$\mathsf{LLF}_{\mathcal{P}}$: A LOGICAL FRAMEWORK FOR MODELING EXTERNAL EVIDENCE, SIDE CONDITIONS, AND PROOF IRRELEVANCE USING MONADS

FURIO HONSELL, LUIGI LIQUORI, PETAR MAKSIMOVIĆ, AND IVAN SCAGNETTO

Università di Udine, Italy
e-mail address: furio.honsell@uniud.it

Inria, France
e-mail address: luigi.liquori@inria.fr

Inria, France and
Mathematical Institute of the Serbian Academy of Sciences and Arts, Serbia
e-mail address: petarmax@mi.sanu.ac.rs

Università di Udine, Italy
e-mail address: ivan.scagnetto@uniud.it

Abstract. We extend the constructive dependent type theory of the Logical Framework LF with *monadic*, *dependent* type constructors indexed with predicates over judgements, called *Locks*. These monads capture various possible *proof attitudes* in establishing the judgment of the object logic encoded by an LF type. Standard examples are *factoring-out* the verification of a *constraint* or *delegating* it to an *external oracle*, or supplying some *non-apodictic* epistemic evidence, or simply *discarding* the proof witness of a precondition deeming it irrelevant. This new framework, called *Lax Logical Framework*, $\mathsf{LLF}_{\mathcal{P}}$, is a conservative extension of LF, and hence it is the appropriate metalanguage for dealing formally with *side-conditions* in rules or *external evidence* in logical systems. $\mathsf{LLF}_{\mathcal{P}}$ arises once the *monadic* nature of the *lock* type-constructor, $\mathcal{L}^{\mathcal{P}}_{M,\sigma}[\cdot]$, introduced by the authors in a series of papers, together with Marina Lenisa, is fully exploited. The nature of the lock monads permits to utilize the very *Lock* destructor, $\mathcal{U}^{\mathcal{P}}_{M,\sigma}[\cdot]$, in place of Moggi's monadic let_T, thus simplifying the equational theory. The rules for $\mathcal{U}^{\mathcal{P}}_{M,\sigma}[\cdot]$ permit also the removal of the monad once the constraint is satisfied. We derive the meta-theory of $\mathsf{LLF}_{\mathcal{P}}$ by a novel indirect method based on the encoding of $\mathsf{LLF}_{\mathcal{P}}$ in LF. We discuss encodings in $\mathsf{LLF}_{\mathcal{P}}$ of call-by-value λ-calculi, Hoare's Logic, and Fitch-Prawitz Naive Set Theory.

dedicated to Pierre-Louis Curien

Contents

1998 ACM Subject Classification: F.4.1 [Mathematical Logic]: Mechanical theorem proving.
Key words and phrases: Computer aided formal verification, type theory, logical frameworks, typed lambda calculus.

LOGICAL METHODS
IN COMPUTER SCIENCE DOI:10.2168/LMCS-???

1. Introduction

A mathematician, half way through a proof, during a seminar, said "...and this trivially holds". But after a few seconds of silence, somewhat to himself, he mumbled: "... but is this really trivial, here? ... Hmm ... ". He kept silent for 5 minutes. And finally triumphantly exclaimed "Yes, it is indeed trivial!"

In this paper we introduce, develop the metatheory, and give applications of the *Lax Logical Framework*, $\mathsf{LLF}_{\mathcal{P}}$. $\mathsf{LLF}_{\mathcal{P}}$ is a conservative extension of LF which was first outlined in the Symposium in honour of Pierre Louis Curien, held in Venice in September 2013. A preliminary version of $\mathsf{LLF}_{\mathcal{P}}$ was presented in [27]. This system has grown out of a series of papers on extensions of LF published by the authors, together with Marina Lenisa, in recent years [9, 24, 26, 25, 22]. The idea underpinning these systems is to be able to express explicitly, by means of a new type-constructor $\mathcal{L}^{\mathcal{P}}_{M,\sigma}[\cdot]$, called a *lock*, the fact that in order to obtain a term of a given type it is necessary to verify the constraint $\mathcal{P}(\Gamma \vdash_{\Sigma} M : \sigma)$. By using this type constructor, one can capture various *proof attitudes* which arise in practice, such as *factoring-out* or *postponing* the verification of certain judgements whose evidence we do not want to derive in the standard way. This occurs when the evidence for the justification of that judgement is supplied by an *external proof search tool* or an *external oracle*, or some other *non-apodictic* epistemic sources of evidence such as diagrams, physical analogies, or explicit computations according to the *Poincaré Principle* [8]. These proof attitudes are ultimately similar to that which occurs in proof irrelevant approaches when one is only interested to know that some evidence is there, but the precise nature of the proof witness is declared immaterial. Therefore, locked types allow for a straightforward accommodation within the Logical Framework of many different *proof cultures* that otherwise can be embedded only very deeply [10, 21] or axiomatically [28]. Locked types support the main

motivation of $\mathsf{LLF}_{\mathcal{P}}$, namely that external tools may be *invoked* and *recorded* uniformly in an LF type-theoretic framework.

The main novelty of $\mathsf{LLF}_{\mathcal{P}}$ w.r.t. previous systems using locked types introduced by the authors, is that $\mathsf{LLF}_{\mathcal{P}}$ capitalizes on a monadic understanding of $\mathcal{L}^{\mathcal{P}}_{M,\sigma}[\cdot]$ constructors. An extended abstract of the present paper appears in [27][1]. Hence, $\mathsf{LLF}_{\mathcal{P}}$ can be viewed as the extension of LF with a family of *monads* indexed with predicates over typed terms, which capture the *effect* of factoring out, or postponing, or delegating to an external oracle the task of providing a proof witness of the verification of the *side-condition* $\mathcal{P}(\Gamma \vdash_\Sigma M : \sigma)$. The basic idea is that any constraint $\mathcal{P}$ can be viewed as a monad $T_{\mathcal{P}}$. Its natural transformation $\eta_{T_{\mathcal{P}}} : A \to T_{\mathcal{P}}(A)$ amounts to a sort of *weakening*, namely any judgement can always be asserted subject to the satisfaction of a given constraint. Correspondingly, the other canonical natural transformation $\mu_{T_{\mathcal{P}}} : T^2_{\mathcal{P}}(A) \to T_{\mathcal{P}}(A)$, amounts to a sort of *contraction*, corresponding to the fact that we trust the verifier, and hence verifying a given constraint twice is redundant.

Being a conservative extension of LF, $\mathsf{LLF}_{\mathcal{P}}$ can be used as a metalanguage for defining logics and proofs. Furthermore, $\mathsf{LLF}_{\mathcal{P}}$ can be used as a *platform* for checking proof arguments that combine different systems or invoke external oracles. Correctness of proofs in $\mathsf{LLF}_{\mathcal{P}}$ is, therefore, *conditionally decidable*, *i.e.* it is decidable provided the external predicate is decidable.

Following the paradigm of Constructive Type Theory, once the new locked type constructor is introduced, we introduce also the corresponding lock constructor for *terms*, which we continue to denote as $\mathcal{L}^{\mathcal{P}}_{M,\sigma}[\cdot]$, together with the *unlock* destructor for terms $\mathcal{U}^{\mathcal{P}}_{M,\sigma}[\cdot]$. This latter term constructor allows one to exit the monadic world once the constraint has been satisfied. Because of the peculiar nature of the lock-monad, which set-theoretically corresponds to taking the *singleton* elements of a set, we can use the very unlock destructor instead of Moggi's let_T destructor [31], normally used in dealing with monads. This greatly simplifies the equational theory.

In this paper, we establish the full language theory of the Lax Logical Framework, $\mathsf{LLF}_{\mathcal{P}}$, by *reducing* it to that of LF itself, *i.e.*, by means of a *metacircular* interpretation of $\mathsf{LLF}_{\mathcal{P}}$-derivations as LF derivations. This encoding is adequate and *shallow* enough so that we can transfer to $\mathsf{LLF}_{\mathcal{P}}$ all the main properties of LF. This approach generalizes to *derivations* the idea underpinning the mapping normally used in the literature to prove normalization of terms in LF-like systems [19, 7].

Differently from earlier systems with locked types, *e.g.*, $\mathsf{LF}_{\mathcal{P}}$, the system $\mathsf{LLF}_{\mathcal{P}}$ allows one to reason "under locks". This allows for natural encodings of side conditions as appear for instance in the ξ_v rule of the call-by-value λ_v-calculus, see Section 5.1.

We discuss encodings in $\mathsf{LLF}_{\mathcal{P}}$ of various logical systems, thereby showing that $\mathsf{LLF}_{\mathcal{P}}$ is the appropriate metalanguage for dealing formally with side-conditions, as well as external

[1]The version of $\mathsf{LLF}_{\mathcal{P}}$ introduced here is both a restriction and an *errata corrigenda* of the system in [27]. The present system is a restriction w.r.t. [27], in that the assumptions of the ($O{\cdot}Guarded{\cdot}Unlock$) rule are less general than the one in [27], but it is an *errata corrigenda* in that the present rule is slightly rephrased and the new rule ($F{\cdot}Guarded{\cdot}Unlock$) is introduced, so as to allow one to prove the subject-reduction without any assumptions. Hence we discard the system in [27], and replace it by the present one even in [27]. We call, therefore, the system in the present paper *the* Lax Logical Framework, even if this name was already used for the one in [27]. Signatures and derivations discussed in [27] carry through in the present version "as is".

and non-apodictic evidence. These examples illustrate the extra expressiveness w.r.t. previous systems given by the monadic understanding of locks, namely the possibility of using *guarded* unlocks $\mathcal{U}^{\mathcal{P}}_{M,\sigma}[\cdot]$, even if the property has not been yet established. Thus, signatures become much more flexible, hence achieving the full modularity that we have been looking for in recent years. We briefly discuss also a famous system introduced by Fitch [16] of a consistent *Naive Set Theory*.

In conclusion, in this paper:

(1) we extend the well understood principle of the LF *paradigm* for explaining a logic, *i.e.* *judgments as types*, *rules* or *hypothetical judgements as higher-order types*, *schemata as higher-order functions*, and *quantified variables as bound metalanguage variables*, with the new clauses: *side conditions as monads* and *external evidence as monads*;
(2) we support the capacity of combining logical systems and relating them to software tools using a simple communication paradigm via "wrappers".

1.1. **Related work.** This paper builds on the earlier work of the authors [24, 26, 25, 22] and was inspired by the very extensive work on Logical Frameworks by [33, 40, 12, 32, 34, 35]. The term *"Lax"* is borrowed from [13, 29], and indeed our system can be viewed as a generalization, to a family of dependent lax operators, of the work carried out there, as well as Moggi's *partial* λ-calculus [30]. A correspondence between lax modalities and monads in functional programming was pointed out in [2, 17]. The connection between constraints and monads in logic programming was considered in the past, *e.g.*, in [32, 15, 14], but to our knowledge, this is the first paper which clearly establishes the correspondence between side conditions and monads in a *higher-order dependent type theory* and in logical frameworks.

In [32], the authors introduce a contextual modal logic, where the notion of context is rendered by means of monadic constructs. There are points of contact with our work which should be explored. Here, we only point out that also in their approach they could have done away with the `let` construct in favour of a deeper substitution as we have done.

Schröder Heister has discussed in a number of papers, see *e.g.* [39, 38], various restrictions and side conditions on rules and on the nature of assumptions that one can add to logical systems to prevent the arising of paradoxes. There are some connections between his work and ours and it would be interesting to compare the bearing of his requirements on side conditions being "closed under substitution" to our notion of *well-behaved* predicate. Similarly, there are commonalities between his distinction between *specific* and *unspecific* variables, and our treatment of free variables in well-behaved predicates.

1.2. **Some methodological and philosophical remarks on *non-apodictic* evidence and formalization.** By the term *non-apodictic evidence* we denote the kind of evidence which is not derived within the formal system itself. This is the kind of evidence which normally justifies assumptions or axioms. Often, it finds its roots in the heuristics which originally inspire the argument. Many heuristics are derived from *Physics* or *analogy*. Archimedes was a champion of the former, as it is well documented in his Organon [3], where he anticipates integral calculus by conceiving a geometrical figure as composed of thin slices of a physical object hanging on a balance scale and subject to gravity. Rather than developing *mathematical physics*, he is, in fact, performing *physical mathematics*.

Arguments by authority have never been allowed, but the beauty of some one-line proofs, or of some proofs-without-words, like the jig-saw puzzle proofs of Pythagoras Theorem, lies precisely in the capacity that these justifications have of conveying the intuition of why the statement is plausible. Schopenhauer's [37](ch.15) criticism of Euclid's "brilliant abstract nonsense" proof of Pythagoras Theorem goes precisely in the direction of defending *intuitive evidence*. In order to have a feel for the kind of evidence we term as non-apodictic, consider the following problem: given a point inside a convex polyhedron, there exists a face of the polyhedron such that the projection of the point onto the plane of that face lies inside the face. How can you formalize adequately the following non-apodictic argument: such a face has to exist otherwise we would have a *perpetuum mobile*?

The approach that we put forward in this paper for handling non-apodictic evidence is simple, but not at all simplistic, given the fact that the quest for absolute justification leads to an *infinite regress*. The very *adequacy* of a given formalization rests ultimately on unformalizable justifications and even the very *execution* of a rule relies on some *external unformalizable* convention, which is manifested only when the rule is put into practice. As Alain Badiou puts it in [5]: *"ce qui identifie la philosophie ce ne sont pas les règles d'un discourse, mais la singularité d'un acte"*. The inevitable infinite regress is captured by the *Münchausen trilemma* [1] or by the story of *Achilles and the Tortoise* narrated by Lewis Carroll [11][2]. Ultimately, we can only "Just do it!".

The irreducible and ineliminable role of *conventions* in human activities, even the apparently most formalizable, has been the object of interest of many philosophers in the XXth century, *e.g.* Wittgenstein or Heidegger. We believe that the first one to point this out was the italian political philosopher Antonio Gramsci, who wrote in his Prison Notebooks, 323-43 (Q1112), 1932 "In acquiring one's conception of the world, one always belongs to a particular grouping, which is that of all the social elements that share the same mode of thinking and acting. We are all conformists of some conformism or other, always man-in-the-mass or collective man. The question is this: of what historical type is the conformism, the mass humanity to which one belongs?"

Different proof tools, or proof search mechanisms are simply other kinds of conformisms. Summing up, $\mathsf{LLF}_{\mathcal{P}}$ makes it possible to invoke our conformism within a Logical Framework, and it is formally rigorous in keeping track of when we do that and in permitting us to explain it away when we can.

2. The system $\mathsf{LLF}_{\mathcal{P}}$

In this section, following the standard pattern and conventions of [19], we introduce the syntax and the rules of $\mathsf{LLF}_{\mathcal{P}}$: in Figure 1, we give the syntactic categories of $\mathsf{LLF}_{\mathcal{P}}$, namely signatures, contexts, kinds, families (*i.e.*, types) and objects (*i.e.*, terms), while the main one-step $\beta\mathcal{L}$-reduction rules appear in Figure 2.

