Yale University
Department of Computer Science

A Temporal-Logic Approach to Functional Calculi for Dependent Types and Higher-Order Encodings

Adam Poswolsky

YALEU/DCS/TR-1364
July 2006

A Temporal-Logic Approach to Functional Calculi for Dependent Types and Higher-Order Encodings

Adam Poswolsky*

Abstract

Dependent-types and higher-order encodings lead to concise and elegant representations of complex data structures as evidenced by the success of the logical framework LF [HHP93].

In this work we first design a functional calculus utilizing LF to represent its data objects. To avoid problems commonly associated with using the same function space for both representation (LF objects) and computation, we separate the two as influenced by our previous work [Sch05]. We then exploit the power of the past time connective from temporal logic to design a meta-logic to reason about higher-order abstract syntax. Sample programs that we discuss in this paper include bracket abstraction and a theorem prover.

It is important to note that this technical report is an enhancement/ simplification of a previous technical report [Pos06] where past-time was used as a modal operator in the meta-logic for LF. Here, we use past-time on the meta-meta-level for LF.

1 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 calculus with functions ranging over the usual dependently typed and higher-order encodings of deductive systems in the logical framework LF [HHP93, Pfe99]. This technical report is an enhancement/ simplification of a previous technical report [Pos06] where past-time was used as a modal operator in the meta-logic for LF. Here, we use past-time on the meta-meta-level for LF.

*Department of Computer Science, Yale University, CT

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 challenge in designing a functional calculus for LF is that it must provide two different function spaces [SDP01, Sch05]. In Section 2, we begin by defining the logical framework LF. We then give a first-order meta-logic for LF and prove soundness.

Rather than use temporal logic to represent propositions occurring at different times, we use it to represent *derivations* occurring at different times. To this end our temporal logic serves as a meta logic for the logic defined in Section 2. We define our temporal meta logic in Section 3, discuss properties, and prove soundness. All proofs in this paper have been encoded and verified in Twelf [PS99] except that the underlying calculus is the simply-typed version. Twelf code is digitally available upon request and can also be read in Appendix A.

In Section 4, we discuss how the resulting calculus handles functions over higher-order abstract encodings. The operators for programming with higher-order abstract syntax correspond to admissible rules. This allows us to *statically* reason about when parameters can be accessed. This compile-time guarantee that parameters cannot escape their scope is the driving force of this work. Current research with coverage checking and termination checking will allow us to statically check that all cases are covered and hence cumulatively get a static guarantee that a function is total without the cumbersome caveat of possible runtime exceptions.

Next we derive a logically equivalent, but motivated for programming, system in Section 5. In Section 6 we add a mechanism to conduct case analysis, and finally in Section 7 we add recursion.

We illustrate the resulting $\lambda^{\mathcal{D}}$ calculus, or Delphin, with examples of bracket abstraction and theorem proving in Section 8. We describe related work in Section 9 before we conclude and assess results in Section 10.

Finally we invite the reader to consult the Delphin homepage for more information: www.cs.yale.edu/∼delphin.

2 Meta-Logic for Dependently-Typed LF

We will start with a brief introduction to the Edinburgh logical framework [HHP93], or LF, and then discuss our logic $\mathcal{L}^{\langle\rangle\Pi}$ to reason about LF objects.

We will define syntactic categories of objects M and types A to which we assign the usual logic meaning. In our system we provide a clean divide between the meta-level (computation) and the representation-level (LF). The justification for this design is based on our previous work with the ∇-calculus [Sch05], but simply stated it is necessary because we cannot reason about arbitrary functions on the meta-level so our system syntactically enforces that representation-level functions cannot see computation-level ones. 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 $\langle A \rangle$.

2.1 Preliminaries

Types are defined as: $A, B ::= a \mid A\ M \mid \Pi x : A\,.B$. Function types assign names to their arguments in $\Pi x : A\,.B$. We reserve the notation $A \to B$ as syntactic sugar when the return type is not dependent on its argument. Types may be indexed by objects and we provide the construct $A\ M$ to represent such types. The syntactic category for objects is: $M, N ::= x \mid c \mid M\ N \mid \lambda x : A\,.N$. We write x for variables while a and c are type and object constants, respectively. These constants are provided a priori in a collection $\Sigma ::= \cdot \mid \Sigma, a : \text{type} \mid \Sigma, c : A$. This 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.

With dependent types, not all types are valid. The *kind* system of LF acts as a type system for types. Valid kinds are defined a follows: $K ::= \text{type} \mid \Pi x : A\,.K$. 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 context $\Gamma ::= \cdot \mid \Gamma, x : A$ which assigns types to variables. 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 2.1 (Canonical forms) *Every well-typed object in the dependently-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 2.2 (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.

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

Theorem 2.3 (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 $x_1 : \mathrm{exp} \ldots x_n : \mathrm{exp} \vdash \ulcorner t \urcorner : \mathrm{exp}$.*

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. □

Example 2.4 (Natural deduction calculus) Let $A, B ::= A \Rightarrow B \mid p$ be the language of formulas.

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

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.

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

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.

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

Theorem 2.5 (Substitution [HHP93]) *If* $\Gamma, x : A \vdash B : K$ *and* $\Gamma \vdash M : A$ *then* $\Gamma \vdash [M/x]B : [M/x]K$.

The dependently typed logical framework is elegantly designed for representation. For instance, it permits adequate encodings of proof systems, type systems, and operational semantics that arise in logic design and programming languages theory. Hence, we next design a logic adopting LF to represent its datatypes.

2.2 Meta Logic, $\mathcal{L}^{\langle\rangle\Pi}$

We derive our meta-logic from the sequent calculus for the implicational fragment of propositional logic with $\langle A \rangle$. We write $\Omega \vdash \tau$ for the central derivability judgment and we cannot permit reorderings of Ω because of dependencies. In a slight abuse of notation, we write Ω_1, Ω_2 for the concatenation of two contexts.

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

Our context, $\Omega ::= \cdot \mid \Omega, u \in \tau$ is simply a collection of τ's. As already stated, we write $\langle A \rangle$ to indicate that A is an inhabited type in our logical framework. For functional types we use $\Pi u \in \tau \,.\sigma$ (not to be confused with LF's dependent function type $\Pi x : A \,.B$). Similarly, we use $\Sigma u \in \tau \,.\sigma$ for dependent products. We will also reserve the notation $\tau \supset \sigma$ and $\tau \star \sigma$ as syntactic sugar for non-dependent versions of functions and products, respectively.

Expressions are defined as:

$$\begin{array}{rcl} e, f & ::= & u \mid \text{unit} \mid \text{void } u \\ & & \mid \lambda u \in \tau \,.e \mid \text{let } w = u \cdot e \text{ in } f \\ & & \mid (e_1 \; e_2) \mid \text{let } (w_1, w_2) = u \text{ in } e \\ & & \mid M \end{array}$$

The proof term algebra almost resembles the λ-calculus. Variables are represented by u, w. Instead of application, we use the more awkward looking, yet cleanly motivated let $w = u \cdot e$ in$f$. Similarly for pairs we use let $(w_1, w_2) = u$ in e. As discussed earlier, M is the witness to $\langle A \rangle$.

We write $[e/u]\tau$ as notation for an explicit substitution on the 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'

In addition, and most importantly, is how we reason about LF. We define $[\Omega]_{\text{LF}}$ 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:

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

We call this operation thinning and define it as follows.

Definition 2.6 (Thinning)

$$[\Omega]_{LF} = \begin{cases} \cdot & \textit{if } \Omega = \cdot \\ [\Omega']_{LF}, u : A & \textit{if } \Omega = \Omega', u \in \langle A \rangle \\ [\Omega']_{LF} & \textit{if } \Omega = \Omega', u \in \tau \ \textit{and} \ \tau \neq \langle A \rangle \end{cases}$$

An alternative approach would use two contexts, Ω and Γ, and have an *explicit* construct to introduce declarations into Γ.However, this computational construct would have to be duplicated on the type-level now forcing us to handle computation on the type-level. We have previously studied such a construct taken the form of let $\langle x \rangle = u$ in e, but find the elegance of keeping the type-level computation free justification enough. In addition, with this paradigm we have that dependencies only occur on LF-objects instead of opening a can of worms by permitting dependencies on arbitrary expressions.

By disallowing dependencies on arbitrary expressions we must point out that substitution on types is *not* defined everywhere. If a function is dependent upon its argument, then its argument must be a variable or an LF object, otherwise we cannot express its type unless if we were to add an ominous let on the type-level. For example, from Example 2.4 we can imagine a function $f \in \Pi u \in \langle \mathrm{o} \rangle . \langle \mathrm{nd}\ u \rangle$. This function can only be called on a variable or on an expression M. Assuming that e is neither a variable nor an LF expression, then although we cannot apply f directly to e, we can do (with syntactic sugar defined later) let $x = \ e$ in $(f\ x)$. Here we explicitly provide a disconnect between e and x. The alternative of having the let on the type-level is impractical because that would also entail reasoning about equality over arbitrary expressions to decide if types are equal.

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

Here notice that we just push our substitution to LF substitution of the form $[M/u]_{\mathrm{LF}}N$. This is the only place that dependencies can occur. Just as dependent-types caused us to be concerned with ill-formed LF types, we now need to distinguish between well-formed and ill-formed formulas. We write $\Omega \vdash \tau$ wff for well-formed formulas.

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

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

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

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

The main work is handled in $\langle\rangle\mathsf{wffR}$ which utilizes thinning to appeal to LF's wff relation. All other cases are standard.

Theorem 2.7 (Admissibility of cut$\mathsf{T}$) *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 defined and* $[e/u]\varrho$ *is defined, then* $\Omega_1, [e/u]\Omega_2 \vdash [e/u]\varrho$ wff.

Proof: This proof 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 present $\mathcal{L}^{\langle\rangle\Pi}$. 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.

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

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

$$\text{no rule } \bot\mathsf{R} \qquad \frac{}{\Omega_1, u \in \bot, \Omega_2 \vdash \text{void } u \in \sigma}\,\bot\mathsf{L}$$

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

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

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

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

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

All rules are standard except for $\langle\rangle\mathsf{R}$. This is the rule that drives the system by bridging LF. One would notice that there is no corresponding L rule. The reason for this is that it is handled by the thinning of the context. The other rules handle dependencies standardly but by this design they simply are pushing the dependencies to be handled by $\langle\rangle\mathsf{R}$ and $\langle\rangle\mathsf{wffR}$.

The cut-rule (where substitution is defined) is admissible for this calculus.

$$\frac{\begin{array}{c}[e/u]\Omega_2 \text{ is defined} \quad [e/u]\sigma \text{ is defined} \\ \Omega_1 \vdash e \in \tau \quad \Omega_1, u \in \tau, \Omega_2 \vdash f \in \sigma\end{array}}{\Omega_1, [e/u]\Omega_2 \vdash [e/u]f \in [e/u]\sigma}\,\mathsf{cut}$$

2.3 Preliminaries

Before proving the admissibility of cut, we need some lemmas relating to substitution.

Although the definition of substitution on expressions is determined by the cut proof, we present it first since we will use it to prove some lemmas regarding substitution.

$$
\begin{array}{rcl}
[e/u]u & \equiv & e \\
[(\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'}{w}]([\dfrac{\lambda u' \in \tau\,.e'}{u}]f) \\
[M/u]N & \equiv & [M/u]_{\mathrm{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
[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 & \text{void } w' \\
[(\text{void } w')/u] \\ (\text{let } (w_1, w_2) = u \text{ in } f) & \equiv & \text{void } 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, 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}
$$

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

Lemma 2.8 (Weakening) *Let $\Omega \leq \Omega'$. If $\Omega \vdash e \in \sigma$ then $\Omega' \vdash e \in \sigma$.*

Proof: *By induction on typing judgment.* □

Lemma 2.9 (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 2.10 (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 2.11 (Not Undefined Property)
If $[e/u]\sigma$ is defined and $[f/w]\sigma$ is also defined, then $[e/u]([f/w]\sigma)$ is also defined.

