [e/u]\Omega_2 \vdash M \in \langle A \rangle$ by $\langle\rangle\mathsf{R}$
$\Omega_1, [e/u]\Omega_2 \vdash M \in [e/u]\langle A \rangle$ by Above

Now we handle commutative conversion cases on the right.

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{(w \in \sigma) \text{ in } (\Omega_1, u \in \tau, \Omega_2)}{\Omega_1, u \in \tau, \Omega_2 \vdash w \in [\![\sigma]\!]}\,\mathsf{ax}, \text{ where } u \neq w$$

$[e/u]\Omega_2$ is not undefined. by Assumption
$[e/u][\![\sigma]\!]$ is not undefined. by Assumption
$(w \in [e/u]\sigma)$ in $(\Omega_1, [e/u]\Omega_2)$ since $u \neq w$ and Prop. of Sub.
$\Omega_1, [e/u]\Omega_2 \vdash w \in [\![[e/u]\sigma]\!]$ by ax
$[\![[e/u]\sigma]\!] = [[\![e]\!]^*_\diamond/u][\![\sigma]\!] = [e/u][\![\sigma]\!]$
by Sub. Shift and DeShift Properties (Lemmas C.2 and C.4)
$\Omega_1, [e/u]\Omega_2 \vdash w \in [e/u][\![\sigma]\!]$ by Above

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{}{\Omega_1, u \in \tau, \Omega_2 \vdash \text{unit} \in \top}\,\top\mathsf{R}$$

$[e/u]\Omega_2$ is not undefined. by Assumption
$\Omega_1, [e/u]\Omega_2 \vdash \text{unit} \in \top$ by $\top\mathsf{R}$
$\Omega_1, [e/u]\Omega_2 \vdash \text{unit} \in [e/u]\top$ by Definition of Sub.

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{(w \in \ominus^k \bot) \in (\Omega_1, u \in \tau, \Omega_2)}{\Omega_1, u \in \tau, \Omega_2 \vdash \text{void } w \in [\![\sigma]\!]}\,\bot\mathsf{L}, \text{ where } u \neq w$$

$[e/u]\Omega_2$ is not undefined. by Assumption
$[e/u][\![\sigma]\!]$ is not undefined. by Assumption
$[e/u][\![\sigma]\!] = [\![\sigma^*]\!]$ by Property of Sub. (Lemma C.2)
$(w \in \ominus^k \bot) \in (\Omega_1, [e/u]\Omega_2)$ since $u \neq w$ and Prop. of Sub.
$\Omega_1, [e/u]\Omega_2 \vdash \text{void } w \in [\![\sigma^*]\!]$ by $\bot\mathsf{L}$
$\Omega_1, [e/u]\Omega_2 \vdash \text{void } w \in [e/u][\![\sigma]\!]$ by Above

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{\mathcal{E}_1 :: \Omega_1, u \in \tau, \Omega_2, u' \in \tau' \vdash e' \in \sigma}{\Omega_1, u \in \tau, \Omega_2 \vdash \lambda u' \in \tau' .e' \in (\Pi u'' \in \tau' .\sigma)}\ \Pi\mathsf{R}$$

$[e/u]\Omega_2$ is not undefined. by Assumption
$[e/u](\Pi u'' \in \tau' .\sigma))$ is not undefined. by Assumption
$[e/u](\Pi u'' \in \tau' .\sigma) = \Pi u'' \in [e/u]\tau' .[e/u]\sigma$
by Definition of Sub.
$\Omega_1, [e/u]\Omega_2, u' \in [e/u]\tau' \vdash [e/u]e' \in [e/u]\sigma$
by IH on $\mathcal{D}$ and $\mathcal{E}_1$
$\Omega_1, [e/u]\Omega_2$
$\quad \vdash \lambda u' \in [e/u]\tau' .([e/u]e') \in \Pi u'' \in [e/u]\tau' .[e/u]\sigma$
by $\Pi\mathsf{R}$
$\Omega_1, [e/u]\Omega_2$
$\quad \vdash \lambda u' \in [e/u]\tau' .([e/u]e') \in [e/u](\Pi u'' \in \tau' .\sigma)$
by Above

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{\begin{array}{l} u \neq u' \\ \mathcal{E}_0 :: (u' \in \ominus^k(\Pi u'' \in \tau' .\sigma')) \in (\Omega_1, u \in \tau, \Omega_2) \\ \mathcal{E}_1 :: \Omega_1, u \in \tau, \Omega_2 \vdash e' \in \tau' \\ \mathcal{E}_2 :: \Omega_1, u \in \tau, \Omega_2, w \in [\![[e'/u'']\sigma']\!] \vdash f \in \varrho \end{array}}{\Omega_1, u \in \tau, \Omega_2 \vdash \text{let } w = u' \cdot\ e' \text{ in } f \in \varrho}\ \Pi\mathsf{L}$$

$[e/u]\Omega_2$ is not undefined. by Assumption
$[e/u]\varrho$ is not undefined. by Assumption
$\mathcal{F}_0 :: (u' \in \ominus^k(\Pi u'' \in [e/u]\tau' .[e/u]\sigma')) \in (\Omega_1, [e/u]\Omega_2)$
since $u \neq u'$, Def. of Sub., and Sub DeShift Property (Lemma C.4) considering that either u is not free or $e = \text{prev}^k\ (e^*)$
$\mathcal{F}_1 :: \Omega_1, [e/u]\Omega_2 \vdash [e/u]e' \in [e/u]\tau'$
by IH on $\mathcal{D}$ and $\mathcal{E}_1$
$[e/u]([e'/u'']\sigma')$ is not undefined.
by Not Undefined Prop. (Lemma C.7)
$[e/u]([e'/u'']\sigma') = [([e/u]e')/u'']([e/u]\sigma')$
by Sub. Distribute Prop. (Lemma C.6)
$[\![[([e/u]e')/u'']([e/u]\sigma')]\!] = [\![[e/u]([e'/u'']\sigma')]\!]$
$= [e/u][\![[e'/u'']\sigma']\!]$

by Above and Sub. DeShift. Prop. (Lemma C.4)

$\mathcal{F}_2 :: \Omega_1, [e/u]\Omega_2, w \in [\![([([e/u]e')/u'']([e/u]\sigma')]\!]$
$\quad \vdash [e/u]f \in [e/u]\varrho$

by IH on $\mathcal{D}$ and $\mathcal{E}_2$

$\Omega_1, [e/u]\Omega_2 \vdash \text{let } w = u' \cdot\ ([e/u]e') \text{ in } ([e/u]f) \in [e/u]\varrho$

by $\Pi\mathsf{L}$

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{\begin{array}{l}\mathcal{E}_1 :: \Omega_1, u \in \tau, \Omega_2 \vdash e_1 \in \sigma_1 \\ \mathcal{E}_2 :: \Omega_1, u \in \tau, \Omega_2, \vdash e_2 \in [e_1/u']\sigma_2\end{array}}{\Omega_1, u \in \tau, \Omega_2 \vdash (e_1, e_2) \in \Sigma u' \in \sigma_1 .\sigma_2}\ \Sigma\mathsf{R}$$

$[e/u]\Omega_2$ is not undefined. by Assumption

$[e/u](\Sigma u' \in \sigma_1 .\sigma_2)$ is not undefined. by Assumption

$[e/u](\Sigma u' \in \sigma_1 .\sigma_2) = \Sigma u' \in [e/u]\sigma_1 .[e/u]\sigma_2$

by Above and Def. of Sub.

$\mathcal{F}_1 :: \Omega_1, [e/u]\Omega_2 \vdash [e/u]e_1 \in [e/u]\sigma_1$

by IH on $\mathcal{D}$ and $\mathcal{E}_1$

$[e/u]([e_1/u']\sigma_2)$ is not undefined.

by Not Undefined Prop. (Lemma C.7)

$[e/u]([e_1/u']\sigma_2) = [([e/u]e_1)/u']([e/u]\sigma_2)$

by Sub. Distribute Prop. (Lemma C.6)

$\mathcal{F}_2 :: \Omega_1, [e/u]\Omega_2 \vdash [e/u]e_2 \in [([e/u]e_1)/u']([e/u]\sigma_2)$

by IH on $\mathcal{D}$ and $\mathcal{E}_2$

$\Omega_1, [e/u]\Omega_2 \vdash ([e/u]e_1, [e/u]e_2) \in \Sigma u' \in [e/u]\sigma_1 .[e/u]\sigma_2$

by $\Sigma\mathsf{R}$

$\Omega_1, [e/u]\Omega_2 \vdash ([e/u]e_1, [e/u]e_2) \in [e/u](\Sigma u' \in \sigma_1 .\sigma_2)$

by Above

Case: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

$$\mathcal{E} = \frac{\begin{array}{l}u \neq u' \\ \mathcal{E}_0 :: (u' \in \ominus^k(\Sigma u'' \in \tau' .\sigma')) \in (\Omega_1, u \in \tau, \Omega_2) \\ \mathcal{E}_1 :: \Omega_1, u \in \tau, \Omega_2, w_1 \in [\![\tau']\!], w_2 \in [w_1/u''][\![\sigma']\!] \vdash f \in \varrho\end{array}}{\Omega_1, u \in \tau, \Omega_2 \vdash \text{let } (w_1, w_2) = u' \text{ in } f \in \varrho}\ \Sigma\mathsf{L}$$

$[e/u]\Omega_2$ is not undefined. by Assumption

$[e/u]\varrho$ is not undefined. by Assumption

$\mathcal{F}_0 :: (u' \in \ominus^k(\Sigma u'' \in [e/u]\tau' . [e/u]\sigma')) \in (\Omega_1, [e/u]\Omega_2)$
since $u \neq u'$, Def. of Sub., and Sub DeShift Property (Lemma C.4) considering that either u is not free or $e = \text{prev}^k\ (e^*)$

$[\![[e/u]\tau']\!] = [[\![e]\!]^*_\diamond/u][\![\tau']\!] = [e/u][\![\tau']\!]$
by Sub. Shift and DeShift Prop. (Lemmas C.2 and C.4)

$[\![[e/u]\sigma']\!] = [[\![e]\!]^*_\diamond/u][\![\sigma']\!] = [e/u][\![\sigma']\!]$
by Sub. Shift and DeShift Prop. (Lemmas C.2 and C.4)

$[e/u]([w_1/u'']\![\sigma']\!])$ is not undefined.
by Not Undefined Prop. (Lemma C.7)

$[e/u]([w_1/u'']\![\sigma']\!]) = [w_1/u''](e/u][\![\sigma']\!])$
by Def. of Sub. and Sub. Distribute Prop. (Lemma C.6)

$[w_1/u''](e/u][\![\sigma']\!]) = [w_1/u''][\![e/u]\sigma']\!]$
by Sub. Equality Prop. C.3

$[e/u]([w_1/u'']\![\sigma']\!]) = [w_1/u''](\![[e/u]\sigma']\!]$
by Above

$\mathcal{F}_1 :: \Omega_1, [e/u]\Omega_2, w_1 \in [\![[e/u]\tau']\!], w_2 \in [w_1/u'']\![[e/u]\sigma']\!]$
$\vdash [e/u]f \in [e/u]\varrho$
by IH on $\mathcal{D}$ and $\mathcal{E}_1$

$\Omega_1, [e/u]\Omega_2 \vdash \text{let } (w_1, w_2) = u' \text{ in } ([e/u]f) \in [e/u]\varrho$
by $\Sigma\mathsf{L}$

□

D Appendix: Twelf Proofs

D.1 sources.cfg

```
num.elf
formulas.elf
seq.elf
lfProps.elf
seqProps.elf
shiftSeq.elf
cutAdmis.elf
natDed.elf
natToSeq.elf
seqToNat.elf
examples.elf
```

D.2 num.elf

```
% Adam Poswolsky
% Properties of numbers
% Here we encode natural numbers and prove some properties.


% Representation of Natural Numbers
% We use X,Y to stand for natural numbers.
nat : type.                       %name nat (X Y) (x y).
z : nat.
s : nat -> nat.

% Encoding of less than or equal to relation.
le : nat -> nat -> type.  %name le L.
%mode le +X +Y.
base : le X X.
one  : le X Y -> le X (s Y).


leAddOneRight : le X Y -> le X (s Y) -> type.
%mode leAddOneRight +L -L'.
leAddOneRightCase : leAddOneRight L (one L).
%worlds () (leAddOneRight _ _).
%total {L} (leAddOneRight L _).


leRemoveOneLeft : le (s X) Y -> le X Y -> type.
%mode leRemoveOneLeft +L -L'.
leRemoveOneLeftBase : leRemoveOneLeft base (one base).
leRemoveOneLeftInd : leRemoveOneLeft (one L) (one L')
       <- leRemoveOneLeft L L'.
%worlds () (leRemoveOneLeft _ _).
%total {L} (leRemoveOneLeft L _).


leAddOneBoth : le X Y -> le (s X) (s Y) -> type.
%mode leAddOneBoth +L -L'.
leAddOneBoth_base: leAddOneBoth base base.
leAddOneBoth_ind: leAddOneBoth (one L) (one L')
                  <- leAddOneBoth L L'.
%worlds () (leAddOneBoth _ _).
%total {L} (leAddOneBoth L _).

leRemoveOneBoth : le (s X) (s Y) -> le X Y -> type.
%mode leRemoveOneBoth +L -L'.
leRemoveOneBothBase : leRemoveOneBoth base base.
leRemoveOneBothInd : leRemoveOneBoth (one L) L'
      <- leRemoveOneLeft L L'.
%worlds () (leRemoveOneBoth _ _).
%total {L} (leRemoveOneBoth L _).


leTrans: le X1 X2 -> le X2 X3 -> le X1 X3 -> type.
%mode leTrans +L1 +L2 -L3.
leTransBase : leTrans L base L.
leTransInd : leTrans L1 (one L2) (one L3)
       <- leTrans L1 L2 L3.
%worlds () (leTrans _ _ _).
%total {L1 L2} (leTrans L1 L2 _).


%
% Show it is impossible for le (s X) X
%
emptyType : type.            %name emptyType EE.

impossibleLessNum : {X} le (s X) X -> emptyType -> type.
%mode +{X:nat} +{L:le (s X) X} -{EE:emptyType}
   (impossibleLessNum X L EE).

impossibleLessNumOne : impossibleLessNum (s X) (one L) EE
       <- leRemoveOneLeft L L'
       <- impossibleLessNum X L' EE.

%worlds () (impossibleLessNum _ _ _).
```

```
%terminates {X} (impossibleLessNum X _ _).
%total {X} (impossibleLessNum X _ _).
```