The rules for one-step closure under context for kinds are presented in Figure 4 on page 6, while those for families and objects are presented in Figure 3 on page 6, and Figure 5 on page 7. We denote the reflexive and transitive closure of $\rightarrow_{\beta\mathcal{L}}$ by $\twoheadrightarrow_{\beta\mathcal{L}}$. Hence, $\beta\mathcal{L}$-definitional equality is defined in the standard way, as the reflexive, symmetric, and transitive closure of $\beta\mathcal{L}$-reduction on kinds, families, and objects, as illustrated in Figure 6. The language of $\mathsf{LLF}_{\mathcal{P}}$ is the same as that of $\mathsf{LF}_{\mathcal{P}}$ [25]. In particular, w.r.t. classical LF, we add the *lock-types* constructor ($\mathcal{L}$) for building types of the shape $\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho]$, where $\mathcal{P}$ is

[2]Notice that Girard in The Blind Spot [18] provides a possibly different appraisal of the same story.

$$\begin{array}{rcllrcl}
\Sigma & \in & \textit{Signatures} & \Sigma & ::= & \emptyset \mid \Sigma, a{:}K \mid \Sigma, c{:}\sigma \\
K & \in & \textit{Kinds} & K & ::= & \mathsf{Type} \mid \Pi x{:}\sigma.K \\
\sigma, \tau, \rho & \in & \textit{Families (Types)} & \sigma & ::= & a \mid \Pi x{:}\sigma.\tau \mid \sigma\, N \mid \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho] \\
M, N & \in & \textit{Objects} & M & ::= & c \mid x \mid \lambda x{:}\sigma.M \mid M\, N \mid \mathcal{L}^{\mathcal{P}}_{N,\sigma}[M] \mid \mathcal{U}^{\mathcal{P}}_{N,\sigma}[M]
\end{array}$$

Figure 1: The pseudo-syntax of $\mathsf{LLF}_{\mathcal{P}}$

$$(\lambda x{:}\sigma.M)\, N \to_{\beta\mathcal{L}} M[N/x]\ (\beta{\cdot}O{\cdot}Main) \qquad \mathcal{U}^{\mathcal{P}}_{N,\sigma}[\mathcal{L}^{\mathcal{P}}_{N,\sigma}[M]] \to_{\beta\mathcal{L}} M\ (\mathcal{L}{\cdot}O{\cdot}Main)$$

Figure 2: Main one-step-$\beta\mathcal{L}$-reduction rules

$$\frac{\sigma \to_{\beta\mathcal{L}} \sigma'}{\Pi x{:}\sigma.\tau \to_{\beta\mathcal{L}} \Pi x{:}\sigma'.\tau}\ (F{\cdot}\Pi_1{\cdot}\beta\mathcal{L}) \qquad \frac{\tau \to_{\beta\mathcal{L}} \tau'}{\Pi x{:}\sigma.\tau \to_{\beta\mathcal{L}} \Pi x{:}\sigma.\tau'}\ (F{\cdot}\Pi_2{\cdot}\beta\mathcal{L})$$

$$\frac{\sigma \to_{\beta\mathcal{L}} \sigma'}{\sigma\, N \to_{\beta\mathcal{L}} \sigma'\, N}\ (F{\cdot}A_1{\cdot}\beta\mathcal{L}) \qquad \frac{N \to_{\beta\mathcal{L}} N'}{\sigma\, N \to_{\beta\mathcal{L}} \sigma\, N'}\ (F{\cdot}A_2{\cdot}\beta\mathcal{L})$$

$$\frac{N \to_{\beta\mathcal{L}} N'}{\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho] \to_{\beta\mathcal{L}} \mathcal{L}^{\mathcal{P}}_{N',\sigma}[\rho]}\ (F{\cdot}\mathcal{L}_1{\cdot}\beta\mathcal{L}) \qquad \frac{\sigma \to_{\beta\mathcal{L}} \sigma'}{\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho] \to_{\beta\mathcal{L}} \mathcal{L}^{\mathcal{P}}_{N,\sigma'}[\rho]}\ (F{\cdot}\mathcal{L}_2{\cdot}\beta\mathcal{L})$$

$$\frac{\rho \to_{\beta\mathcal{L}} \rho'}{\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho] \to_{\beta\mathcal{L}} \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho']}\ (F{\cdot}\mathcal{L}_3{\cdot}\beta\mathcal{L})$$

Figure 3: $\beta\mathcal{L}$-closure-under-context for families

a predicate on typed judgements. Correspondingly, at the object level, we introduce the *constructor* lock ($\mathcal{L}$) and the *destructor* unlock ($\mathcal{U}$). The intended meaning of the $\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\cdot]$ constructors is that of *logical filters*. Locks can be viewed also as a generalization of the *Lax* modality of [13, 29]. One of the points of this paper is to show that they can be viewed also as *monads*.

$$\frac{\sigma \to_{\beta\mathcal{L}} \sigma'}{\Pi x{:}\sigma.K \to_{\beta\mathcal{L}} \Pi x{:}\sigma'.K}\ (K{\cdot}\Pi_1{\cdot}\beta\mathcal{L}) \qquad \frac{K \to_{\beta\mathcal{L}} K'}{\Pi x{:}\sigma.K \to_{\beta\mathcal{L}} \Pi x{:}\sigma.K'}\ (K{\cdot}\Pi_2{\cdot}\beta\mathcal{L})$$

Figure 4: $\beta\mathcal{L}$-closure-under-context for kinds

For the sake of generality, we allow declarations of the form $x{:}\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau]$ in contexts, *i.e.*, we allow one to declare variables ranging over lock-types, albeit this is not used in practice.

Following the standard specification paradigm of Constructive Type Theory, we define lock-types using *introduction*, *elimination*, and *equality rules*. Namely, we introduce a lock-*constructor* for building objects $\mathcal{L}^{\mathcal{P}}_{N,\sigma}[M]$ of type $\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho]$, via the *introduction rule* $(O{\cdot}Lock)$. Correspondingly, we introduce an unlock-*destructor* $\mathcal{U}^{\mathcal{P}}_{N,\sigma}[M]$ via the *elimination rule* $(O{\cdot}Guarded{\cdot}Unlock)$.

$$\frac{\sigma \to_{\beta\mathcal{L}} \sigma'}{\lambda x{:}\sigma.M \to_{\beta\mathcal{L}} \lambda x{:}\sigma'.M}\ (O{\cdot}\lambda_1{\cdot}\beta\mathcal{L}) \qquad \frac{M \to_{\beta\mathcal{L}} M'}{\lambda x{:}\sigma.M \to_{\beta\mathcal{L}} \lambda x{:}\sigma.M'}\ (O{\cdot}\lambda_2{\cdot}\beta\mathcal{L})$$

$$\frac{M \to_{\beta\mathcal{L}} M'}{M\,N \to_{\beta\mathcal{L}} M'\,N}\ (O{\cdot}A_1{\cdot}\beta\mathcal{L}) \qquad \frac{N \to_{\beta\mathcal{L}} N'}{M\,N \to_{\beta\mathcal{L}} M\,N'}\ (O{\cdot}A_2{\cdot}\beta\mathcal{L})$$

$$\frac{N \to_{\beta\mathcal{L}} N'}{\mathcal{L}^{\mathcal{P}}_{N,\sigma}[M] \to_{\beta\mathcal{L}} \mathcal{L}^{\mathcal{P}}_{N',\sigma}[M]}\ (O{\cdot}\mathcal{L}_1{\cdot}\beta\mathcal{L}) \qquad \frac{\sigma \to_{\beta\mathcal{L}} \sigma'}{\mathcal{L}^{\mathcal{P}}_{N,\sigma}[M] \to_{\beta\mathcal{L}} \mathcal{L}^{\mathcal{P}}_{N,\sigma'}[M]}\ (O{\cdot}\mathcal{L}_2{\cdot}\beta\mathcal{L})$$

$$\frac{M \to_{\beta\mathcal{L}} M'}{\mathcal{L}^{\mathcal{P}}_{N,\sigma}[M] \to_{\beta\mathcal{L}} \mathcal{L}^{\mathcal{P}}_{N,\sigma}[M']}\ (O{\cdot}\mathcal{L}_3{\cdot}\beta\mathcal{L}) \qquad \frac{N \to_{\beta\mathcal{L}} N'}{\mathcal{U}^{\mathcal{P}}_{N,\sigma}[M] \to_{\beta\mathcal{L}} \mathcal{U}^{\mathcal{P}}_{N',\sigma}[M]}\ (O{\cdot}\mathcal{U}_1{\cdot}\beta\mathcal{L})$$

$$\frac{\sigma \to_{\beta\mathcal{L}} \sigma'}{\mathcal{U}^{\mathcal{P}}_{N,\sigma}[M] \to_{\beta\mathcal{L}} \mathcal{U}^{\mathcal{P}}_{N,\sigma'}[M]}\ (O{\cdot}\mathcal{U}_1{\cdot}\beta\mathcal{L}) \qquad \frac{M \to_{\beta\mathcal{L}} M'}{\mathcal{U}^{\mathcal{P}}_{N,\sigma}[M] \to_{\beta\mathcal{L}} \mathcal{U}^{\mathcal{P}}_{N,\sigma}[M']}\ (O{\cdot}\mathcal{U}_1{\cdot}\beta\mathcal{L})$$

Figure 5: $\beta\mathcal{L}$-closure-under-context for objects

$$\frac{T \to_{\beta\mathcal{L}} T'}{T =_{\beta\mathcal{L}} T'}\ (\beta\mathcal{L}{\cdot}Eq{\cdot}Main) \qquad \frac{}{T =_{\beta\mathcal{L}} T}\ (\beta\mathcal{L}{\cdot}Eq{\cdot}Refl)$$

$$\frac{T =_{\beta\mathcal{L}} T'}{T' =_{\beta\mathcal{L}} T}\ (\beta\mathcal{L}{\cdot}Eq{\cdot}Sym) \qquad \frac{T =_{\beta\mathcal{L}} T' \qquad T' =_{\beta\mathcal{L}} T''}{T =_{\beta\mathcal{L}} T''}\ (\beta\mathcal{L}{\cdot}Eq{\cdot}Trans)$$

Figure 6: $\beta\mathcal{L}$-definitional equality

The introduction rule of lock-types corresponds to the introduction rule of monads. The correspondence with the elimination rule for monads is not so immediate because the latter is normally given using a let_T-construct. The correspondence becomes clear once we realize that $let_{T_{\mathcal{P}(\Gamma\vdash S:\sigma)}} x = M\ in\ N$ can be safely replaced by $N[\mathcal{U}^{\mathcal{P}}_{S,\sigma}[M]/x]$ since the $\mathcal{L}^{\mathcal{P}}_{S,\sigma}[\cdot]$-monads satisfy the property $let_{T_{\mathcal{P}}}\ x = M\ in\ N \to N$ if $x \notin FV(N)$, provided x occurs *guarded* in N, *i.e.* within subterms of the appropriate locked-type.

But, since we do not use the traditional let_T construct in elimination rules, we have to take care of elimination also at the level of types by means of the rule *(F·Guarded·Unlock)*. Moreover both rules *(F·Guarded·Unlock)* and *(O·Guarded·Unlock)* need to be merged with *equality* to preserve subject reduction.

These rules give evidence to the understanding of *locks as monads*. Indeed, given a predicate $\mathcal{P}$ and $\Gamma \vdash_\Sigma N : \sigma$, the intended monad $(T_{\mathcal{P}}, \eta, \mu)$ can be naturally defined on the term model of $\mathsf{LLF}_{\mathcal{P}}$ viewed as a category. In particular $\eta_\rho \triangleq \lambda x{:}\rho.\mathcal{L}^{\mathcal{P}}_{N,\sigma}[x]$ and $\mu_\rho \triangleq \lambda x{:}\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho]].\ \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\mathcal{U}^{\mathcal{P}}_{N,\sigma}[\mathcal{U}^{\mathcal{P}}_{N,\sigma}[x]]]$. Indeed, if $\Gamma, x{:}\rho \vdash_\Sigma N : \sigma$ is derivable, the term for η can be easily inferred by applying rules $(O{\cdot}Var)$, $(O{\cdot}Lock)$, and $(O{\cdot}Abs)$ as follows:

$$\frac{\dfrac{\Gamma, x{:}\rho \vdash_\Sigma x : \rho \quad \Gamma, x{:}\rho \vdash_\Sigma N : \sigma}{\Gamma, x{:}\rho \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{N,\sigma}[x] : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho]}}{\Gamma \vdash_\Sigma \lambda x{:}\rho.\mathcal{L}^{\mathcal{P}}_{N,\sigma}[x] : \Pi x{:}\rho.\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho]}$$

Signature rules

$$\frac{}{\emptyset\ \mathsf{sig}}\ (S{\cdot}Empty) \qquad \frac{\vdash_\Sigma K \quad a \notin \mathsf{Dom}(\Sigma)}{\Sigma, a{:}K\ \mathsf{sig}}\ (S{\cdot}Kind) \qquad \frac{\vdash_\Sigma \sigma{:}\mathsf{Type} \quad c \notin \mathsf{Dom}(\Sigma)}{\Sigma, c{:}\sigma\ \mathsf{sig}}\ (S{\cdot}Type)$$

Context rules

$$\frac{\Sigma\ \mathsf{sig}}{\vdash_\Sigma \emptyset}\ (C{\cdot}Empty) \qquad \frac{\Gamma \vdash_\Sigma \sigma{:}\mathsf{Type} \quad x \notin \mathsf{Dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}\sigma}\ (C{\cdot}Type)$$

Kind rules

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \mathsf{Type}}\ (K{\cdot}Type) \qquad \frac{\Gamma, x{:}\sigma \vdash_\Sigma K}{\Gamma \vdash_\Sigma \Pi x{:}\sigma.K}\ (K{\cdot}Pi)$$

Family rules

$$\frac{\vdash_\Sigma \Gamma \quad a{:}K \in \Sigma}{\Gamma \vdash_\Sigma a : K}\ (F{\cdot}Const) \qquad \frac{\Gamma, x{:}\sigma \vdash_\Sigma \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \Pi x{:}\sigma.\tau : \mathsf{Type}}\ (F{\cdot}Pi)$$

$$\frac{\Gamma \vdash_\Sigma \sigma : \Pi x{:}\tau.K \quad \Gamma \vdash_\Sigma N : \tau}{\Gamma \vdash_\Sigma \sigma\, N : K[N/x]}\ (F{\cdot}App) \qquad \frac{\Gamma \vdash_\Sigma \rho : \mathsf{Type} \quad \Gamma \vdash_\Sigma N : \sigma}{\Gamma \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho] : \mathsf{Type}}\ (F{\cdot}Lock)$$

$$\frac{\Gamma \vdash_\Sigma \sigma : K \quad \Gamma \vdash_\Sigma K' \quad K{=}_{\beta\mathcal{L}}K'}{\Gamma \vdash_\Sigma \sigma : K'}\ (F{\cdot}Conv)$$

Object rules

$$\frac{\vdash_\Sigma \Gamma \quad c{:}\sigma \in \Sigma}{\Gamma \vdash_\Sigma c : \sigma}\ (O{\cdot}Const) \qquad \frac{\vdash_\Sigma \Gamma \quad x{:}\sigma \in \Gamma}{\Gamma \vdash_\Sigma x : \sigma}\ (O{\cdot}Var)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma M : \tau}{\Gamma \vdash_\Sigma \lambda x{:}\sigma.M : \Pi x{:}\sigma.\tau}\ (O{\cdot}Abs) \qquad \frac{\Gamma \vdash_\Sigma M : \Pi x{:}\sigma.\tau \quad \Gamma \vdash_\Sigma N : \sigma}{\Gamma \vdash_\Sigma M\, N : \tau[N/x]}\ (O{\cdot}App)$$

$$\frac{\Gamma \vdash_\Sigma M : \sigma \quad \Gamma \vdash_\Sigma \tau : \mathsf{Type} \quad \sigma{=}_{\beta\mathcal{L}}\tau}{\Gamma \vdash_\Sigma M : \tau}\ (O{\cdot}Conv) \qquad \frac{\Gamma \vdash_\Sigma M : \rho \quad \Gamma \vdash_\Sigma N : \sigma}{\Gamma \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{N,\sigma}[M] : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho]}\ (O{\cdot}Lock)$$

$$\frac{\Gamma \vdash_\Sigma M : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho] \quad \mathcal{P}(\Gamma \vdash_\Sigma N : \sigma)}{\Gamma \vdash_\Sigma \mathcal{U}^{\mathcal{P}}_{N,\sigma}[M] : \rho}\ (O{\cdot}Top{\cdot}Unlock)$$

$$\frac{\Gamma, x{:}\tau \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\rho] : \mathsf{Type} \quad \Gamma \vdash_\Sigma N : \mathcal{L}^{\mathcal{P}}_{S',\sigma'}[\tau] \quad \sigma{=}_{\beta\mathcal{L}}\sigma' \quad S{=}_{\beta\mathcal{L}}S'}{\Gamma \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\rho[\mathcal{U}^{\mathcal{P}}_{S',\sigma'}[N]/x]] : \mathsf{Type}}\ (F{\cdot}Guarded{\cdot}Unlock)$$

$$\frac{\Gamma, x{:}\tau \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{S,\sigma}[M] : \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\rho] \quad \Gamma \vdash_\Sigma N : \mathcal{L}^{\mathcal{P}}_{S',\sigma'}[\tau] \quad \sigma{=}_{\beta\mathcal{L}}\sigma' \quad S{=}_{\beta\mathcal{L}}S'}{\Gamma \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{S,\sigma}[M[\mathcal{U}^{\mathcal{P}}_{S',\sigma'}[N]/x]] : \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\rho[\mathcal{U}^{\mathcal{P}}_{S',\sigma'}[N]/x]]}\ (O{\cdot}Guarded{\cdot}Unlock)$$