Proof: *Trivial.* □

Lemma 2.12 (Redundant Substitution on τ)

- *If u is not free in τ, then $[e/u]\tau = \tau$*
- *If $\Omega \vdash \tau$ wff and $(u \in \tau) \in \Omega$ and $\tau \neq \langle A \rangle$, then $[e/u]\tau = \tau$*
- *If $e \neq M$ and $[e/u]\tau$ is well-formed, then $[e/u]\tau = \tau$ (or equivalently u does not occur free in τ).*

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

Lemma 2.13 (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 the beginning of this section, Section 2.3).*

Therefore, If u' is not free in τ, then it follows from Redundant Sub Lemma (Lemma 2.12). Therefore, we now assume that u' occurs free in τ. which also means that e' must be of the form 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*

□

2.4 Cut Admissibility Proof

Theorem 2.14 (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* $[e/u]\Omega_2$ *is defined and* $[e/u]\sigma$ *is defined, then* $\Omega_1, [e/u]\Omega_2 \vdash [e/u]f \in [e/u]\sigma$ *for some proof term* $[e/u]f$.
Note that this lemma defines $[e/u]f$*, i.e. substitution on terms of* $\mathcal{L}^{\langle\rangle\Pi}$*, but we presented the results earlier, Section 2.3.*

Proof: By induction on 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 defined.	by Assumption
$[e/u]\tau$ is defined.	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 2.8)
$\Omega_1, [e/u]\Omega_2 \vdash e \in [e/u]\tau$	By Redundant Sub., Since u not free in τ (Lemma 2.12)

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

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

$[\frac{\lambda u' \in \tau\,.e'}{u}]\Omega_2$ is defined. by Assumption

$[\frac{\lambda u' \in \tau\,.e'}{u}]\varrho$ is defined. 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 2.12)

$\mathcal{E}_2' :: \Omega_1, [\frac{\lambda u' \in \tau\,.e'}{u}]\Omega_2, w \in [e/u']\sigma \vdash [\frac{\lambda u' \in \tau\,.e'}{u}]f \in \varrho$
by IH on $\mathcal{D}$ and $\mathcal{E}_2$ and Redundant Sub. Prop. (Lemma 2.12)

$\Omega_1, u' \in \tau \leq \Omega_1, [\frac{\lambda u' \in \tau\,.e'}{u}]\Omega_2, u' \in \tau$ 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 2.8) 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 2.13 and 2.12)

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

$[\frac{[\frac{[\frac{\lambda u' \in \tau\,.e'}{u}]e}{u'}]e'}{w}]\varrho = \varrho$
by Redundant Sub., since w not free in ϱ (Lemma 2.12)

$\mathcal{F} :: \Omega_1, [\frac{\lambda u' \in \tau\,.e'}{u}]\Omega_2 \vdash [\frac{[\frac{[\frac{\lambda u' \in \tau\,.e'}{u}]e}{u'}]e'}{w}]([\frac{\lambda u' \in \tau\,.e'}{u}]f) \in \varrho$
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 2.12)

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

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

$[M/u]\Omega_2$ is defined. by Assumption (or by Def. of Sub)

$[M/u]\langle B \rangle$ is defined. by Assumption (or by Def. of Sub)

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

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

$\mathcal{F}' :: [(\Omega_1, [M/u]_{\mathrm{LF}}\Omega_2)]_{\mathrm{LF}} \vdash [M/u]_{\mathrm{LF}}N : [M/u]_{\mathrm{LF}}B$
by Property of $[\_]_{\mathrm{LF}}$ on $\mathcal{F}$

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

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

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

$[(e_1, e_2)/u]\Omega_2$ is defined. by Assumption

$[(e_1, e_2)/u]\varrho$ is defined. 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 2.12)

$\mathcal{D}_2' :: \Omega_1, [(e_1, e_2)/u]\Omega_2 \vdash e_2 \in [e_1/u']\sigma$
by Weakening (Lemma 2.8)

$[e_1/w_1]\varrho = \varrho$
by Redundant Sub., since w_1 not free in ϱ (Lemma 2.12)

$[e_1/w_1]([w_1/u']\sigma) = [e_1/u']\sigma$
By Prop. of Sub., Distributivity and Redundancy (on w_1) (Lemmas 2.13 and 2.12)

$\mathcal{F}_2 :: \Omega_1, [(e_1, e_2)/u]\Omega_2, w_2 \in [e_1/u']\sigma$
$\quad \vdash [e_1/w_1]([(e_1, e_2)/u]f) \in \varrho$
by IH on $\mathcal{D}_1$ and $\mathcal{F}_1$

$[e_2/w_2]\varrho = \varrho$
by Redundant Sub., since w_2 not free in ϱ (Lemma 2.12)

$\Omega_1, [(e_1, e_2)/u]\Omega_2$
$\quad \vdash [e_2/w_2]([e_1/w_1]([(e_1, e_2)/u]f)) \in \varrho$
by IH on $\mathcal{D}_2'$ and $\mathcal{F}_2$

$\Omega_1, [(e_1, e_2)/u]\Omega_2$
$\quad \vdash [e_2/w_2]([e_1/w_1]([(e_1, e_2)/u]f))$
$\quad \in [(e_1, e_2)/u]\varrho$ by Redundant Sub (Lemma 2.12)

Now we handle commutative conversion on the left.

Case: $\mathcal{D} = \dfrac{(w \in \tau) \in \Omega_1}{\Omega_1 \vdash w \in \tau}\,\mathsf{ax}$ $\qquad \mathcal{E} = \Omega_1, u \in \tau, \Omega_2 \vdash e \in \sigma$

$[w/u]\Omega_2$ is defined. by Assumption
$[w/u]\sigma$ is defined. by Assumption
$\Omega_1, [w/u]\Omega_2 \vdash$ rename u to w in $e \in [w/u]\sigma$
by variable renaming

Case:

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

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

$[(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\Omega_2$ is defined. by Assumption
$[(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\varrho'$ is defined. by Assumption
$\mathcal{F}_0 :: (u' \in (\Pi u'' \in \tau\,.\sigma))$
$\quad \in \Omega_1, [(\text{let } w = u' \cdot\ e' \text{ in } f)/u]\Omega_2$ 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 2.8) 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 2.8) 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 2.9

Case:

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

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

$[(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\Omega_2$ is defined.
by Assumption

$[(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\varrho'$ is defined.
by Assumption

$\mathcal{F}_0 :: (u' \in (\Sigma u'' \in \tau\,.\sigma))$
$\quad \in \Omega_1, [(\text{let } (w_1, w_2) = u' \text{ in } f)/u]\Omega_2$ 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 2.8) 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 2.9

Case:

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

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

$[(\text{void } w')/u]\Omega_2$ is defined.	by Assumption
$[(\text{void } w')/u]\varrho$ is defined.	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.
$\Omega_1, \Omega_2 \vdash \text{void } w' \in \varrho$	by $\bot\mathsf{L}$
$\Omega_1, [(\text{void } w')/u]\Omega_2 \vdash \text{void } w' \in [(\text{void } w')/u]\varrho$	by Above

Case:

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

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

$[(\text{void } w')/u]\Omega_2$ is defined.	by Assumption
$[(\text{void } w')/u]\varrho$ is defined.	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.
$\Omega_1, \Omega_2 \vdash \text{void } w' \in \varrho$	by $\bot\mathsf{L}$
$\Omega_1, [(\text{void } w')/u]\Omega_2 \vdash \text{void } w' \in [(\text{void } w')/u]\varrho$	by Above

Case:

$$\mathcal{D} = \frac{(w \in \bot) \in \Omega_1}{\Omega_1 \vdash \text{void } w \in \bot}\ \bot\mathsf{L}$$

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

$[(\text{void } w)/u]\Omega_2$ is defined.	by Assumption
$[(\text{void } w)/u]\sigma$ is defined.	by Assumption
$\Omega_1, [(\text{void } w)/u]\Omega_2 \vdash \text{void } w \in \sigma$	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:

$$\mathcal{D} = \frac{(w \in \bot) \in \Omega_1}{\Omega_1 \vdash \text{void } w \in \tau}\ \bot\mathsf{L}$$

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

$[(\text{void } w)/u]\Omega_2$ is defined.	by Assumption
$[(\text{void } w)/u]\langle A \rangle$ is defined.	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 2.10
$\Omega_1, [(\text{void } w)/u]\Omega_2 \vdash M \in [(\text{void } w)/u]\langle A \rangle$	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 defined.	by Assumption
$[e/u]\sigma$ is defined.	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

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

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

$[e/u]\Omega_2$ is defined.	by Assumption
$[e/u]\sigma$ is defined.	by Assumption
$(w \in \bot) \in (\Omega_1, [e/u]\Omega_2)$	since $u \neq w$ and Prop. of Sub.
$\Omega_1, [e/u]\Omega_2 \vdash \text{void } w \in [e/u]\sigma$	by $\bot\mathsf{L}$

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

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

$[e/u]\Omega_2$ is defined. by Assumption

$[e/u](\Pi u'' \in \tau' .\sigma))$ is defined. 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: $\mathcal{D} = \Omega_1 \vdash e \in \tau$

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

$[e/u]\Omega_2$ is defined. by Assumption

$[e/u]\varrho$ is defined. by Assumption

$\mathcal{F}_0 :: (u' \in (\Pi u'' \in [e/u]\tau' . [e/u]\sigma')) \in (\Omega_1, [e/u]\Omega_2)$
since $u \neq u'$ and Def. of Sub.

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

$[e/u]([e'/u'']\sigma')$ is defined.
by Not Undefined Prop. (Lemma 2.11)

$[e/u]([e'/u'']\sigma') = [([e/u]e')/u'']([e/u]\sigma')$
by Sub. Distribute Prop. (Lemma 2.13)

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

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

$[e/u]\Omega_2$ is defined. by Assumption

$[e/u](\Sigma u' \in \sigma_1 \,.\sigma_2)$ is defined. 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 defined.
by Not Undefined Prop. (Lemma 2.11)

$[e/u]([e_1/u']\sigma_2) = [([e/u]e_1)/u']([e/u]\sigma_2)$
by Sub. Distribute Prop. (Lemma 2.13)

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

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

$[e/u]\Omega_2$ is defined. by Assumption

$[e/u]\varrho$ is defined. by Assumption

$\mathcal{F}_0 :: (u' \in (\Sigma u'' \in [e/u]\tau' . [e/u]\sigma')) \in (\Omega_1, [e/u]\Omega_2)$
since $u \neq u'$ and Def. of Sub.

$[e/u]([w_1/u'']\sigma')$ is defined.
by Not Undefined Prop. (Lemma 2.11)

$[e/u]([w_1/u'']\sigma') = [w_1/u''][e/u]\sigma')$
by Def. of Sub. and Sub. Distribute Prop. (Lemma 2.13)

$\mathcal{F}_1 :: \Omega_1, [e/u]\Omega_2, w_1 \in [e/u]\tau', w_2 \in [w_1/u''][e/u]\sigma'$
$\vdash [e/u]f \in [e/u]\varrho$
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 $\Sigma\mathsf{L}$

□

2.5 Result

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

Theorem 2.15 (Cut-Elimination) *Let $\Omega \vdash e \in \tau$ be a derivation in $\mathcal{L}^{\langle\rangle\Pi\mathsf{cut}}$. Then there exists an e' such that $\Omega \vdash e' \in \tau$ in $\mathcal{L}^{\langle\rangle\Pi}$.*

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

Corollary 2.16 (Soundness) *The logics $\mathcal{L}^{\langle\rangle\Pi}$ and $\mathcal{L}^{\langle\rangle\Pi\mathsf{cut}}$ are sound.*

2.6 Curry-Howard Isomorphism, $\lambda^{\langle\rangle\Pi}$

Our sequent-calculus 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.

The operational behavior of the $\lambda^{\langle\rangle\Pi}$-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 2.14) as equivalences on proof terms. A detailed table can be found in Section 2.3 but we repeat some of the interesting equivalences below.

$$
\begin{array}{rcl}
[e/u]w & \equiv & \begin{cases} e & \text{if } u = w \\ w & \text{otherwise} \end{cases} \\
[(\lambda u' \in \tau\,.e')/u] & & \\
(\text{let } w = u \cdot e \text{ in } f) & \equiv & [\dfrac{[\frac{\lambda u' \in \tau\,.e'}{u}]e}{u'}]e'}{w}](\dfrac{\lambda u' \in \tau\,.e'}{u}]f) \\
[\dfrac{(\text{let } w = u' \cdot e' \text{ in } f)}{u}]e & \equiv & \text{let } w = u' \cdot e' \text{ in } [f/u]e \\
[(e_1, e_2)/u] & \equiv & [e_2/w_2]([e_1/w_1] \\
(\text{let } (w_1, w_2) = u \text{ in } f) & & ([(e_1, e_2)/u]f)) \\
[M/u]N & \equiv & [M/u]_{\mathrm{LF}} N
\end{array}
$$

At first glance this definition of substitution looks needlessly complicated. However, it is simply because we are using a sequent-calculus instead of natural deduction.

Definition 2.17 (Values) *The following syntactic category denotes all values of the $\lambda^{\langle\rangle\Pi}$-calculus: $v ::=\mid \lambda u \in \tau\,.e \mid \text{unit} \mid M \mid (v_1, v_2)$.*

With this computational view of $\mathcal{L}^{\langle\rangle\Pi}$ we can show that for every expression e can always be reduced to a value v. We define reduction as the reflexive, transitive closure of $\equiv$. This property is called strong normalization.

Theorem 2.18 (Strong Normalization) *If $\cdot \vdash e \in \tau$ then e reduces to a value v.*

Therefore we see that we can interpret the judgments of $\mathcal{L}^{\langle\rangle\Pi}$ in a computational manner. We will use $\lambda^{\langle\rangle\Pi}$ when we refer to the system computationally. The relationship between a logic and its corresponding computational behavior is also called the Curry-Howard isomorphism.

Finally, we point out that although substitution on types is not defined everywhere, it is total for substitutions of values.

Theorem 2.19 (Completeness of Substitution on Values)
$[v/u]\tau$ is always defined.

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

3 Past-Time Temporal Logic

We now present our temporal meta logic to reason about *derivations* of $\mathcal{L}^{\langle\rangle\Pi}$ at different times. Here we utilize a past-time modal operator $\ominus$ over the types τ of $\mathcal{L}^{\langle\rangle\Pi}$.

$$\kappa, \alpha ::= \ominus\kappa \mid \tau$$

We write $\Phi \Vdash \kappa$ for the central derivability judgment and adopt the shorthand $\ominus^i\tau$ to indicate there are i leading $\ominus$'s to τ. Our context, $\Phi ::= \cdot \mid \Phi, u \in \kappa$ is simply a collection of κ's, . Time travel is defined with respect to adding and removing $\ominus$'s to all hypothesis in Φ. Namely, $\Phi^{-\ominus}$ 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 (MovePast)

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

Similarly, we define $\Phi^{+\ominus}$ to take us one step to the future by adding a $\ominus$ to everything in Φ. Finally, we also define a thinning operation $[\Phi]$ that strips all $\ominus$'s from Φ to give us a context of the form Ω.

Definition 3.2 (Temporal Thinning)

$$[\Phi] = \begin{cases} \cdot & \textit{if } \Phi = \cdot \\ [\Phi'], u \in \tau & \textit{if } \Phi = \Phi', u \in \ominus^i \tau \end{cases}$$

It is important to discuss why we create a meta logic for $\mathcal{L}^{\langle\rangle\Pi}$ instead of just adding a modal $\ominus$ operator directly to $\mathcal{L}^{\langle\rangle\Pi}$. The former uses time to reason over *derivations* whilst the latter would use time to reason over *propositions*. For the use of Higher-Order Abstract Syntax, we need to reason about entire derivations to argue that parameters do not escape their scope as will be shown in Section 4. The meaning of $\ominus\tau$ is that τ is derivable under the logic $\mathcal{L}^{\langle\rangle\Pi}$ without making use of any hypothesis that does not exist in the past.