D.3 formulas.elf

```
% Adam Poswolsky
% Encoding of our language
% And Properties on Formulas themselves.


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Representation-Level
% A,B ::= a (type constant) | A arrow B
% Note:  We will use M,N for the objects.
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lftp : type.                              %name  lftp (A B) (a b).
arrow : lftp -> lftp -> lftp.             %infix right 10 arrow.
% We include a block to represent type constants
% that may exist.
%block lftypeConsBlock : block {A:lftp}.



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Meta-Level
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Formulas T,S ::= top, bot, T imp S, past T, inj A
% Note:  We will use D,E for the objects.
%
% However, for our encoding we make a distinction
% between those types starting with "past".  All
% proofs will be over all formulas, "o", but this
% distinction helps because we often want to distinguish
% between it being "past" or anything else.
%
% The reason for this is that our system requires
% that changes are made to the context.  Namely, the
% number of pasts in front of everything in the context
% can go up or down.  The trick we will use to capture this
% (see seq.elf) takes advantage of this restriction.
%
%

o : type. %name o (T S) (t s).
pureO : type.                             %name pureO (TP SP) (tp sp).

top : pureO.
bot : pureO.
imp : o -> o -> pureO.                    %infix  right 10 imp.
inj : lftp -> pureO.
pair : o -> o -> pureO.

! : pureO -> o.
past: o -> o.


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Properties
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
% stripPast +X1 +T1 -X2 -T2
% This will strip away all pasts in front of T1 yielding T2.
% And X2 = X1 - (number of leading pasts of T1)
%
stripPast : nat -> o -> nat -> pureO -> type.    %name stripPast SP.
stripPastPure : stripPast X (! TP) X TP.
stripPastPast : stripPast X1 T1 X2 TP -> stripPast (s X1) (past T1) X2 TP.


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Some Props of stripPast
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

stripToLess : stripPast X1 T1 X2 TP -> le X2 X1 -> type.
%mode stripToLess +SP -L.
stripToLessPure : stripToLess stripPastPure base.
stripToLessPast : stripToLess (stripPastPast SP) L'
```

```
   <- stripToLess SP L
   <- leAddOneRight L L'.
%worlds (lftypeConsBlock) (stripToLess _ _).
%total {SP} (stripToLess SP _).


stripAddOne : stripPast X1 T1 X2 TP -> stripPast (s X1) T1 (s X2) TP -> type.
%mode stripAddOne +SP -SP'.
stripAddOnePure : stripAddOne stripPastPure stripPastPure.
stripAddOnePast : stripAddOne (stripPastPast SP) (stripPastPast SP')
   <- stripAddOne SP SP'.
%worlds (lftypeConsBlock) (stripAddOne _ _).
%total {SP} (stripAddOne SP _).
```

D.4 seq.elf

```
% Adam Poswolsky
% Encoding of Sequent Calculus version

%{
IMPORTANT:  How to read the rules:

Our system needs to be able to add or subtract "past"
from all elements in the context.  In order to encode this
we design our judgment with an index N such that if N goes up
it is interpreted as adding a past to everything in the context.

Therefore, when reading the rules one must keep in mind that:

% !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
% !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
hyp X T -> conc Y S
    is meant to be interpreted as:

(past ... past T)  |---  S
where the number of leading pasts to T is (Y - X)
% !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
% !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Notice that this also means that
    hyp X T -> conc Y S
is the same as
    hyp (X+1) (past T) -> conc Y S

In other encodings of this sytem we allowed this
and then had an "equiv" relation to tell when they
are equal.  However, that introduced a lot of overhead
(see old directory).

Instead, here we disallow the second representation.
This explains us distinguishing between types
that do not start with a past (pure0) and those that
can be anything (o).  "hyp" allows contain a
type without any leading pasts (so the fact that there
were leading pasts is captures in the index).

}%

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Context (explained above)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
hyp : nat -> pure0 -> type.         %name hyp H (h sh).
%block hypBlock : some {X:nat}{TP:pure0} block {h:hyp X TP}.



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Representation-Level
% M,N ::= c (object constant) | lam x:A.M | app M N
%
% The "casting" operation is captures by castHyp with
% shiftLF
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

exp : nat -> lftp -> type.           %name exp (M N) (m n).
lam : (exp X A -> exp X B) -> exp X (A arrow B).
app : exp X (A arrow B)-> exp X A -> exp X B.
shiftLF : exp X A -> exp (s X) A.
castHyp : hyp X (inj A) -> exp X A.
%
% %%%%%%%%%%%%%%%%%
% The lfexpBlock can also be thought of
% as representing the constants "c" that
% exist in the signature.
% %%%%%%%%%%%%%%%%%
%block lfexpBlock : some {X:nat}{A:lftp} block {m:exp X A}.



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Meta-Level
```

```
% (actual rule is placed in comments before encoded rule)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
conc : nat -> o -> type.             %name conc (D E F) (d e f).


%{ AXIOM
T is in Omega
--------------- axiom, where |T| removes leading pasts of T
Omega |-- |T|
}%
axiom : le X Y -> hyp X TP -> conc Y (! TP).


%{ IMPR
Omega, T |-- S
-------------------- impr
Omega |-- T imp S
}%
impr : stripPast X T X' TP -> (hyp X' TP -> conc X S) -> conc X (! (T imp S)).


%{ IMPL
(past...past)(T1 imp T2) in Omega,
Omega |-- T1   Omega,|T2| |--- S
---------------------------------- impl, where |T2| removes leading pasts of T2
Omega |-- S
}%
impl : le X Y -> hyp X (T1 imp T2) -> conc Y T1
       -> stripPast Y T2 _ TP2 -> (hyp Y TP2 -> conc Y S) -> conc Y S.


%{ TOP

--------------- topr
Omega |--- top
}%
topr : conc _ (! top).


%{ BOTL

------------------------------------ botl, where |T| removes all leadings pasts to T
Omega,(past...past)bot,Omega' |--- |T|
}%
botl : le X Y -> hyp X bot -> stripPast Y T _ TP -> conc Y (! TP).


%{ PASTR
(remove past from Omega) |-- T
------------------------------------ pastr
              Omega      |-- past T

}%
pastr : conc X T -> conc (s X) (past T).


%{ INJR
(cast Omega to LF) |-- A
------------------------------------ injr
             Omega |-- inj A

}%
injr : exp X A -> conc X (! (inj A)).


%{ PAIRR
Omega |-- T1          Omega |-- T2
---------------------------------- pairr
      Omega |-- pair T1 T2
}%
pairr : conc X T1 -> conc X T2 -> conc X (! (pair T1 T2)).

%{ PAIRL
(past...past)(pair T1 T2) in Omega,
Omega, |T1|, |T2| |--- S
---------------------------------- pairl, where |T*| removes all leadings pasts to T*
```

```
Omega |-- S
}%
pairl : le X Y  -> hyp X (pair T1 T2) -> stripPast Y T1 _ TP1 -> stripPast Y T2 _ TP2
                -> (hyp Y TP1 -> hyp Y TP2 -> conc Y S)
-> conc Y S.
```

D.5 lfProps.elf

```
% Adam Poswolsky
% Properties of the Representation-Level (LF) of our system.


% This is used when we encounter a case that is impossible
fromEmptyComesAllLF : emptyType -> exp X A -> type.
%mode +{X:nat} +{A:lftp} +{EE:emptyType} -{M:exp X A} (fromEmptyComesAllLF EE M).
%worlds (hypBlock | lfexpBlock | lftypeConsBlock) (fromEmptyComesAllLF _ _).
%total {EE} (fromEmptyComesAllLF EE _).


%
% There is no "past" on the LF level itself, so we have more freedom
% with the index.  Namely if (exp Y A -> exp X B) and Y <= X, then
% we can change it to (exp Y' A -> exp X B) where Y' can be anything <= X.
%
shiftNegExpLF : le (s Y) X -> (exp Y A -> exp X B) -> (exp (s Y) A -> exp X B) -> type.
%mode shiftNegExpLF +L +M -M2.
shiftNegExpLFLam : shiftNegExpLF L ([n] lam (M n)) ([n] lam (M' n))
   <- ({m} shiftNegExpLF L ([n] M n m) ([n] M' n m)).
shiftNegExpLFApp : shiftNegExpLF L ([n] app (M1 n) (M2 n)) ([n] app (M1' n) (M2' n))
   <- shiftNegExpLF L M1 M1'
   <- shiftNegExpLF L M2 M2'.
shiftNegExpLFShift1 : shiftNegExpLF base ([n] shiftLF (M n)) ([n] app (shiftLF (lam M)) n).

shiftNegExpLFShift2 : shiftNegExpLF (one L) ([n] shiftLF (M n)) ([n] shiftLF (M' n))
    <- shiftNegExpLF L M M'.

shiftNegExpLFCast1 : shiftNegExpLF L ([n] castHyp H) ([n] castHyp H).
shiftNegExpLFBlockCase : shiftNegExpLF L ([_] M) ([_] M).
shiftNegExpLFidentity : shiftNegExpLF L ([n] n) ([n] M n)
     <- impossibleLessNum _ L EE
     <- ({m} fromEmptyComesAllLF EE (M m)).

%worlds ( hypBlock | lfexpBlock | lftypeConsBlock) (shiftNegExpLF _ _ _).
%total {L M} (shiftNegExpLF L M _).


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is an important judgment which says that if we have a function
% from (hyp Y (inj A) -> exp X B) then we can create (exp Y A -> exp X B)
%
% Note that the reverse direction is trivial (just use castHyp)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
convertHypExp : (hyp Y (inj A) -> exp X B) -> (exp Y A -> exp X B) -> type.
%mode convertHypExp +M -M2.
convertHypExpLam : convertHypExp ([h] lam (M h)) ([n] lam (M' n))
   <- ({m} convertHypExp ([h] M h m) ([n] M' n m)).
convertHypExpApp : convertHypExp ([h] app (M1 h) (M2 h)) ([n] app (M1' n) (M2' n))
   <- convertHypExp M1 M1'
   <- convertHypExp M2 M2'.
convertHypExpShift : convertHypExp ([h] shiftLF (M h)) ([n] shiftLF (M' n))
      <- convertHypExp M M'.

convertHypExpCast1 : convertHypExp ([_] castHyp H) ([_] castHyp H).
convertHypExpCast2 : convertHypExp ([h] castHyp h) ([n] n).

convertHypExpBlockCase : convertHypExp ([_] M) ([_] M).
%worlds ( hypBlock | lfexpBlock | lftypeConsBlock) (convertHypExp _ _).
%total {M} (convertHypExp M _).



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Strengthening (#1)
%
%   if (hyp Y TP -> exp X A) and TP is not an Inj, then the function
%       cannot use its argument.
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
notInj : pureO -> type.              %name notInj NI ni.
notInjImp : notInj (T1 imp T2).
notInjPair : notInj (pair T1 T2).
notInjTop : notInj top.
notInjBot : notInj bot.
```

```
strengthenLFNI : notInj TP -> (hyp Y TP -> exp X A) -> exp X A -> type.
%mode strengthenLFNI +NI +M -M2.
strengthenLFNILam : strengthenLFNI NI ([h] lam (N h)) (lam N')
   <- ({m} strengthenLFNI NI ([h] N h m) (N' m)).
strengthenLFNIApp : strengthenLFNI NI ([h] app (N1 h) (N2 h)) (app N1' N2')
   <- strengthenLFNI NI N1 N1'
   <- strengthenLFNI NI N2 N2'.
strengthenLFNIShift : strengthenLFNI NI ([h] shiftLF (N h)) (shiftLF N')
     <- strengthenLFNI NI N N'.
strengthenLFNICast : strengthenLFNI NI ([h] castHyp H) (castHyp H).
strengthenLFNIBlockCase : strengthenLFNI NI ([h] N) N.
%worlds ( hypBlock | lfexpBlock | lftypeConsBlock) (strengthenLFNI _ _ _).
%total {N} (strengthenLFNI _ N _).



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Strengthening (#2)
%
%   If (hyp Y TP -> exp X A) and Y > X, then the function cannot
%       use its argument.
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

strengthenLF : le Y X -> (hyp (s X) TP -> exp Y B) -> exp Y B -> type.
%mode strengthenLF +L +M -M2.
strengthenLFLam : strengthenLF L ([h] lam (M h)) (lam M')
   <- ({m} strengthenLF L ([h] M h m) (M' m)).
strengthenLFApp : strengthenLF L ([h] app (M1 h) (M2 h)) (app M1' M2')
   <- strengthenLF L M1 M1'
   <- strengthenLF L M2 M2'.
strengthenLFShift : strengthenLF L ([h] shiftLF (M h)) (shiftLF M')
     <- leRemoveOneLeft L L'
     <- strengthenLF L' M M'.
strengthenLFCast1 : strengthenLF L ([h] castHyp H) (castHyp H).
strengthenLFCast2 : strengthenLF L ([h] castHyp h) M
     <- impossibleLessNum _ L EE
     <- fromEmptyComesAllLF EE M.
strengthenLFBlockCase : strengthenLF L ([h] M) M.
%worlds ( hypBlock | lfexpBlock | lftypeConsBlock) (strengthenLF _ _ _).
%total {M} (strengthenLF _ M _).

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Strengthening (#3)
%
%   If (exp Y B -> exp X A) and Y > X, then the function cannot
%       use its argument.
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
strengthenLF2 : le Y X -> (exp (s X) A -> exp Y B) -> exp Y B -> type.
%mode strengthenLF2 +L +M -M2.
strengthenLF2Lam : strengthenLF2 L ([n] lam (M n)) (lam M')
   <- ({m} strengthenLF2 L ([n] M n m) (M' m)).
strengthenLF2App : strengthenLF2 L ([n] app (M1 n) (M2 n)) (app M1' M2')
   <- strengthenLF2 L M1 M1'
   <- strengthenLF2 L M2 M2'.
strengthenLF2Shift : strengthenLF2 L ([n] shiftLF (M n)) (shiftLF M')
     <- leRemoveOneLeft L L'
     <- strengthenLF2 L' M M'.
strengthenLF2Identity : strengthenLF2 L ([n] n) M
 <- impossibleLessNum _ L EE
 <- fromEmptyComesAllLF EE M.
strengthenLF2BlockAndCast : strengthenLF2 L ([m] M) M.
%worlds ( hypBlock | lfexpBlock | lftypeConsBlock) (strengthenLF2 _ _ _).
%total {M} (strengthenLF2 _ M _).
```