Figure 7: The $\mathsf{LLF}_{\mathcal{P}}$ Type System

As for the term for μ, if $\Gamma \vdash_\Sigma N : \sigma$ is derivable, applying weakening and the rules $(O{\cdot}Var)$, $(O{\cdot}Lock)$, and $(O{\cdot}Guarded{\cdot}Unlock)$, we can derive the following:

$$\frac{\dfrac{\Gamma, z_2{:}\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau], z_1{:}\tau \vdash_\Sigma z_1 : \tau \quad \Gamma, z_2{:}\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau], z_1{:}\tau \vdash_\Sigma N : \sigma}{\Gamma, z_2{:}\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau], z_1{:}\tau \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{N,\sigma}[z_1] : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau]} \qquad \Gamma, z_2{:}\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau] \vdash_\Sigma z_2 : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau]}{\Gamma, z_2{:}\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau] \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\mathcal{U}^{\mathcal{P}}_{N,\sigma}[z_2]] : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau]}$$

Whence, if $x{:}\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau]] \in \Gamma$, we can derive the following, applying again rules $(O{\cdot}Var)$, and $(O{\cdot}Guarded{\cdot}Unlock)$:

$$\frac{\Gamma, z_2{:}\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau] \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\mathcal{U}^{\mathcal{P}}_{N,\sigma}[z_2]] : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau] \quad \Gamma, z_2{:}\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau] \vdash_\Sigma x : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau]]}{\Gamma \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\mathcal{U}^{\mathcal{P}}_{N,\sigma}[\mathcal{U}^{\mathcal{P}}_{N,\sigma}[x]]] : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau]}$$

And, finally, applying rule $(O{\cdot}Abs)$, we get the term $\lambda x{:}\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau]].\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\mathcal{U}^{\mathcal{P}}_{N,\sigma}[\mathcal{U}^{\mathcal{P}}_{N,\sigma}[x]]]$. Finally, to provide the intended meaning of $\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\cdot]$, we need to introduce in $\mathsf{LLF}_{\mathcal{P}}$ also the rule $(O{\cdot}Top{\cdot}Unlock)$, which allows for the elimination of the lock-type constructor if the predicate $\mathcal{P}$ is verified, possibly *externally*, on an appropriate and derivable judgement. Figure 7 shows the full typing system of $\mathsf{LLF}_{\mathcal{P}}$. All *type equality rules* of $\mathsf{LLF}_{\mathcal{P}}$ use a notion of conversion which is a combination of standard β-reduction, $(\beta{\cdot}O{\cdot}Main)$, with another notion of reduction $(\mathcal{L}{\cdot}O{\cdot}Main)$, called $\mathcal{L}$-reduction. The latter behaves as a lock-releasing mechanism, erasing the $\mathcal{U}$-$\mathcal{L}$ pair in a term of the form $\mathcal{U}^{\mathcal{P}}_{N,\sigma}[\mathcal{L}^{\mathcal{P}}_{N,\sigma}[M]]$. Lock-types have been discussed by the authors in a series of papers [9, 24, 26, 25, 22], but *Guarded Unlock* rules, first suggested in [27] have not been fully discussed before. These rules are crucial, because otherwise in order to release a locked term it is necessary to query the external oracle *explicitly*, by means of the rule $(O{\cdot}Top{\cdot}Unlock)$, and obtain a positive answer. This is rather heavy from the practical point of view, because it might force the invocation of an external tool more than once for the same property. Moreover, such properties are not essential to the main thrust of the proof and one would like to be free to proceed with the main argument, postponing the verification of "details" as much as possible. But, more importantly, such rules allow us to exploit hypothetic-general locked judgements in encoding rules, as in the case of the call-by-value λ-calculus, see [4], and refer to terms in locked types by pattern matching. The improvement in all the case studies is neat w.r.t. plain old $\mathsf{LF}_{\mathcal{P}}$ [25]. Namely, even if at a given stage of the proof development we assume (or are not able, or we do not want to waste time to verify) a side-condition, we can *postpone* such a task, by unlocking immediately the given term and by proceeding with the proof. The lock-type of the term into which we release the unlocked term will keep track that the verification has to be carried out, sooner or later.

The Guarded Unlock rules, namely $(O{\cdot}Guarded{\cdot}Unlock)$ and $(F{\cdot}Guarded{\cdot}Unlock)$ are the novelty w.r.t. the extended abstract of the present paper which appeared in [27]. First of all in [27] there was no Guarded Unlock rule at the level of Type Families, but this appears to be necessary to recover a standard proof of the sub-derivation property. As far as the Guarded Unlock rule at the level of Objects, the new $(O{\cdot}Guarded{\cdot}Unlock)$-rule is, first of all, a restriction of the one in [27]. Namely, we require that the subject of the first premise has an *explicit* outermost lock, otherwise we can derive unlocked terms also at the top level, if locked variables appear in the assumptions. This external lock forces the establishment of all pending constraints before the nested unlock can surface. We could have ruled out locked assumptions, but this restriction allows for a smoother formulation of the language theory of $\mathsf{LLF}_{\mathcal{P}}$, as will be shown in Section 4. Furthermore, the new version of the $(O{\cdot}Guarded{\cdot}Unlock)$-rule uses type equality judgements explicitly. Namely the two minor premises ($\sigma{=}_{\beta\mathcal{L}}\sigma'$ and $S{=}_{\beta\mathcal{L}}S'$) in the $(O{\cdot}Guarded{\cdot}Unlock)$-rule allow for $\beta\mathcal{L}$-conversion in the subscripts σ and S of the lock/unlock operators. This appears to be necessary for subject reduction.

We conclude this section by recalling that, since external predicates affect reductions in $\mathsf{LLF}_{\mathcal{P}}$, they must be *well-behaved* in order to preserve subject reduction. And this property

is needed for *decidability*, *relative to* an oracle, which is essential in LF's. Let α be a shorthand for any "subject of type predicate", we introduce the crucial definition:

Definition 2.1 (Well-behaved predicates, [25])**.** A finite set of predicates $\{\mathcal{P}_i\}_{i\in I}$ is *well-behaved* if each $\mathcal{P}$ in the set satisfies the following conditions:

(1) ***Closure under signature and context weakening and permutation:***
   (a) If Σ and Ω are valid signatures such that $\Sigma \subseteq \Omega$ and $\mathcal{P}(\Gamma \vdash_\Sigma \alpha)$, then $\mathcal{P}(\Gamma \vdash_\Omega \alpha)$.
   (b) If Γ and Δ are valid contexts such that $\Gamma \subseteq \Delta$ and $\mathcal{P}(\Gamma \vdash_\Sigma \alpha)$, then $\mathcal{P}(\Delta \vdash_\Sigma \alpha)$.

(2) ***Closure under substitution:*** If $\mathcal{P}(\Gamma, x{:}\sigma', \Gamma' \vdash_\Sigma N : \sigma)$ and $\Gamma \vdash_\Sigma N' : \sigma'$, then $\mathcal{P}(\Gamma, \Gamma'[N'/x] \vdash_\Sigma N[N'/x] : \sigma[N'/x])$.

(3) ***Closure under reduction:***
   (a) If $\mathcal{P}(\Gamma \vdash_\Sigma N : \sigma)$ and $N \to_{\beta\mathcal{L}} N'$, then $\mathcal{P}(\Gamma \vdash_\Sigma N' : \sigma)$.
   (b) If $\mathcal{P}(\Gamma \vdash_\Sigma N : \sigma)$ and $\sigma \to_{\beta\mathcal{L}} \sigma'$, then $\mathcal{P}(\Gamma \vdash_\Sigma N : \sigma')$.

3. Encoding $\mathsf{LLF}_\mathcal{P}$ in LF

In this section we define a very *shallow encoding* of $\mathsf{LLF}_\mathcal{P}$ in Edinburgh LF [19]. This translation has two purposes. On one hand we *explain* the "gist" of $\mathsf{LLF}_\mathcal{P}$, using the *normative* LF paradigm. On the other hand, we provide a tool for transferring properties such as *confluence, normalization* and *subject reduction* from LF to $\mathsf{LLF}_\mathcal{P}$. This approach generalizes the proof technique used in the literature for proving normalization of dependent type systems relative to their corresponding purely propositional variant, *e.g.*, LF relative to the simply typed λ-calculus, or the Calculus of Constructions relative to F_ω [19, 7].

The embedding of $\mathsf{LLF}_\mathcal{P}$ into LF is given by an inductive, *i.e. compositional*, function which maps derivations in $\mathsf{LLF}_\mathcal{P}$ to derivations in LF. The critical instances occur in relation to lock-types, as was to be expected. The key idea of the encoding is based on the analogy *locks as abstractions* and *unlocks as applications*. To this end we introduce new type-constants in LF to represent lock-types in $\mathsf{LLF}_\mathcal{P}$, appropriate object constants to represent external evidence, and use appropriate object variables to represent hypothetical external evidence. Hence locked types become Π-types over such new types and locked terms become abstractions over such new types.

Before entering into the intricacies of the encoding, we illustrate, suggestively, how the translation of the basic lock-related rules would appear in a non-dependent purely propositional fragment, if there were just one single predicate represented by the proposition, *i.e.* type, L:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \lambda x{:}L.M : L \to A}\ (O{\cdot}Lock)$$

$$\frac{\Gamma \vdash M : L \to A \quad \Gamma \vdash c : L}{\Gamma \vdash Mc : A}\ (O{\cdot}Top{\cdot}Unlock)$$

$$\frac{\Gamma, x{:}B \vdash \lambda y{:}L.M : L \to A \quad \Gamma \vdash N : L \to B}{\Gamma \vdash \lambda y{:}L.M[Ny/x] : L \to A}\ (O{\cdot}Guarded{\cdot}Unlock)$$

Resuming full generality, each use of the predicate $\mathcal{P}$ in an $\mathsf{LLF}_\mathcal{P}$ derivation, relative to a context Γ, term N and type σ is encoded by a corresponding LF-type denoted by

$\mathcal{P}(x_1, \ldots, x_n, \sigma', N')$ where $\{x_1, \ldots, x_n\} \equiv \mathsf{Dom}(\Gamma)$[3] and σ', N' are the encodings in LF of σ and N, respectively. However, since LF is not a polymorphic type theory, we cannot feed σ' directly to the constant $\mathcal{P}$. Hence, we use a simple "trick" representing σ' indirectly by means of the identity function $\lambda x{:}\sigma'.x$ (or $I_{\sigma'}$ for short). Thus for each predicate $\mathcal{P}$ in $\mathsf{LLF}_{\mathcal{P}}$, we introduce in LF two families of constants depending on the environment $\Gamma \equiv x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$, the signature Σ, and the type $\Gamma \vdash_\Sigma \sigma : \mathsf{Type}$ as follows:

$$P^{\Sigma}_{\Gamma} : \Pi x_1{:}\sigma'_1 \ldots x_n{:}\sigma'_n.(\sigma' \to \sigma') \to \sigma' \to \mathsf{Type}$$

and

$$c_{\mathcal{P}^{\Sigma}_{\Gamma}} : \Pi x_1{:}\sigma'_1 \ldots x_n{:}\sigma'_n.\Pi x{:}\sigma' \to \sigma'.y{:}\sigma'.(\mathcal{P}^{\Sigma}_{\Gamma} x_1 \ldots x_n x y).$$

where σ'_i $(1 \leq i \leq n)$ and σ' are the encodings in LF of σ_i and σ, respectively. The former constants are used to encode the lock-type, in such a way that the derivation of the term $(c_{\mathcal{P}^{\Sigma}_{\Gamma}}\ x_1 \ldots x_n\ I_{\sigma'}\ N')$ or of a variable of type $(P^{\Sigma}_{\Gamma}\ x_1 \ldots x_n\ I_{\sigma'}\ N')$ will encode in LF the fact that the external judgment $\mathcal{P}(\Gamma \vdash_\Sigma N : \sigma)$ of $\mathsf{LLF}_{\mathcal{P}}$ holds or it is assumed to hold. Notice that the properties of *well-behaved* predicates ensure precisely that such encodings can be safely introduced without implicitly enforcing the validity of any spurious judgement. In the following, we will abbreviate the list $x_1, x_2, \ldots, x_n$ as $\vec{x}$, whenever it will be clear from the context the origin of the x_i's. Moreover, we will drop the Σ and Γ in the notation of the constants $\mathcal{P}^{\Sigma}_{\Gamma}$ and $c_{\mathcal{P}^{\Sigma}_{\Gamma}}$.

For the above reasons, if the judgment labelling the root of a derivation tree in $\mathsf{LLF}_{\mathcal{P}}$ is, $\Gamma \vdash_\Sigma \ M : \sigma$, the signature of the corresponding judgement in LF is not, in general, a one-to-one translation of the declarations contained in Σ. Further constants are needed for encoding predicates and external evidence, be it concrete if it derives from the oracle's call and a $(Top{\cdot}Unlock)$ rule, or hypothetical if it derives from a $(Guarded{\cdot}Unlock)$ rule.

More precisely, the encoding function, denoted by ϵ in the following, will yield, as the translation progresses, an LF-signature which possibly increases from the initially empty one, with a

(1) possibly fresh $\mathcal{P}$-like constant whenever a lock or unlock operator is introduced in rules $(F{\cdot}Lock)$ and $(O{\cdot}Lock)$ and $(O{\cdot}Top{\cdot}Unlock)$ and $(F{\cdot}Guarded{\cdot}Unlock)$, and $(O{\cdot}Guarded{\cdot}Unlock)$;
(2) possibly fresh $c_{\mathcal{P}}$ constant, witnessing the *external evidence*, introduced in rule $(O{\cdot}Top{\cdot}Unlock)$. Notice that the translation of the first premise of that rule, which involves the lock-type already provides the constant $\mathcal{P}$.

As a consequence, in translating a rule which has two or more premises it is necessary to *merge* the resulting signatures from the corresponding translations. The function Merge concatenates the declarations in the input signatures passed as arguments, pruning out possible duplications. Merging signatures requires *engrafting* subtrees, of the appropriate derivations, in the derivations of the original signatures, thus establishing the validity of the "augmented" counterparts. We denote this, ultimately straightforward, "rearrangement" with the notation $(\mathcal{D})^+$ in Figures 8, 9, 10, 11, 12, 13, 14, 15, 16, and 17.

In the following, for the sake of simplicity and readability, we will denote the result of the application of the mapping function ϵ (see Figure 18) on terms with an overline

[3]By inspection on the clauses of the encoding function ϵ (introduced later in this section), it is clear that, if $\{x_1, \ldots, x_n\}$ is the domain of the original typing context in $\mathsf{LLF}_{\mathcal{P}}$, then it will also be the domain of its encoding in LF.