Alternatively, 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 will call this logic $\mathcal{L}^{\langle\rangle\Pi\ominus}$ (or $\lambda^{\langle\rangle\Pi\ominus}$ as a calculus) and it comprises just two judgments.

$$\frac{\Phi^{-\ominus} \Vdash e \in \kappa}{\Phi \Vdash \text{prev } e \in \ominus\kappa}\ominus\mathsf{R} \qquad \frac{[\Phi] \vdash e \in \tau}{\Phi \Vdash e \in \tau}\,\mathsf{bridge}$$

This meta logic is very minimal (although it will be extended shortly) and is notably missing constructs for abstraction and application. However, we are designing this logic to reason about the derivability of $\mathcal{L}^{\langle\rangle\Pi}$ and the study of functions at this meta-meta-level are left for future research. The $\ominus\mathsf{R}$ rule allows us to travel back in time and the bridge rule ties $\mathcal{L}^{\langle\rangle\Pi}$ into our system.

Theorem 3.3 (Future)
If $\Phi^{+\ominus} \Vdash e \in \ominus\kappa$, then there exists an e' such that $\Phi \vdash e' \in \kappa$.

Proof: *By induction on the typing judgment of* $\mathcal{L}^{\langle\rangle\Pi\ominus}$ *and* $\mathcal{L}^{\langle\rangle\Pi}$. □

Thus, the following rule is admissible, which also serves as the elimination rule for $\ominus$.

$$\frac{\Phi^{+\ominus} \Vdash e \in \ominus\kappa}{\Phi \Vdash \text{next } e \in \kappa}\,\mathsf{future}$$

We first need to show that if κ is derivable in the past then it is also derivable now. We call this shifting and due to our design we can also show that the proof-term stays the same. Since we are dealing with dependent-types, this property proves to be crucial.

Lemma 3.4 (Shifting)

- *If* $\Phi^{-\ominus} \vdash e \in \kappa$ *then* $\Phi \Vdash e \in \kappa$.
- *If* $\Phi \vdash \text{prev}^i\ e \in \ominus^i\tau$ *then* $[\Phi] \vdash e \in \tau$.

Proof: *By induction on typing judgment of* $\mathcal{L}^{\langle\rangle\Pi\ominus}$ *and* $\mathcal{L}^{\langle\rangle\Pi}$. □

Recall that since we are dealing with dependent-types we need to extend our definition of substitution for κ.

$$\begin{array}{rcl} [(\text{prev } e)/u](\ominus\kappa) & = & \ominus([e/u]\kappa) \\ [(\text{prev } e)/u]\tau & = & [e/u]\tau \\ [e/u](\ominus\kappa), u \text{ not free in } \kappa & = & \ominus\kappa \\ \text{None of the above match, } [e/u]\kappa & = & \text{undefined} \end{array}$$

We also need to extend our definition of well-formedness with

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

Before proving the admissibility of cut, we update Weakening and our Redundant Substitution lemmas.

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

Lemma 3.5 (Weakening) *Let* $\Phi \leq \Phi'$. *If* $\Phi \vdash e \in \kappa$ *then* $\Phi' \vdash e \in \kappa$.

Proof: *By induction on typing judgment.* □

Lemma 3.6 (Redundant Substitution on κ)

- *If u is not free in κ, then $[e/u]\kappa = \kappa$*
- *If $\Omega \vdash \kappa$ wff and $(u \in \kappa) \in \Omega$ and $\kappa \neq \ominus^k\langle A\rangle$, then $[e/u]\kappa = \kappa$*
- *If $e \neq \mathrm{prev}^k\ M$ and $[e/u]\kappa$ is well-formed, then $[e/u]\kappa = \kappa$ (or equivalently u does not occur free in κ).*

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

Theorem 3.7 (Admissibility of cut$\mathsf{T}$) *If $\mathcal{D} :: \Phi_1 \vdash e \in \kappa$ and $\mathcal{E} :: \Phi_1, u \in \kappa, \Phi_2 \vdash \kappa'$ wff and $[e/u]\Phi_2$ is defined and $[e/u]\kappa'$ is defined, then $\Phi_1, [e/u]\Phi_2 \vdash [e/u]\kappa'$ wff.*

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

Finally we show that the following cut rule is admissible:

$$\frac{\begin{array}{c}[e/u]\Phi_2 \text{ is defined} \quad [e/u]\kappa' \text{ is defined} \\ \Phi_1 \Vdash e \in \kappa \quad \Phi_1, u \in \kappa, \Phi_2 \vdash f \in \kappa'\end{array}}{\Phi_1, [e/u]\Phi_2 \Vdash [e/u]f \in [e/u]\kappa'}\ \mathsf{cut}$$

Theorem 3.8 (Admissibility of cut) *If* $\mathcal{D} :: \Phi_1 \Vdash e \in \kappa$ *and* $\mathcal{E} :: \Phi_1, \kappa, \Phi_2 \Vdash f \in \kappa'$ *and* $[e/u]\Phi_2$ *is defined and* $[e/u]\kappa'$ *is defined, then* $\Phi_1, [e/u]\Phi_2 \Vdash [e/u]f \in [e/u]\kappa'$.
Note that this lemma defines $[e/u]f$*, i.e. substitution on terms of* $\mathcal{L}^{\langle\rangle\Pi\ominus}$.

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

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 :: {\Phi_1}^{-\ominus} \Vdash e \in \kappa}{\Phi_1 \Vdash \text{prev } e \in \ominus\kappa}\ominus\mathsf{R}$

$$\mathcal{E} = \frac{\mathcal{E}_1 :: (\Phi_1, u \in \ominus\kappa, \Phi_2)^{-\ominus} \Vdash f \in \kappa'}{\Phi_1, u \in \ominus\kappa, \Phi_2 \Vdash \text{prev } f \in \ominus\kappa'}\ominus\mathsf{R}$$

$\mathcal{E}_1' :: {\Phi_1}^{-\ominus}, u \in \kappa, {\Phi_2}^{-\ominus} \Vdash f \in \kappa'$
by Property of $(\_)^{-\ominus}$ on $\mathcal{E}_1$

$\mathcal{F} :: {\Phi_1}^{-\ominus}, [e/u]{\Phi_2}^{-\ominus} \Vdash [e/u]f \in [e/u]\kappa'$
by Induction Hypothesis on $\mathcal{D}_1$ and $\mathcal{E}_1'$

$\mathcal{F}' :: (\Phi_1, [(\text{prev } e)/u]\Phi_2)^{-\ominus} \Vdash [e/u]f \in [e/u]\kappa'$
by Def. of Sub.

$\mathcal{F}'' :: \Phi_1, [(\text{prev } e)/u]\Phi_2 \Vdash \text{prev } ([e/u]f) \in [(\text{prev } e)/u]\kappa'$
by $\ominus$R and Def. of Sub.

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 :: {\Phi_1}^{-\ominus} \Vdash \text{prev}^i\ e \in \ominus^i\sigma}{\Phi_1 \Vdash \text{prev}^{i+1}\ e \in \ominus^{i+1}\sigma}\ominus\mathsf{R}$

$$\mathcal{E} = \frac{\mathcal{E}_1 :: [\Phi_1, u \in \ominus^{i+1}\sigma, \Phi_2] \vdash f \in \tau}{\Phi_1, u \in \ominus^{i+1}\sigma, \Phi_2 \Vdash f \in \tau}\,\mathsf{bridge}$$

$\mathcal{D}' :: [\Phi_1] \vdash e \in \sigma$
by Shifting (Lemma 3.4) on $\mathcal{D}$

$\mathcal{E}_1' :: [\Phi_1], u \in \sigma, [\Phi_2] \vdash f \in \tau$
by Property of $[\_]$ on $\mathcal{E}_1$

$\mathcal{F} :: [\Phi_1], [e/u][\Phi_2] \vdash [e/u]f \in [e/u]\tau$
by Admissibility of Cut in $\mathcal{L}^{\langle\rangle\Pi}$ (Theorem 2.14)

$\mathcal{F}' :: [\Phi_1, [(\text{prev}^{i+1}\ e)/u]\Phi_2] \vdash [e/u]f \in [e/u]\tau$
by Def. of Sub.

$\mathcal{F}'' :: \Phi_1, [(\text{prev}^{i+1}\ e)/u]\Phi_2 \Vdash [e/u]f \in [(\text{prev}^{i+1}\ e)/u]\tau$
by bridge and Def. of Sub.

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 :: [\Phi_1] \vdash e \in \tau}{\Phi_1 \Vdash e \in \tau}\,\mathsf{bridge}$

$\mathcal{E} = \dfrac{\mathcal{E}_1 :: [\Phi_1, u \in \tau, \Phi_2] \vdash f \in \sigma}{\Phi_1, u \in \tau, \Phi_2 \Vdash f \in \sigma}\,\mathsf{bridge}$

$\mathcal{E}_1' :: [\Phi_1], u \in \tau, [\Phi_2] \vdash f \in \sigma$

by Property of $[\_]$ on $\mathcal{E}_1$

$\mathcal{F} :: [\Phi_1], [e/u][\Phi_2] \vdash [e/u]f \in [e/u]\tau$

by Admissibility of Cut in $\mathcal{L}^{\langle\rangle\Pi}$ (Theorem 2.14)

$\mathcal{F}' :: [\Phi_1, [e/u]\Phi_2] \vdash [e/u]f \in [e/u]\tau$

by Def. of Sub.

$\mathcal{F}'' :: \Phi_1, [e/u]\Phi_2 \Vdash [e/u]f \in [e/u]\tau$

by bridge

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 :: [\Phi_1] \vdash e \in \tau}{\Phi_1 \Vdash e \in \tau}\,\mathsf{bridge}$

$\mathcal{E} = \dfrac{\mathcal{E}_1 :: (\Phi_1, u \in \tau, \Phi_2)^{-\ominus} \Vdash f \in \kappa}{\Phi_1, u \in \tau, \Phi_2 \Vdash \mathrm{prev}\ f \in \ominus\kappa}\,\ominus\mathsf{R}$

$\mathcal{E}_1' :: (\Phi_1, \Phi_2)^{-\ominus} \Vdash f \in \kappa$

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

$\mathcal{F} :: \Phi_1, \Phi_2 \Vdash \mathrm{prev}\ f \in \kappa$

by $\ominus\mathsf{R}$

$[e/u]\Phi_2 = \Phi_2 \quad \text{and} \quad [e/u]\kappa = \kappa$

By Redundant Substitution Lemma (Lemma 3.6) since u cannot occur free by well-formedness of $\mathcal{E}_1'$

$\mathcal{F}' :: \Phi_1, [e/u]\Phi_2 \Vdash \mathrm{prev}\ f \in [e/u]\kappa$

by Above

□

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

Corollary 3.9 (Soundness) *The logics $\mathcal{L}^{\langle\rangle\Pi\ominus}$ and $\mathcal{L}^{\langle\rangle\Pi\ominus\mathsf{cut}}$ are sound.*

3.1 Operational Behavior

As we did in Section 2.6 we interpret the proof of the admissibility of cut as equivalences on proof terms.

$$\begin{array}{rcl} [(\text{prev } e)/u](\text{prev } f) & \equiv & \text{prev } ([e/u]f) \\ f \neq \text{prev } f', [(\text{prev } e)/u]f & \equiv & [e/u]f \\ e \neq \text{prev } e', [e/u](\text{prev } f) & \equiv & \text{prev } f \end{array}$$

Finally, we extend the definition of Values in Definition 2.17 to include prev v.

Definition 3.10 (Values) *The following syntactic category denotes all values of the $\lambda^{\langle\rangle\Pi\ominus}$-calculus:* $v ::= | \ \lambda u \in \tau\,.e \mid \text{unit} \mid M \mid (v_1, v_2) \mid \text{prev } v$.

3.2 Bridging bidirectionally

It is clear to see that $\mathcal{L}^{\langle\rangle\Pi\ominus}$ can be used to reason about $\mathcal{L}^{\langle\rangle\Pi}$ through the bridge rule. However, inside $\mathcal{L}^{\langle\rangle\Pi}$ it is completely unaware of $\mathcal{L}^{\langle\rangle\Pi\ominus}$. Since we are reasoning about derivations this was the natural thing to do. We have defined $\mathcal{L}^{\langle\rangle\Pi\ominus}$ as a meta-logic for $\mathcal{L}^{\langle\rangle\Pi}$.

However, it is desirable from within $\mathcal{L}^{\langle\rangle\Pi}$ to make some meta arguments. Recall that Ω is a valid context of the form Φ and consider the following rule.

$$\frac{\Omega \Vdash e \in \tau}{\Omega \vdash e \in \tau}\ \mathsf{meta}$$

Recall that $\Omega \Vdash \tau$ is a meta-level ($\mathcal{L}^{\langle\rangle\Pi\ominus}$) statement that τ is derivable in $\mathcal{L}^{\langle\rangle\Pi}$ in the present. Therefore, in prose this rule is stating that if $\mathcal{L}^{\langle\rangle\Pi\ominus}$ can show that τ is derivable in $\mathcal{L}^{\langle\rangle\Pi}$, then we can conclude that there exists a proof in $\mathcal{L}^{\langle\rangle\Pi}$. Therefore, it is not surprising that this is an admissible rule. Notice as well that both bridge (in $\mathcal{L}^{\langle\rangle\Pi\ominus}$) and now meta (in $\mathcal{L}^{\langle\rangle\Pi}$) do not change the proof-term.