D.6 seqProps.elf

```
% Adam Poswolsky
% Properties of our Meta-Level
% This includes reversing pastr (future rule),
% Weakening, and Strengthening.

% This is used when we encounter a case that is impossible
fromEmptyComesAll : emptyType -> conc X T -> type.
%mode +{X:nat} +{T:o} +{EE:emptyType} -{D:conc X T} (fromEmptyComesAll EE D).
%worlds (hypBlock | lfexpBlock | lftypeConsBlock) (fromEmptyComesAll _ _).
%total {EE} (fromEmptyComesAll EE _).


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Weakening... for example, if Omega,A |-- B then Omega,(past A) |-- B
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
weakenHypLF' : (hyp (s X) TP -> exp Y B) -> (hyp X TP -> exp Y B) -> type.
%mode weakenHypLF' +M1 -M2.
weakenHypLF'Lam : weakenHypLF' ([sh] lam (M sh)) ([h] lam (M' h))
   <- ({m} weakenHypLF' ([sh] M sh m) ([h] M' h m)).
weakenHypLF'App : weakenHypLF' ([sh] app (M1 sh) (M2 sh)) ([h] app (M1' h) (M2' h))
   <- weakenHypLF' M1 M1'
   <- weakenHypLF' M2 M2'.
weakenHypLF'Shift : weakenHypLF' ([sh] shiftLF (M sh)) ([h] shiftLF (M' h))
     <- weakenHypLF' M M'.
weakenHypLF'Cast1 : weakenHypLF' ([sh] castHyp H) ([h] castHyp H).
weakenHypLF'Cast2 : weakenHypLF' ([sh] castHyp sh) ([h] (shiftLF (castHyp h))).
weakenHypLF'BlockCase : weakenHypLF' ([sh] M) ([h] M).
%worlds (hypBlock | lfexpBlock | lftypeConsBlock) (weakenHypLF' _ _).
%total {M} (weakenHypLF' M _).


weakenHyp' : (hyp (s X) TP -> conc Y S) -> (hyp X TP -> conc Y S) -> type.
%mode weakenHyp' +F1 -F2.
weakenHyp'Axiom1 : weakenHyp' ([sh] axiom L H) ([h] axiom L H).
weakenHyp'Axiom2 : weakenHyp' ([sh] axiom L sh) ([h] axiom L' h)
    <- leRemoveOneLeft L L'.

weakenHyp'Impr : weakenHyp' ([sh] impr SP (D sh)) ([h] impr SP (D' h))
  <- ({h2} weakenHyp' ([sh] D sh h2) ([h] D' h h2)).

weakenHyp'Impl1 : weakenHyp' ([sh] impl L H (D1 sh) SP (D2 sh))
   ([h] impl L H (D1' h) SP (D2' h))
   <- weakenHyp' D1 D1'
   <- ({h2} weakenHyp' ([sh] D2 sh h2) ([h] D2' h h2)).

weakenHyp'Impl2 : weakenHyp' ([sh] impl L sh (D1 sh) SP (D2 sh))
   ([h] impl L' h (D1' h) SP (D2' h))
   <- weakenHyp' D1 D1'
   <- ({h2} weakenHyp' ([sh] D2 sh h2) ([h] D2' h h2))
   <- leRemoveOneLeft L L'.

weakenHyp'Top : weakenHyp' ([sh] topr) ([h] topr).
weakenHyp'Botl1 : weakenHyp' ([sh] botl L H SP) ([h] botl L H SP).
weakenHyp'Botl2 : weakenHyp' ([sh] botl L sh SP) ([h] botl L' h SP)
   <- leRemoveOneLeft L L'.

weakenHyp'Pastr : weakenHyp' ([sh] pastr (D sh)) ([h] pastr (D' h))
   <- weakenHyp' D D'.

weakenHyp'Injr : weakenHyp' ([sh] injr (M sh)) ([h] injr (M' h))
  <- weakenHypLF' M M'.

weakenHyp'Pairr : weakenHyp' ([sh] pairr (D1 sh) (D2 sh)) ([h] pairr (D1' h) (D2' h))
   <- weakenHyp' D1 D1'
   <- weakenHyp' D2 D2'.

weakenHyp'Pairl1 : weakenHyp' ([sh] pairl L H SP1 SP2 (D sh))
   ([h] pairl L H SP1 SP2 (D' h))
   <- ({h2}{h3} weakenHyp' ([sh] D sh h2 h3) ([h] D' h h2 h3)).

weakenHyp'Pairl2 : weakenHyp' ([sh] pairl L sh SP1 SP2 (D sh))
   ([h] pairl L' h SP1 SP2 (D' h))
   <- ({h2}{h3} weakenHyp' ([sh] D sh h2 h3) ([h] D' h h2 h3))
   <- leRemoveOneLeft L L'.
```

```
%worlds (hypBlock | lfexpBlock | lftypeConsBlock) (weakenHyp' _ _).
%total {F} (weakenHyp' F _).


weakenHyp : le X2 X1 -> (hyp X1 TP -> conc X* S*) -> (hyp X2 TP -> conc X* S*) -> type.
%mode weakenHyp +L +E1 -E2.
weakenHypBase : weakenHyp base E1 E1.
weakenHypOne : weakenHyp (one L) E1 E2
 <- weakenHyp' E1 E1'
 <- weakenHyp L E1' E2.
%worlds (hypBlock | lfexpBlock | lftypeConsBlock) (weakenHyp _ _ _).
%total {L} (weakenHyp L _ _).


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Strengthening... (Part 1 of 2)
%           if (hyp X TP -> conc Y T)  and X > Y, then
%            the function doesn't use its argument.
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
strengthen : le Y X -> (hyp (s X) TP -> conc Y T) -> conc Y T -> type.
%mode strengthen +L +E -D.
strengthenAxiom1 : strengthen L ([h] axiom L' H) (axiom L' H).
strengthenAxiom2 : strengthen L ([h] axiom L' h) D
    <- leAddOneBoth L L1
    <- leTrans L1 L' L2
    <- impossibleLessNum _ L2 EE
    <- fromEmptyComesAll EE D.

strengthenImpr  : strengthen L ([h] impr SP (D h)) (impr SP D')
   <- ({h'} strengthen L ([h] D h h') (D' h')).

strengthenImpl1  : strengthen L ([h] impl L' H (D1 h) SP (D2 h)) (impl L' H D1' SP D2')
   <- strengthen L D1 D1'
   <- ({h'} strengthen L ([h] D2 h h') (D2' h')).
strengthenImpl2  : strengthen L ([h] impl L' h (D1 h) SP (D2 h)) D
    <- leAddOneBoth L L1
    <- leTrans L1 L' L2
    <- impossibleLessNum _ L2 EE
    <- fromEmptyComesAll EE D.

strengthenTopr : strengthen L ([h] topr) (topr).
strengthenBotl1 : strengthen L ([h] botl L' H SP) (botl L' H SP).
strengthenBotl2 : strengthen L ([h] botl L' h SP) D
    <- leAddOneBoth L L1
    <- leTrans L1 L' L2
    <- impossibleLessNum _ L2 EE
    <- fromEmptyComesAll EE D.

strengthenPastr : strengthen L ([h] pastr (D h)) (pastr D')
   <- leRemoveOneLeft L L'
   <- strengthen L' D D'.

strengthenInjr : strengthen L ([h] injr (M h)) (injr M')
  <- strengthenLF L M M'.

strengthenPairr : strengthen L ([h] pairr (D1 h) (D2 h)) (pairr D1' D2')
   <- strengthen L D1 D1'
   <- strengthen L D2 D2'.

strengthenPairl1  : strengthen L ([h] pairl L' H SP1 SP2 (D h)) (pairl L' H SP1 SP2 D')
     <- ({h'}{h''} strengthen L ([h] D h h' h'') (D' h' h'')).
strengthenPairl2  : strengthen L ([h] pairl L' h SP1 SP2 (D h)) D'
     <- leAddOneBoth L L1
     <- leTrans L1 L' L2
     <- impossibleLessNum _ L2 EE
     <- fromEmptyComesAll EE D'.

%worlds ( hypBlock | lfexpBlock | lftypeConsBlock) (strengthen _ _ _).
% reduces won't work here ...   %reduces D' <= D (strengthen _ D D').
%total {D} (strengthen _ D _).


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Important Admissible Rule:
% We now define the future rule (reverse of pastr).
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
futureRule : conc (s X) (past T) -> conc X T -> type.
%mode futureRule +D -D'.

futureRulePastr : futureRule (pastr D) D.

futureRuleImpl : futureRule (impl L H D1 SP D2) E
  <- ({h} futureRule (D2 h) (D2' h))
  <- strengthen base D2' E.

futureRulePairl : futureRule (pairl L H SP1 SP2 D) E2
   <- ({h}{h2} futureRule (D h h2) (D' h h2))
   <- ({h2} strengthen base ([h] D' h h2) (E1 h2))
   <- strengthen base E1 E2.

%worlds (hypBlock | lfexpBlock | lftypeConsBlock)
        (futureRule _ _).
%covers (futureRule  +D -D').
%terminates {D} (futureRule  D _).
%total {D} (futureRule D _).



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Strengthening... (Part 2 of 2)
%           if Omega,A |-- (past B)
%                  and A is not (past A*), then Omega |-- (past B)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


impossibleConc : conc z (past T) -> emptyType -> type.
impossibleConcImpl : impossibleConc (impl L H D1 SP D2) EE
      <- ({h} impossibleConc (D2 h) EE).
impossibleConcPairl : impossibleConc (pairl L H SP1 SP2 D) EE
       <- ({h}{h2} impossibleConc (D h h2) EE).
%mode impossibleConc +D -EE.
%worlds (hypBlock | lfexpBlock | lftypeConsBlock)
        (impossibleConc _ _).
%total {D} (impossibleConc D _).


strengthenP : (hyp X _ -> conc X (past T)) -> conc X (past T) -> type.
%mode strengthenP  +D -D'.
strengthenPcase : strengthenP D (pastr E')
   <- ({h} futureRule (D h) (D' h))
   <- strengthen base D' E'.

strengthenPimpossible : strengthenP D E
 <- ({h} impossibleConc (D h) EE)
 <- fromEmptyComesAll EE E.

%worlds(hypBlock | lfexpBlock | lftypeConsBlock) (strengthenP _ _).
%covers (strengthenP +D -D').
%total {D} (strengthenP D _).


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% equivalent Conclusion
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
equivConc : conc X T -> stripPast X T Y TP -> conc Y (! TP) -> type.
%mode equivConc +D1 +SP -E.
equivConcSPbase : equivConc E SP E.
equivConcSPpast : equivConc D1 (stripPastPast SP) E
   <- futureRule D1 D1'
   <- equivConc D1' SP E.
%worlds (hypBlock | lfexpBlock | lftypeConsBlock) (equivConc _ _ _).
%total {SP} (equivConc _ SP _).
```