Figure 8: Encoding of signature rules

$$\epsilon\left(\ \frac{}{\emptyset\ \mathsf{sig}}\ (S{\cdot}Empty)\ \right) \Longrightarrow\ \overline{\emptyset\ \mathsf{sig}}$$

$$\epsilon\left(\ \frac{\dfrac{\mathcal{D}}{\vdash_{\Sigma'} K}\quad a \notin \mathsf{Dom}(\Sigma')}{\Sigma', a{:}K\ \mathsf{sig}}\ (S{\cdot}Kind)\ \right) \Longrightarrow\ \frac{\dfrac{\mathcal{D}'}{\vdash_{\Sigma''} K'}\quad a \notin \mathsf{Dom}(\Sigma'')}{\Sigma'', a{:}K'\ \mathsf{sig}}$$

$$\text{where } \epsilon\left(\ \frac{\mathcal{D}}{\vdash_{\Sigma'} K}\ \right) \Longrightarrow\ \frac{\mathcal{D}'}{\vdash_{\Sigma''} K'}$$

$$\epsilon\left(\ \frac{\dfrac{\mathcal{D}}{\vdash_{\Sigma'} \sigma : \mathsf{Type}}\quad c \notin \mathsf{Dom}(\Sigma')}{\Sigma', c{:}\sigma\ \mathsf{sig}}\ (S{\cdot}Type)\ \right) \Longrightarrow\ \frac{\dfrac{\mathcal{D}'}{\vdash_{\Sigma''} \sigma' : \mathsf{Type}}\quad c \notin \mathsf{Dom}(\Sigma'')}{\Sigma'', c{:}\sigma'\ \mathsf{sig}}$$

$$\text{where } \epsilon\left(\ \frac{\mathcal{D}}{\vdash_{\Sigma'} \sigma : \mathsf{Type}}\ \right) \Longrightarrow\ \frac{\mathcal{D}'}{\vdash_{\Sigma''} \sigma' : \mathsf{Type}}$$

Figure 9: Encoding of typing context rules

$$\epsilon\left(\ \frac{\dfrac{\mathcal{D}}{\Sigma'\ \mathsf{sig}}}{\vdash_{\Sigma} \emptyset}\ (C{\cdot}Empty)\ \right) \Longrightarrow\ \frac{\dfrac{\mathcal{D}'}{\Sigma''\ \mathsf{sig}}}{\vdash_{\Sigma''} \emptyset} \qquad \text{where } \epsilon\left(\ \frac{\mathcal{D}}{\Sigma'\ \mathsf{sig}}\ \right) \Longrightarrow\ \frac{\mathcal{D}'}{\Sigma''\ \mathsf{sig}}$$

$$\epsilon\left(\ \frac{\dfrac{\mathcal{D}_1}{\vdash_{\Sigma'} \Gamma'}\quad \dfrac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} \sigma : \mathsf{Type}}\ (1)}{\vdash_{\Sigma'} \Gamma', x{:}\sigma}\ (C{\cdot}Type)\ \right) \Longrightarrow\ \frac{\dfrac{(\mathcal{D}_1')^{+}}{\vdash_{\Sigma^{iv}} \Gamma''}\quad \dfrac{(\mathcal{D}_2')^{+}}{\Gamma'' \vdash_{\Sigma^{iv}} \sigma' : \mathsf{Type}}\ (2)}{\vdash_{\Sigma^{iv}} \Gamma'', x{:}\sigma'}$$

$$\text{where } \epsilon\left(\ \frac{\mathcal{D}_1}{\vdash_{\Sigma'} \Gamma'}\ \right) \Longrightarrow\ \frac{\mathcal{D}_1'}{\vdash_{\Sigma''} \Gamma''}\ \text{, and } \epsilon\left(\ \frac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} \sigma : \mathsf{Type}}\ \right) \Longrightarrow\ \frac{\mathcal{D}_2'}{\Gamma'' \vdash_{\Sigma'''} \sigma' : \mathsf{Type}}\ \text{, and}$$

$$\Sigma^{iv} \triangleq \mathsf{Merge}(\Sigma'', \Sigma'''), \text{ and } (1) \triangleq x \notin \mathsf{Dom}(\Gamma'), \text{ and } (2) \triangleq x \notin \mathsf{Dom}(\Gamma'')$$

$(\overline{\ })$, whenever it will be clear which are the signature and the environment involved. The notation is also extended to signatures and typing environments in the obvious way.

Finally we point out that the function ϵ induces a *compositional* map from kinds, families, and objects in $\mathsf{LLF}_{\mathcal{P}}$ to the corresponding categories in LF. We denote such a map by θ_{Γ}^{Σ} and we provide an independent inductive definition in Figure 18. It receives as input parameters the signature Σ and the typing context Γ synthesized by the map ϵ encoding derivations.

Starting from signatures and contexts, we have the encoding clauses of Figures 8 and 9. In particular, notice how rules having two premises, *i.e.* rules $(S{\cdot}Kind)$, $(S{\cdot}Type)$, and $(C{\cdot}Type)$, are dealt with. In encoding locks in LF we extend the signature, therefore we have to reflect on the encoding of the derivation of the first premise the effects of encoding the second premise and viceversa.

In Figures 10, 11, 12, 13, 14, 15, 16 and 17 appear the clauses defining the encoding of derivations concerning terms (*i.e.*, kinds, families, and objects). The *key clause* of our

Figure 10: Encoding of kind rules

$$\epsilon\left(\frac{\dfrac{\mathcal{D}}{\vdash_{\Sigma'}\Gamma'}}{\Gamma'\vdash_{\Sigma'}\mathsf{Type}}\ (K{\cdot}Type)\right)\Longrightarrow\frac{\dfrac{\mathcal{D}'}{\vdash_{\Sigma''}\Gamma''}}{\Gamma''\vdash_{\Sigma''}\mathsf{Type}}\quad\text{where }\epsilon\left(\frac{\mathcal{D}}{\vdash_{\Sigma'}\Gamma'}\right)\Longrightarrow\frac{\mathcal{D}'}{\vdash_{\Sigma''}\Gamma''}$$

$$\epsilon\left(\frac{\dfrac{\mathcal{D}}{\Gamma',x{:}\sigma\vdash_{\Sigma'}K}}{\Gamma'\vdash_{\Sigma'}\Pi x{:}\sigma.K}\ (K{\cdot}Pi)\right)\Longrightarrow\frac{\dfrac{\mathcal{D}'}{\Gamma'',x{:}\sigma'\vdash_{\Sigma''}K'}}{\Gamma''\vdash_{\Sigma''}\Pi x{:}\sigma'.K'}\quad\text{where }\epsilon\left(\frac{\mathcal{D}}{\Gamma',x{:}\sigma\vdash_{\Sigma'}K}\right)\Longrightarrow\frac{\mathcal{D}'}{\Gamma'',x:\sigma'\vdash_{\Sigma''}K'}$$

Figure 11: Encoding of family rules - Pt.1

$$\epsilon\left(\frac{\dfrac{\mathcal{D}}{\vdash_{\Sigma'}\Gamma'}\quad a{:}K\in\Sigma'}{\Gamma'\vdash_{\Sigma'}a:K}\ (F{\cdot}Const)\right)\Longrightarrow\frac{\dfrac{\mathcal{D}'}{\vdash_{\Sigma''}\Gamma''}\quad a{:}K'\in\Sigma''}{\Gamma''\vdash_{\Sigma''}a:K'}\quad\text{where }\epsilon\left(\frac{\mathcal{D}}{\vdash_{\Sigma'}\Gamma'}\right)\Longrightarrow\frac{\mathcal{D}'}{\vdash_{\Sigma''}\Gamma''}$$

$$\epsilon\left(\frac{\dfrac{\mathcal{D}}{\Gamma',x{:}\sigma\vdash_{\Sigma'}\tau:\mathsf{Type}}}{\Gamma'\vdash_{\Sigma'}\Pi x{:}\sigma.\tau:\mathsf{Type}}\ (F{\cdot}Pi)\right)\Longrightarrow\frac{\dfrac{\mathcal{D}'}{\Gamma'',x{:}\sigma'\vdash_{\Sigma''}\tau':\mathsf{Type}}}{\Gamma''\vdash_{\Sigma''}\Pi x{:}\sigma.'\tau':\mathsf{Type}}\quad\text{where }\epsilon\left(\frac{\mathcal{D}}{\Gamma',x{:}\sigma\vdash_{\Sigma'}\tau:\mathsf{Type}}\right)\Longrightarrow\frac{\mathcal{D}'}{\Gamma'',x{:}\sigma'\vdash_{\Sigma''}\tau':\mathsf{Type}}$$

$$\epsilon\left(\frac{\dfrac{\mathcal{D}_1}{\Gamma'\vdash_{\Sigma'}\sigma:\Pi x{:}\tau.K}\quad\dfrac{\mathcal{D}_2}{\Gamma'\vdash_{\Sigma'}N:\tau}}{\Gamma'\vdash_{\Sigma'}\sigma\,N:K[N/x]}\ (F{\cdot}App)\right)\Longrightarrow\frac{\dfrac{(\mathcal{D}_1')^+}{\Gamma''\vdash_{\Sigma^{iv}}\sigma':\Pi x{:}\tau.'K'}\quad\dfrac{(\mathcal{D}_2')^+}{\Gamma''\vdash_{\Sigma^{iv}}N':\tau'}}{\Gamma''\vdash_{\Sigma^{iv}}\sigma\,'N':K'[N'/x]}$$

$$\text{where }\epsilon\left(\frac{\mathcal{D}_1}{\Gamma'\vdash_{\Sigma'}\sigma:\Pi x{:}\tau.K}\right)\Longrightarrow\frac{\mathcal{D}_1'}{\Gamma''\vdash_{\Sigma''}\sigma':\Pi x{:}\tau.'K'}\text{ , and}$$

$$\epsilon\left(\frac{\mathcal{D}_2}{\Gamma'\vdash_{\Sigma'}N:\tau}\right)\Longrightarrow\frac{\mathcal{D}_2'}{\Gamma''\vdash_{\Sigma'''}N':\tau'}\text{ , and }\Sigma^{iv}\triangleq\mathsf{Merge}(\Sigma'',\Sigma''')$$

encoding is $(F{\cdot}Lock)$, mapping a lock type in $\mathsf{LLF}_{\mathcal{P}}$ to a Π-type in LF:

$$\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho]\rightsquigarrow\Pi y{:}(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\overline{\sigma}}\ \overline{N}).\overline{\rho}$$

Correspondingly at the level of objects we have the following key steps (where Σ' and Γ' are, respectively, the signature and the typing context in LF coming from the corresponding encoding of the derivation):

$$\mathcal{L}^{\mathcal{P}}_{N,\sigma}[M]\ \rightsquigarrow\ \left\{\ \lambda x{:}(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\overline{\sigma}}\ \overline{N}).\overline{M}\qquad\text{if }\mathcal{P}^{\Sigma}_{\Gamma}\in\mathsf{Dom}(\Sigma')\right.$$

$$\mathcal{U}^{\mathcal{P}}_{N,\sigma}[M]\ \rightsquigarrow\ \begin{cases}(\overline{M}\,x) & \text{if }\mathcal{P}^{\Sigma}_{\Gamma}\in\mathsf{Dom}(\Sigma')\text{ and }x{:}(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\overline{\sigma}}\ \overline{N})\in\Gamma'\\ (\overline{M}\,(c_{\mathcal{P}^{\Sigma}_{\Gamma}}\ \vec{x}\ I_{\overline{\sigma}}\ \overline{N})) & \text{if }c_{\mathcal{P}^{\Sigma}_{\Gamma}},\mathcal{P}^{\Sigma}_{\Gamma}\in\mathsf{Dom}(\Sigma')\end{cases}$$

Of course, this amounts to a form of *proof irrelevance*, as it should be, since in $\mathsf{LLF}_{\mathcal{P}}$ the verification of $\mathcal{P}(\Gamma\vdash_{\Sigma}N:\sigma)$ is carried out by an external system (*i.e.*, the *oracle*) which only returns a yes/no answer. Hence the external proof argument is not important and it

Figure 12: Encoding of family rules - Pt.2

$$\epsilon\left(\frac{\mathcal{D}_1}{\Gamma' \vdash_{\Sigma'} \rho : \mathsf{Type}} \quad \frac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} N : \sigma}\ (*)\right) \Longrightarrow \frac{(\mathcal{D}'_1)^+}{\Gamma'' \vdash_{\Sigma^v} \rho' : \mathsf{Type}} \quad \frac{(\mathcal{D}'_2)^+}{\Gamma'' \vdash_{\Sigma^v} N' : \sigma'} \Big/ \Gamma'' \vdash_{\Sigma^v} \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'}\ N').\rho' : \mathsf{Type}$$

$* \triangleq (F{\cdot}Lock)$ where $\epsilon\left(\frac{\mathcal{D}_1}{\Gamma' \vdash_{\Sigma'} \rho : \mathsf{Type}}\right) \Longrightarrow \frac{\mathcal{D}'_1}{\Gamma'' \vdash_{\Sigma''} \rho' : \mathsf{Type}}$, and

$\epsilon\left(\frac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} N : \sigma}\right) \Longrightarrow \frac{\mathcal{D}'_2}{\Gamma'' \vdash_{\Sigma'''} N' : \sigma'}$, and

$x_1, \ldots, x_n \triangleq \vec{x} \triangleq \mathsf{Dom}(\Gamma')$, and $\Sigma^{iv} \triangleq \mathsf{Merge}(\Sigma'', \Sigma''')$, and

$$\Sigma^v \triangleq \begin{cases} \Sigma^{iv} & \text{if } \mathcal{P} \in \mathsf{Dom}(\Sigma^{iv}) \\ \Sigma^{iv}, \mathcal{P}{:}\Pi x_1{:}\sigma_1 \ldots\ x_n{:}\sigma_n.(\sigma' \to \sigma') \to \sigma' \to \mathsf{Type} & \text{otherwise} \end{cases}$$

$$\epsilon\left(\frac{\mathcal{D}_1}{\Gamma' \vdash_{\Sigma'} \sigma : K} \quad \frac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} K'} \quad \frac{\mathcal{D}_3}{K =_{\beta\mathcal{L}} K'}\ (*)\right) \Longrightarrow \frac{(\mathcal{D}'_1)^+}{\Gamma'' \vdash_{\Sigma^{iv}} \sigma' : K''} \quad \frac{(\mathcal{D}'_2)^+}{\Gamma'' \vdash_{\Sigma^{iv}} K'''} \quad \frac{\mathcal{D}'_3}{K'' =_\beta K'''} \Big/ \Gamma'' \vdash_{\Sigma^{iv}} \sigma' : K'''$$

$* \triangleq (F{\cdot}Conv)$ where $\epsilon\left(\frac{\mathcal{D}_1}{\Gamma' \vdash_{\Sigma'} \sigma : K}\right) \Longrightarrow \frac{\mathcal{D}'_1}{\Gamma'' \vdash_{\Sigma''} \sigma' : K''}$, and

$\epsilon\left(\frac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} K'}\right) \Longrightarrow \frac{\mathcal{D}'_2}{\Gamma'' \vdash_{\Sigma'''} K'''}$, and

$\Sigma^{iv} \triangleq \mathsf{Merge}(\Sigma'', \Sigma''')$, and

$\epsilon\left(\frac{\mathcal{D}_3}{K =_{\beta\mathcal{L}} K'}\right) \Longrightarrow \frac{\mathcal{D}'_3}{K'' =_\beta K'''}$

Figure 13: Encoding of the "standard" object rules - Pt.1

$$\epsilon\left(\frac{\frac{\mathcal{D}}{\vdash_{\Sigma'} \Gamma'} \quad c{:}\sigma \in \Sigma'}{\Gamma' \vdash_{\Sigma'} c : \sigma}\ (O{\cdot}Const)\right) \Longrightarrow \frac{\frac{\mathcal{D}'}{\vdash_{\Sigma''} \Gamma''} \quad c{:}\sigma' \in \Sigma''}{\Gamma'' \vdash_{\Sigma''} c : \sigma'} \quad \text{where } \epsilon\left(\frac{\mathcal{D}}{\vdash_{\Sigma'} \Gamma'}\right) \Longrightarrow \frac{\mathcal{D}'}{\vdash_{\Sigma''} \Gamma''}$$

$$\epsilon\left(\frac{\frac{\mathcal{D}}{\vdash_{\Sigma} \Gamma'} \quad x{:}\sigma \in \Gamma'}{\Gamma' \vdash_{\Sigma'} x : \sigma}\ (O{\cdot}Var)\right) \Longrightarrow \frac{\frac{\mathcal{D}'}{\vdash_{\Sigma''} \Gamma''} \quad x{:}\sigma' \in \Gamma''}{\Gamma'' \vdash_{\Sigma''} x : \sigma'} \quad \text{where } \epsilon\left(\frac{\mathcal{D}}{\vdash_{\Sigma'} \Gamma'}\right) \Longrightarrow \frac{\mathcal{D}'}{\vdash_{\Sigma''} \Gamma''}$$

$$\epsilon\left(\frac{\frac{\mathcal{D}}{\Gamma', x{:}\sigma \vdash_{\Sigma'} M : \tau}}{\Gamma' \vdash_{\Sigma'} \lambda x{:}\sigma.M : \Pi x{:}\sigma.\tau}\ (O{\cdot}Abs)\right) \Longrightarrow \frac{\frac{\mathcal{D}'}{\Gamma'', x{:}\sigma' \vdash_{\Sigma''} M' : \tau'}}{\Gamma'' \vdash_{\Sigma''} \lambda x{:}\sigma'.M' : \Pi x{:}\sigma'.\tau'}$$

where $\epsilon\left(\frac{\mathcal{D}}{\Gamma', x{:}\sigma \vdash_{\Sigma'} M : \tau}\right) \Longrightarrow \frac{\mathcal{D}'}{\Gamma'', x{:}\sigma' \vdash_{\Sigma''} M' : \tau'}$

can be represented by a constant or variable. In the remaining cases, the encoding function behaves in a straightforward way propagating itself to inner subderivations, *e.g.* in the rules of application, abstraction etc.