Lemma 3.11 (Admissible meta) *If $\Omega \Vdash e \in \tau$ then $\Omega \vdash e \in \tau$*

Proof: *By Inversion on $\Omega \Vdash e \in \tau$ using* bridge □

This concludes the logical motivation for this paper. The remainder is dedicated to interpreting the corresponding $\lambda^{\langle\rangle\Pi\ominus}$ calculus as a functional programming language. We first add an operator for programming with

higher-order encodings, which is just an admissible rule. We will then simplify our logic to an equivalent form called Delphin ($\lambda^{\mathcal{D}}$) and 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.

4 Higher-Order Abstract Syntax

The temporal features of the $\lambda^{\langle\rangle\Pi\ominus}$-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 $\lambda^{\langle\rangle\Pi\ominus}$-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 Φ 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 4.1 (Strengthen)
If $\Phi, u \in \tau \Vdash e \in \ominus\kappa$, *then* $\Omega \Vdash e \in \ominus\kappa$

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

Thus, the following rule is admissible.

$$\frac{\Phi, u \in \tau \Vdash e \in \ominus\kappa}{\Omega \Vdash \nu u \in \tau\, .e \in \ominus\kappa}\ \mathsf{new}$$

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

Thus the final version of our logic, $\mathcal{L}^{\langle\rangle\Pi\ominus\mathsf{cut}+}$, is sound and defined as follows:

$$\frac{\Phi^{-\ominus} \Vdash e \in \kappa}{\Phi \Vdash \text{prev } e \in \ominus\kappa}\ominus\mathsf{R} \quad \frac{[\Phi] \vdash e \in \tau}{\Phi \Vdash e \in \tau}\,\mathsf{bridge}$$

$$\frac{\Phi^{+\ominus} \Vdash e \in \ominus\kappa}{\Phi \Vdash \text{next } e \in \kappa}\,\mathsf{future} \quad \frac{\Phi, u \in \tau \Vdash e \in \ominus\kappa}{\Omega \Vdash \nu u \in \tau\,.e \in \ominus\kappa}\,\mathsf{new}$$

$$\frac{\begin{array}{c}[e/u]\Phi_2 \text{ is defined} \quad [e/u]\kappa' \text{ is defined} \\ \Phi_1 \Vdash e \in \kappa \quad \Phi_1, u \in \kappa, \Phi_2 \vdash f \in \kappa'\end{array}}{\Phi_1, [e/u]\Phi_2 \Vdash [e/u]f \in [e/u]\kappa'}\,\mathsf{cut}$$

In addition, the logic we use in bridge is $\mathcal{L}^{\langle\rangle\Pi\mathsf{cut}}$ with the admissible meta rule.

5 Delphin

Our design has led us to a three-tiered system with an explicit bridge and meta rule, called $\mathcal{L}^{\langle\rangle\Pi\ominus\mathsf{cut}+}$. In addition we have three contexts to reason about. We have Γ at the bottom, Ω at its meta-level, and Φ at its meta-meta-level. However, since both logics can communicate, we present an equivalent simplified version so we don't need bridge and meta and in addition get rid of dealing with Ω. Since we are also shifting our focus to computation we take the liberty of replacing the sequent let version of application with the more natural version, $e_1\ e_2$.

Therefore, we present here the equivalent logic $\mathcal{L}^{\mathcal{D}}$ (or calculus $\lambda^{\mathcal{D}}$), which will be the focus for the rest of this paper. Note that the purpose of this section is twofold – it both summarizes the logical system to this point and simplifies it into one judgment, $\Phi \vdash^{d} e \in \kappa$

Types remain unchanged and are:

$$\begin{array}{lcl}\kappa, \alpha & ::= & \ominus\kappa \mid \tau \\ \tau, \sigma, \varrho & ::= & \top \mid \bot \mid \Pi u \in \tau\,.\sigma \mid \Sigma u \in \tau\,.\sigma \mid \langle A \rangle\end{array}$$

Expressions are now:

$$\begin{array}{lcl}e, f & ::= & u \mid \text{unit} \mid \text{void } e \mid \lambda u \in \tau\,.e \mid e_1\ e_2 \mid \text{prev } e \mid \text{next } e \\ & & \mid (e_1, e_2) \mid \text{let } (w_1, w_2) = e \text{ in } f \mid \nu u \in \tau\,.e \mid M\end{array}$$

We collapse the logics together by having the ax rule do the work of merge in transforming the context. Note that we can interpret the ax rule as shifting all hypothesis to the present thereby enforcing that only the temporal rules have any access to time information, which is our desired semantics.

$$\frac{(u \in \ominus^i \tau) \text{ in } \Phi}{\Phi \Vdash^d u \in \tau}\ \mathsf{ax} \qquad \frac{}{\Phi \Vdash^d \text{unit} \in \top}\ \mathsf{top}$$

$$\frac{\Phi, u \in \tau \Vdash^d e \in \sigma}{\Phi \Vdash^d \lambda u \in \tau\,.e \in \Pi u \in \tau\,.\sigma}\ \mathsf{lam}$$

$$\frac{\Phi \Vdash^d e_1 \in (\Pi u' \in \tau\,.\sigma) \quad \Phi \Vdash^d e_2 \in \tau}{\Phi \Vdash^d e_1\ e_2 \in [e_2/u']\sigma}\ \mathsf{app}$$

$$\frac{\Phi \Vdash^d e_1 \in \tau \quad \Phi \Vdash^d e_2 \in [e_1/u]\sigma}{\Phi \Vdash^d (e_1, e_2) \in \Sigma u \in \tau\,.\sigma}\ \mathsf{pair}$$

$$\frac{\Phi \Vdash^d e \in (\Sigma w_1 \in \tau\,.\sigma) \quad \Phi, w_1 \in \tau, w_2 \in \sigma \Vdash^d f \in \varrho}{\Phi \Vdash^d \text{let } (w_1, w_2) = e \text{ in } f \in \varrho}\ \mathsf{openPair}$$

$$\frac{\Phi \Vdash^d e \in \bot}{\Phi \Vdash^d \text{void } e \in \sigma}\ \mathsf{bot} \qquad \frac{[\Phi]_{\text{LF}} \vdash M : A}{\Phi \Vdash^d M \in \langle A \rangle}\ \langle\rangle\mathsf{I}$$

$$\frac{\Phi^{-\ominus} \Vdash^d e \in \kappa}{\Phi \Vdash^d \text{prev } e \in \ominus\kappa}\ \mathsf{past} \qquad \frac{\Phi^{+\ominus} \Vdash^d e \in \ominus\kappa}{\Phi \Vdash^d \text{next } e \in \kappa}\ \mathsf{future}$$

$$\frac{\Phi, u \in \tau \Vdash^d e \in \ominus\kappa}{\Phi \Vdash^d \nu u \in \tau\,.e \in \ominus\kappa}\ \mathsf{new}$$

It is important to note that our openPair rule which corresponds to ΣL 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 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. Now if LF

supported pairs, then we would just define our thinning operation to cast it appropriately. However, LF does not support pairing. We could alternatively adopt a fst and snd notation and have the thinning operation *extend* the context by opening up all the pairs, but this would just be a hack and make things needlessly more convoluted.

Substitution on types remains unchanged (only defined for LF terms) and is as follows:

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

Currently LF thinning is only defined on Ω but it is easy to combine the definition of $[\Phi]$ and $[\Omega]_{\mathrm{LF}}$ into $[\Phi]_{\mathrm{LF}}$ below.

Definition 5.1 (Thinning for Natural Deduction)

$$[\Phi]_{LF} = \begin{cases} \cdot & \text{if } \Phi = \cdot \\ [\Phi']_{LF}, u : A & \text{if } \Phi = \Phi', u \in \ominus^i\langle A\rangle \\ [\Phi']_{LF} & \text{if } \Phi = \Phi', u \in \tau \text{ and } \tau \neq \ominus^i\langle A\rangle \end{cases}$$

5.1 Equivalence Proof Details

First we need to prove some preliminaries.

Lemma 5.2 (DeShifting) *If* $[\Phi] \vdash^d e \in \tau$ *then* $\Phi \vdash^d e \in \tau$

Proof: *By Induction on* $[\Phi] \vdash^d e \in \tau$*. Note that the only interesting case is* ax *where it demonstrates this property since it throws out all time information (all* $\ominus$*'s).* □

Lemma 5.3 (Weakening) *Let* $\Phi \leq \Phi'$*. If* $\Phi \vdash^d e \in \kappa$ *then* $\Phi' \vdash^d e \in \kappa$*.*

Proof: *By induction on typing judgment.* □

In order to prove that $\mathcal{L}^{\langle\rangle\Pi\ominus\mathsf{cut}+}$ is equivalent to $\mathcal{L}^{\mathcal{D}}$ we proceed by proving this in both directions separately.

Lemma 5.4 ($\mathcal{L}^{\langle\rangle\Pi\ominus\mathsf{cut}+}$ to $\mathcal{L}^{\mathcal{D}}$)

1. *If* $\Omega \vdash e \in \tau$ *then there exists an* e' *such that* $\Omega \Vdash^{d} e' \in \tau$ *and if* $e = M$ *then* $e' = M$.
2. *If* $\Phi \Vdash e \in \kappa$ *then there exists an* e' *such that* $\Phi \Vdash^{d} e' \in \kappa$ *and if* $e = \mathrm{prev}^{i}\ M$ *then* $e' = \mathrm{prev}^{i}\ M$.

Proof: *By Induction on* $\mathcal{D} :: \Omega \vdash e \in \tau$ *for Part 1 and by Induction on* $\mathcal{D} :: \Phi \Vdash e \in \kappa$ *for Part 2. Note that since the* cut *rules are admissible rules from the others, we observe that for any derivation* $\mathcal{D}$*, there exists a* cut-free *derivation. Therefore, we do induction just over* cut-free *derivations.*

Part 1: By Induction on cut-free *derivations* $\mathcal{D} :: \Omega \vdash e \in \tau$.

Case: $\mathcal{D} = \dfrac{(u \in \tau)\ in\ \Omega}{\Omega \vdash u \in \tau}\ \mathsf{ax}$

$\Omega \Vdash^{d} u \in \tau$ *by* ax

Case: $\mathcal{D} = \dfrac{}{\Omega \vdash \mathrm{unit} \in \top}\ \top\mathsf{R}$

$\Omega \Vdash^{d} \mathrm{unit} \in \top$ *by* top

Case: $\mathcal{D} = \dfrac{}{\Omega_1, u \in \bot, \Omega_2 \vdash \text{void } u \in \sigma}\ \bot\mathsf{L}$

$\Omega \vdash^d u \in \bot$ *by* ax
$\Omega \vdash^d \text{void } u \in \sigma$ *by* bot

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

$\Omega, u \in \tau \vdash^d e' \in \sigma$ *by IH on* $\mathcal{D}_1$
$\Omega \vdash^d \lambda u \in \tau\,.e' \in \Pi u \in \tau\,.\sigma$ *by* lam

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

$\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash^d u \in \Pi u' \in \tau\,.\sigma$ *by* ax
$\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash^d e' \in \tau$ *by IH on* $\mathcal{D}_1$
$\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash^d (u\ e') \in [e'/u']\sigma$ *by* app
If $e = M$ *then* $e' = M$ *by IH on* $\mathcal{D}_1$
$[e'/u']\sigma = [e/u']\sigma$ *by Above and Redundant Sub. (Lemma 2.12)*
$\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash^d (u\ e') \in [e/u']\sigma$ *by Above*
$\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2, w \in [e/u']\sigma \vdash^d f' \in \varrho$ *by IH on* $\mathcal{D}_2$
$\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash^d \lambda w \in [e/u']\sigma\,.f' \in \Pi w \in [e/u']\sigma\,.\varrho$ *by* lam
$[(u\ e')/w]\varrho = \varrho$ *By Redundant Sub., Since* w *not free in* ϱ. *(Lemma 2.12)*
$\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \vdash^d (\lambda w \in [e/u']\sigma\,.f')\ (u\ e') \in \varrho$ *by* app

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

$\Omega \Vdash^d e_1' \in \tau$ *by IH on* $\mathcal{D}_1$
If $e_1 = M_1$ *then* $e_1' = M_1$ *by IH on* $\mathcal{D}_1$
$[e_1'/u]\sigma = [e_1/u]\sigma$ *by Above and Redundant Sub. (Lemma 2.12)*
$\Omega \Vdash^d e_2' \in [e_1'/u]\sigma$ *by IH on* $\mathcal{D}_2$
$\Omega \Vdash^d (e_1', e_2') \in \Sigma u \in \tau\,.\sigma$ *by* pair

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1 :: \Omega_1, u \in (\Sigma u' \in \tau\,.\sigma), \Omega_2, w_1 \in \tau, w_2 \in [w_1/u']\sigma \vdash e \in \varrho}{\Omega_1, u \in (\Sigma u' \in \tau\,.\sigma), \Omega_2 \vdash \text{let } (w_1, w_2) = u \text{ in } e \in \varrho}\,\Sigma\mathsf{L}$

$\Omega_1, u \in (\Pi u' \in \tau\,.\sigma), \Omega_2 \Vdash^d u \in \Pi u' \in \tau\,.\sigma$ *by* ax
$\Omega_1, u \in (\Sigma u' \in \tau\,.\sigma), \Omega_2 \Vdash^d u \in (\Sigma u' \in \tau\,.\sigma)$ *by* ax
$\Omega_1, u \in (\Sigma u' \in \tau\,.\sigma), \Omega_2, w_1 \in \tau, w_2 \in [w_1/u']\sigma \Vdash^d e' \in \varrho$ *by IH on* $\mathcal{D}_1$
$\Omega_1, u \in (\Sigma u' \in \tau\,.\sigma), \Omega_2 \Vdash^d \text{let } (w_1, w_2) = u \text{ in } e' \in \varrho$ *by* openPair

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ [\Omega]_{LF} \vdash M : A\end{array}}{\Omega \vdash M \in \langle A \rangle}\,\langle\rangle\mathsf{R}$

$\Omega \Vdash^d M \in \langle A \rangle$ *by* $\langle\rangle\mathsf{I}$ *on* $\mathcal{D}_1$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Omega \Vdash e \in \tau\end{array}}{\Omega \vdash e \in \tau}\,\mathsf{meta}$