D.7 shiftSeq.elf

```
% Adam Poswolsky
% Encoding of Shift Property in Sequent Calculus


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Shifting:
%   Namely, if Omega |-- (past T) then Omega |-- T
%     and/or if Omega |-- T  then  (past Omega) |-- T
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% This captures the property that if we can prove something
% in the past, then you can also prove it is true in the present.
% There are a lot of "messy" details in this proof
% but is easily proved on paper.

% This proof proceeds in three steps.
% First, we create conc' and show that we can copy from conc' to conc.
% Second, we shift fron conc X T to conc' (s X) T.
% Finally, we put the first two together to go from conc X T to conc (s X) T.


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% STEP 1 :  Copy from a conc' to conc
% Note that this system is identical except we added !! (to shift hyps)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

hyp' : nat -> pureO -> type.         %name hyp' H' h'.
!! : hyp' X TP -> hyp' (s X) TP.

conc' : nat -> o -> type.    %name conc' (D' E') (d' e').
exp' : nat -> lftp -> type.           %name exp' (M' N') (m' n').


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% LF Level
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lam' : (exp' X A -> exp' X B) -> exp' X (A arrow B).
app' : exp' X (A arrow B)-> exp' X A -> exp' X B.
shiftLF' : exp' X A -> exp' (s X) A.
castHyp' : hyp' X (inj A) -> exp' X A.


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Meta Level
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

axiom' : le X Y -> hyp' X TP -> conc' Y (! TP).
impr' : stripPast X T X' TP -> (hyp' X' TP -> conc' X S)
        -> conc' X (! (T imp S)).
impl' : le X Y -> hyp' X (T1 imp T2) -> conc' Y T1
        -> stripPast Y T2 _ TP2 -> (hyp' Y TP2 -> conc' Y S) -> conc' Y S.
topr' : conc' _ (! top).
botl' : le X Y -> hyp' X bot -> stripPast Y T _ TP -> conc' Y (! TP).
pastr' : conc' X T -> conc' (s X) (past T).
injr' : exp' X A -> conc' X (! (inj A)).
pairr' : conc' X T1 -> conc' X T2 -> conc' X (! (pair T1 T2)).
pairl' : le X Y -> hyp' X (pair T1 T2) -> stripPast Y T1 _ TP1 ->
 stripPast Y T2 _ TP2 -> (hyp' Y TP1 -> hyp' Y TP2 -> conc' Y S)
 -> conc' Y S.

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Copying from conc' to conc
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
copyLF' : exp' X A -> exp X A -> type.      %name copyLF' C c.
%mode copyLF' +M' -M.
copyC' : conc' X T -> conc X T -> type.     %name copyC' C c.
%mode copyC' +D' -D.
copyH' : hyp' Y TP ->  le X Y -> hyp X TP -> type.   %name copyH' C c.
%mode copyH' +H' -L -H.


%block lfcopyBlock : some {Y:nat}{A:lftp} block {m:exp Y A}{n:exp' Y A}{m:copyLF' n m}.
%block copyBlock : some {Y:nat}{TP:pureO}
                   block {h:hyp Y TP} {h':hyp' Y TP} {m: copyH' h' base h}.
```

```
shiftLFGeneral : le X Y -> exp X A -> exp Y A -> type.
%mode shiftLFGeneral +L +M -M'.
shiftLFGeneralBase : shiftLFGeneral base M M.
shiftLFGeneralInd : shiftLFGeneral (one L) M (shiftLF M')
     <- shiftLFGeneral L M M'.
%worlds (copyBlock | lfcopyBlock | lftypeConsBlock )   (shiftLFGeneral _ _ _).
%total {L} (shiftLFGeneral L _ _).


copy-!!'  :copyH' (!! H') L' H
    <- copyH' H' L H
    <- leAddOneRight L L'.


copyLF'lam : copyLF' (lam' M') (lam M)
      <- ({m}{m'} copyLF' m' m -> copyLF' (M' m') (M m)).
copyLF'app : copyLF' (app' M1' M2') (app M1 M2)
      <- copyLF' M1' M1
      <- copyLF' M2' M2.
copyLF'shiftLF : copyLF' (shiftLF' M') (shiftLF M)
  <- copyLF' M' M.

copyLF'castHyp : copyLF' (castHyp' H') M
  <- copyH' H' L H
  <- shiftLFGeneral L (castHyp H) M.

copyC'_axiom : copyC' (axiom' L1 H') (axiom L3 H)
<- copyH' H' L2 H
<- leTrans L2 L1 L3.

copyC'_impr : copyC' (impr' SP D') (impr SP D)
       <- ({h}{h'} copyH' h' base h
     -> copyC' (D' h') (D h)).

copy-impl': copyC' (impl' L1 H' D1' SP D2') (impl L3 H D1 SP D2)
    <- copyC' D1' D1
     <- ({h} {h'}  copyH' h' base h
  -> copyC' (D2' h') (D2 h))
    <- copyH' H' L2 H
    <- leTrans L2 L1 L3.


copyC'_topr : copyC' topr' topr.

copy-botl' : copyC' (botl' L1 H' SP) (botl L3 H SP)
             <- copyH' H' L2 H
     <- leTrans L2 L1 L3.

copy-pastr' : copyC' (pastr' D') (pastr D)
       <- copyC' D' D.

copy-injr' : copyC' (injr' M') (injr M)
      <- copyLF' M' M.

copy-pairr' : copyC' (pairr' D1' D2') (pairr D1 D2)
       <- copyC' D1' D1
       <- copyC' D2' D2.

copy-pairl': copyC' (pairl' L1 H' SP1 SP2 D') (pairl L3 H SP1 SP2 D)
     <- ({h} {h'}  copyH' h' base h
  -> ({h2} {h2'}  copyH' h2' base h2
-> copyC' (D' h2' h') (D h2 h)))
     <- copyH' H' L2 H
     <- leTrans L2 L1 L3.


%worlds (copyBlock | lfcopyBlock | lftypeConsBlock )
        (copyC' _ _) (copyH' _ _ _) (copyLF' _ _).
%terminates (D' H' M') (copyC' D'  _) (copyH' H' _ _) (copyLF' M' _).
%covers (copyC' +D' -D) (copyH' +H' -L -H) (copyLF' +M' -M).
%total (D' H' M') (copyC' D'  _) (copyH' H' _ _) (copyLF' M' _).


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% STEP 2
% Shifting conc Y T to conc' (s Y) T
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map' : hyp X TP -> le X Y -> hyp' (s Y) TP -> type. %name map' W w.
%mode map' +H +L -H'.

calcShiftLF' : exp Y A -> exp' (s Y) A -> type.    %name calcShiftLF' S s.
%mode calcShiftLF' +M -M'.

shift' : conc Y T -> conc' (s Y) T -> type.        %name shift' S s.
%mode shift' +D -D'.


%block lfmapBlock : some {Y:nat}{A:lftp}
                    block {m:exp Y A}{n:exp' (s Y) A}{s:calcShiftLF' m n}.
%block mapBlock : some {Y:nat}{TP:pureO}
  block {h:hyp Y TP}{h':hyp' (s Y) TP}{s:map' h base h'}.
%block lfmapcopyBlock : some {Y:nat}{A:lftp}
                        block {m:exp Y A}{n:exp' Y A}{c:copyLF' n m}{s:calcShiftLF' m (shiftLF' n)}.
%block mapcopyBlock : some {Y:nat}{TP:pureO}
                      block {h:hyp Y TP}{h':hyp' Y TP}{c:copyH' h' base h}{s:map' h base (!! h')}.


calcShiftLF'lam : calcShiftLF' (lam M) (lam' M')
   <- ({n}{n'} calcShiftLF' n n' -> calcShiftLF' (M n) (M' n')).

calcShiftLF'app : calcShiftLF' (app M1 M2) (app' M1' M2')
   <- calcShiftLF' M1 M1'
   <- calcShiftLF' M2 M2'.

calcShiftLF'shiftLF : calcShiftLF' (shiftLF M1) (shiftLF' M1')
       <- calcShiftLF' M1 M1'.

calcShiftLF'castHyp : calcShiftLF' (castHyp H) (castHyp' H')
       <- map' H base H'.


map'_ind : map' H (one L) (!! H')
    <- map' H L H'.


shift'-axiom: shift' (axiom L H) (axiom' L' H')
    <- map' H base H'
    <- leAddOneBoth L L'.


shift'-impr: shift' (impr SP D) (impr' SP' D')
   <- ({h} {h'}  map' h base h'
 -> shift' (D h) (D' h'))
   <- stripAddOne SP SP'.


shift'-impl: shift' (impl L H D1 SP D2) (impl' L' H' D1' SP' D2')
      <- shift' D1 D1'
      <- ({h} {h'} map' h base h'
    -> shift' (D2 h) (D2' h'))
      <- map' H base H'
      <- leAddOneBoth L L'
      <- stripAddOne SP SP'.


shift'-topr : shift' (topr) (topr').

shift'-botl : shift' (botl L H SP) (botl' L' H' SP')
       <- map' H base  H'
       <- leAddOneBoth L L'
       <- stripAddOne SP SP'.

shift'-pastr : shift' (pastr D) (pastr' D')
<- shift' D D'.


shift'-injr : shift' (injr M) (injr' M')
       <- calcShiftLF' M M'.
```

```
shift'-pairr : shift' (pairr D1 D2) (pairr' D1' D2')
<- shift' D1 D1'
<- shift' D2 D2'.


shift'-pairl: shift' (pairl L H SP1 SP2 D) (pairl' L' H' SP1' SP2' D')
       <- ({h} {h'} map' h base h'
     -> ({h2} {h2'} map' h2 base h2'
   -> shift' (D h2 h) (D' h2' h')))
       <- map' H base H'
       <- leAddOneBoth L L'
       <- stripAddOne SP1 SP1'
       <- stripAddOne SP2 SP2'.


%worlds (mapBlock | mapcopyBlock | lfmapBlock | lfmapcopyBlock | lftypeConsBlock )
        (shift' _ _ ) (map' _ _ _) (calcShiftLF' _ _).
%terminates {H L} (map' H L _).
%total {H L} (map' H L _).
%terminates {M} (calcShiftLF' M _).
%total {M} (calcShiftLF' M _).
%terminates {D} (shift' D _).
%total {D} (shift' D _).


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Final Shift Stage :  Put it together to get from conc Y A to conc (s Y) A
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

shift : conc Y T -> conc (s Y) T -> type.
%mode shift +D -D2.
shiftCase : shift D D2
     <- shift' D  D'
     <- copyC' D' D2.
%worlds (mapcopyBlock | lfmapcopyBlock | lftypeConsBlock) (shift _ _).
%terminates {D} (shift D _) .
%total {D} (shift D _).



shiftPast : conc Y (past T) -> conc Y T -> type.
%mode shiftPast +D -D2.

shiftPastImpl : shiftPast (impl L H D1 SP D2) (impl L H D1 SP D2*)
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') ->
shiftPast (D2 h) (D2* h)).

shiftPastPairl : shiftPast (pairl L H SP1 SP2 D) (pairl L H SP1 SP2 D*)
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h')
               -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
    shiftPast (D h2 h) (D* h2 h))).

shiftPastPastr : shiftPast (pastr D) E
  <- shift D E.

%worlds (mapcopyBlock | lfmapcopyBlock | lftypeConsBlock) (shiftPast _ _).
%terminates {D} (shiftPast D _) .
%covers (shiftPast +D -D2).
%total {D} (shiftPast D _).



shiftGeneral : le X Y -> conc X T -> conc Y T -> type.
%mode shiftGeneral +L +D -D2.
shiftGeneralBase : shiftGeneral base D D.
shiftGeneralInd : shiftGeneral (one L) D E
   <- shiftGeneral L D D2
   <- shift D2 E.
%worlds(mapcopyBlock | lfmapcopyBlock | lftypeConsBlock ) (shiftGeneral _ _ _).
%total {L} (shiftGeneral L _ _).
```