Figure 14: Encoding of the "standard" object rules - Pt.2

$$\epsilon\left(\frac{\dfrac{\mathcal{D}_1}{\Gamma' \vdash_{\Sigma'} M : \Pi x{:}\sigma.\tau} \quad \dfrac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} N : \sigma}\ (O{\cdot}App)}{\Gamma' \vdash_{\Sigma'} M\ N : \tau[N/x]}\right) \Longrightarrow \frac{\dfrac{(\mathcal{D}'_1)^+}{\Gamma'' \vdash_{\Sigma^{iv}} M' : \Pi x{:}\sigma.'\tau'} \quad \dfrac{(\mathcal{D}'_2)^+}{\Gamma'' \vdash_{\Sigma^{iv}} N' : \sigma'}}{\Gamma'' \vdash_{\Sigma^{iv}} M'\ N' : \tau'[N'/x]}$$

where $\epsilon\left(\dfrac{\mathcal{D}_1}{\Gamma' \vdash_{\Sigma'} M : \Pi x{:}\sigma.\tau}\right) \Longrightarrow \dfrac{\mathcal{D}'_1}{\Gamma'' \vdash_{\Sigma''} M' : \Pi x{:}\sigma.'\tau'}$, and

$\epsilon\left(\dfrac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} N : \sigma}\right) \Longrightarrow \dfrac{\mathcal{D}'_2}{\Gamma'' \vdash_{\Sigma'''} N' : \sigma'}$, and $\Sigma^{iv} \triangleq \mathsf{Merge}(\Sigma'', \Sigma''')$

$$\epsilon\left(\frac{\dfrac{\mathcal{D}_1}{\Gamma' \vdash_{\Sigma'} M : \sigma} \quad \dfrac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} \tau : \mathsf{Type}} \quad \dfrac{\mathcal{D}_3}{\sigma =_\beta \mathcal{L}\tau}\ (*)}{\Gamma' \vdash_{\Sigma'} M : \tau}\right) \Longrightarrow \frac{\dfrac{(\mathcal{D}'_1)^+}{\Gamma'' \vdash_{\Sigma^{iv}} M' : \sigma'} \quad \dfrac{(\mathcal{D}'_2)^+}{\Gamma'' \vdash_{\Sigma^{iv}} \tau' : \mathsf{Type}} \quad \dfrac{\mathcal{D}'_3}{\sigma' =_\beta \tau'}}{\Gamma^{iv} \vdash_{\Sigma^{iv}} M' : \tau'}$$

$(*) \triangleq (O{\cdot}Conv)$ where $\epsilon\left(\dfrac{\mathcal{D}_1}{\Gamma' \vdash_{\Sigma'} M : \sigma}\right) \Longrightarrow \dfrac{\mathcal{D}'_1}{\Gamma'' \vdash_{\Sigma''} M' : \sigma'}$, and

$\epsilon\left(\dfrac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} \tau : \mathsf{Type}}\right) \Longrightarrow \dfrac{\mathcal{D}'_2}{\Gamma'' \vdash_{\Sigma'''} \tau' : \mathsf{Type}}$, and

$\Sigma^{iv} \triangleq \mathsf{Merge}(\Sigma'', \Sigma''')$, and $\epsilon\left(\dfrac{s\mathcal{D}_3}{\sigma =_\beta \mathcal{L}\tau}\right) \Longrightarrow \dfrac{\mathcal{D}'_3}{\sigma' =_\beta \tau'}$

Figure 15: Encoding of the object rules involving locks and unlocks - Pt. 1

$$\epsilon\left(\frac{\dfrac{\mathcal{D}_1}{\Gamma' \vdash_{\Sigma'} M : \rho} \quad \dfrac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} N : \sigma}\ (O{\cdot}Lock)}{\Gamma' \vdash_{\Sigma'} \mathcal{L}^{\mathcal{P}}_{N,\sigma}[M] : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho]}\right) \Longrightarrow \frac{\dfrac{(\mathcal{D}'_1)^+}{\Gamma'' \vdash_{\Sigma^v} M' : \rho'} \quad \dfrac{(\mathcal{D}'_2)^+}{\Gamma'' \vdash_{\Sigma^v} N' : \sigma'}}{\Gamma'' \vdash_{\Sigma^v} \lambda y : (\mathcal{P}\ \vec{x}\ I_{\sigma'}\ N').M' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'}\ N').\rho'}$$

where $\epsilon\left(\dfrac{\mathcal{D}_1}{\Gamma' \vdash_{\Sigma'} M : \rho}\right) \Longrightarrow \dfrac{\mathcal{D}'_1}{\Gamma'' \vdash_{\Sigma''} M' : \rho'}$, and

$\epsilon\left(\dfrac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} N : \sigma}\right) \Longrightarrow \dfrac{\mathcal{D}'_2}{\Gamma'' \vdash_{\Sigma'''} N' : \sigma'}$, and

$\Sigma^{iv} \triangleq \mathsf{Merge}(\Sigma'', \Sigma''')$, and $x_1, \ldots, x_n \triangleq \vec{x} \triangleq \mathsf{Dom}(\Gamma')$, and

$$\Sigma^v = \begin{cases} \Sigma^{iv} & \text{if } \mathcal{P} \in \mathsf{Dom}(\Sigma^{iv}) \\ (\Sigma^{iv}, \mathcal{P}{:}\Pi x_1{:}\sigma_1 \ldots x_n{:}\sigma_n.(\sigma' \to \sigma') \to \sigma' \to \mathsf{Type}) & \text{otherwise} \end{cases}$$

$$\epsilon\left(\frac{\dfrac{\mathcal{D}}{\Gamma' \vdash_{\Sigma'} M : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho]} \quad \mathcal{P}(\Gamma' \vdash_{\Sigma'} N : \sigma)\ (*)}{\Gamma' \vdash_{\Sigma'} \mathcal{U}^{\mathcal{P}}_{N,\sigma}[M] : \rho}\right) \Longrightarrow \frac{(\mathcal{D}')^+}{\Gamma'' \vdash_{\Sigma'''} M' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'}\ N').\ \rho'} \over \Gamma'' \vdash_{\Sigma'''} (M'\ (c_{\mathcal{P}}\ \vec{x}\ I_{\sigma'}\ N')) : \rho'$$

$(*) \triangleq (O{\cdot}Top{\cdot}Unlock)$ where $\epsilon\left(\dfrac{\mathcal{D}}{\Gamma' \vdash_{\Sigma'} M : \mathcal{L}^{\mathcal{P}}_{N,\sigma}[\rho]}\right) \Longrightarrow \dfrac{\mathcal{D}'}{\Gamma'' \vdash_{\Sigma''} M' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'}\ N').\ \rho'}$, and

$x_1, \ldots, x_n \triangleq \vec{x} \triangleq \mathsf{Dom}(\Gamma')$, and

$$\Sigma''' \triangleq \begin{cases} \Sigma'' & \text{if } c_P \in \mathsf{Dom}(\Sigma'') \\ \Sigma'', c_P{:}\Pi x_1{:}\sigma_1 \ldots x_n{:}\sigma_n.\Pi x{:}(\sigma' \to \sigma'), y{:}\sigma'.(\mathcal{P}\ \vec{x}\ x\ y)) & \text{otherwise} \end{cases}$$

Figure 16: Encoding of the ($F{\cdot}Guarded{\cdot}Unlock$) rule

$$\epsilon\left(\dfrac{\dfrac{\mathcal{D}_1}{\Gamma', x{:}\tau \vdash_{\Sigma'} \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\rho] : \mathsf{Type}} \quad \dfrac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} N : \mathcal{L}^{\mathcal{P}}_{S',\sigma'}[\tau]} \quad \dfrac{\mathcal{D}_3}{\sigma =_{\beta\mathcal{L}} \sigma'} \quad \dfrac{\mathcal{D}_4}{S =_{\beta\mathcal{L}} S'}}{\Gamma' \vdash_{\Sigma'} \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\rho[\mathcal{U}^{\mathcal{P}}_{S',\sigma'}[N]/x]] : \mathsf{Type}}\ (F{\cdot}Guarded{\cdot}Unlock)\right)$$

$$\Longrightarrow$$

$$\dfrac{\dfrac{\mathcal{D}'_1 \qquad \mathcal{D}}{\Gamma'', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'') \vdash_{\Sigma^{iv}} \rho'[N'y/x] : \mathsf{Type}}\ (T)}{\Gamma'' \vdash_{\Sigma^{iv}} \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').\rho'[N'y/x] : \mathsf{Type}}$$

where $\mathcal{D} \triangleq$

$$\dfrac{\dfrac{\dfrac{\dfrac{(\mathcal{D}'_2)^+}{\Gamma'' \vdash_{\Sigma^{iv}} N' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'''}\ S''').\tau'} \quad \dfrac{\dfrac{\dfrac{\mathcal{D}'_3}{\sigma'' =_\beta \sigma'''} \quad \dfrac{\mathcal{D}'_4}{S'' =_\beta S'''}}{\vdots}}{\Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'''}\ S''').\tau' =_{\beta\mathcal{L}} \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').\tau'}}{\Gamma'' \vdash_{\Sigma^{iv}} N' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').\tau'}}{\Gamma'', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'') \vdash_{\Sigma^{iv}} N' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').\tau'}\ (w+\alpha) \qquad \mathcal{D}_5}{\Gamma'', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'') \vdash_{\Sigma^{iv}} N'y : \tau'}$$

, and

$\mathcal{D}_5 : \Gamma'', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'') \vdash_{\Sigma^{iv}} y : (\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'')$, and $\Delta \triangleq x{:}\tau', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'')$, and

$(w+\alpha)$ stands for an application of weakening and α-conversion in LF, and

(T) stands for an application of the transitivity theorem in LF, and

vertical dots $\left(\vdots\right)$ stand for applications of context-closure and definitional equality rules, and

$$\epsilon\left(\dfrac{\mathcal{D}_1}{\Gamma', x{:}\tau \vdash_{\Sigma'} \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\rho] : \mathsf{Type}}\right) \Longrightarrow \dfrac{\mathcal{D}'_1}{\Gamma'', x{:}\tau' \vdash_{\Sigma''} \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').\rho' : \mathsf{Type}}$$

, and

whence (for the Generation Lemma on Pure Type Systems [7]) there exists a derivation $\mathcal{D}'_1$

$\mathcal{D}'_1 : \Gamma'', x{:}\tau', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'') \vdash_{\Sigma''} \rho' : \mathsf{Type}$, and

$$\epsilon\left(\dfrac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} N : \mathcal{L}^{\mathcal{P}}_{S',\sigma'}[\tau]}\right) \Longrightarrow \dfrac{\mathcal{D}'_2}{\Gamma'' \vdash_{\Sigma'''} N' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'''}\ S''').\rho'}$$

, and

$$\epsilon\left(\dfrac{\mathcal{D}_3}{\sigma =_{\beta\mathcal{L}} \sigma'}\right) \Longrightarrow \dfrac{\mathcal{D}'_3}{\sigma'' =_\beta \sigma'''}$$

, and

$$\epsilon\left(\dfrac{\mathcal{D}_4}{S =_{\beta\mathcal{L}} S'}\right) \Longrightarrow \dfrac{\mathcal{D}'_4}{S'' =_\beta S'''}$$

, and

$\Sigma^{iv} \triangleq \mathsf{Merge}(\Sigma'', \Sigma''')$, and $x_1, \ldots, x_n \triangleq \vec{x} \triangleq \mathsf{Dom}(\Gamma')$

As far as the typing rules, we point out the possible extension of the signature in the encoding of rules ($F{\cdot}Lock$) (Figure 12) and ($O{\cdot}Lock$) (Figure 15) when a, possibly new, lock constant is introduced and in rule ($O{\cdot}Top{\cdot}Unlock$) (Figure 15) when a term is unlocked by

Figure 17: Encoding of the object rules involving locks and unlocks - Pt. 2

$$\epsilon\left(\frac{\mathcal{D}_1}{\Gamma', x{:}\tau \vdash_{\Sigma'} \mathcal{L}^{\mathcal{P}}_{S,\sigma}[M] : \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\rho]} \quad \frac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} N : \mathcal{L}^{\mathcal{P}}_{S',\sigma'}[\tau]} \quad \frac{\mathcal{D}_3}{\sigma =_{\beta\mathcal{L}} \sigma'} \quad \frac{\mathcal{D}_4}{S =_{\beta\mathcal{L}} S'}}{\Gamma' \vdash_{\Sigma'} \mathcal{L}^{\mathcal{P}}_{S,\sigma}[M[\mathcal{U}^{\mathcal{P}}_{S',\sigma'}[N]/x]] : \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\rho[\mathcal{U}^{\mathcal{P}}_{S',\sigma'}[N]/x]]}\ (O{\cdot}Guarded{\cdot}Unlock)\right)$$

$$\Longrightarrow$$

$$\frac{\dfrac{\dfrac{\mathcal{D}}{\Gamma'', \Delta \vdash_{\Sigma^{iv}} M' : \rho'} \quad \mathcal{D}'}{\Gamma'', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'') \vdash_{\Sigma^{iv}} M'[N'y/x] : \rho'[N'y/x]}\ (T)}{\Gamma'' \vdash_{\Sigma^{iv}} \lambda y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').M'[N'y/x] : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').\rho'[N'y/x]}$$

where $\mathcal{D} \triangleq \dfrac{\dfrac{\dfrac{(\mathcal{D}'_1)^+}{\Gamma'', \Delta \vdash_{\Sigma^{iv}} \lambda y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').M' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').\rho'}\ (w+\alpha) \quad \mathcal{D}_5}{\Gamma'', \Delta \vdash_{\Sigma^{iv}} (\lambda y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').M')y : \rho'}}{\Gamma'', \Delta \vdash_{\Sigma^{iv}} M' : \rho'}\ (SR)$, and

$$\mathcal{D}' \triangleq \dfrac{\dfrac{\dfrac{(\mathcal{D}'_2)^+}{\Gamma'' \vdash_{\Sigma^{iv}} N' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'''}\ S'''){.}\tau'} \quad \dfrac{\dfrac{\dfrac{\mathcal{D}'_3}{\sigma'' =_\beta \sigma'''} \quad \dfrac{\mathcal{D}'_4}{S'' =_\beta S'''}}{\vdots}}{\Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'''}\ S'''){.}\tau' =_{\beta\mathcal{L}} \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S''){.}\tau'}}{\Gamma'' \vdash_{\Sigma^{iv}} N' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S''){.}\tau'}}{\Gamma'', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'') \vdash_{\Sigma^{iv}} N' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S''){.}\tau'}\ (w+\alpha)}{\Gamma'', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'') \vdash_{\Sigma^{iv}} N'y : \tau'}\ \mathcal{D}_6$$, and

$\mathcal{D}_5 : \Gamma'', x{:}\tau', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'') \vdash_{\Sigma^{iv}} y : (\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'')$, and $\Delta \triangleq x{:}\tau', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'')$, and

$\mathcal{D}_6 : \Gamma'', y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'') \vdash_{\Sigma^{iv}} y : (\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'')$, and

$(w+\alpha)$ stands for an application of weakening and α-conversion in LF,

(SR) stands for an application of subject reduction in LF, and

(T) stands for an application of the transitivity theorem in LF, and

vertical dots $\left(\vdots\right)$ stand for applications of context-closure and definitional equality rules, and

$$\epsilon\left(\frac{\mathcal{D}_1}{\Gamma', x{:}\tau \vdash_{\Sigma'} \mathcal{L}^{\mathcal{P}}_{S,\sigma}[M] : \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\rho]}\right) \Longrightarrow \frac{\mathcal{D}'_1}{\Gamma'', x{:}\tau' \vdash_{\Sigma''} \lambda y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').M' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma''}\ S'').\rho'}\ ,\ \text{and}$$

$$\epsilon\left(\frac{\mathcal{D}_2}{\Gamma' \vdash_{\Sigma'} N : \mathcal{L}^{\mathcal{P}}_{S',\sigma'}[\tau]}\right) \Longrightarrow \frac{\mathcal{D}'_2}{\Gamma'' \vdash_{\Sigma'''} N' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'''}\ S'''){.}\rho'}\ ,\ \text{and}$$

$$\epsilon\left(\frac{\mathcal{D}_3}{\sigma =_{\beta\mathcal{L}} \sigma'}\right) \Longrightarrow \frac{\mathcal{D}'_3}{\sigma'' =_\beta \sigma'''}\ ,\ \text{and}\ \epsilon\left(\frac{\mathcal{D}_4}{S =_{\beta\mathcal{L}} S'}\right) \Longrightarrow \frac{\mathcal{D}'_4}{S'' =_\beta S'''}\ ,\ \text{and}$$