$\Phi \Vdash^d e' \in \tau$
and if $e = M$ *then* $e' = M$. *by IH on* $\mathcal{D}_1$

Part 2: By Induction on cut-free *derivations* $\mathcal{D} :: \Phi \Vdash e \in \kappa$.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Phi^{-\ominus} \Vdash e \in \kappa\end{array}}{\Phi \Vdash \text{prev } e \in \ominus\kappa}\ \ominus\mathsf{R}$

$\Phi^{-\ominus} \models^d e' \in \kappa$	*by IH on* $\mathcal{D}_1$
If $e = \text{prev}^i\ M$ *then* $e' = \text{prev}^i\ M$	*by IH on* $\mathcal{D}_1$
$\Phi \models^d \text{prev } e' \in \ominus\kappa$	*by* past
If (prev $e = \text{prev}^i\ M$) *then* (prev $e' = \text{prev}^i\ M$)	*by Above*

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ [\Phi] \vdash e \in \tau\end{array}}{\Phi \Vdash e \in \tau}\ \mathsf{bridge}$

$[\Phi] \models^d e' \in \tau$	*by IH on* $\mathcal{D}_1$
If $e = M$ *then* $e' = M$	*by IH on* $\mathcal{D}_1$
$\Phi \models^d e' \in \tau$	*by DeShifting (Lemma 5.2)*
If $e = \text{prev}^i\ M$ *then* $e' = \text{prev}^i\ M$	*by Above*

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Phi^{+\ominus} \Vdash e \in \ominus\kappa\end{array}}{\Phi \Vdash \text{next } e \in \kappa}\ \mathsf{future}$

$\Phi^{+\ominus} \models^d e' \in \kappa$	*by IH on* $\mathcal{D}_1$
$\Phi \models^d \text{next } e' \in \ominus\kappa$	*by* future

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Phi, u \in \tau \Vdash e \in \ominus\kappa\end{array}}{\Omega \Vdash \nu u \in \tau\, .e \in \ominus\kappa}\ \mathsf{new}$

$\Phi, u \in \tau \models^d e' \in \ominus\kappa$	*by IH on* $\mathcal{D}_1$
$\Phi \models^d \nu u \in \tau\, .e' \in \ominus\kappa$	*by* new

□

Lemma 5.5 ($\mathcal{L}^{\mathcal{D}}$ to $\mathcal{L}^{\langle\rangle\Pi\ominus\mathbf{cut}+}$)
If $\Phi \Vdash^d e \in \kappa$ then there exists an e' such that $\Phi \Vdash e' \in \kappa$ and

- *if $\kappa = \tau$ then there exists an f such that $[\Phi] \vdash f \in \tau$.*
- *if $e = \mathrm{prev}^i\ M$ then $e' = \mathrm{prev}^i\ M$.*

Proof: *By Induction on $\mathcal{D} :: \Phi \Vdash^d e \in \kappa$.*

Case: $\mathcal{D} = \dfrac{(u \in \ominus^i \tau)\ in\ \Phi}{\Phi \Vdash^d u \in \tau}\ \mathsf{ax}$

$[\Phi] \vdash u \in \tau$	*by* ax *and Definition of* $[\_]$
$\Phi \Vdash u \in \tau$	*by* bridge

Case: $\mathcal{D} = \dfrac{}{\Phi \Vdash^d \mathrm{unit} \in \top}\ \mathsf{top}$

$[\Phi] \vdash \mathrm{unit} \in \top$	*by* top
$\Phi \Vdash \mathrm{unit} \in \top$	*by* bridge

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Phi, u \in \tau \Vdash^d e \in \sigma\end{array}}{\Phi \Vdash^d \lambda u \in \tau\, .e \in \Pi u \in \tau\, .\sigma}\ \mathsf{lam}$

$[\Phi], u \in \tau \vdash e' \in \sigma$	*by IH on* $\mathcal{D}_1$ *and Def. of* $[\_]$
$[\Phi] \vdash \lambda u \in \tau\, .e' \in \Pi u \in \tau\, .\sigma$	*by* $\supset$ R
$\Phi \Vdash \lambda u \in \tau\, .e' \in \Pi u \in \tau\, .\sigma$	*by* bridge

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Phi \stackrel{d}{\models} e_1 \in (\Pi u' \in \tau\,.\sigma) & \Phi \stackrel{d}{\models} e_2 \in \tau\end{array}}{\Phi \stackrel{d}{\models} e_1\ e_2 \in [e_2/u']\sigma}\ \mathsf{app}$

$[\Phi] \vdash e_1' \in \Pi u' \in \tau\,.\sigma$	*by IH on* $\mathcal{D}_1$
$[\Phi] \vdash e_2' \in \tau$	*by IH on* $\mathcal{D}_2$
$[\Phi], u \in (\Pi u' \in \tau\,.\sigma) \vdash e_2' \in \tau$	*by Weakening (Lemma 2.8)*
$[\Phi], u \in (\Pi u' \in \tau\,.\sigma) \vdash u \in \Pi u' \in \tau\,.\sigma$	*by* ax
$[\Phi], u \in (\Pi u' \in \tau\,.\sigma) \vdash \text{let } w = u \cdot\ e_2' \text{ in } w \in [e_2'/u']\sigma$	*by* $\supset$ L
$[e_1'/u][e_2'/u']\sigma = [e_2'/u']\sigma$	*by Redundant Sub (Lemma 2.12)*
$[\Phi] \vdash [e_1'/u](\text{let } w = u \cdot\ e_2' \text{ in } w) \in [e_2'/u']\sigma$	*by* cut
If $e_2 = M$ *then* $e_2' = M$	*by IH on* $\mathcal{D}_2$
$[e_2'/u']\sigma = [e_2/u']\sigma$	*by Above and Redundant Sub. (Lemma 2.12)*
$[\Phi] \vdash [e_1'/u](\text{let } w = u \cdot\ e_2' \text{ in } w) \in [e_2/u']\sigma$	*by Above*
$\Phi \Vdash [e_1'/u](\text{let } w = u \cdot\ e_2' \text{ in } w) \in [e_2/u']\sigma$	*by* bridge

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Phi \stackrel{d}{\models} e_1 \in \tau & \Phi \stackrel{d}{\models} e_2 \in [e_1/u]\sigma\end{array}}{\Phi \stackrel{d}{\models} (e_1, e_2) \in \Sigma u \in \tau\,.\sigma}\ \mathsf{pair}$

$[\Phi] \vdash e_1' \in \tau$	*by IH on* $\mathcal{D}_1$
If $e_1 = M$ *then* $e_1' = M$	*by IH on* $\mathcal{D}_1$
$[e_1'/u]\sigma = [e_1/u]\sigma$	*by Above and Redundant Sub. (Lemma 2.12)*
$[\Phi] \vdash e_2' \in [e_1'/u]\sigma$	*by IH on* $\mathcal{D}_2$
$[\Phi] \vdash (e_1', e_2') \in \Sigma u \in \tau\,.\sigma$	*by* ΣR
$\Phi \Vdash (e_1', e_2') \in \Sigma u \in \tau\,.\sigma$	*by* bridge

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Phi \Vdash^d e \in (\Sigma w_1 \in \tau\,.\sigma) & \Phi, w_1 \in \tau, w_2 \in \sigma \Vdash^d f \in \varrho\end{array}}{\Phi \Vdash^d \text{let } (w_1, w_2) = e \text{ in } f \in \varrho}\ \mathsf{openPair}$

$[\Phi] \vdash e' \in (\Sigma w_1 \in \tau\,.\sigma)$	*by IH on* $\mathcal{D}_1$
$[\Phi], w_1 \in \tau, w_2 \in \sigma \vdash f' \in \varrho$	*by IH on* $\mathcal{D}_2$
$[\Phi], u \in (\Sigma w_1 \in \tau\,.\sigma), w_1 \in \tau, w_2 \in \sigma \vdash f' \in \varrho$	*by Weakening (Lemma 2.8)*
$[\Phi] \vdash \text{let } (w_1, w_2) = u \text{ in } f' \in \varrho$	*by* $\Sigma\mathsf{L}$
$\Phi \Vdash \text{let } (w_1, w_2) = u \text{ in } f' \in \varrho$	*by* bridge

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1}{\Phi \Vdash^d \text{void } e \in \sigma}\ \mathsf{bot}$

$[\Phi] \vdash e' \in \bot$	*by IH on* $\mathcal{D}_1$
$[\Phi], u \in \bot \vdash \text{void } u \in \sigma$	*by* $\bot\mathsf{L}$
$[\Phi] \vdash [e'/u](\text{void } u) \in \sigma$	*by* cut
$\Phi \Vdash [e'/u](\text{void } u) \in \sigma$	*by* bridge

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1}{\Phi \Vdash^d M \in \langle A \rangle}\ \langle\rangle\mathsf{I}$

$[\Phi] \vdash M \in \langle A \rangle$	*by* $\langle\rangle\mathsf{R}$ *and Property of* $[\_]$
$\Phi \Vdash M \in \langle A \rangle$	*by* bridge

Case: $\mathcal{D} = \dfrac{\mathcal{D}_1}{\Phi \Vdash^d \text{prev } e \in \ominus\kappa}\ \mathsf{past}$

$\Phi^{-\ominus} \Vdash e' \in \kappa$	*by IH on* $\mathcal{D}_1$
If $e = \text{prev}^i\ M$ *then* $e' = \text{prev}^i\ M$	*by IH on* $\mathcal{D}_1$
$\Phi \Vdash \text{prev } e' \in \ominus\kappa$	*by* $\ominus\mathsf{R}$
If (prev $e = \text{prev}^i\ M$) *then* (prev $e' = \text{prev}^i\ M$)	*by Above*

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Phi^{+\ominus} \Vdash^{d} e \in \ominus\kappa\end{array}}{\Phi \Vdash^{\underline{d}} \text{next } e \in \kappa}\ \mathsf{future}$

$\Phi^{+\ominus} \Vdash e' \in \ominus\kappa$ *by IH on* $\mathcal{D}_1$
$\Phi \Vdash \text{next } e' \in \kappa$ *by* future

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Phi, u \in \tau \Vdash^{d} e \in \ominus\kappa\end{array}}{\Phi \Vdash^{\underline{d}} \nu u \in \tau\,.e \in \ominus\kappa}\ \mathsf{new}$

$\Phi, u \in \tau \Vdash e' \in \ominus\kappa$ *by IH on* $\mathcal{D}_1$
$\Phi \Vdash \nu u \in \tau\,.e' \in \ominus\kappa$ *by* new

□

5.2 Result

Theorem 5.6 (Delphin is Equivalent to $\mathcal{L}^{\langle\rangle\Pi\ominus\mathbf{cut}+}$)
The logic presented here, $\mathcal{L}^{\mathcal{D}}$ is equivalent to $\mathcal{L}^{\langle\rangle\Pi\ominus\mathsf{cut}+}$.

Proof: *By Lemma 5.4 and Lemma 5.5* □

When looking at term-level substitution of $\mathcal{L}^{\langle\rangle\Pi\ominus\mathsf{cut}+}$ 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 here 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. Due to space limitations we omit the operational semantics, but note that any operational semantics respecting the term equivalences (which we get from the proof of the admissibility of cut) is acceptable.
Values in $\lambda^{\mathcal{D}}$ remain the same as in Definition 3.10, and are:

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

By inspection of the cases we see that type-level substitution is always 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 type-level substitution is defined as follows.

Definition 5.7 *Substitution of Delphin Values on Delphin Types.*

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

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 Delphin.

Definition 5.8 *Substitution of Delphin Values on Delphin Terms. Note that when we write "*$\text{prev}^i\ v$*" we refer to the instance where v is stripped of*

all its leading prev*'s.*

$$
\begin{array}{rcl}
[(\text{prev}^i\ v)/u]u' & \equiv & \begin{cases} v & \textit{if } u = u' \\ u' & \textit{otherwise} \end{cases} \\
[v/u]\text{unit} & \equiv & \text{unit} \\
[v/u](\text{void } e) & \equiv & \text{void } (v/u)e \\
[v/u](\lambda u' \in \tau\,.e) & \equiv & \lambda u' \in [v/u]\tau\,.[v/u]e \\
[v/u](e_1\ e_2) & \equiv & ([v/u]e_1)\ ([v/u]e_2) \\
[v/u](\text{prev } e) & \equiv & \begin{cases} \text{prev } ([v'/u]e) & \textit{if } v = \text{prev } v' \\ \text{prev } e & \textit{otherwise} \end{cases} \\
[v/u](\text{next } e) & \equiv & \text{next}([\text{prev } v/u]e) \\
[v/u](e_1, e_2) & \equiv & ([v/u]e_1,\ [v/u]e_2) \\
[v/u]\text{let } (w_1, w_2) = e \textit{ in } f & \equiv & \text{let } (w_1, w_2) = [v/u]e \textit{ in } [v/u]f \\
[v/u](\nu u' \in \tau\,.e) & \equiv & \nu u' \in [v/u]\tau\,.[v/u]e \\
[v/u]N & \equiv & \begin{cases} [M/u]_{LF}N & \textit{if } v = M \\ N & \textit{otherwise} \end{cases}
\end{array}
$$

Therefore, we will define Delphin to utilize an eager operational semantics. With this choice we are guaranteed that substitution is always defined.

6 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 2.2, u will be instantiated during run-time by a canonical object $\Gamma \vdash M : \text{exp}$. As our logical framework possesses the canonical form property, 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).