D.8 cutAdmis.elf

```
% Adam Poswolsky
% Encoding of Admissibility of Cut


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Admissibility of Cut:
%   Namely, if Omega |-- T and Omega, T |-- S  then Omega |-- S
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


ca : {T}  conc X T -> stripPast X T Y TP -> (hyp Y TP -> conc X S) -> conc X S -> type.
%mode (ca +T +D +SP +E -F).

% %%%%%%%%%%%%%%%%%%%%%%
% Essential Conversions
% %%%%%%%%%%%%%%%%%%%%%%

ca_axiom_r : ca T D SP ([h] axiom L h) F
      <- equivConc D SP D'
      <- shiftGeneral L D' F.

ca_impl_r : ca (!(T imp S)) (impr SP' D) stripPastPure ([h] impl L h (E1 h) SP* (E2 h)) F3
     <- ca (! (T imp S)) (impr SP' D) stripPastPure E1 E1'
     <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
   ca (! (T imp S)) (impr SP' D) stripPastPure ([h] E2 h h2) (E2' h2))
     <- ca T E1' SP' D F1
     <- stripToLess SP* L*
     <- weakenHyp L* E2' F2
     <- ca S F1 SP* F2 F3.

ca_pastr_l_pastr_r : ca (past T) (pastr D) (stripPastPast SP) ([h] pastr (E h)) (pastr F)
          <- ca T D SP E F.

ca_injr_r : ca (! (inj A)) (injr N) stripPastPure ([h] injr (M h)) (injr (M' N))
     <- convertHypExp M M'.

ca_pairl_r : ca (!(pair T S)) (pairr D1 D2) stripPastPure ([h] pairl L h SP1 SP2 (E h)) F*
      <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2')
   -> ({h3}{h3'} copyH' h3' base h3 -> map' h3 base (!! h3') ->
       ca (! (pair T S)) (pairr D1 D2) stripPastPure ([h] E h h2 h3) (E' h2 h3)))
      <- stripToLess SP1 L1
      <- ({h2} weakenHyp L1 ([h] E' h h2) ([h] F1 h h2))
      <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2')
    -> ca T D1 SP1 ([h] F1 h h2) (F2 h2))
      <- stripToLess SP2 L2
      <- weakenHyp L2 F2 F3
      <- ca S D2 SP2 F3 F*.


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Shift on left.
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% If we have pastr on the left but not pastr on the right,
% then we shift the result on the left and call the IH.
%

ca_pastr_l1 : ca (past T) (pastr D) (stripPastPast SP) ([h] botl L h SP*) F
       <- shift D D'
       <- stripAddOne SP SP'
       <- stripToLess SP' L'
       <- ca T D' SP' ([h] botl L' h SP*) F.

ca_pastr_l2 : ca (past T) (pastr D) (stripPastPast SP) ([h] impl L h (E1 h) SP* (E2 h)) F
       <- ca (past T) (pastr D) (stripPastPast SP) E1 E1'
       <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
     ca (past T) (pastr D) (stripPastPast SP) ([h] E2 h h2) (E2' h2))
       <- shift D D'
       <- stripAddOne SP SP'
       <- stripToLess SP' L'
       <- ca T D' SP' ([h] impl L' h E1' SP* E2') F.


ca_pastr_l3a : ca (past T) (pastr D) (stripPastPast (SP : stripPast X1 T X2 (inj B))) ([h] injr (M h)) F
       <- shift D D'
```

```
       <- stripAddOne SP SP'
       <- stripToLess SP' L'
       <- convertHypExp M M'
       <- shiftNegExpLF L' M' M''
       <- ca T D' SP' ([h] injr (([x] M'' (castHyp x)) h)) F.

ca_pastr_l3b : ca (past T) (pastr D) (stripPastPast SP) ([h] injr (M h)) (injr N)
<- strengthenLFNI notInjTop M N.

ca_pastr_l3c : ca (past T) (pastr D) (stripPastPast SP) ([h] injr (M h)) (injr N)
<- strengthenLFNI notInjImp M N.

ca_pastr_l3d : ca (past T) (pastr D) (stripPastPast SP) ([h] injr (M h)) (injr N)
<- strengthenLFNI notInjBot M N.

ca_pastr_l3e : ca (past T) (pastr D) (stripPastPast SP) ([h] injr (M h)) (injr N)
<- strengthenLFNI notInjPair M N.

ca_pastr_l4 : ca (past T) (pastr D) (stripPastPast SP) ([h] pairl L h SP1 SP2 (E h)) F
       <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
     ({h3}{h3'} copyH' h3' base h3 -> map' h3 base (!! h3') ->
          ca (past T) (pastr D) (stripPastPast SP) ([h] E h h2 h3) (E' h2 h3)))
       <- shift D D'
       <- stripAddOne SP SP'
       <- stripToLess SP' L'
       <- ca T D' SP' ([h] pairl L' h SP1 SP2 E') F.


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Commutative Conversions (on left)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ca_axiom_l : ca (! TP) (axiom L H) stripPastPure E (E' H)
      <- weakenHyp L E E'.


ca_impl_l : ca T (impl L H D1 SP' D2) SP E (impl L H D1 SP' F)
     <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
   ca T (D2 h2) SP E (F h2)).

ca_pairl_l : ca T (pairl L H SP1 SP2 D) SP E (pairl L H SP1 SP2 F)
      <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
             ({h3}{h3'} copyH' h3' base h3 -> map' h3 base (!! h3') ->
        ca T (D h2 h3) SP E (F h2 h3))).

%{
% Here we need to take care of the case when we have botl on the left
% and an E of the form ([h] impl _ h _ _), ([h] pairl _ h _ _ _),
%                      or ([h] botl _ h _), or ([h] injr (M h))
}%
ca_botl_l_impl_r1 : ca (! TP) (botl L H SP') SP ([h] impl L* h _ SP* _) (botl L H stripPastPure).
ca_botl_l_impl_rStrengthen : ca _ (botl _ _ _) SP ([h] impl L h (E1 h) SP* (E2 h)) F
      <- strengthenP ([h] impl L h (E1 h) SP* (E2 h)) F.

ca_botl_l_pairl_r1 : ca (! TP) (botl L H SP') SP ([h] pairl L* h SP1 SP2 _) (botl L H stripPastPure).
ca_botl_l_pairl_rStrengthen : ca _ (botl _ _ _) SP ([h] pairl L h SP1 SP2 (E h)) F
      <- strengthenP ([h] pairl L h SP1 SP2 (E h)) F.

ca_botl_l_botl_r : ca (! TP) (botl L H SP') SP ([h] botl L* h SP*) (botl L H stripPastPure).
ca_botl_l_injr_r : ca (! TP) (botl L H SP') SP ([h] injr _) (botl L H stripPastPure).



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Strengthening Conversion for past and inj
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Strengthen Away when Pastr is on the right and left is in present.
% Here D can be impr, topr, botl, injr, or pairr

ca_strengthenImprPastr1 : ca _ (impr _ _) SP ([h] pastr (E' h)) F
  <- strengthenP ([h] pastr (E' h)) F.
ca_strengthenToprPastr2 : ca _ (topr) SP ([h] pastr (E' h)) F
  <- strengthenP ([h] pastr (E' h)) F.
ca_strengthenBotlPastr3 : ca _ (botl _ _ _) SP ([h] pastr (E' h)) F
  <- strengthenP ([h] pastr (E' h)) F.
```

```
ca_strengthenInjrPastr4 : ca _ (injr _) SP ([h] pastr (E' h)) F
  <- strengthenP ([h] pastr (E' h)) F.
ca_strengthenInjrPastr5 : ca _ (pairr _ _) SP ([h] pastr (E' h)) F
   <- strengthenP ([h] pastr (E' h)) F.


% Now for inject on right...
ca_strengthen_ImprInjr : ca _ (impr _ _) SP ([h] injr (M h)) (injr N)
  <- strengthenLFNI notInjImp M N.
ca_strengthen_TopInjr :  ca _ (topr) SP ([h] injr (M h)) (injr N)
  <- strengthenLFNI notInjTop M N.
ca_strengthen_PairrInjr :  ca _ (pairr _ _) SP ([h] injr (M h)) (injr N)
   <- strengthenLFNI notInjPair M N.



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Commutative Conversions (on right)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ca_axiom_rCommute : ca T D SP ([h] axiom L H) (axiom L H).

ca_topr_rCommute : ca T D SP ([h] topr) topr.

ca_botl_rCommute : ca T D SP ([h] botl L H SP') (botl L H SP').

ca_impr_rCommute : ca T D SP ([h] impr SP' (E h)) (impr SP' F)
    <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
        ca T D SP ([h] (E h h2)) (F h2)).

ca_impl_rCommute : ca T D SP ([h] impl L H (E1 h) SP' (E2 h)) (impl L H E1' SP' E2')
    <- ca T D SP E1 E1'
    <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
  ca T D SP ([h] E2 h h2) (E2' h2)).

ca_pairr_rCommute : ca T D SP ([h] pairr (E1 h) (E2 h)) (pairr F1 F2)
     <- ca T D SP E1 F1
     <- ca T D SP E2 F2.

ca_pairl_rCommute : ca T D SP ([h] pairl L H SP1 SP2 (E h)) (pairl L H SP1 SP2 E')
    <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
          ({h3}{h3'} copyH' h3' base h3 -> map' h3 base (!! h3') ->
      ca T D SP ([h] E h h2 h3) (E' h2 h3))).

%worlds (mapcopyBlock | lfmapcopyBlock | lftypeConsBlock) (ca T D SP E F).
%terminates {T [D E]} (ca T D _ E _).
%covers (ca +T +D +SP +E -F).
%total {T [D E]} (ca T D _ E _).
```