$\Sigma^{iv} \triangleq \mathsf{Merge}(\Sigma'', \Sigma''')$, and $x_1, \ldots, x_n \triangleq \vec{x} \triangleq \mathsf{Dom}(\Gamma')$

Figure 18: Induced encoding of terms (kinds, families and objects)

$\theta^{\Sigma}_{\Gamma}(\mathsf{Type})$	$\triangleq$	Type
$\theta^{\Sigma}_{\Gamma}(\Pi x{:}\sigma.K)$	$\triangleq$	$\Pi x{:}\theta^{\Sigma}_{\Gamma}(\sigma).\theta^{\Sigma}_{\Gamma,x:\theta^{\Sigma}_{\Gamma}(\sigma)}(K)$
$\theta^{\Sigma}_{\Gamma}(a)$	$\triangleq$	a if $a \in \mathsf{Dom}(\Sigma)$
$\theta^{\Sigma}_{\Gamma}(\Pi x{:}\sigma.\tau)$	$\triangleq$	$\Pi x{:}\theta^{\Sigma}_{\Gamma}(\sigma).\theta^{\Sigma}_{\Gamma,x:\theta^{\Sigma}_{\Gamma}(\sigma)}(\tau)$
$\theta^{\Sigma}_{\Gamma}(\sigma N)$	$\triangleq$	$\theta^{\Sigma}_{\Gamma}(\sigma)\,\theta^{\Sigma}_{\Gamma}(N)$
$\theta^{\Sigma}_{\Gamma}(\mathcal{L}^{\mathcal{P}}_{N,\sigma}[\tau])$	$\triangleq$	$\Pi x{:}(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\theta^{\Sigma}_{\Gamma}(\sigma)}\ \theta^{\Sigma}_{\Gamma}(N)).\theta^{\Sigma}_{\Gamma,x:(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\theta^{\Sigma}_{\Gamma}(\sigma)}\ \theta^{\Sigma}_{\Gamma}(N))}(\tau)$ if $\mathcal{P}^{\Sigma}_{\Gamma} \in \mathsf{Dom}(\Sigma)$, and $\vec{x} \in \mathsf{Dom}(\Gamma)$
$\theta^{\Sigma}_{\Gamma}(c)$	$\triangleq$	c if $c \in \mathsf{Dom}(\Sigma)$
$\theta^{\Sigma}_{\Gamma}(x)$	$\triangleq$	x if $x \in \mathsf{Dom}(\Gamma)$
$\theta^{\Sigma}_{\Gamma}(\lambda x{:}\tau.M)$	$\triangleq$	$\lambda x{:}\theta^{\Sigma}_{\Gamma}(\tau).\theta^{\Sigma}_{\Gamma,x:\theta^{\Sigma}_{\Gamma}(\tau)}(M)$
$\theta^{\Sigma}_{\Gamma}(MN)$	$\triangleq$	$\theta^{\Sigma}_{\Gamma}(M)\,\theta^{\Sigma}_{\Gamma}(N)$
$\theta^{\Sigma}_{\Gamma}(\mathcal{L}^{\mathcal{P}}_{N,\sigma}[M])$	$\triangleq$	$\lambda x{:}(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\theta^{\Sigma}_{\Gamma}(\sigma)}\ \theta^{\Sigma}_{\Gamma}(N)).\theta^{\Sigma}_{\Gamma,x:(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\theta^{\Sigma}_{\Gamma}(\sigma)}\ \theta^{\Sigma}_{\Gamma}(N))}(M)$ if $\mathcal{P}^{\Sigma}_{\Gamma} \in \mathsf{Dom}(\Sigma)$, and $\vec{x} \in \mathsf{Dom}(\Gamma)$)
$\theta^{\Sigma}_{\Gamma}(\mathcal{U}^{\mathcal{P}}_{N,\sigma}[M])$	$\triangleq$	$\begin{cases} \theta^{\Sigma}_{\Gamma}(M)\,x \quad \text{if } \mathcal{P}^{\Sigma}_{\Gamma} \in \mathsf{Dom}(\Sigma), \text{ and } x{:}(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\theta^{\Sigma}_{\Gamma}(\sigma)}\ \theta^{\Sigma}_{\Gamma}(N)) \in \Gamma \\ \theta^{\Sigma}_{\Gamma}(M)\,(c_{\mathcal{P}^{\Sigma}_{\Gamma}}\ \vec{x}\ I_{\theta^{\Sigma}_{\Gamma}(\sigma)}\ \theta^{\Sigma}_{\Gamma}(N)) \quad \text{if } c_{\mathcal{P}^{\Sigma}_{\Gamma}}, \mathcal{P}^{\Sigma}_{\Gamma} \in \mathsf{Dom}(\Sigma) \end{cases}$

the external oracle call. Finally we can establish the following crucial theorem concerning the encoding functions ϵ and θ^{Σ}_{Γ}:

Theorem 3.1 (Properties of ϵ and θ^{Σ}_{Γ})**.** *The encoding function ϵ defined in Figures 8, 9, 10, 11, 12, 13, 14, 15, 16, and 17 satisfies the following properties:*

(1) ϵ encodes derivations coherently, i.e., derivations of signatures/contexts well-formedness in $\mathsf{LLF}_{\mathcal{P}}$ are mapped to derivations of signatures/contexts well-formedness in LF, typing derivations in $\mathsf{LLF}_{\mathcal{P}}$ are mapped to typing derivations in LF, etc.

(2) ϵ is a total compositional function.

(3) Given a valid derivation $\mathcal{D} : \Gamma \vdash_{\Sigma} A : B$ in $\mathsf{LLF}_{\mathcal{P}}$, then the derivation, denoted by $\epsilon(\mathcal{D}) : \overline{\Gamma} \vdash_{\overline{\Sigma}} \overline{A} : \overline{B}$, is a valid derivation in LF.

(4) Given a valid derivation $\mathcal{D} : A \to_{\beta\mathcal{L}} B$ (resp. $\mathcal{D} : A =_{\beta\mathcal{L}} B$) in $\mathsf{LLF}_{\mathcal{P}}$, then the derivation, denoted by $\epsilon(\mathcal{D}) : \overline{A} \to_{\beta} \overline{B}$ (resp. $\epsilon(\mathcal{D}) : \overline{A} =_{\beta} \overline{B}$), is a valid derivation in LF.

(5) The compositional function ϵ induces the function θ^{Σ}_{Γ} between terms (kinds, families and objects) of $\mathsf{LLF}_{\mathcal{P}}$ and of LF defined in Figure 18. The signature Σ and the typing context Γ passed as parameters to the induced map are precisely the final ones generated by the translation process of the derivations.

(6) *If $\mathcal{D}$ is a valid derivation in $\mathsf{LLF}_{\mathcal{P}}$ of the typing judgment $\Gamma \vdash_\Sigma A : B$, then $\epsilon(\mathcal{D} : \Gamma \vdash_\Sigma A : B) = \mathcal{D}' : \overline{\Gamma} \vdash_{\overline{\Sigma}} \theta^{\overline{\Gamma}}_{\overline{\Sigma}}(A) : \theta^{\overline{\Gamma}}_{\overline{\Sigma}}(B)$.*

(7) *If the judgments $\Gamma \vdash_\Sigma A : B$ and $A \to_{\beta\mathcal{L}} A'$ (resp. $A =_{\beta\mathcal{L}} A'$) hold in $\mathsf{LLF}_{\mathcal{P}}$, then we have $\theta^{\overline{\Sigma}}_{\overline{\Gamma}}(A) \to_\beta \theta^{\overline{\Sigma}}_{\overline{\Gamma}}(A')$ (resp. $\theta^{\overline{\Sigma}}_{\overline{\Gamma}}(A) =_\beta \theta^{\overline{\Sigma}}_{\overline{\Gamma}}(A')$) in LF.*

Proof : By a tedious but ultimately straightforward induction on derivations in $\mathsf{LLF}_{\mathcal{P}}$, taking into account the clauses of ϵ and θ^{Σ}_{Γ}.

The judgements generated by the encoding ϵ make a very special use of variables of type $(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\overline{\sigma}}\ \overline{N})$. Namely such variables can λ-bind only objects and Π-bind only types, *i.e.* terms of kind `Type`. In particular families are never λ-abstracted and kinds are never Π-bound by such variables. Furthermore such variables are introduced in the encoding only in a controlled way by the lock-unlock rules. This is crucial to preserve the reversibility of the encoding necessary for transferring the properties from LF to $\mathsf{LLF}_{\mathcal{P}}$. Notice that the terms produced by the encoding θ^{Σ}_{Γ} satisfy similar properties. This kind of closure property, which we call *goodness*, is defined as follows:

Definition 3.2 (Good Judgements and good terms)**.** We call *good judgements* the LF-judgements whose subterms of type $(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\overline{\sigma}}\ \overline{N})$ (for some σ and N in $\mathsf{LLF}_{\mathcal{P}}$) are only terms of the shape $(c_{\mathcal{P}^{\Sigma}_{\Gamma}}\ \vec{x}\ I_{\overline{\sigma}}\ \overline{N})$ or variables, which moreover appear as proper subterms always in applied position. Moreover only object terms are *lambda*-bound by variables of type $(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\overline{\sigma}}\ \overline{N})$ (for some σ and N in $\mathsf{LLF}_{\mathcal{P}}$) and only type terms, *i.e.* objects of kind `Type`, are Π-bounded over variables of type $(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\overline{\sigma}}\ \overline{N})$ (for some σ and N in $\mathsf{LLF}_{\mathcal{P}}$). Terms occurring in good judgements are called *good terms*.

The introduction of good judgements is motivated by the following:

Theorem 3.3 (Good judgements)**.** *The LF-judgements in the codomain of ϵ are good judgements.*

Proof : By induction on the definition of ϵ, one can easily check that since whenever variables, or terms, of the shape $(c_{\mathcal{P}^{\Sigma}_{\Gamma}}\ \vec{x}\ I_{\overline{\sigma}}\ \overline{N})$ are introduced these are always applied to the translation of the body of an unlock-term in $\mathsf{LLF}_{\mathcal{P}}$.

Moreover, β-reductions carried out in LF preserve the "goodness" property. Indeed, if the term in functional position is a λ-term, it will either erase the variable/term (representing an unlock/lock dissolution) or replace it into another term, again, in argument position. This latter case arises in derivations involving the $(O \cdot Guarded \cdot Unlock)$-rule and the $(F \cdot Guarded \cdot Unlock)$-rule in $\mathsf{LLF}_{\mathcal{P}}$. This remark leads immediately to the following proposition:

Theorem 3.4 (Closure by reduction of good terms)**.** *Good terms are closed by β-reduction in LF, i.e., if M is good and $M \to_\beta M'$ in LF, then also M' is a good term.*

To be able to transfer back to $\mathsf{LLF}_{\mathcal{P}}$ properties of LF we need to "invert" the encoding function ϵ. This can be done only starting from a good judgment in LF. To this end we establish the following proposition.

Theorem 3.5. *If $\Gamma \vdash_\Sigma M : \sigma$ is a good LF judgement, then there is a derivation of that judgment in LF, all whose judgements are good LF judgements.*

Proof By induction on derivations in LF. The only critical cases being type equality rules, which are nonetheless straightforward.

Given the ϵ-encoding of derivations and the induced θ^{Σ}_{Γ}-encoding defined in Figure 18, we can define an inverse function η^{Σ}_{Γ} on terms (see Figure 19), where Σ and Γ are, respectively the signature and the typing context of the corresponding typing judgment in LF. The following fundamental *invertibility* theorem establishes the above claim more precisely. This theorem will be the main tool for transferring the metatheoretic properties of LF to $\mathsf{LLF}_{\mathcal{P}}$. Notice that it amounts to a form of *adequacy* of the encoding ϵ.

Theorem 3.6 (Invertibility)**.** *There exists a total function δ mapping derivations of good judgements in LF into derivations of $\mathsf{LLF}_{\mathcal{P}}$. More precisely if $\mathcal{D}$ is a valid derivation in LF of the* good judgment $\Gamma \vdash_{\Sigma} A : B$*, then $\delta(\mathcal{D} : \Gamma \vdash_{\Sigma} A : B) = \mathcal{D}' : \Gamma' \vdash_{\Sigma'} \eta^{\Gamma}_{\Sigma}(A) : \eta^{\Gamma}_{\Sigma}(B)$, where the function η^{Σ}_{Γ} between terms (kinds, families and objects) of LF and terms of $\mathsf{LLF}_{\mathcal{P}}$ is defined in Figure 19. In particular the function η^{Σ}_{Γ} is left inverse to θ^{Σ}_{Γ}, i.e., for each derivation term M such that $\Gamma \vdash_{\Sigma} M : \tau$ in $\mathsf{LLF}_{\mathcal{P}}$, we have that $\eta^{\overline{\Sigma}}_{\overline{\Gamma}}(\theta^{\overline{\Sigma}}_{\overline{\Gamma}}(M)) = M$ holds.*

Proof (Sketch): The definition of $\mathcal{D}$ is syntax-driven by the structure of LF terms which are also *good terms* (see Theorem 3.3). The only critical cases are the following:

(1) application: it can represent an *ordinary* application in $\mathsf{LLF}_{\mathcal{P}}$ if the argument is not of the shape $(c_{\mathcal{P}} \ldots)$. The case of a variable cannot arise at "top level" since such variable is explained away making use of a *guarded unlock*. Instead, if the argument is of the shape $(c_{\mathcal{P}} \ldots)$, *i.e.*, a constant of type $(\mathcal{P} \ldots)$, then we can invert the derivation by means of a $(O{\cdot}Top{\cdot}Unlock)$-rule.

(2) abstraction introduction: it can lead to an *ordinary* abstraction in $\mathsf{LLF}_{\mathcal{P}}$ if the type of the abstracted variable is not of the shape $(\mathcal{P} \ldots)$. Otherwise we have to introduce either a *lock* or a *guarded unlock*, depending on the occurrences of the abstracted variable of type $(\mathcal{P} \ldots)$ in the abstraction's body. Indeed, if there are no occurrences of the free variable, *i.e.*, we have a *dummy* abstraction, we invert it by a simple lock introduction; on the other hand, if there are one or more occurrences of the variable, we are in the second subcase. Indeed, let us suppose we have the following derivation in LF:

$$\dfrac{\dfrac{\mathcal{D}}{\Gamma, y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma}\ S) \vdash_{\Sigma} M : \rho}}{\Gamma \vdash_{\Sigma} \lambda y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma}\ S).M : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma}\ S).\rho} \tag{3.1}$$

Then, if y occurs in M, it must occur in argument position since we are working with *good terms*, *i.e.*, as a subterm of $Ny : \tau$ for a suitable N and τ, *i.e.*, $M \equiv M^*[Ny]$ (where $M^*[\cdot]$ denotes a *context*, *i.e.*, term with a "hole" which can be filled by another term). For the sake of simplicity we discuss first the case in which all occurrences of y are applied to the same term N. From above we can define $M' \equiv M[x/Ny]$ and $\rho' \equiv \rho[x/Ny]$ for a suitable x of type τ such that $x \notin \mathsf{Dom}(\Gamma)$ and $x \not\equiv y$. Thus, we can infer the existence of a derivation $\mathcal{D}' : \Gamma, y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma}\ S), x{:}\tau \vdash_{\Sigma} M' : \rho'$. Moreover, there must be a derivation $\mathcal{D}_2 : \Gamma, y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma}\ S) \vdash_{\Sigma} Ny : \tau$ from $\mathcal{D}_3 : \Gamma, y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma}\ S) \vdash_{\Sigma} N : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'}\ S').\tau$, and $\mathcal{D}_4 : \Gamma, y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma}\ S) \vdash_{\Sigma} y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma}\ S)$, and $\mathcal{D}_5 : S =_{\beta} S'$, and $\mathcal{D}_6 : \sigma =_{\beta} \sigma'$.
Therefore we can perform the following proof "manipulation":

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}}{\Gamma, y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S) \vdash_\Sigma M : \rho}}{\Gamma \vdash_\Sigma \lambda y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).M : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).\rho}}{\begin{array}{c}\rightsquigarrow\\ \mathcal{D}_5 \quad \mathcal{D}_6\\ \vdots\end{array}}}{\dfrac{\mathcal{D}_3 \quad \dfrac{}{\Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'}\ S').\tau =_\beta \Pi y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).\tau}}{\dfrac{\Gamma, y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S) \vdash_\Sigma N : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).\tau \quad \mathcal{D}_4}{\Gamma, y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S) \vdash_\Sigma Ny : \tau}}}$$

$$\dfrac{\dfrac{\mathcal{D}' \qquad\qquad}{\Gamma, y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S) \vdash_\Sigma M'[Ny/x] : \rho'[Ny/x]}\ (transitivity)}{\Gamma \vdash_\Sigma \lambda y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).M'[Ny/x] : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).\rho'[Ny/x]}$$