Therefore we add a case statement to our language.

$$\begin{array}{rcl} e, f & ::= & \ldots \mid \text{case } w \text{ of } (p_1 \mid \ldots \mid p_k) \\ p & ::= & \epsilon u \in \kappa\, .p \mid \nabla u \in \kappa\, .p \mid e \text{ in } f \end{array}$$

We write $p_1 \mid \ldots \mid p_k$ simply to refer to a list of possible cases (so $k \geq 0$). It may be useful to have 0 cases if the type is uninhabited but this is deferred to our current work on coverage checking. The ϵ and ∇-quantified variables represent the pattern variables that can occur in e. We use ∇ to refer to parameters (introduced by new) and ϵ stands for typical pattern-variables. Both 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). This distinction will be illustrated in the examples (Section 8) and is explained thoroughly in our previous work [Sch05].

We create an auxiliary judgment $\Phi \vdash_w p$ to represent typing inside a case of variable w. This judgment collects the pattern variables to the left of w.

$$\frac{\begin{array}{c}\text{for all } 1 \leq i \leq k \\ \cdot \vdash \alpha \text{ wff} \qquad \Phi_1, w \in \alpha, \Phi_2 \vdash_w p_i \in \kappa\end{array}}{\Phi_1, w \in \alpha, \Phi_2 \stackrel{d}{\vdash} \text{case } w \text{ of } (p_1 \mid \ldots \mid p_k) \in \kappa}\ \mathsf{case}, k \geq 0$$

$$\frac{\Phi_1 \stackrel{d}{\vdash} e \in \alpha \qquad \Phi_1, [e/w]\Phi_2 \stackrel{d}{\vdash} f \in [e/w]\kappa}{\Phi_1, w \in \alpha, \Phi_2 \vdash_w (e \text{ in } f) \in \kappa}\ \mathsf{base}$$

$$\frac{\Phi_1, u \in \kappa, w \in \alpha, \Phi_2 \vdash_w p \in \kappa'}{\Phi_1, w \in \alpha, \Phi_2 \vdash_w (\epsilon u \in \kappa\, .p) \in \kappa'}\ \mathsf{patternVar}$$

$$\frac{\Phi_1, u \in \kappa, w \in \alpha, \Phi_2 \vdash_w p \in \kappa'}{\Phi_1, w \in \alpha, \Phi_2 \vdash_w (\nabla u \in \kappa\, .p) \in \kappa'}\ \mathsf{patternParam}$$

The rules patternVar and patternParam simply add pattern variables to the context and base substitutes the pattern for the variable. Notice importantly that pattern-variables carry time information which is crucial.

The most important characteristic of the above rules is that in case we require that we only do cases on closed types, i.e. $\cdot \vdash \alpha$ wff. At first glance this may seem problematic for dependently-typed systems. Variables in Φ may occur in other types as index arguments to LF type families, therefore it appears that we cannot do case analysis over variables whose types are not

closed. For example, based on Example 2.4, Φ could be $\cdot, u \in \langle \mathrm{o} \rangle, w \in \langle \mathrm{nd}\ u \rangle$. Therefore we can do case analysis over u but not w. However, we can do case analysis over Σ types. Therefore, if $\Phi = \cdot, u' \in \Sigma u \in \langle \mathrm{o} \rangle . \langle \mathrm{nd}\ u \rangle$ then we can do case analysis over u'. The key insight to case is that in order to case on $\langle \mathrm{nd}\ u \rangle$ we are simultaneously doing case also on u.

Therefore, we extend our case statement in a straightforward way to accept multiple variables imitating the behavior of doing case on a variable with a Σ type.

$$\begin{array}{rcl} e, f & ::= & \ldots \mid \mathrm{case}\ (w_1, \ldots, w_m)\ \mathrm{of}\ (p_1 \mid \ldots \mid p_k) \\ p & ::= & \epsilon u \in \kappa\, .p \mid \nabla u \in \kappa\, .p \mid (e_1, \ldots, e_m)\ \mathrm{in}\ f \end{array}$$

And the typing derivation is now.

$$\frac{\begin{array}{l} \Phi = \Phi_1, w_1 \in \alpha_1, \ldots, w_2 \in \alpha_2, \ldots w_m \in \alpha_m, \Phi_2 \\ \cdot \vdash (\Sigma w_1 \in \alpha_1\, .\Sigma w_2 \in \alpha_2 \ldots . \alpha_m)\ \mathrm{wff} \\ \text{for all } 1 \leq i \leq k, \quad \Phi \vdash_{(w_1,\ldots,w_m)} p_i \in \kappa \end{array}}{\Phi \Vdash^d \mathrm{case}\ (w_1, \ldots, w_m)\ \mathrm{of}\ (p_1 \mid \ldots \mid p_k) \in \kappa}\ \mathsf{case}, k \geq 0$$

$$\frac{\begin{array}{l} \Phi = \Phi_1, w_1 \in \alpha_1, \ldots, w_2 \in \alpha_2, \ldots w_m \in \alpha_m, \Phi_2 \\ \Phi_1 \Vdash^d (e_1, \ldots, e_m) \in (\Sigma w_1 \in \alpha_1\, .\Sigma w_2 \in \alpha_2 \ldots . \alpha_m) \\ [\frac{e_1,\ldots,e_m}{w_1,\ldots,w_m}]\Phi \Vdash^d f \in [\frac{e_1,\ldots,e_m}{w_1,\ldots,w_m}]\kappa \end{array}}{\Phi \vdash_{(w_1,\ldots,w_m)} ((e_1, \ldots, e_m)\ \mathrm{in}\ f) \in \kappa}\ \mathsf{base}$$

$$\frac{\Phi_1, u \in \kappa, w_1 \in \alpha_1, \Phi_2 \vdash_{(w_1,\ldots,w_m)} p \in \kappa'}{\Phi_1, w_1 \in \alpha_1, \Phi_2 \vdash_{(w_1,\ldots,w_m)} (\epsilon u \in \kappa\, .p) \in \kappa'}\ \mathsf{patternVar}$$

$$\frac{\Phi_1, u \in \kappa, w_1 \in \alpha_1, \Phi_2 \vdash_{(w_1,\ldots,w_m)} p \in \kappa'}{\Phi_1, w_1 \in \alpha_1, \Phi_2 \vdash_{(w_1,\ldots,w_m)} (\nabla u \in \kappa\, .p) \in \kappa'}\ \mathsf{patternParam}$$

Note that we introduced a shorthand for multiple substitutions $[\frac{e_1,\ldots,e_m}{w_1,\ldots,w_m}]$. When applied to a type α this is equivalent to to $[e_1/w_1] \ldots [e_m/w_m]\alpha$, and is similarly defined when applied to Φ except that the result of the substitution removes $w_1, \ldots, w_m$ from the context. It is best to view these rules as just an extension of the previous ones from one variable to many.

Due to space limitations we will only briefly talk about the operational behavior of pattern matching. We defer to the LF unification algorithm for pattern-matching on terms of type $\langle A \rangle$. Pattern matching on other

types are straightforward. For instance, pattern matching on pairs is pairwise. Matching prev e with prev e' is successful if we can match e with e'. And pattern matching on computational (non-LF) function types is defined weakly just as syntactic equality. Due to the higher-order nature of our logical framework, pattern-matching on terms of type $\langle A \rangle$ is in general not decidable. In our experience, a decidable fragment of higher-order pattern matching (which also works for the dependently typed LF) is that of Miller patterns [Mil91], into which all of our examples fall (especially those in Section 8). In addition, a strictness analysis [PS98] on ϵ bound variables can ensure that their respective instantiations are uniquely determined during program execution.

In the same spirit of presenting the operational behavior in terms of equivalence rules, we present the following equivalence rules to illustrate the behavior of pattern variables.

$$\nabla w \in \ominus^k \langle B \rangle \,.p \;\equiv\; [(\text{prev}^k\ w')/w]p \tag{1}$$

$$\epsilon w \in \ominus^k \langle B \rangle \,.p \;\equiv\; [(\text{prev}^k\ N)/w]p \tag{2}$$

In the above equivalences, both w' and N must be appropriately well-typed.

With the constructs of ν and case defined, we turn our attention to provide an example illustrating its behavior.

Example 6.1 (Derivability) Recall from example 2.2 the definition of untyped λ-terms in the logical framework as objects of type "exp".

1. The expression $\nu u \in \langle \text{exp} \rangle \,.\langle u \rangle$ is not well-formed, because $u \in \langle \text{exp} \rangle \Vdash^d \langle u \rangle \in \ominus\langle \text{exp} \rangle$ is not derivable.

2. The expression $\nu u \in \langle \text{exp} \rangle \,.\text{prev}\ \langle u \rangle$ is not well-formed, because $\cdot \Vdash^d \langle u \rangle \in \langle \text{exp} \rangle$ is not derivable.

3. The expression

   $$\nu x \in \langle \text{exp} \rangle \,.[x/y]\text{case } y \text{ of } (\epsilon e \in \ominus\langle \text{exp} \to \text{exp} \rangle \,.\\ e\ x \text{ in } (\text{prev } \langle \text{lam } e \rangle))$$

   is well-formed.

   $e : (\text{exp} \to \text{exp}) \vdash \text{lam } e : \text{exp}$ by LF
   $e \in \langle \text{exp} \to \text{exp} \rangle \Vdash^d \langle \text{lam } e \rangle \in \langle \text{exp} \rangle$ by $\langle\rangle\mathsf{R}$
   $x \in \langle \text{exp} \rangle, e \in \ominus\langle \text{exp} \to \text{exp} \rangle$

$$
\begin{array}{lr}
\quad \Vdash^d \text{prev } \langle \text{lam } e \rangle \in \ominus\langle \text{exp} \rangle & \text{by } \ominus\mathsf{R} \\
x \in \langle \text{exp} \rangle, e \in \ominus\langle \text{exp} \to \text{exp} \rangle & \\
\quad \Vdash^d (\text{prev } \langle \text{lam } e \rangle) \in [e\ x/y](\ominus\langle \text{exp} \rangle) & \text{by Substitution} \\
\\
x : \text{exp}, e : (\text{exp} \to \text{exp}) \vdash e\ x : \text{exp} & \text{by LF} \\
x \in \langle \text{exp} \rangle, e \in \ominus\langle \text{exp} \to \text{exp} \rangle \Vdash^d e\ x \in \langle \text{exp} \rangle & \text{by } \langle\rangle\mathsf{R} \\
\\
x \in \langle \text{exp} \rangle, e \in \ominus\langle \text{exp} \to \text{exp} \rangle, y \in \langle \text{exp} \rangle & \\
\quad \vdash_y e\ x \text{ in } (\text{prev } \langle \text{lam } e \rangle) \in \ominus\langle \text{exp} \rangle & \text{by } \mathsf{base} \\
x \in \langle \text{exp} \rangle, y \in \langle \text{exp} \rangle \vdash_y (\epsilon e \in \ominus\langle \text{exp} \to \text{exp} \rangle\,. & \\
\quad e\ x \text{ in } (\text{prev } \langle \text{lam } e \rangle)) \in \ominus\langle \text{exp} \rangle & \text{by } \mathsf{patternVar} \\
\\
\cdot \vdash \text{exp wff} & \text{by LF} \\
\cdot \Vdash^d \langle \text{exp} \rangle \text{ wff} & \text{by } \langle\rangle\mathsf{wffR} \\
\\
x \in \langle \text{exp} \rangle, y \in \langle \text{exp} \rangle \Vdash^d \text{case } y \text{ of } (\epsilon e \in \ominus\langle \text{exp} \to \text{exp} \rangle\,. & \\
\quad e\ x \text{ in } (\text{prev } \langle \text{lam } e \rangle)) \in \ominus\langle \text{exp} \rangle & \text{by } \mathsf{case} \\
x \in \langle \text{exp} \rangle \Vdash^d [x/y]\text{case } y \text{ of } (\epsilon e \in \ominus\langle \text{exp} \to \text{exp} \rangle\,. & \\
\quad e\ x \text{ in } (\text{prev } \langle \text{lam } e \rangle)) \in \ominus\langle \text{exp} \rangle & \text{by Substitution} \\
\cdot \Vdash^d \nu x \in \langle \text{exp} \rangle\,.[x/y]\text{case } y \text{ of } (\epsilon e \in \ominus\langle \text{exp} \to \text{exp} \rangle\,. & \\
\quad\quad e\ x \text{ in } (\text{prev } \langle \text{lam } e \rangle)) \in \ominus\langle \text{exp} \rangle & \text{by } \mathsf{new}
\end{array}
$$

Example 6.2 (Evaluation) In continuation of the previous example, we show the evaluation of expression 3 above.

$$
\begin{array}{l}
[x/y]\text{case } y \text{ of } (\epsilon e \in \ominus\langle \text{exp} \to \text{exp} \rangle\,.e\ x \text{ in } (\text{prev } \langle \text{lam } e \rangle)) \\
\quad \equiv\ [x/y]\text{case } y \text{ of } x \text{ in } (\text{prev } \langle \text{lam } (\lambda x' : \text{exp}\,.x') \rangle) \\
\quad \equiv\ (\text{prev } \langle \text{lam } (\lambda x' : \text{exp}\,.x') \rangle)
\end{array}
$$

In Example 6.2 the pattern variable e is substituted with the identity function. We would *not* be able to substitute a function where x occurred free because then it would not type-check in our system. This is exactly the behavior that we want illustrating how we statically enforce that parameters cannot escape their scope.

7 General Recursion

Next, we extend Delphin 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, we are now focusing on the computational aspects. Thus we add a general recursion principle.

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

which satisfies the usual specification.

$$\frac{\Phi, u \in \kappa \vdash^{d} e \in \kappa}{\Phi \vdash^{d} (\mu u \in \kappa \,.e) \in \kappa}\ \mathsf{rec}$$

and is interpreted operationally as

$$(\mu u \in \kappa \,.e) \quad \equiv \quad [\mu u \in \kappa \,.e/u]e. \tag{3}$$

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 next, in Section 8.