D.9 natDed.elf

```
% Adam Poswolsky
% Formulation of Natural Deduction version
% And some basic properties.



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Natural Deduction Meta Level
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
concND : nat -> o -> type.            %name concND (D E F) (d e f).

botND : concND X (! bot) -> stripPast X T _ TP -> concND X (! TP).

%
% When introducing a function whose argument has type (inj A)
% we need to also introduce an (exp _ A) for use by the LF level.
% Therefore, when the argument is an (inj A), the natural extension
% would look like (concND X' (inj AA) -> exp X' AA -> concND X T)
% But this is no better than just (exp X' AA -> concND X T), so
% we just use that.
%
MlamInj : stripPast X T X' (inj A) ->
          (exp X' A -> concND X S) -> concND X (! (T imp S)).
MlamNI : stripPast X T X' TP -> notInj TP ->
          (concND X' (! TP) -> concND X S) -> concND X (! (T imp S)).

Mapp : concND X (!(S imp T)) -> concND X S  -> stripPast X T _ TP  -> concND X (! TP).
topND : concND X (! top).
prev : concND X T -> concND (s X) (past T).
inject : exp X A -> concND X (!(inj A)).
pairI : concND X T1 -> concND X T2 -> concND X (! (pair T1 T2)).

% Similarly to the lam case, we need 4 cases for pairE to distinguish
% between inj and notInj
% Notice that this encoding differs from what is in the paper!
% Here, instead of creating e.fst and e.snd, we just have one rule:
% let (x,y) = e in f.
%
% Therefore, we need to do another (easy conversion) where
% let (x,y) = e in f  is converted to [e.fst/x][e.snd/y] f
% and in the other direction
% e.fst is converted to let (x,y) = e in x
% e.snd is converted to let (x,y) = e in y
%
% However, the manipulation of the context to perform this is
% difficult and unnecessary, so we omit it.
pairE00 : concND X (! (pair T1 T2)) -> stripPast X T1 _ TP1 -> stripPast X T2 _ TP2
  -> notInj TP1 -> notInj TP2
          -> (concND X (! TP1) -> concND X (! TP2) -> concND X S)
          -> concND X S.
pairE01 : concND X (! (pair T1 T2)) -> stripPast X T1 _ TP1 -> stripPast X T2 _ (inj B)
  -> notInj TP1
          -> (concND X (! TP1) -> exp X B -> concND X S)
          -> concND X S.
pairE10 : concND X (! (pair T1 T2)) -> stripPast X T1 _ (inj A) -> stripPast X T2 _ TP2
  -> notInj TP2
          -> (exp X A -> concND X (! TP2) -> concND X S)
          -> concND X S.
pairE11 : concND X (! (pair T1 T2)) -> stripPast X T1 _ (inj A) -> stripPast X T2 _ (inj B)
          -> (exp X A -> exp X B -> concND X S)
          -> concND X S.


% Note that shift is an admissible rule from the others... it is trivial to prove on paper
% But we won't go crazy again (see shiftSeq.elf) and prove it here.
shiftND : concND X T -> concND (s X) T.




% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% BASIC PROPERTIES
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%block NDPureblock :  some {X:nat}{TP:pureO}
      block {d:concND X (! TP)}.



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Strengthening
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

strengthenFunND : notInj TP -> (hyp X TP -> concND X (! TP) -> concND Y T) -> (concND X (! TP) -> concND Y T) -> type.
%mode strengthenFunND +NI +F -F'.
strengthenFunND_BotND : strengthenFunND NI ([h][d] botND (D h d) SP) ([d] botND (D' d) SP)
 <- strengthenFunND NI D D'.
strengthenFunND_MlamInj : strengthenFunND NI ([h][d] MlamInj SP (F h d)) ([d] MlamInj SP (F' d))
   <- ({n'} strengthenFunND NI ([h][d] F h d n') ([d] F' d n')).
strengthenFunND_MlamNI : strengthenFunND NI ([h][d] MlamNI SP NI' (F h d)) ([d] MlamNI SP NI' (F' d))
  <- ({e} strengthenFunND NI ([h][d] F h d e) ([d] F' d e)).

strengthenFunND_Mapp : strengthenFunND NI ([h][d] Mapp (D1 h d) (D2 h d) SP) ([d] Mapp (D1' d) (D2' d) SP)
<- strengthenFunND NI D1 D1'
<- strengthenFunND NI D2 D2'.
strengthenFunND_TopND : strengthenFunND NI ([h][d] topND) ([d] topND).
strengthenFunND_Prev : strengthenFunND NI ([h][d] prev (D h d)) ([d] prev (D' d))
<- strengthenFunND NI D D'.
strengthenFunND_Inject : strengthenFunND NI ([h][_] inject (M h)) ([d] inject M')
    <- strengthenLFNI NI M M'.

strengthenFunND_PairI : strengthenFunND NI ([h][d] pairI (D1 h d) (D2 h d)) ([d] pairI (D1' d) (D2' d))
 <- strengthenFunND NI D1 D1'
 <- strengthenFunND NI D2 D2'.

strengthenFunND_PairE00 : strengthenFunND NI ([h][d] pairE00 (D1 h d) SP1 SP2 NI1 NI2 (D2 h d))
                     ([d] pairE00 (D1' d) SP1 SP2 NI1 NI2 (D2' d))
   <- strengthenFunND NI D1 D1'
   <- ({e1}{e2} strengthenFunND NI ([h][d] D2 h d e1 e2) ([d] D2' d e1 e2)).

strengthenFunND_PairE01 : strengthenFunND NI ([h][d] pairE01 (D1 h d) SP1 SP2 NI1 (D2 h d))
                     ([d] pairE01 (D1' d) SP1 SP2 NI1 (D2' d))
   <- strengthenFunND NI D1 D1'
   <- ({e1}{m2} strengthenFunND NI ([h][d] D2 h d e1 m2) ([d] D2' d e1 m2)).

strengthenFunND_PairE10 : strengthenFunND NI ([h][d] pairE10 (D1 h d) SP1 SP2 NI2 (D2 h d))
                     ([d] pairE10 (D1' d) SP1 SP2 NI2 (D2' d))
   <- strengthenFunND NI D1 D1'
   <- ({m1}{e2} strengthenFunND NI ([h][d] D2 h d m1 e2) ([d] D2' d m1 e2)).

strengthenFunND_PairE11 : strengthenFunND NI ([h][d] pairE11 (D1 h d) SP1 SP2 (D2 h d))
                     ([d] pairE11 (D1' d) SP1 SP2 (D2' d))
   <- strengthenFunND NI D1 D1'
   <- ({m1}{m2} strengthenFunND NI ([h][d] D2 h d m1 m2) ([d] D2' d m1 m2)).

strengthenFunND_ShiftND : strengthenFunND NI ([h][d] shiftND (D h d)) ([d] shiftND (D' d))
   <- strengthenFunND NI D D'.
strengthenFunND_Block : strengthenFunND NI ([h][d] (D d)) ([d] D d).
%worlds (NDPureblock | hypBlock | lfexpBlock | lftypeConsBlock) (strengthenFunND _ _ _).
%total {F} (strengthenFunND _ F _).

% This is used when we encounter a case that is impossible
fromEmptyComesAllND : emptyType -> concND X T -> type.
%mode +{X:nat} +{T:o} +{EE:emptyType} -{D:concND X T} (fromEmptyComesAllND EE D).
%worlds (NDPureblock | hypBlock | lfexpBlock | lftypeConsBlock) (fromEmptyComesAllND _ _).
%total {EE} (fromEmptyComesAllND EE _).


strengthenND : le X Y -> (concND (s Y) (! TP) -> concND X T) -> concND X T -> type.
%mode strengthenND +L +F -F'.
strengthenND_BotND : strengthenND L ([d] botND (D d) SP) (botND D' SP)
      <- strengthenND L D D'.
strengthenND_MlamInj : strengthenND L ([d] MlamInj SP (F d)) (MlamInj SP F')
<- ({n'} strengthenND L ([d] F d n') (F' n')).
strengthenND_MlamNI : strengthenND L ([d] MlamNI SP NI' (F d)) (MlamNI SP NI' F')
```

```
       <- ({e} strengthenND L ([d] F d e) (F' e)).

strengthenND_Mapp : strengthenND L ([d] Mapp (D1 d) (D2 d) SP) (Mapp D1' D2' SP)
     <- strengthenND L D1 D1'
     <- strengthenND L D2 D2'.
strengthenND_TopND : strengthenND L ([d] topND) (topND).
strengthenND_Prev : strengthenND L ([d] prev (D d)) (prev D')
     <- leRemoveOneLeft L L'
     <- strengthenND L' D D'.
strengthenND_Inject : strengthenND L ([_] inject M) (inject M).

strengthenND_PairI : strengthenND L ([d] pairI (D1 d) (D2 d)) (pairI D1' D2')
      <- strengthenND L D1 D1'
      <- strengthenND L D2 D2'.

strengthenND_PairE00 : strengthenND L ([d] pairE00 (D1 d) SP1 SP2 NI1 NI2 (D2 d))
                 (pairE00 D1' SP1 SP2 NI1 NI2 D2')
<- strengthenND L D1 D1'
<- ({e1}{e2} strengthenND L ([d] D2 d e1 e2) (D2' e1 e2)).

strengthenND_PairE01 : strengthenND L ([d] pairE01 (D1 d) SP1 SP2 NI1 (D2 d))
                 (pairE01 D1' SP1 SP2 NI1 D2')
<- strengthenND L D1 D1'
<- ({e1}{m2} strengthenND L ([d] D2 d e1 m2) (D2' e1 m2)).

strengthenND_PairE10 : strengthenND L ([d] pairE10 (D1 d) SP1 SP2 NI2 (D2 d))
                 (pairE10 D1' SP1 SP2 NI2 D2')
<- strengthenND L D1 D1'
<- ({m1}{e2} strengthenND L ([d] D2 d m1 e2) (D2' m1 e2)).

strengthenND_PairE11 : strengthenND L ([d] pairE11 (D1 d) SP1 SP2 (D2 d))
                 (pairE11 D1' SP1 SP2 D2')
<- strengthenND L D1 D1'
<- ({m1}{m2} strengthenND L ([d] D2 d m1 m2) (D2' m1 m2)).

strengthenND_ShiftND : strengthenND L ([d] shiftND (D d)) (shiftND D')
<- leRemoveOneLeft L L'
<- strengthenND L' D D'.

strengthenND_Impossible : strengthenND (L: le (s X) X) F D
   <- impossibleLessNum _ L EE
   <- fromEmptyComesAllND EE D.

strengthenND_Block : strengthenND L ([d] D) D.

%worlds (NDPureblock | hypBlock | lfexpBlock | lftypeConsBlock) (strengthenND _ _ _).
%total {F} (strengthenND _ F _).


strengthenFunNDLF : (hyp X (inj A) -> exp X A -> concND Y T) -> (exp X A -> concND Y T) -> type.
%mode strengthenFunNDLF +F -F'.
strengthenFunNDLF_BotND : strengthenFunNDLF ([h][n] botND (D h n) SP) ([n] botND (D' n) SP)
   <- strengthenFunNDLF D D'.
strengthenFunNDLF_MlamInj : strengthenFunNDLF ([h][n] MlamInj SP (F h n)) ([n] MlamInj SP (F' n))
     <- ({n'} strengthenFunNDLF ([h][n] F h n n') ([n] F' n n')).
strengthenFunNDLF_MlamNI : strengthenFunNDLF ([h][n] MlamNI SP NI (F h n)) ([n] MlamNI SP NI (F' n))
    <- ({e} strengthenFunNDLF ([h][n] F h n e) ([n] F' n e)).

strengthenFunNDLF_Mapp : strengthenFunNDLF ([h][n] Mapp (D1 h n) (D2 h n) SP) ([n] Mapp (D1' n) (D2' n) SP)
  <- strengthenFunNDLF D1 D1'
  <- strengthenFunNDLF D2 D2'.
strengthenFunNDLF_TopND : strengthenFunNDLF ([h][n] topND) ([n] topND).
strengthenFunNDLF_Prev : strengthenFunNDLF ([h][n] prev (D h n)) ([n] prev (D' n))
  <- strengthenFunNDLF D D'.
strengthenFunNDLF_Inject : strengthenFunNDLF ([h][n] inject (M h n)) ([n] inject (([n'] M' n' n') n))
    <- ({n} convertHypExp ([h] M h n) (M' n)).

strengthenFunNDLF_PairI : strengthenFunNDLF ([h][n] pairI (D1 h n) (D2 h n)) ([n] pairI (D1' n) (D2' n))
   <- strengthenFunNDLF D1 D1'
   <- strengthenFunNDLF D2 D2'.

strengthenFunNDLF_PairE00 : strengthenFunNDLF ([h][n] pairE00 (D1 h n) SP1 SP2 NI1 NI2 (D2 h n))
                      ([n] pairE00 (D1' n) SP1 SP2 NI1 NI2 (D2' n))
     <- strengthenFunNDLF D1 D1'
     <- ({e1}{e2} strengthenFunNDLF ([h][n] D2 h n e1 e2) ([n] D2' n e1 e2)).
```

```
strengthenFunNDLF_PairE01 : strengthenFunNDLF ([h][n] pairE01 (D1 h n) SP1 SP2 NI1 (D2 h n))
                      ([n] pairE01 (D1' n) SP1 SP2 NI1 (D2' n))
     <- strengthenFunNDLF D1 D1'
     <- ({e1}{m2} strengthenFunNDLF ([h][n] D2 h n e1 m2) ([n] D2' n e1 m2)).

strengthenFunNDLF_PairE10 : strengthenFunNDLF ([h][n] pairE10 (D1 h n) SP1 SP2 NI2 (D2 h n))
                      ([n] pairE10 (D1' n) SP1 SP2 NI2 (D2' n))
     <- strengthenFunNDLF D1 D1'
     <- ({m1}{e2} strengthenFunNDLF ([h][n] D2 h n m1 e2) ([n] D2' n m1 e2)).

strengthenFunNDLF_PairE11 : strengthenFunNDLF ([h][n] pairE11 (D1 h n) SP1 SP2 (D2 h n))
                      ([n] pairE11 (D1' n) SP1 SP2 (D2' n))
     <- strengthenFunNDLF D1 D1'
     <- ({m1}{m2} strengthenFunNDLF ([h][n] D2 h n m1 m2) ([n] D2' n m1 m2)).

strengthenFunNDLF_ShiftND : strengthenFunNDLF ([h][n] shiftND (D h n)) ([n] shiftND (D' n))
     <- strengthenFunNDLF D D'.
strengthenFunNDLF_Block : strengthenFunNDLF ([h][n] D) ([n] D).
%worlds (NDPureblock | hypBlock | lfexpBlock | lftypeConsBlock) (strengthenFunNDLF _ _).
%total {F} (strengthenFunNDLF F _).


strengthenNDLF : le X Y -> (exp (s Y) A -> concND X T) -> concND X T -> type.
%mode strengthenNDLF +L +F -D.
strengthenNDLF_botND : strengthenNDLF L ([n] botND (D n) SP) (botND D' SP)
      <- strengthenNDLF L D D'.

strengthenNDLF_MlamInj : strengthenNDLF L ([n] MlamInj SP (F n)) (MlamInj SP F')
<- ({m}
            strengthenNDLF L ([n] F n m) (F' m)).

strengthenNDLF_Mlam : strengthenNDLF L ([n] MlamNI SP NI (F n)) (MlamNI SP NI F')
       <- ({e}
  strengthenNDLF L ([n] F n e) (F' e)).

strengthenNDLF_Mapp : strengthenNDLF L ([n] Mapp (D1 n) (D2 n) SP) (Mapp D1' D2' SP)
     <- strengthenNDLF L D1 D1'
     <- strengthenNDLF L D2 D2'.

strengthenNDLF_topND : strengthenNDLF L ([n] topND) topND.
strengthenNDLF_prev : strengthenNDLF L ([n] prev (D n)) (prev D')
     <- leRemoveOneLeft L L'
     <- strengthenNDLF L' D D'.
strengthenNDLF_inject : strengthenNDLF L ([n] inject (M n)) (inject M')
       <- strengthenLF2 L M M'.

strengthenNDLF_PairI : strengthenNDLF L ([n] pairI (D1 n) (D2 n)) (pairI D1' D2')
<- strengthenNDLF L D1 D1'
<- strengthenNDLF L D2 D2'.

strengthenNDLF_PairE00 : strengthenNDLF L ([n] pairE00 (D1 n) SP1 SP2 NI1 NI2 (D2 n))
                  (pairE00 D1' SP1 SP2 NI1 NI2 D2')
  <- strengthenNDLF L D1 D1'
  <- ({e1}{e2} strengthenNDLF L ([n] D2 n e1 e2) (D2' e1 e2)).

strengthenNDLF_PairE01 : strengthenNDLF L ([n] pairE01 (D1 n) SP1 SP2 NI1 (D2 n))
                  (pairE01 D1' SP1 SP2 NI1 D2')
  <- strengthenNDLF L D1 D1'
  <- ({e1}{m2} strengthenNDLF L ([n] D2 n e1 m2) (D2' e1 m2)).

strengthenNDLF_PairE10 : strengthenNDLF L ([n] pairE10 (D1 n) SP1 SP2 NI2 (D2 n))
                  (pairE10 D1' SP1 SP2 NI2 D2')
  <- strengthenNDLF L D1 D1'
  <- ({m1}{e2} strengthenNDLF L ([n] D2 n m1 e2) (D2' m1 e2)).

strengthenNDLF_PairE11 : strengthenNDLF L ([n] pairE11 (D1 n) SP1 SP2 (D2 n))
                  (pairE11 D1' SP1 SP2 D2')
     <- strengthenNDLF L D1 D1'
     <- ({m1}{m2} strengthenNDLF L ([n] D2 n m1 m2) (D2' m1 m2)).


strengthenNDLF_shiftND : strengthenNDLF L ([n] shiftND (D n)) (shiftND D')
<- leRemoveOneLeft L L'
<- strengthenNDLF L' D D'.

strengthenNDLF_block : strengthenNDLF L ([n] E) E.
```

```
%worlds (NDPureblock | hypBlock | lfexpBlock | lftypeConsBlock) (strengthenNDLF _ _ _).
%total {F} (strengthenNDLF _ F _).

impossibleConcND : concND z (past T) -> emptyType -> type.

impossibleConcND_PairE00 : impossibleConcND (pairE00 D1 SP1 SP2 NI1 NI2 D2) EE
    <- ({e1}{e2} impossibleConcND (D2 e1 e2) EE).

impossibleConcND_PairE01 : impossibleConcND (pairE01 D1 SP1 SP2 NI1 D2) EE
    <- ({e1}{m2} impossibleConcND (D2 e1 m2) EE).

impossibleConcND_PairE10 : impossibleConcND (pairE10 D1 SP1 SP2 NI2 D2) EE
    <- ({m1}{e2} impossibleConcND (D2 m1 e2) EE).

impossibleConcND_PairE11 : impossibleConcND (pairE11 D1 SP1 SP2 D2) EE
    <- ({m1}{m2} impossibleConcND (D2 m1 m2) EE).

%mode impossibleConcND +D -EE.
%worlds (NDPureblock | hypBlock | lfexpBlock | lftypeConsBlock) (impossibleConcND _ _).
%total {D} (impossibleConcND D _).


futureRuleND : concND (s X) (past T) -> concND X T -> type.
%mode futureRuleND +D -D'.

futureRuleND_Prev : futureRuleND (prev D) D.

futureRuleND_PairE00 : futureRuleND (pairE00 D1 SP1 SP2 NI1 NI2 D2) E2
<- ({e1}{e2} futureRuleND (D2 e1 e2) (D2' e1 e2))
<- ({e1} strengthenND base (D2' e1) (E1 e1))
<- strengthenND base E1 E2.

futureRuleND_PairE01 : futureRuleND (pairE01 D1 SP1 SP2 NI1 D2) E2
<- ({e1}{m2} futureRuleND (D2 e1 m2) (D2' e1 m2))
<- ({e1} strengthenNDLF base (D2' e1) (E1 e1))
<- strengthenND base E1 E2.

futureRuleND_PairE10 : futureRuleND (pairE10 D1 SP1 SP2 NI2 D2) E2
<- ({m1}{e2} futureRuleND (D2 m1 e2) (D2' m1 e2))
<- ({m1} strengthenND base (D2' m1) (E1 m1))
<- strengthenNDLF base E1 E2.

futureRuleND_PairE11 : futureRuleND (pairE11 D1 SP1 SP2 D2) E2
<- ({m1}{m2} futureRuleND (D2 m1 m2) (D2' m1 m2))
<- ({m1} strengthenNDLF base (D2' m1) (E1 m1))
<- strengthenNDLF base E1 E2.

futureRuleND_ShiftND : futureRuleND (shiftND D) (shiftND D')
<- futureRuleND D D'.

futureRuleND_Impossible : futureRuleND (shiftND D) D'
   <- impossibleConcND D EE
   <- fromEmptyComesAllND EE D'.

%worlds (NDPureblock | hypBlock | lfexpBlock | lftypeConsBlock) (futureRuleND _ _).
%total {D} (futureRuleND D _).



strengthenNDpast : (concND X (! TP) -> concND X (past T)) -> concND X (past T) -> type.
%mode strengthenNDpast  +D -D'.
strengthenNDpastcase : strengthenNDpast D (prev E')
<- ({e} futureRuleND (D e) (D' e))
<- strengthenND base D' E'.

strengthenNDpastimpossible : strengthenNDpast D E
      <- ({e} impossibleConcND (D e) EE)
      <- fromEmptyComesAllND EE E.

%worlds (NDPureblock | hypBlock | lfexpBlock | lftypeConsBlock) (strengthenNDpast _ _).
%covers (strengthenNDpast +D -D').
%total {D} (strengthenNDpast D _).


strengthenNDLFpast : (exp X A -> concND X (past T)) -> concND X (past T) -> type.
```

```
%mode strengthenNDLFpast  +D -D'.
strengthenNDLFpastcase : strengthenNDLFpast D (prev E')
  <- ({m} futureRuleND (D m) (D' m))
  <- strengthenNDLF base D' E'.

strengthenNDLFpastimpossible : strengthenNDLFpast D E
<- ({m} impossibleConcND (D m) EE)
<- fromEmptyComesAllND EE E.

%worlds (NDPureblock | hypBlock | lfexpBlock | lftypeConsBlock) (strengthenNDLFpast _ _).
%covers (strengthenNDLFpast +D -D').
%total {D} (strengthenNDLFpast D _).
```