Obviously, $M'[Ny/x] \equiv M[x/Ny][Ny/x] \equiv M$ and $\rho'[Ny/x] \equiv \rho[x/Ny]\ [Ny/x] \equiv \rho$. Whence, we can define the following "decoding" of the abstraction introduction (3.1), where $\mathcal{D}'_3$ is essentially the derivation

$$\mathcal{D}_3 : \Gamma, y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S) \vdash_\Sigma N : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_{\sigma'}\ S').\tau$$

with a final application of *strengthening* to prune the variable y from the typing context:

$$\dfrac{\delta\left(\dfrac{\dfrac{\mathcal{D}'}{\Gamma, x{:}\tau, y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S) \vdash_\Sigma M' : \rho'}}{\Gamma, x{:}\tau \vdash_\Sigma \lambda y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).M' : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).\rho'}\right) \quad \delta(\mathcal{D}'_3) \quad \delta(\mathcal{D}_5) \quad \delta(\mathcal{D}_6)}{\Gamma' \vdash_{\Sigma'} \mathcal{L}^{\mathcal{P}}_{S'',\sigma''}[M''[\mathcal{U}^{\mathcal{P}}_{S''',\sigma'''}[N']/x]] : \mathcal{L}^{\mathcal{P}}_{S'',\sigma''}[\rho''[\mathcal{U}^{\mathcal{P}}_{S''',\sigma'''}[N']/x]]}$$

$$\Longrightarrow$$

$$\dfrac{\dfrac{\delta(\mathcal{D}'')}{\Gamma', x{:}\tau' \vdash_{\Sigma'} \mathcal{L}^{\mathcal{P}}_{S'',\sigma''}[M''] : \mathcal{L}^{\mathcal{P}}_{S'',\sigma''}[\rho'']} \quad \dfrac{\delta(\mathcal{D}''_3)}{\Gamma' \vdash_{\Sigma'} N' : \mathcal{L}^{\mathcal{P}}_{S''',\sigma'''}[\tau']} \quad \dfrac{\delta(\mathcal{D}'_5)}{S'' =_{\beta\mathcal{L}} S'''} \quad \dfrac{\delta(\mathcal{D}'_6)}{\sigma'' =_{\beta\mathcal{L}} \sigma'''}}{\Gamma' \vdash_{\Sigma'} \mathcal{L}^{\mathcal{P}}_{S'',\sigma''}[M''[\mathcal{U}^{\mathcal{P}}_{S''',\sigma'''}[N']/x]] : \mathcal{L}^{\mathcal{P}}_{S'',\sigma''}[\rho''[\mathcal{U}^{\mathcal{P}}_{S''',\sigma'''}[N']/x]]}$$

where $\mathcal{D}''$, $\mathcal{D}''_3$, $\mathcal{D}'_5$ and $\mathcal{D}'_6$ are, respectively, the residuals of derivations $\mathcal{D}'$, $\mathcal{D}'_3$, $\mathcal{D}_5$ and $\mathcal{D}_6$ once we have got rid of the last applied rule. The signature Σ' and the typing context Γ' in $\mathsf{LLF}_\mathcal{P}$ are obtained from the corresponding entities Σ and Γ in LF by pruning all the constants $\mathcal{P}$ and $c_\mathcal{P}$ and variables of type $(\mathcal{P}\ldots)$.

The case in which the bound variable y occurs as argument of several different $N_1,\ldots,N_k$, the above decoding has to be carried out repeatedly for each N_i.

The fact that the decoding via δ of the derivation in LF of the typing judgment $\Gamma \vdash_\Sigma \lambda y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).M'[Ny/x] : \Pi y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).\rho'[Ny/x]$ corresponds to a derivation in $\mathsf{LLF}_\mathcal{P}$ of type $\Gamma' \vdash_{\Sigma'} \eta^\Sigma_\Gamma(\lambda y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).M'[Ny/x]) : \eta^\Sigma_\Gamma(\Pi y{:}(\mathcal{P}\ \vec{x}\ I_\sigma\ S).\rho'[Ny/x])$ can be easily verified, by looking at the defining clauses in Figure 19 and exploiting the fact that $\eta^\Sigma_\Gamma(M[N/x]) = \eta^\Sigma_\Gamma(M)[\eta^\Sigma_\Gamma(N)/x]$.

All the properties of the decoding function η are proved by induction taking into account the fact that all terms are good.

Figure 19: Induced decoding of terms (kinds, families and objects)

$$
\begin{array}{lcl}
\eta^{\Sigma}_{\Gamma}(\mathsf{Type}) & \triangleq & \mathsf{Type} \\
\eta^{\Sigma}_{\Gamma}(\Pi x{:}\sigma.K) & \triangleq & \Pi x{:}\eta^{\Sigma}_{\Gamma}(\sigma).\eta^{\Sigma}_{\Gamma,x:\eta^{\Sigma}_{\Gamma}(\sigma)}(K) \\
\eta^{\Sigma}_{\Gamma}(a) & \triangleq & a \qquad \text{if } a \in \mathsf{Dom}(\Sigma), \text{ and } a \not\equiv \mathcal{P}^{\Sigma'}_{\Gamma'} \\
\eta^{\Sigma}_{\Gamma}(\Pi x{:}\sigma.\tau) & \triangleq & \begin{cases} \mathcal{L}^{\mathcal{P}}_{\eta^{\Sigma}_{\Gamma}(N),\eta^{\Sigma}_{\Gamma}(\sigma')}[\eta^{\Sigma}_{\Gamma,x:\sigma}(\tau)] & \text{if } \sigma \equiv (\mathcal{P}^{\Sigma'}_{\Gamma'}\ \vec{x}\ I_{\sigma'}\ N) \\ \Pi x{:}\eta^{\Sigma}_{\Gamma}(\sigma).\eta^{\Sigma}_{\Gamma,x:\eta^{\Sigma}_{\Gamma}(\sigma)}(\tau) & \text{otherwise} \end{cases} \\
\eta^{\Sigma}_{\Gamma}(\sigma N) & \triangleq & \eta^{\Sigma}_{\Gamma}(\sigma)\,\eta^{\Sigma}_{\Gamma}(N) \\
\eta^{\Sigma}_{\Gamma}(c) & \triangleq & c \qquad \text{if } c \in \mathsf{Dom}(\Sigma), \text{ and } c \neq c_{\mathcal{P}^{\Sigma'}_{\Gamma'}}) \\
\eta^{\Sigma}_{\Gamma}(x) & \triangleq & x \quad \text{if } x \in \mathsf{Dom}(\Gamma), \text{ and the type of } x \text{ is not of the form } (\mathcal{P}^{\Sigma'}_{\Gamma'}\ \vec{x}\ I_{\sigma'}\ N) \\
\eta^{\Sigma}_{\Gamma}(\lambda x{:}\tau.M) & \triangleq & \begin{cases} \mathcal{L}^{\mathcal{P}}_{\eta^{\Sigma}_{\Gamma}(N),\eta^{\Sigma}_{\Gamma}(\sigma')}[\eta^{\Sigma}_{\Gamma,x:\tau}(M)] & \text{if } \tau \equiv (\mathcal{P}^{\Sigma'}_{\Gamma'}\ \vec{x}\ I_{\sigma'}\ N) \\ \lambda x{:}\eta^{\Sigma}_{\Gamma}(\tau).\eta^{\Sigma}_{\Gamma,x:\eta^{\Sigma}_{\Gamma}(\tau)}(M) & \text{otherwise} \end{cases} \\
\eta^{\Sigma}_{\Gamma}(MN) & \triangleq & \begin{cases} \mathcal{U}^{\mathcal{P}}_{\eta^{\Sigma}_{\Gamma}(N'),\eta^{\Sigma}_{\Gamma}(\sigma)}[\eta^{\Sigma}_{\Gamma}(M)] & \text{if } N \equiv (c_{\mathcal{P}^{\Sigma}_{\Gamma}}\ \vec{x}\ I_{\sigma}\ N') \\ \mathcal{U}^{\mathcal{P}}_{\eta^{\Sigma}_{\Gamma}(N'),\eta^{\Sigma}_{\Gamma}(\sigma)}[\eta^{\Sigma}_{\Gamma}(M)] & \text{if } N \equiv y, \text{ and } y{:}(\mathcal{P}^{\Sigma}_{\Gamma}\ \vec{x}\ I_{\sigma}\ N') \in \Gamma \\ \eta^{\Sigma}_{\Gamma}(M)\,\eta^{\Sigma}_{\Gamma}(N) & \text{otherwise} \end{cases}
\end{array}
$$

4. Metatheory of $\mathsf{LLF}_{\mathcal{P}}$

We are now ready to prove all the most significant metatheoretic properties of $\mathsf{LLF}_{\mathcal{P}}$. As we remarked earlier, most of them, and most notably *Strong Normalization* and *Subject Reduction*, can be inherited uniformly from those of LF using Theorems 3.1 and 3.6 and the other results of the previous section.

4.1. **Confluence.** As it is often the case for systems without η-like conversions, confluence can be proved directly on *raw terms* as in [19, 25]. Namely using *Newman's Lemma* ([6], Chapter 3), and showing that the reduction on "raw terms" is *locally confluent*. But we can also make use of the decoding function η on closed good terms. Hence, we have:

Theorem 4.1 (Confluence of $\mathsf{LLF}_{\mathcal{P}}$)**.** *$\beta\mathcal{L}$-reduction is confluent, i.e.:*

1. *If $K \twoheadrightarrow_{\beta\mathcal{L}} K'$ and $K \twoheadrightarrow_{\beta\mathcal{L}} K''$, then there exists a K''' such that $K' \twoheadrightarrow_{\beta\mathcal{L}} K'''$ and $K'' \twoheadrightarrow_{\beta\mathcal{L}} K'''$.*
2. *If $\sigma \twoheadrightarrow_{\beta\mathcal{L}} \sigma'$ and $\sigma \twoheadrightarrow_{\beta\mathcal{L}} \sigma''$, then there exists a σ''' such that $\sigma' \twoheadrightarrow_{\beta\mathcal{L}} \sigma'''$ and $\sigma'' \twoheadrightarrow_{\beta\mathcal{L}} \sigma'''$.*
3. *If $M \twoheadrightarrow_{\beta\mathcal{L}} M'$ and $M \twoheadrightarrow_{\beta\mathcal{L}} M''$, then there exists an M''' such that $M' \twoheadrightarrow_{\beta\mathcal{L}} M'''$ and $M'' \twoheadrightarrow_{\beta\mathcal{L}} M'''$.*

4.2. **Strong Normalisation.** Strong normalisation can be proved following the same pattern used in [25], relying on the strong normalization of LF, as proven in [19]. However, we do not use a suitable extension of the "forgetful" function $^{-\mathcal{UL}} : \mathsf{LLF}_{\mathcal{P}} \to \mathsf{LF}$ (introduced in [25]), which maps $\mathsf{LLF}_{\mathcal{P}}$ terms into LF terms essentially deleting the $\mathcal{L}$ and $\mathcal{U}$ symbols, but rather we use the very encoding functions introduced in Section 3. Consider a well typed term M of $\mathsf{LLF}_{\mathcal{P}}$ such that $\Gamma \vdash_\Sigma M : \sigma$. Without loss of generality we can assume that M is closed. It is immediate to check that any sequence of $\to_{\beta\mathcal{L}}$-reductions starting from M can be reflected in an $\to_\beta$ reduction starting from $\theta^\Sigma_\Gamma(T)$ of the same length. This is a serendipitous consequence of the choice of encoding "locks as abstractions" and "unlocks as applications". Therefore, an infinite $\to_{\beta\mathcal{L}}$-reduction in $\mathsf{LLF}_{\mathcal{P}}$ would produce an infinite $\to_\beta$-reduction in LF, which is impossible, because LF is strongly normalizing.

Theorem 4.2 (Strong normalization of $\mathsf{LLF}_{\mathcal{P}}$)**.**

(1) If $\Gamma \vdash_\Sigma K$, then K is $\to_{\beta\mathcal{L}}$-strongly normalizing.
(2) if $\Gamma \vdash_\Sigma \sigma : K$, then σ is $\to_{\beta\mathcal{L}}$-strongly normalizing.
(3) if $\Gamma \vdash_\Sigma M : \sigma$, then M is $\to_{\beta\mathcal{L}}$-strongly normalizing.

4.3. **Subject Reduction.** Using Theorems 3.1 and 3.6, and in particular the property concerning the interplay between ϵ and θ^Σ_Γ and the one between δ and η^Σ_Γ, it is easy to argue that the inversion function δ commutes with reduction, *i.e.* inverting into $\mathsf{LLF}_{\mathcal{P}}$ the reduct in LF produces the reduct in $\mathsf{LLF}_{\mathcal{P}}$.

Whence, we can deduce the fundamental theorem of subject reduction:

Theorem 4.3 (Subject reduction of $\mathsf{LLF}_{\mathcal{P}}$)**.** *If predicates are well-behaved, then:*

(1) If $\Gamma \vdash_\Sigma K$, and $K \to_{\beta\mathcal{L}} K'$, then $\Gamma \vdash_\Sigma K'$.
(2) If $\Gamma \vdash_\Sigma \sigma : K$, and $\sigma \to_{\beta\mathcal{L}} \sigma'$, then $\Gamma \vdash_\Sigma \sigma' : K$.
(3) If $\Gamma \vdash_\Sigma M : \sigma$, and $M \to_{\beta\mathcal{L}} M'$, then $\Gamma \vdash_\Sigma M' : \sigma$.

Proof (sketch): Let us assume to have a derivation $\mathcal{D}_1 : \Gamma \vdash_\Sigma M : \sigma$, and a reduction for $M \to_{\beta\mathcal{L}} M'$ in $\mathsf{LLF}_{\mathcal{P}}$. Then we encode $\mathcal{D}_1$ with ϵ, yielding $\mathcal{D}'_1 : \Gamma' \vdash_{\Sigma'} M'' : \sigma'$, and the terms with $\theta^{\Sigma'}_{\Gamma'}$, yielding $M'' \to_\beta M'''$ in LF, where $\theta^{\Gamma'}_{\Sigma'}(M) = M''$, and $\theta^{\Gamma'}_{\Sigma'}(M') = M'''$, and $\theta^{\Gamma'}_{\Sigma'}(\sigma) = \sigma'$. Since in LF subject reduction holds, then there is a derivation $\mathcal{D}_3 : \Gamma' \vdash_{\Sigma'} M''' : \sigma'$. Thus, we can "decode" $\mathcal{D}_3$ in $\mathsf{LLF}_{\mathcal{P}}$ (via δ), yielding a derivation $\mathcal{D}'_3 : \Gamma \vdash_\Sigma \eta^{\Sigma'}_{\Gamma'}(M''') : \eta^{\Sigma'}_{\Gamma'}(\sigma')$, *i.e.*, a derivation $\mathcal{D}'_3 : \Gamma \vdash_\Sigma M' : \sigma$.

4.4. **Other properties.** In a similar way we can prove also other standard metatheoretic results:

Proposition 4.4 (Weakening and permutation)**.** *If predicates are closed under signature/context weakening and permutation, then:*

(1) If Σ and Ω are valid signatures, and every declaration occurring in Σ also occurs in Ω, then $\Gamma \vdash_\Sigma \alpha$ implies $\Gamma \vdash_\Omega \alpha$.
(2) If Γ and Δ are valid contexts w.r.t. the signature Σ, and every declaration occurring in Γ also occurs in Δ, then $\Gamma \vdash_\Sigma \alpha$ implies $\Delta \vdash_\Sigma \alpha$.

Proposition 4.5 (Transitivity)**.** *If predicates are closed under signature/context weakening and permutation and under substitution, then: if $\Gamma, x{:}\sigma, \Gamma' \vdash_\Sigma \alpha$, and $\Gamma \vdash_\Sigma N : \sigma$, then $\Gamma, \Gamma'[N/x] \vdash_\Sigma \alpha[N/x]$.*

As for the so-called *subderivation properties*, we need to be more careful, as is shown in the following section. The issue of decidability for $\mathsf{LLF}_{\mathcal{P}}$ can be addressed as that for $\mathsf{LF}_{\mathcal{P}}$ in [25].