8 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.

$$\begin{array}{rcl} \text{let } u = e \text{ in } f & \equiv & (\lambda u \in \tau \,.f)e \\ & & \quad \text{when } \tau \text{ is inferable.} \end{array}$$

$$\begin{array}{rcl} \textbf{fun } f\ (u_1 \in \tau_1)\ (u_2 \in \tau_2) & \equiv & \mu f \in (\Pi u_1 \in \tau_1 \,.\Pi u_2 \in \tau_2 \,. \\ \ldots (u_n \in \tau_n) : \sigma & & \ldots \Pi u_n \in \tau_n \,.\sigma) \,. \\ = e & & \lambda u_1 \in \tau_1 \ldots . \lambda u_n \in \tau_n \,.e \end{array}$$

We also should mention that for illustrative purposes we *explicitly* declare all pattern-variables since the time information on them is crucial in understanding the examples. However, ϵ-bound pattern variables are relatively easy to infer (even with the extra time information) and they can be implicit in Delphin.

8.1 Structural Identity

Recall the encoding of the untyped λ-calculus from Example 2.2. We use the hypothetical judgment $E = F$ to define when two λ-expressions are equal.

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

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⟩
= case e of
    ϵE1 ∈ ⟨exp⟩ .ϵE2 ∈ ⟨exp⟩ .(app E1 E2) in
      let D1 = cpfun E1 in
      let D2 = cpfun E2 in
      cp_app D1 D2
  | ϵE' ∈ ⟨exp → exp⟩ .(lam E') in
      next (νx ∈ ⟨exp⟩ .νu ∈ ⟨cp x x⟩ .
              let D' = cpfun (E' x) in
                case D' of ϵD ∈ ⊖⟨Πx : exp .cp x x
                                    → cp (E' x) (E' x)⟩ .
                           (D x u) in
              prev (cp_lam D)
  | ∇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. 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. Variables bound by ∇ are only instantiated by parameters, which are introduced by new. Variables bound by ϵ can be instantiated with anything. If we replaced the ∇ quantified variables in the last case with ϵ's then since the pattern is just x, this branch could non deterministically return anything.

In this example there is no nondeterministic behavior because all pattern variables occur *strict* [PS98] in the pattern. We will exploit the nondeterministic behavior later in our theorem prover example. However, we can eliminate the nondeterminism by making sure that pattern variables occur strict in the pattern. In the implementation we can generate a warning if it doesn't.

8.2 Combinators

Recall the definition of the natural deduction calculus from Example 2.4. 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 into a combinator with one less parameter . If M has type $\langle$comb A $\rightarrow$ comb B$\rangle$ and N has type $\langle$comb A$\rangle$ then we can use ba to get a combinator, u, of type $\langle$comb $A \Rightarrow B\rangle$. Then we can do MP u N to get a term that is equivalent to $\langle MN\rangle$ in combinator logic. Formally, ba is written as.

```
fun ba (A ∈ ⟨o⟩) (B ∈ ⟨o⟩) (F ∈ ⟨comb A → comb B⟩)
          : ⟨comb A ⇒ B⟩
=  case (A, B, F) of
          ϵA ∈ ⟨o⟩ .ϵB ∈ ⟨o⟩ .∇z ∈ ⟨comb B⟩ .(A, B, λx : comb A .z)
          in (MP K z)

        | ϵA ∈ ⟨o⟩ .(A, A, λx : comb A .x)
            in (MP (MP S K) K)

        | ϵA ∈ ⟨o⟩ .ϵC ∈ ⟨o⟩ .ϵD ∈ ⟨o⟩ .
              (A, C ⇒ D ⇒ C, λx : comb A .K)
           in (MP K K)

        | ϵA ∈ ⟨o⟩ .ϵC ∈ ⟨o⟩ .ϵD ∈ ⟨o⟩ .ϵE ∈ ⟨o⟩ .
              (A,
               (C ⇒ D ⇒ E) ⇒ (C ⇒ D) ⇒ C ⇒ E,
               λx : comb A .S)
           in (MP K S)

        | ϵA ∈ ⟨o⟩ .ϵB ∈ ⟨o⟩ .ϵC ∈ ⟨o⟩ .
            ϵD1 ∈ ⟨comb A → comb (C ⇒ B)⟩ .
            ϵD2 ∈ ⟨comb A → comb C⟩ .
              (A, B, λx : comb A .MP (D1 x) (D2 x))
           in
                let D1' = ba A (C ⇒ B) D1 in
                let D2' = ba A C 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 (∇z) 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⟩
= case (A, D) of
        ϵA ∈ ⟨o⟩ .∇u ∈ ⟨comb A → nd A⟩ .∇c ∈ ⟨comb A⟩ .
          (A, u c)
        in c

    | ϵA ∈ ⟨o⟩ .ϵB ∈ ⟨o⟩ .
        ϵD1 ∈ ⟨nd (B ⇒ A)⟩ .ϵD2 ∈ ⟨nd B⟩ .
          (A, impe D1 D2)
        in
          let C1 = convert (B ⇒ A) D1 in
          let C2 = convert B D2 in
            (MP C1 C2)

    | ϵB ∈ ⟨o⟩ .ϵC ∈ ⟨o⟩ .ϵD ∈ ⟨nd B → nd C⟩ .
          (B ⇒ C, impi D)
        in
          next (νu ∈ ⟨comb B → nd B⟩ .νb ∈ ⟨comb B⟩ .
                  let D′ = convert B (D (u b)) in
                  case D′ of ϵD″ ∈ ⊖⟨comb B → comb C⟩ .
                                  (D″ b) 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 u of type $\langle \text{nd } B\rangle$, we introduce a parameter of type $\langle \text{comb } B \to \text{nd } B\rangle$ that carries a combinator as "payload." In our example, the payload is another parameter $b \in \langle \text{comb } B\rangle$, the image of u under convert. This technique is known as payload-carrying parameters and is applicable to a wide range of examples [Sch05].

8.3 Theorem Prover

Next we illustrate how the non-deterministic instantiation of ϵ bound parameters can be used to implement a very simple and naive proof search procedure. Our theorem prover constructs derivation in the natural deduction calculus from Example 2.4 from a formula A. The function prove $\in \Pi A \in \langle \text{o}\rangle .\langle \text{nd } A\rangle$ and and solve $\in \Pi A \in \langle \text{o}\rangle .\Pi C \in \langle \text{o}\rangle .\langle \text{nd } C\rangle \supset \langle \text{nd } A\rangle$ are defined mutually recursively. The former constructs a proof for A, and the

latter applies impe to proofs of C in the hope to discover a proof of A.

fun prove $(A \in \langle \mathrm{o} \rangle) : \langle \mathrm{nd}\ A \rangle$

$=$ case A of

$$\begin{array}{l}
\quad \nabla p \in \langle \mathrm{o} \rangle\,.\epsilon C \in \langle \mathrm{o} \rangle\,.\nabla u \in \langle \mathrm{nd}\ C \rangle\,. \\
\qquad p \text{ in } \mathsf{solve}\ \ \langle p \rangle\ \ \langle C \rangle\ \ u \\
\\
\quad \mid\ \nabla A' \in \langle \mathrm{o} \rangle\,.\nabla B' \in \langle \mathrm{o} \rangle\,. \\
\qquad (A' \Rightarrow B') \text{ in} \\
\qquad \text{next } (\nu u \in \langle \mathrm{nd}\ A' \rangle\,. \\
\qquad\qquad \text{let } D' = \mathsf{prove}\ \ B'\ \text{ in} \\
\qquad\qquad \text{case } D' \text{ of } \epsilon D \in \ominus \langle \mathrm{nd}\ A' \to \mathrm{nd}\ B' \rangle\,. \\
\qquad\qquad\qquad\qquad (D\ u) \text{ in } \quad \text{prev } (\text{impi } D))
\end{array}$$

and **fun** $\mathsf{solve}\ (A \in \langle \mathrm{o} \rangle)\ \ (C \in \langle \mathrm{o} \rangle)\ \ (D \in \langle \mathrm{nd}\ C \rangle) : \langle \mathrm{nd}\ A \rangle$

$$\begin{array}{l}
= \text{ case } (A,\ C) \text{ of} \\
\qquad \epsilon A \in \langle \mathrm{o} \rangle\,.(A,\ A)\ \text{ in }\ D \\
\quad \mid\ \epsilon A \in \langle \mathrm{o} \rangle\,.\nabla A' \in \langle \mathrm{o} \rangle\,.\nabla B' \in \langle \mathrm{o} \rangle\,. \\
\qquad (A,\ A' \Rightarrow B')\ \text{ in} \\
\qquad\qquad \text{let } D' = \mathsf{prove}\ \ A'\ \text{ in} \\
\qquad\qquad \mathsf{solve}\ \ A\ \ B'\ \ (\text{impe } D'\ D)
\end{array}$$

9 Related Work

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

Higher-order abstract syntax. The $\lambda^{\mathcal{D}}$-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 $\lambda^{\mathcal{D}}$-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 $\lambda^{\mathcal{D}}$-calculus in their structural design. Our calculus is a three-tiered language. Its upper two layers, 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 $\lambda^{\mathcal{D}}$-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.

Our work is based upon our earlier work with the ∇-calculus [Sch05] which lacked dependent-types and followed a more ad-hoc approach than our logically motivated work here. Our work with the ∇-calculus has also inspired Westbrook's similar work called λ^{FV} [Wes06]. This calculus is designed to include coverage checking in the type-system. Work on λ^{FV} is still ongoing, including a proof for type safety.

10 Conclusion

In this paper we have given a formulation of temporal logic designed to reason about derivations under a different logic at different times. It is being turned into a novel programming language, which is called Delphin and currently under development. Please visit www.cs.yale.edu/∼delphin for more information. And finally, with the appropriate syntactical enforceable side conditions of coverage and termination, $\lambda^{\mathcal{D}}$ 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. We are also concentrating on the implementation efforts.

A Appendix: Twelf Proofs

A.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
```

A.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 _ _).
```

A.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 : pureO -> pureO -> pureO.                  %infix  right 10 imp.
inj : lftp -> pureO.
pair : pureO -> pureO -> 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 _).
```

A.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, TP1 |-- TP2
-------------------- impr
Omega |-- TP1 imp TP2
}%
impr : (hyp X TP1 -> conc X (! TP2))
       -> conc X (! (TP1 imp TP2)).


%{ IMPL
(past...past)(TP1 imp TP2) in Omega,
Omega |-- TP1   Omega,TP2 |--- S
--------------------------------- impl
Omega |-- S
}%
impl : le X Y -> hyp X (TP1 imp TP2) -> conc Y (! TP1)
       -> (hyp Y TP2 -> conc Y (! S)) -> conc Y (! S).


%{ TOP

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


%{ BOTL

------------------------------------- botl
Omega,(past...past)bot,Omega' |--- TP
}%
botl : le X Y -> hyp X bot -> 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 |-- TP1          Omega |-- TP2
---------------------------------- pairr
      Omega |-- pair TP1 TP2
}%
pairr : conc X (! TP1) -> conc X (! TP2) -> conc X (! (pair TP1 TP2)).

%{ PAIRL
(past...past)(pair TP1 TP2) in Omega,
Omega, TP1, TP2 |--- S
```

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

A.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 (TP1 imp TP2).
notInjPair : notInj (pair TP1 TP2).
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 _).
```

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

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

weakenHyp'Impl2 : weakenHyp' ([sh] impl L sh (D1 sh) (D2 sh))
   ([h] impl L' h (D1' h) (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) ([h] botl L H).
weakenHyp'Botl2 : weakenHyp' ([sh] botl L sh) ([h] botl L' h)
   <- 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 (D sh))
   ([h] pairl L H (D' h))
   <- ({h2}{h3} weakenHyp' ([sh] D sh h2 h3) ([h] D' h h2 h3)).

weakenHyp'Pairl2 : weakenHyp' ([sh] pairl L sh (D sh))
   ([h] pairl L' h (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 (D h)) (impr D')
   <- ({h'} strengthen L ([h] D h h') (D' h')).

strengthenImpl1  : strengthen L ([h] impl L' H (D1 h) (D2 h)) (impl L' H D1' D2')
   <- strengthen L D1 D1'
   <- ({h'} strengthen L ([h] D2 h h') (D2' h')).
strengthenImpl2  : strengthen L ([h] impl L' h (D1 h) (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) (botl L' H).
strengthenBotl2 : strengthen L ([h] botl L' h) 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 (D h)) (pairl L' H D')
     <- ({h'}{h''} strengthen L ([h] D h h' h'') (D' h' h'')).

strengthenPairl2  : strengthen L ([h] pairl L' h (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.
%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)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


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'.

%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 _).
```

A.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' : (hyp' X TP1 -> conc' X (! TP2))
        -> conc' X (! (TP1 imp TP2)).
impl' : le X Y -> hyp' X (TP1 imp TP2) -> conc' Y (! TP1)
        -> (hyp' Y TP2 -> conc' Y (! S)) -> conc' Y (! S).
topr' : conc' _ (! top).
botl' : le X Y -> hyp' X bot -> conc' Y (! TP).
pastr' : conc' X T -> conc' (s X) (past T).
injr' : exp' X A -> conc' X (! (inj A)).
pairr' : conc' X (! TP1) -> conc' X (! TP2) -> conc' X (! (pair TP1 TP2)).
pairl' : le X Y -> hyp' X (pair TP1 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' D') (impr D)
       <- ({h}{h'} copyH' h' base h
     -> copyC' (D' h') (D h)).

copy-impl': copyC' (impl' L1 H' D1' D2') (impl L3 H D1 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') (botl L3 H)
             <- 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' D') (pairl L3 H 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 D) (impr' D')
   <- ({h} {h'}  map' h base h'
 -> shift' (D h) (D' h')).


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


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

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

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 D) (pairl' L' H' 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'.


%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.

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 _ _).
```

A.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.
caPure : {TP}  conc X (! TP) -> (hyp X TP -> conc X S) -> conc X S -> type.

%mode (ca +T +D +SP +E -F).
%mode (caPure +TP +D +E -F).