D.10 natToSeq.elf

```
% Adam Poswolsky
% Conversion from Natural Deduction to Sequent Calculus.


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Convert Natural Deduction to Sequent Calculus
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
NDtoSQ : concND X T -> conc X T -> type.
%mode NDtoSQ +DN -D.

%block NDtoSQblockNI : some {X:nat}{TP:pureO}{NI:notInj TP}
       block {h:hyp X TP}{h':hyp' X TP}{c:copyH' h' base h}{m:map' h base (!! h')}
                             {e:concND X (! TP)}{w:NDtoSQ e (axiom base h)}.
%block NDtoSQblockInj : some {X:nat}{A:lftp}
                        block {h:hyp X (inj A)}{h':hyp' X (inj A)}{c:copyH' h' base h}{m:map' h base (!! h')}
                              {m:exp X A}{n:exp' X A}{w:copyLF' n m}{s:calcShiftLF' m (shiftLF' n)}.


% %%%%%%%%%%%%%%%%%%%%
% Meta Level
% %%%%%%%%%%%%%%%%%%%%

NDtoSQ_botND : NDtoSQ (botND D SP) F
      <- NDtoSQ D D'
      <- ca (! bot) D' stripPastPure ([h] botl base h SP) F.

NDtoSQ_MlamNI : NDtoSQ (MlamNI SP NI D) (impr SP D')
      <- ({h}{h'} copyH' h' base h -> map' h base (!! h')
    -> {e}(NDtoSQ e (axiom base h))
    -> NDtoSQ (D e) (D' h)).

NDtoSQ_MlamInj : NDtoSQ (MlamInj SP D) (impr SP ([h] D' h (castHyp h)))
      <- ({h}{h'} copyH' h' base h -> map' h base (!! h')
    -> {n}{n'} copyLF' n' n -> calcShiftLF' n (shiftLF' n')
    -> NDtoSQ (D n) (D' h n)).

NDtoSQ_Mapp : NDtoSQ (Mapp E D SP) F
      <- NDtoSQ E E'
      <- NDtoSQ D D'
      <- ca _ E' stripPastPure ([h] impl base h D' SP ([h'] axiom base h')) F.

NDtoSQ_topND : NDtoSQ topND topr.
NDtoSQ_prev : NDtoSQ (prev D) (pastr D')
       <- NDtoSQ D D'.
NDtoSQ_inject : NDtoSQ (inject M) (injr M).

NDtoSQ_pairI : NDtoSQ (pairI D1 D2) (pairr D1' D2')
<- NDtoSQ D1 D1'
<- NDtoSQ D2 D2'.

NDtoSQ_pairE00 : NDtoSQ (pairE00 D1 SP1 SP2 NI1 NI2 D2) F
  <- NDtoSQ D1 D1'
                 <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e}(NDtoSQ e (axiom base h))
     -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2}(NDtoSQ e2 (axiom base h2))
-> NDtoSQ (D2 e e2) (D2' h h2)))
  <- ca _ D1' stripPastPure ([h] pairl base h SP1 SP2 D2') F.

NDtoSQ_pairE01 : NDtoSQ (pairE01 D1 SP1 SP2 NI1 D2) F
  <- NDtoSQ D1 D1'
                 <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e}(NDtoSQ e (axiom base h))
     -> ({h2: hyp X (inj A)}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {n2}{n2'} copyLF' n2' n2 -> calcShiftLF' n2 (shiftLF' n2')
-> NDtoSQ (D2 e n2) (D2' h h2 n2)))
  <- ca _ D1' stripPastPure ([h] pairl base h SP1 SP2 ([h1][h2] D2' h1 h2 (castHyp h2))) F.

NDtoSQ_pairE10 : NDtoSQ (pairE10 D1 SP1 SP2 NI2 D2) F
  <- NDtoSQ D1 D1'
  <- ({h: hyp X (inj A)}{h'} copyH' h' base h -> map' h base (!! h') -> {n}{n'} copyLF' n' n -> calcShiftLF' n (shiftLF' n')
                    -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2}(NDtoSQ e2 (axiom base h2))
                       -> NDtoSQ (D2 n e2) (D2' h n h2)))
  <- ca _ D1' stripPastPure ([h] pairl base h SP1 SP2 ([h1][h2] D2' h1 (castHyp h1) h2)) F.

NDtoSQ_pairE11 : NDtoSQ (pairE11 D1 SP1 SP2 D2) F
  <- NDtoSQ D1 D1'
  <- ({h: hyp X (inj A)}{h'} copyH' h' base h -> map' h base (!! h') -> {n}{n'} copyLF' n' n -> calcShiftLF' n (shiftLF' n')
     -> ({h2: hyp X (inj B)}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {n2}{n2'} copyLF' n2' n2 -> calcShiftLF' n2 (shiftLF' n2')
```

```
                       -> NDtoSQ (D2 n n2) (D2' h n h2 n2)))
  <- ca _ D1' stripPastPure ([h] pairl base h SP1 SP2 ([h1][h2] D2' h1 (castHyp h1) h2 (castHyp h2))) F.


NDtoSQ_shift : NDtoSQ (shiftND D) D''
<- NDtoSQ D D'
<- shift D' D''.


%worlds (NDtoSQblockNI | NDtoSQblockInj | lftypeConsBlock) (NDtoSQ _ _).
%terminates {DN} (NDtoSQ DN _).
%total {DN} (NDtoSQ DN _).
```

D.11 seqToNat.elf

```
% Adam Poswolsky
% Conversion from Sequent Calculus to Natural Deduction

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% First, a property we need when converting Sequent to Natural Deduction
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
NDprop : concND X (!(inj A)) -> (exp X A -> concND X T) -> concND X T -> type.
%mode NDprop +D +F -D'.
NDprop_pure : NDprop D E (Mapp (MlamInj stripPastPure E) D stripPastPure).
NDprop_impure : NDprop D E F
 <- strengthenNDLFpast E F.
%worlds (hypBlock | lfexpBlock | NDPureblock | lftypeConsBlock) (NDprop _ _ _).
%terminates {F} (NDprop _ F _).
%total {F} (NDprop _ F _).



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Now Convert Sequent Calculus to Natural Deduction
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SQtoND : conc X T -> concND X T -> type.
hypToCN : le X Y -> hyp X TP -> concND Y (! TP) -> type.

%mode SQtoND +D -DN.
%mode hypToCN +L +H -CN.

%block SQtoNDblockNI : some {X:nat}{TP:pureO}{NI:notInj TP}
                       block {h:hyp X TP}{h':hyp' X TP}{c:copyH' h' base h}{m:map' h base (!! h')}
                          {e:concND X (! TP)}{w:SQtoND (axiom base h) e}
          {z:hypToCN base h e}.


%block SQtoNDblockInj : some {X:nat}{A:lftp}
                        block {h:hyp X (inj A)}{h':hyp' X (inj A)}
                           {c:copyH' h' base h}{m:map' h base (!! h')}
                           {m:exp X A}{n:exp' X A}{w:copyLF' n m}
                           {s:calcShiftLF' m (shiftLF' n)}{z:hypToCN base h (inject m)}.


hypToCN_Ind : hypToCN (one L) H (shiftND D)
       <- hypToCN L H D.

%worlds (SQtoNDblockNI | SQtoNDblockInj | lftypeConsBlock) (hypToCN _ _ _).
%terminates {L} (hypToCN L _ _).
%total {L} (hypToCN L _ _).


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Meta Cases
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SQtoND_axiom : SQtoND (axiom L H) F
    <- hypToCN L H F.

SQtoND_imprTop : SQtoND (impr SP F) (MlamNI SP notInjTop F*)
       <- ({h:hyp _ top}{h'} copyH' h' base h -> map' h base (!! h') ->
                  {e} SQtoND (axiom base h) e -> hypToCN base h e ->
  SQtoND (F h) (F' h e))
       <- strengthenFunND notInjTop F' F*.

SQtoND_imprTop : SQtoND (impr SP F) (MlamNI SP notInjTop F*)
       <- ({h:hyp _ top}{h'} copyH' h' base h -> map' h base (!! h') ->
                  {e} SQtoND (axiom base h) e -> hypToCN base h e ->
  SQtoND (F h) (F' h e))
       <- strengthenFunND notInjTop F' F*.
SQtoND_imprBot : SQtoND (impr SP F) (MlamNI SP notInjBot F*)
       <- ({h:hyp _ bot}{h'} copyH' h' base h -> map' h base (!! h') ->
                  {e} SQtoND (axiom base h) e -> hypToCN base h e ->
  SQtoND (F h) (F' h e))
       <- strengthenFunND notInjBot F' F*.
SQtoND_imprImp : SQtoND (impr SP F) (MlamNI SP notInjImp F*)
       <- ({h:hyp _ (T1 imp T2)}{h'} copyH' h' base h -> map' h base (!! h') ->
                  {e} SQtoND (axiom base h) e -> hypToCN base h e ->
  SQtoND (F h) (F' h e))
```

```
       <- strengthenFunND notInjImp F' F*.

SQtoND_imprPair : SQtoND (impr SP F) (MlamNI SP notInjPair F*)
       <- ({h:hyp _ (pair T1 T2)}{h'} copyH' h' base h -> map' h base (!! h') ->
                  {e} SQtoND (axiom base h) e -> hypToCN base h e ->
  SQtoND (F h) (F' h e))
       <- strengthenFunND notInjPair F' F*.

SQtoND_imprInj : SQtoND (impr SP F) (MlamInj SP F*)
       <- ({h:hyp _ (inj A)}{h'} copyH' h' base h -> map' h base (!! h') ->
                  {m} {n} copyLF' n m -> calcShiftLF' m (shiftLF' n) -> hypToCN base h (inject m) ->
  SQtoND (F h) (F' h m))
       <- strengthenFunNDLF F' F*.


SQtoND_implTop : SQtoND (impl L H D1 SP D2) (F2' (Mapp F0 F1 SP))
       <- hypToCN L H F0
       <- SQtoND D1 F1
       <- ({h:hyp _ top}{h'} copyH' h' base h -> map' h base (!! h') ->
                  {e} SQtoND (axiom base h) e -> hypToCN base h e ->
  SQtoND (D2 h) (F2 h e))
       <- strengthenFunND notInjTop F2 F2'.
SQtoND_implBot : SQtoND (impl L H D1 SP D2) (F2' (Mapp F00 F1 SP))
       <- hypToCN L H F00
       <- SQtoND D1 F1
       <- ({h:hyp _ bot}{h'} copyH' h' base h -> map' h base (!! h') ->
                  {e} SQtoND (axiom base h) e -> hypToCN base h e ->
  SQtoND (D2 h) (F2 h e))
       <- strengthenFunND notInjBot F2 F2'.

SQtoND_implImp : SQtoND (impl L H D1 SP D2) (F2' (Mapp F00 F1 SP))
       <- hypToCN L H F00
       <- SQtoND D1 F1
       <- ({h:hyp _ (T1 imp T2)}{h'} copyH' h' base h -> map' h base (!! h') ->
                  {e} SQtoND (axiom base h) e -> hypToCN base h e ->
  SQtoND (D2 h) (F2 h e))
       <- strengthenFunND notInjImp F2 F2'.

SQtoND_implPair : SQtoND (impl L H D1 SP D2) (F2' (Mapp F00 F1 SP))
       <- hypToCN L H F00
       <- SQtoND D1 F1
       <- ({h:hyp _ (pair T1 T2)}{h'} copyH' h' base h -> map' h base (!! h') ->
                  {e} SQtoND (axiom base h) e -> hypToCN base h e ->
  SQtoND (D2 h) (F2 h e))
       <- strengthenFunND notInjPair F2 F2'.

SQtoND_implInj : SQtoND (impl L H D1 SP D2) E
       <- hypToCN L H F00
       <- SQtoND D1 F1
       <- ({h:hyp _ (inj A)}{h'} copyH' h' base h -> map' h base (!! h') ->
                  {m} {n} copyLF' n m -> calcShiftLF' m (shiftLF' n)
                  ->  hypToCN base h (inject m) ->
  SQtoND (D2 h) (F2 h m))
       <- strengthenFunNDLF F2 F2'
       <- NDprop (Mapp F00 F1 SP) F2' E.

SQtoND_topr : SQtoND topr topND.

SQtoND_botl : SQtoND (botl L H SP) (botND D SP)
       <- hypToCN L H D.

SQtoND_pastr : SQtoND (pastr D) (prev D')
<- SQtoND D D'.

SQtoND_injr : SQtoND (injr M) (inject M).

SQtoND_pairr : SQtoND (pairr D1 D2) (pairI D1' D2')
<- SQtoND D1 D1'
<- SQtoND D2 D2'.

% We have 25 cases for pairl
% For convention we will use 0=top,1=bot,2=imp,3=pair,4=inj
% And we will label each case with XY where X is
% the type of the first element and Y is the type
% of the second.
```

```
SQtoND_pairl00 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjTop notInjTop F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                   -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                      SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjTop (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjTop ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl01 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjTop notInjBot F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                   -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                      SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjBot (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjTop ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl02 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjTop notInjImp F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                   -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                      SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjImp (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjTop ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl03 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjTop notInjPair F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                   -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                      SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjPair (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjTop ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl04 : SQtoND (pairl L H SP1 SP2 D) (pairE01 F00 SP1 SP2 notInjTop F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                   -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {m2} {n2} copyLF' n2 m2 -> calcShiftLF' m2 (shiftLF' n2)
                      ->  hypToCN base h2 (inject m2) -> SQtoND (D h h2) (F h e h2 m2)))
  <- ({h}{e} strengthenFunNDLF (F h e) (F2 h e))
  <- ({m2} strengthenFunND notInjTop ([h][e] F2 h e m2) ([e] F3 e m2)).

SQtoND_pairl10 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjBot notInjTop F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                   -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                      SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjTop (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjBot ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl11 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjBot notInjBot F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                   -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                      SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjBot (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjBot ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl12 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjBot notInjImp F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                   -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                      SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjImp (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjBot ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl13 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjBot notInjPair F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                   -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                      SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjPair (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjBot ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl14 : SQtoND (pairl L H SP1 SP2 D) (pairE01 F00 SP1 SP2 notInjBot F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                   -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {m2} {n2} copyLF' n2 m2 -> calcShiftLF' m2 (shiftLF' n2)
```

```
->  hypToCN base h2 (inject m2) -> SQtoND (D h h2) (F h e h2 m2)))
  <- ({h}{e} strengthenFunNDLF (F h e) (F2 h e))
  <- ({m2} strengthenFunND notInjBot ([h][e] F2 h e m2) ([e] F3 e m2)).


SQtoND_pairl20 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjImp notInjTop F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                    -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                       SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjTop (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjImp ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl21 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjImp notInjBot F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                    -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                       SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjBot (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjImp ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl22 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjImp notInjImp F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                    -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                       SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjImp (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjImp ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl23 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjImp notInjPair F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                    -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                       SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjPair (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjImp ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl24 : SQtoND (pairl L H SP1 SP2 D) (pairE01 F00 SP1 SP2 notInjImp F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                    -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {m2} {n2} copyLF' n2 m2 -> calcShiftLF' m2 (shiftLF' n2)
                       ->  hypToCN base h2 (inject m2) -> SQtoND (D h h2) (F h e h2 m2)))
  <- ({h}{e} strengthenFunNDLF (F h e) (F2 h e))
  <- ({m2} strengthenFunND notInjImp ([h][e] F2 h e m2) ([e] F3 e m2)).


SQtoND_pairl30 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjPair notInjTop F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                    -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                       SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjTop (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjPair ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl31 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjPair notInjBot F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                    -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                       SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjBot (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjPair ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl32 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjPair notInjImp F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                    -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                       SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjImp (F h e) (F2 h e))
  <- ({e2} strengthenFunND notInjPair ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl33 : SQtoND (pairl L H SP1 SP2 D) (pairE00 F00 SP1 SP2 notInjPair notInjPair F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                    -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2 -> hypToCN base h2 e2 ->
                       SQtoND (D h h2) (F h e h2 e2)))
  <- ({h}{e} strengthenFunND notInjPair (F h e) (F2 h e))
```

```
  <- ({e2} strengthenFunND notInjPair ([h][e] F2 h e e2) ([e] F3 e e2)).

SQtoND_pairl34 : SQtoND (pairl L H SP1 SP2 D) (pairE01 F00 SP1 SP2 notInjPair F3)
  <- hypToCN L H F00
  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {e} SQtoND (axiom base h) e -> hypToCN base h e
                    -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {m2} {n2} copyLF' n2 m2 -> calcShiftLF' m2 (shiftLF' n2)
                       ->  hypToCN base h2 (inject m2) -> SQtoND (D h h2) (F h e h2 m2)))
  <- ({h}{e} strengthenFunNDLF (F h e) (F2 h e))
  <- ({m2} strengthenFunND notInjPair ([h][e] F2 h e m2) ([e] F3 e m2)).



SQtoND_pairl40 : SQtoND (pairl L H SP1 SP2 D) (pairE10 F00 SP1 SP2 notInjTop F3)
  <- hypToCN L H F00
                  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {m} {n} copyLF' n m -> calcShiftLF' m (shiftLF' n)
                     ->  hypToCN base h (inject m)
                     -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2
                        -> hypToCN base h2 e2 -> SQtoND (D h h2) (F h m h2 e2)))
  <- ({h}{m} strengthenFunND notInjTop (F h m) (F2 h m))
  <- ({e2} strengthenFunNDLF ([h][m] F2 h m e2) ([m] F3 m e2)).

SQtoND_pairl41 : SQtoND (pairl L H SP1 SP2 D) (pairE10 F00 SP1 SP2 notInjBot F3)
  <- hypToCN L H F00
                  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {m} {n} copyLF' n m -> calcShiftLF' m (shiftLF' n)
                     ->  hypToCN base h (inject m)
                     -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2
                        -> hypToCN base h2 e2 ->  SQtoND (D h h2) (F h m h2 e2)))
  <- ({h}{m} strengthenFunND notInjBot (F h m) (F2 h m))
  <- ({e2} strengthenFunNDLF ([h][m] F2 h m e2) ([m] F3 m e2)).

SQtoND_pairl42 : SQtoND (pairl L H SP1 SP2 D) (pairE10 F00 SP1 SP2 notInjImp F3)
  <- hypToCN L H F00
                  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {m} {n} copyLF' n m -> calcShiftLF' m (shiftLF' n)
                     ->  hypToCN base h (inject m)
                     -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2
                        -> hypToCN base h2 e2 -> SQtoND (D h h2) (F h m h2 e2)))
  <- ({h}{m} strengthenFunND notInjImp (F h m) (F2 h m))
  <- ({e2} strengthenFunNDLF ([h][m] F2 h m e2) ([m] F3 m e2)).

SQtoND_pairl43 : SQtoND (pairl L H SP1 SP2 D) (pairE10 F00 SP1 SP2 notInjPair F3)
  <- hypToCN L H F00
                  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {m} {n} copyLF' n m -> calcShiftLF' m (shiftLF' n)
                     ->  hypToCN base h (inject m)
                     -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {e2} SQtoND (axiom base h2) e2
                        -> hypToCN base h2 e2 -> SQtoND (D h h2) (F h m h2 e2)))
  <- ({h}{m} strengthenFunND notInjPair (F h m) (F2 h m))
  <- ({e2} strengthenFunNDLF ([h][m] F2 h m e2) ([m] F3 m e2)).

SQtoND_pairl44 : SQtoND (pairl L H SP1 SP2 D) (pairE11 F00 SP1 SP2 F3)
  <- hypToCN L H F00
                  <- ({h}{h'} copyH' h' base h -> map' h base (!! h') -> {m} {n} copyLF' n m -> calcShiftLF' m (shiftLF' n)
                     ->  hypToCN base h (inject m)
                     -> ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') -> {m2} {n2} copyLF' n2 m2 -> calcShiftLF' m2 (shiftLF' n2)
                        -> hypToCN base h2 (inject m2) -> SQtoND (D h h2) (F h m h2 m2)))
  <- ({h}{m} strengthenFunNDLF (F h m) (F2 h m))
  <- ({m2} strengthenFunNDLF ([h][m] F2 h m m2) ([m] F3 m m2)).


%worlds (SQtoNDblockNI | SQtoNDblockInj | lftypeConsBlock) (SQtoND _ _).
%terminates {D} (SQtoND D _).
%total {D} (SQtoND D _).
```

D.12 examples.elf

```
% Adam Poswolsky
% Example Section

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Distributivity of Past over implication with inject.
% Type is past (<A> -> <B>) -> <A> -> <B>
distPast : conc (s z) (!
                    (past (!(!(inj A) imp !(inj B)))
                  imp !(past (!(inj A))
                  imp past (!(inj B))))
                    )
 = impr (stripPastPast stripPastPure) ([h1 : hyp z (!(inj A) imp !(inj B))]
           impr (stripPastPast stripPastPure) ([h2 : hyp z (inj A)]
           pastr (impl base h1
                   (axiom base h2) stripPastPure
                   ([h3 : hyp z (inj B)] axiom base h3)))).



% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% LF Application on Meta-Level
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
liftedApp : conc X (!(inj (A arrow B))) -> conc X (!(inj A)) -> conc X (!(inj B)) -> type.
%mode liftedApp +D1 +D2 -F.
liftedAppCase : liftedApp D1 D2 E
       <- ({h1}{h1'} copyH' h1' base h1 -> map' h1 base (!! h1') ->
                 ca _ D2 stripPastPure ([h2] injr (app (castHyp h1) (castHyp h2)))
                                                  (F h1))
<- ca _ D1 stripPastPure F E.
%worlds (mapcopyBlock | lfmapcopyBlock | lftypeConsBlock) (liftedApp _ _ _).
%total {} (liftedApp _ _ _).
```

References

[Aug98] Lennart Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CX05] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Intenational Conference on Functional Programming*, 2005.

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Logic in Computer Science*, pages 184–195, 1996.

[GP99] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, July 1999. IEEE Computer Society Press.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HMS01] Furio Honsell, Marino Miculan, and Ivan Scagnetto. π-calculus in (Co)inductive-type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.

[Hof99] Martin Hofmann. Semantical analysis for higher-order abstract syntax. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 204–213, Trento, Italy, July 1999. IEEE Computer Society Press.

[Lel98] Pierre Leleu. *Induction et Syntaxe Abstraite d'Ordre Supérieur dans les Théories Typées*. PhD thesis, Ecole Nationale des Ponts et Chaussees, Marne-la-Vallee, France, December 1998.

[MAC03] Alberto Momigliano, Simon Ambler, and Roy Crole. A definitional approach to primitive recursion over higher order abstract syntax. In Alberto Momigliano and Marino Miculan, editors, *Proceedings of the Merlin Workshop*, Uppsala, Sweden, June 2003. ACM Press.

[Mil90] Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, Nice, France, May 1990.

[Mil91] Dale Miller. Unification of simply typed lambda-terms as logic programming. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.

[MM04] Conor McBride and James McKinna. The view from the left. *Journale of Functional Programming*, 14(1), 2004.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[Pfe95] Frank Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.

[Pfe99] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. In preparation.

[PS98] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*. Springer-Verlag LNCS 1657, 1998. To appear.

[Sch01] Carsten Schürmann. Recursion for higher-order encodings. In Laurent Fribourg, editor, *Proceedings of the Conference on Computer Science Logic (CSL 2001)*, pages 585–599, Paris, France, August 2001. Springer Verlag LNCS 2142.

[Sch05] *The nabla-calculus. Functional programming with higher-order encodings*, Nara, Japan, 2005.

[SDP01] Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, (266):1–57, 2001.

[SP98] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300, Lindau, Germany, July 1998. Springer-Verlag LNAI 1421.

[Sti92] Colin Stirling. *Handbook of Logic in Computer Science*, volume 2, chapter Modal and Temporal Logics, pages 478–563. Oxford, 1992.

[Tah04] Walid Taha. A gentle introduction to multi-stage programming. In Don Batory, Charles Consel, Christian Lengauer, and Martin Odersky, editors, *Domain-specific Program Generation*, LNCS. Springer-Verlag, 2004. to appear.

[TS00] Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.

[WaIW05] Edwin Westbrook and Aaron Stump adn Ian Wehrman. A language based approach to functionally correct imperative programming. In *Intenational Conference on Functional Programming*, 2005.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.