% %%%%%%%%%%%%%%%%%%%%%%
% Bridge ca and caPure
% %%%%%%%%%%%%%%%%%%%%%%
ca_bridge : ca (! TP) D stripPastPure E F
     <- caPure TP D E F.

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

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

caPure_impl_r : caPure (TP1 imp TP2) (impr D) ([h] impl L h (E1 h) (E2 h)) F3
 <- caPure (TP1 imp TP2) (impr D) E1 E1'
 <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
       caPure (TP1 imp TP2) (impr D) ([h] E2 h h2) (E2' h2))
 <- caPure TP1 E1' D F1
 <- caPure TP2 F1 E2' 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.

caPure_injr_r : caPure (inj A) (injr N) ([h] injr (M h)) (injr (M' N))
 <- convertHypExp M M'.

caPure_pairl_r : caPure (pair TP1 TP2) (pairr D1 D2) ([h] pairl L h (E h)) F*
  <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2')
        -> ({h3}{h3'} copyH' h3' base h3 -> map' h3 base (!! h3') ->
        caPure (pair TP1 TP2) (pairr D1 D2) ([h] E h h2 h3) (E' h2 h3)))
  <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2')
          -> caPure TP1 D1 ([h] E' h h2) (F2 h2))
  <- caPure TP2 D2 F2 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) F
       <- shift D D'
       <- stripAddOne SP SP'
       <- stripToLess SP' L'
       <- ca T D' SP' ([h] botl L' h) F.

ca_pastr_l2 : ca (past T) (pastr D) (stripPastPast SP) ([h] impl L h (E1 h) (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' 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 (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 E') F.


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

caPure_axiom_l : caPure TP (axiom L H) E (E' H)
  <- weakenHyp L E E'.


caPure_impl_l : caPure TP (impl L H D1 D2) E (impl L H D1 F)
 <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
      caPure TP (D2 h2) E (F h2)).


caPure_pairl_l : caPure TP (pairl L H D) E (pairl L H F)
           <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
                  ({h3}{h3'} copyH' h3' base h3 -> map' h3 base (!! h3') ->
             caPure TP (D h2 h3) 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))
}%
caPure_botl_l_impl_r1 : caPure _ (botl L H) ([h] impl L* h _ _) (botl L H).

caPure_botl_l_pairl_r1 : caPure _ (botl L H) ([h] pairl L* h _) (botl L H).

caPure_botl_l_botl_r : caPure _ (botl L H) ([h] botl L* h) (botl L H).
caPure_botl_l_injr_r : caPure _ (botl L H) ([h] injr _) (botl L H).


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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
% or impl or pairl

caPure_strengthenImprPastr1 : caPure _ (impr _) ([h] pastr (E' h)) F
       <- strengthenP ([h] pastr (E' h)) F.
caPure_strengthenToprPastr2 : caPure _ (topr) ([h] pastr (E' h)) F
```

```
       <- strengthenP ([h] pastr (E' h)) F.
caPure_strengthenBotlPastr3 : caPure _ (botl _ _) ([h] pastr (E' h)) F
       <- strengthenP ([h] pastr (E' h)) F.
caPure_strengthenInjrPastr4 : caPure _ (injr _) ([h] pastr (E' h)) F
       <- strengthenP ([h] pastr (E' h)) F.
caPure_strengthenInjrPastr5 : caPure _ (pairr _ _) ([h] pastr (E' h)) F
       <- strengthenP ([h] pastr (E' h)) F.
caPure_strengthenInjrPastr6 : caPure _ (impl _ _ _ _) ([h] pastr (E' h)) F
       <- strengthenP ([h] pastr (E' h)) F.
caPure_strengthenInjrPastr7 : caPure _ (pairl _ _ _) ([h] pastr (E' h)) F
       <- strengthenP ([h] pastr (E' h)) F.


% Now for inject on right...
caPure_strengthen_ImprInjr : caPure _ (impr _) ([h] injr (M h)) (injr N)
      <- strengthenLFNI notInjImp M N.
caPure_strengthen_TopInjr :  caPure _ (topr) ([h] injr (M h)) (injr N)
     <- strengthenLFNI notInjTop M N.
caPure_strengthen_PairrInjr :  caPure _ (pairr _ _) ([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).
caPure_axiom_rCommute : caPure TP D ([h] axiom L H) (axiom L H).

ca_topr_rCommute : ca T D SP ([h] topr) topr.
caPure_topr_rCommute : caPure TP D ([h] topr) topr.

ca_botl_rCommute : ca T D SP ([h] botl L H) (botl L H).
caPure_botl_rCommute : caPure TP D ([h] botl L H) (botl L H).

ca_impr_rCommute : ca T D SP ([h] impr (E h)) (impr F)
    <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
        ca T D SP ([h] (E h h2)) (F h2)).
caPure_impr_rCommute : caPure TP D ([h] impr (E h)) (impr F)
         <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
             caPure TP D ([h] (E h h2)) (F h2)).

ca_impl_rCommute : ca T D SP ([h] impl L H (E1 h) (E2 h)) (impl L H E1' 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)).
caPure_impl_rCommute : caPure TP D ([h] impl L H (E1 h) (E2 h)) (impl L H E1' E2')
         <- caPure TP D E1 E1'
         <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
       caPure TP D ([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.
caPure_pairr_rCommute : caPure TP D ([h] pairr (E1 h) (E2 h)) (pairr F1 F2)
          <- caPure TP D E1 F1
          <- caPure TP D E2 F2.

ca_pairl_rCommute : ca T D SP ([h] pairl L H (E h)) (pairl L H 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))).
caPure_pairl_rCommute : caPure TP D ([h] pairl L H (E h)) (pairl L H E')
         <- ({h2}{h2'} copyH' h2' base h2 -> map' h2 base (!! h2') ->
               ({h3}{h3'} copyH' h3' base h3 -> map' h3 base (!! h3') ->
          caPure TP D ([h] E h h2 h3) (E' h2 h3))).


%worlds (mapcopyBlock | lfmapcopyBlock | lftypeConsBlock) (ca _ _ _ _ _) (caPure _ _ _ _).
%terminates {(T TP') [(D D') (E E')]} (ca T D _ E _) (caPure TP' D' E' _).
%covers (ca +T +D +SP +E -F) (caPure +TP +D +E -F).
%total {(T TP') [(D D') (E E')]} (ca T D _ E _) (caPure TP' D' E' _).
```

A.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) -> 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 : (exp X A -> concND X (! TP2)) -> concND X (! ((inj A) imp TP2)).
MlamNI : notInj TP ->
         (concND X (! TP) -> concND X (! TP2)) -> concND X (! (TP imp TP2)).

Mapp : concND X (!(TP1 imp TP2)) -> concND X (! TP1) -> concND X (! TP2).
topND : concND X (! top).
prev : concND X T -> concND (s X) (past T).
inject : exp X A -> concND X (!(inj A)).
pairI : concND X (! TP1) -> concND X (! TP2) -> concND X (! (pair TP1 TP2)).

% Similarly to the lam case, we need 4 cases for pairE to distinguish
% between inj and notInj
pairE00 : concND X (! (pair TP1 TP2))
  -> notInj TP1 -> notInj TP2
          -> (concND X (! TP1) -> concND X (! TP2) -> concND X (! S))
          -> concND X (! S).
pairE01 : concND X (! (pair TP1 (inj B)))
  -> notInj TP1
          -> (concND X (! TP1) -> exp X B -> concND X (! S))
          -> concND X (! S).
pairE10 : concND X (! (pair (inj A) TP2))
  -> notInj TP2
          -> (exp X A -> concND X (! TP2) -> concND X (! S))
          -> concND X (! S).
pairE11 : concND X (! (pair (inj A) (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)) ([d] botND (D' d))
 <- strengthenFunND NI D D'.
strengthenFunND_MlamInj : strengthenFunND NI ([h][d] MlamInj (F h d)) ([d] MlamInj (F' d))
   <- ({n'} strengthenFunND NI ([h][d] F h d n') ([d] F' d n')).
strengthenFunND_MlamNI : strengthenFunND NI ([h][d] MlamNI NI' (F h d)) ([d] MlamNI 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)) ([d] Mapp (D1' d) (D2' d))
<- 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) NI1 NI2 (D2 h d))
                     ([d] pairE00 (D1' d) 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) NI1 (D2 h d))
                     ([d] pairE01 (D1' d) 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) NI2 (D2 h d))
                     ([d] pairE10 (D1' d)  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) (D2 h d))
                     ([d] pairE11 (D1' d) (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)) (botND D')
      <- strengthenND L D D'.
strengthenND_MlamInj : strengthenND L ([d] MlamInj (F d)) (MlamInj F')
<- ({n'} strengthenND L ([d] F d n') (F' n')).
strengthenND_MlamNI : strengthenND L ([d] MlamNI NI' (F d)) (MlamNI NI' F')
       <- ({e} strengthenND L ([d] F d e) (F' e)).

strengthenND_Mapp : strengthenND L ([d] Mapp (D1 d) (D2 d)) (Mapp D1' D2')
     <- 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) NI1 NI2 (D2 d))
                (pairE00 D1' 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) NI1 (D2 d))
                (pairE01 D1' 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) NI2 (D2 d))
                (pairE10 D1' 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) (D2 d))
                (pairE11 D1' 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)) ([n] botND (D' n))
   <- strengthenFunNDLF D D'.
strengthenFunNDLF_MlamInj : strengthenFunNDLF ([h][n] MlamInj (F h n)) ([n] MlamInj (F' n))
     <- ({n'} strengthenFunNDLF ([h][n] F h n n') ([n] F' n n')).
strengthenFunNDLF_MlamNI : strengthenFunNDLF ([h][n] MlamNI NI (F h n)) ([n] MlamNI 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)) ([n] Mapp (D1' n) (D2' n))
  <- 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) NI1 NI2 (D2 h n))
                     ([n] pairE00 (D1' n) 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) NI1 (D2 h n))
                     ([n] pairE01 (D1' n) 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) NI2 (D2 h n))
                     ([n] pairE10 (D1' n) 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) (D2 h n))
                     ([n] pairE11 (D1' n) (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)) (botND D')
      <- strengthenNDLF L D D'.

strengthenNDLF_MlamInj : strengthenNDLF L ([n] MlamInj (F n)) (MlamInj F')
<- ({m}
            strengthenNDLF L ([n] F n m) (F' m)).

strengthenNDLF_Mlam : strengthenNDLF L ([n] MlamNI NI (F n)) (MlamNI NI F')
       <- ({e}
  strengthenNDLF L ([n] F n e) (F' e)).

strengthenNDLF_Mapp : strengthenNDLF L ([n] Mapp (D1 n) (D2 n)) (Mapp D1' D2')
     <- 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) NI1 NI2 (D2 n))
                  (pairE00 D1' 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) NI1 (D2 n))
                  (pairE01 D1' 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) NI2 (D2 n))
                  (pairE10 D1' 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) (D2 n))
                  (pairE11 D1' 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 _).


futureRuleND : concND (s X) (past T) -> concND X T -> type.
%mode futureRuleND +D -D'.

futureRuleND_Prev : futureRuleND (prev D) D.


futureRuleND_ShiftND : futureRuleND (shiftND D) (shiftND D')
<- futureRuleND D 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'.

%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'.

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

A.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) F
      <- NDtoSQ D D'
      <- ca (! bot) D' stripPastPure ([h] botl base h) F.

NDtoSQ_MlamNI : NDtoSQ (MlamNI NI D) (impr 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 D) (impr ([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) F
      <- NDtoSQ E E'
      <- NDtoSQ D D'
      <- ca _ E' stripPastPure ([h] impl base h D' ([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 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 D2') F.

NDtoSQ_pairE01 : NDtoSQ (pairE01 D1 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 ([h1][h2] D2' h1 h2 (castHyp h2))) F.

NDtoSQ_pairE10 : NDtoSQ (pairE10 D1 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 ([h1][h2] D2' h1 (castHyp h1) h2)) F.

NDtoSQ_pairE11 : NDtoSQ (pairE11 D1 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 ([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 _).
```

A.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 E) D).
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 F) (MlamNI 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 F) (MlamNI 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 F) (MlamNI 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 F) (MlamNI notInjImp F*)
       <- ({h:hyp _ (TP1 imp TP2)}{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 F) (MlamNI notInjPair F*)
       <- ({h:hyp _ (pair TP1 TP2)}{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 F) (MlamInj 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 D2) (F2' (Mapp F0 F1))
       <- 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 D2) (F2' (Mapp F00 F1))
       <- 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 D2) (F2' (Mapp F00 F1))
       <- hypToCN L H F00
       <- SQtoND D1 F1
       <- ({h:hyp _ (TP1 imp TP2)}{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 D2) (F2' (Mapp F00 F1))
       <- hypToCN L H F00
       <- SQtoND D1 F1
       <- ({h:hyp _ (pair TP1 TP2)}{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 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) F2' E.

SQtoND_topr : SQtoND topr topND.

SQtoND_botl : SQtoND (botl L H) (botND D)
       <- 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE01 F00 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE01 F00 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE01 F00 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE00 F00 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 D) (pairE01 F00 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 D) (pairE10 F00 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 D) (pairE10 F00 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 D) (pairE10 F00 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 D) (pairE10 F00 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 D) (pairE11 F00 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 _).
```

A.12 examples.elf

```
% Adam Poswolsky
% Example Section


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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.

[Pos06] Adam Poswolsky and Carsten Schürmann. Extended Report: A Temporal-Logic Approach to Programming with Dependent Types and Higher-Order Encodings. Yale University Tech Report TR-1355, 2006.
http://www.cs.yale.edu/publications/techreports/tr1355.pdf

[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.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Wes06] Edwin Westbrook Free Variable Types Trends in Functional Programming (TFP 2006)
http://www.cs.nott.ac.uk/∼nhn/TFP2006/Papers
/21-Westbrook-FreeVariableTypes.pdf

[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] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The ∇-calculus. Functional programming with higher-order encodings. *Typed Lambda Calculus and Applications, TLCA'05* 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.