4.5. **Expressivity.** We recall that a system $\mathcal{S}'$ is a conservative extension of $\mathcal{S}$ if the language of $\mathcal{S}$ is included in that of $\mathcal{S}'$, and moreover for all judgements $\mathcal{J}$, in the language of $\mathcal{S}$, then $\mathcal{J}$ is provable in $\mathcal{S}'$ if and only if $\mathcal{J}$ is provable in $\mathcal{S}$.

Theorem 4.6. $\mathsf{LLF}_{\mathcal{P}}$ *is a conservative extension of* LF.

Proof (sketch) The *if* part is trivial. For the *only if* part, consider a derivation in $\mathsf{LLF}_{\mathcal{P}}$ and drop all locks/unlocks (*i.e. release* the terms and types originally locked). This pruned derivation is a legal derivation in standard LF.

Notice that the above result holds independently of the particular nature or any property of the external oracles that we may *invoke* during the proof development (in $\mathsf{LLF}_{\mathcal{P}}$), *e.g.* decidability or recursive enumerability of $\mathcal{P}$.

Instead, $\mathsf{LLF}_{\mathcal{P}}$ is *not* a conservative extension of $\mathsf{LF}_{\mathcal{P}}$, since the new typing rule allows us to derive more judgements with unlocked-terms even if the predicate does not hold *e.g.*

$$\frac{\Gamma, x{:}\tau \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{S,\sigma}[x] : \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\tau] \quad \Gamma \vdash_\Sigma N : \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\tau] \quad S{=}_{\beta\mathcal{L}}S \quad \sigma{=}_{\beta\mathcal{L}}\sigma}{\Gamma \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{S,\sigma}[x[\mathcal{U}^{\mathcal{P}}_{S,\sigma}[N]/x]] : \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\tau[\mathcal{U}^{\mathcal{P}}_{S,\sigma}[N]/x]]}\ (O{\cdot}Guarded{\cdot}Unlock)$$

Then, since x does not occur free in τ, $\mathcal{L}^{\mathcal{P}}_{S,\sigma}[\tau[\mathcal{U}^{\mathcal{P}}_{S,\sigma}[N]/x]] \equiv \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\tau]$ and we get $\Gamma \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\mathcal{U}^{\mathcal{P}}_{S,\sigma}[N]] : \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\tau]$. This can be considered as the analogue of an η-expansion of $\Gamma \vdash_\Sigma N : \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\tau]$ and it cannot be carried out in plain $\mathsf{LF}_{\mathcal{P}}$ if $\mathcal{P}(\Gamma \vdash_\Sigma S : \sigma)$ does not hold.

However, as we noticed at the end of Section 2, in the Guarded Unlock Rules we require that the subject of the first premise be necessarily externally locked. This fact, even in the presence of locked variables in the typing context, avoids to derive unlocked terms at top level. Indeed, we have the following:

Theorem 4.7 (Soundness of unlock)**.** *If* $\Gamma \vdash_\Sigma \mathcal{U}^{\mathcal{P}}_{N,\sigma}[M] : \tau$ *is derived in* $\mathsf{LLF}_{\mathcal{P}}$ *then* $\mathcal{P}(\Gamma \vdash_\Sigma N : \sigma)$ *is true.*

Proof The proof can be carried out by a straightforward induction on the derivation of $\Gamma \vdash_\Sigma \mathcal{U}^{\mathcal{P}}_{N,\sigma}[M] : \tau$. So doing, we immediately restrict the possibilities for the last rule used in the derivation to $(O{\cdot}Top{\cdot}Unlock)$ and to $(O{\cdot}Conv)$ which affect only on the type of the judgment.

Nevertheless, we have to phrase the so-called subderivation properties carefully. Indeed, in $\mathsf{LLF}_{\mathcal{P}}$, given a derivation of $\Gamma \vdash_\Sigma \alpha$ and a subterm N occurring in the subject of this judgement, we cannot prove that there always exists a derivation of the form $\Gamma \vdash_\Sigma N : \tau$ (for a suitable τ). Consider, for instance, the previous example concerning the derivation of $\Gamma \vdash_\Sigma \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\mathcal{U}^{\mathcal{P}}_{S,\sigma}[N]] : \mathcal{L}^{\mathcal{P}}_{S,\sigma}[\tau]$. Clearly, if $\mathcal{P}(\Gamma \vdash_\Sigma S : \sigma)$ does not hold, then we cannot derive any judgement whose subject and predicate are $\mathcal{U}^{\mathcal{P}}_{S,\sigma}[N] : \tau$.

Hence we have to restate point 6 of Proposition 3.11 (Subderivation, part 1) of [25] in a way similar to what we did in [27].

Proposition 4.8 (Subderivation, part 1, point 6)**.** *Given a derivation* $\mathcal{D} : \Gamma \vdash_\Sigma \alpha$*, and a subterm* N *occurring in the subject of this judgement, we have that either there exists a subderivation of a judgement having* N *as a subject, or there exists a derivation of a judgment having as subject* $\mathcal{L}^{\mathcal{P}}_{S,\sigma}[N]$ *(for suitable* $\mathcal{P}$*,* S*,* σ*).*

Proof The proof is carried out by induction on the derivation of $\Gamma \vdash_\Sigma \alpha$.

5. Case Studies

In this section we discuss encodings of logics in $\mathsf{LLF}_\mathcal{P}$. Of course, all encodings given in [25] for $\mathsf{LF}_\mathcal{P}$, carry over immediately to the setting of $\mathsf{LLF}_\mathcal{P}$, because the latter is a language extension of the former. So here, we do not present encodings for *modal* and *ordered linear logic*. However, the possibility of using guarded unlocks, *i.e.* the full power of the monad destructor, allows for significant simplifications in several of the encodings of logical systems given in $\mathsf{LF}_\mathcal{P}$. We illustrate this point discussing call-by-value λ_v-calculus, which greatly benefits from the possibility of applying functions to locked-arguments, and Hoare's Logic, which combines various kinds of syntactical and semantical locks in its rules. We do not discuss adequacy of these encodings since it is a trivial variant of the one presented in [25]. One of the crucial problems in designing restricted logical systems is to enforce *incrementally*, *locally*, in rule application *global* constraints on derivations. $\mathsf{LLF}_\mathcal{P}$ can be very useful in this respect since it allows one to focus precisely on the shape of the well-behaved predicate. A classical case in point is *Fitch Naive Set Theory* as formalized *e.g.* in Prawitz [36]. The global constraint is that derivations be normalizable. This means that the elimination rules must not generate non-normalizable derivations. In order to enforce this using a Lock-type, we need to introduce a well-behaved predicate, *i.e.* closed under substitution. This is easy if the proof is closed, *i.e.* it does not involve assumptions, otherwise we must make sure that no future instantiations are made on the variables corresponding to the assumptions in the proof. To achieve this we introduce two kinds of judgements the *apodictic judgements*, *i.e.* those which are actually involved in the proofs and the *generic* ones. Unspecific judgements appear only in assumptions and in order to be used must be "demoted" to an *apodictic* judgement. What happens in terms within the scope of the demoting operator does not matter for the validity of the predicate. Thus variables witnessing generic assumptions, even if replaced, behave as constants while variables witnessing an apodictic judgements can be freely substituted. The local constraint in the elimination rules accesses the input proof-terms, checks that the combination can be normalized and furthermore that all free variables of judgement type are generic, see Subsection 5.3.

5.1. Call-by-value λ_v-calculus.

We encode, using *Higher Order Abstract Syntax* (HOAS), the syntax of untyped λ-calculus: $M, N ::= x \mid M\ N \mid \lambda x.M$ as in [25], where natural numbers (through the constructor `free`) are used to represent free variables, while bound variables are rendered as metavariables of $\mathsf{LLF}_\mathcal{P}$ of type `term`:

Definition 5.1 ($\mathsf{LLF}_\mathcal{P}$ signature Σ_λ for untyped λ-calculus)**.**

```
term : Type                    nat  :  Type                          0 : nat
   S :  nat -> nat             free :   nat -> term
 app : term -> term -> term    lam  : (term -> term) -> term
```

Definition 5.2 (Call-by-value reduction strategy)**.** The call-by-value evaluation strategy is given by:

$$\frac{}{\vdash_{\mathsf{v}} M = M}\ (\mathsf{refl}) \qquad \frac{\vdash_{\mathsf{v}} N = M}{\vdash_{\mathsf{v}} M = N}\ (\mathsf{symm})$$

$$\frac{\vdash_{\mathsf{v}} M = N \quad \vdash_{\mathsf{v}} N = P}{\vdash_{\mathsf{v}} M = P}\ (\mathsf{trans}) \qquad \frac{\vdash_{\mathsf{v}} M = N \quad \vdash_{\mathsf{v}} M' = N'}{\vdash_{\mathsf{v}} M\,M' = N\,N'}\ (\mathsf{app})$$

$$\frac{v \text{ is a value}}{\vdash_{\mathsf{v}} (\lambda x.M)\,v = M[v/x]}\ (\beta_{\mathsf{v}}) \qquad \frac{\vdash_{\mathsf{v}} M = N}{\vdash_{\mathsf{v}} \lambda x.M = \lambda x.N}\ (\xi_{\mathsf{v}})$$

where values are either variables, constants, or abstractions.

The new typing rule (*O·Guarded·Unlock*) of $\mathsf{LLF}_{\mathcal{P}}$, allows to encode naturally the system as follows.

Definition 5.3 ($\mathsf{LLF}_{\mathcal{P}}$ signature Σ_{v} for λ_{v}-calculus)**.** We extend the signature of Definition 5.1 as follows:

```
   eq : term->term->Type
 refl : ΠM:term.(eq M M)
 symm : ΠM:term.ΠN:term.(eq N M)->(eq M N)
trans : ΠM,N,P:term.     (eq M N)->(eq N P) ->(eq M P)
eq_app: ΠM,N,M',N':term. (eq M N)->(eq M'N')->(eq (app M M')(app N N'))
betav : ΠM:(term->term).   ΠN:term.L^Val_N,term[(eq (app (lam M) N)(M N))]
 csiv : ΠM,N:(term->term).(Πx:term.L^Val_x,term[(eq (M x)(N x))])->(eq (lam M)(lam N))
```

where the predicate Val is defined as follows:
– $\mathit{Val}\,(\Gamma \vdash_{\Sigma} \mathtt{N} : \mathtt{term})$ holds iff either `N` is an abstraction or a constant (*i.e.* a term of the shape `(free i)`).

Notice the neat improvement w.r.t. to the encoding of $\mathsf{LF}_{\mathcal{P}}$, given in [25], as far as the rule `csiv`. The encoding of the rule ξ_{v} is problematic if bound variables are encoded using metavariables, because the predicate `Val` appearing in the lock cannot mention explicitly variables, for it to be *well-behaved*. In [25], since we could not apply the rules unless we had explicitly eliminated the `Val`-lock, in order to overcome the difficulty we had to make a detour using constants. In $\mathsf{LLF}_{\mathcal{P}}$, on the other hand, we can apply the rules "under `Val`", so to speak, and postpone the proof of the `Val`-checks till the very end, and then rather than checking `Val` we can get rid of the lock altogether, since the bound variable of the rule `csiv`, is assumed to be locked. Notice that this phrasing of the rule `csiv` amounts precisely to the fact that in λ_{v} variables range over values. As a concrete example of all this, we show how to derive the equation $\lambda x.z\,((\lambda y.y)\,x) = \lambda x.z\,x$. Using "pencil and paper" we would proceed as follows:

$$\dfrac{\dfrac{-}{\vdash_{\mathsf{v}} z = z}\ (\mathsf{refl}) \qquad \dfrac{x \text{ is a value}}{(\lambda y.y)\,x = y[x/y]}\ (\beta_{\mathsf{v}})}{\dfrac{\vdash_{\mathsf{v}} z\,((\lambda y.y)\,x) = z\,x}{\vdash_{\mathsf{v}} \lambda x.z\,((\lambda y.y)\,x) = \lambda x.z\,x}\ (\xi_{\mathsf{v}})}\ (\mathsf{app})$$

Similarly, in $\mathsf{LLF}_{\mathcal{P}}$, we can derive $\mathtt{z{:}term} \vdash_{\Sigma} (\mathtt{refl\ z}) : (\mathtt{eq\ z\ z})$ and

$$\Gamma, \mathtt{x{:}term} \vdash_{\Sigma} (\mathtt{betav}\,(\lambda \mathtt{y{:}term.y})\,\mathtt{x}) : \mathcal{L}^{\mathtt{Val}}_{\mathtt{x,term}}[(\mathtt{eq}\,(\mathtt{app}\,(\mathtt{lam}\,\lambda \mathtt{y{:}term.y})\,\mathtt{x})\,((\lambda \mathtt{y{:}term.y})\,\mathtt{x}))].$$

This far, in old $\mathsf{LF}_{\mathcal{P}}$, we would be blocked if we could not prove that $\mathtt{Val}(\Gamma, \mathtt{x{:}term} \vdash_{\Sigma} \mathtt{x : term})$ holds, since `eq_app` cannot accept an argument with a locked-type. However, in

$\mathsf{LLF}_{\mathcal{P}}$, we can apply the (*O·Guarded·Unlock*) rule obtaining the following proof term (from the typing environment $\Gamma, \mathtt{x{:}term}, \mathtt{z{:}term}$):

$$\mathcal{L}^{\mathtt{Val}}_{\mathtt{x,term}}[(\mathtt{eq\_app\ z\ z\ (app\ (lam}\ \lambda\mathtt{y{:}term.y)\ x)\ x\ (refl\ z)}\ \mathcal{U}^{\mathtt{Val}}_{\mathtt{x,term}}[(\mathtt{betav}\ (\lambda\mathtt{y{:}term.y})\ \mathtt{x})])]$$

of type $\mathcal{L}^{\mathtt{Val}}_{\mathtt{x,term}}[(\mathtt{eq\ (app\ z\ (app\ (lam}\ \lambda\mathtt{y{:}term.y)\ x))\ (app\ z\ x))}]$. And abstracting x, a direct application of `csiv` yields the result.

5.2. **Imp with Hoare Logic.** An area of Logic which can greatly benefit from the new system $\mathsf{LLF}_{\mathcal{P}}$ is *program logics*, because of the many syntactical checks which occur in these systems. To illustrate this fact, we consider a very simple imperative language Imp, whose syntax is:

$$\begin{array}{rcll} p & ::= & \mathit{skip} \mid x := \mathit{expr} \mid p; p \mid & \text{null} \mid \text{assignment} \mid \text{sequence} \\ & & \mathit{if\ cond\ then\ p\ else\ p} \mid \mathit{while\ cond}\ \{p\} & \text{cond} \mid \text{while} \end{array}$$

Other primitive notions of Imp are variables, both integer and *identifier*, and expressions. Identifiers denote locations. For the sake of simplicity, we assume only integers (represented by type `int`) as possible values for identifiers. In this section, we follow as closely as possible the HOAS encoding, originally proposed in [4], in order to illustrate the features and possible advantages of using $\mathsf{LLF}_{\mathcal{P}}$ w.r.t. LF. The main difference with that approach is that here we encode *concrete* identifiers by constants of type `var`, an `int`-like type, of course different from `int` itself, so as to avoid confusion with possible values of locations.

Definition 5.4 ($\mathsf{LLF}_{\mathcal{P}}$ signature Σ_{Imp} for Imp)**.**

```
 int : Type                          bool : Type
 var : Type                        and,imp : bool -> bool -> bool
bang :  var -> int                  0,1,-1 :  int
   = :  int -> int -> bool               + :  int ->  int -> int
 not : bool -> bool                 forall : (int -> bool) -> bool
```

Since variables of type `int` may be bound in expressions (by means of the `forall` constructor), we define explicitly the encoding function $\epsilon^{\mathtt{exp}}_{\mathcal{X}}$ mapping expressions with free variables of type `int` in $\mathcal{X}$ of the source language Imp into the corresponding terms of $\mathsf{LLF}_{\mathcal{P}}$:

$$\epsilon^{\mathtt{exp}}_{\mathcal{X}}(0) \triangleq \mathtt{0} \qquad \epsilon^{\mathtt{exp}}_{\mathcal{X}}(+/-1) \triangleq \mathtt{+/-1}$$

$$\epsilon^{\mathtt{exp}}_{\mathcal{X}}(x) \triangleq \begin{cases} \mathtt{x} & \text{if } x \in \mathcal{X} \\ \mathtt{(bang\ x)} & \text{if } x \notin \mathcal{X} \end{cases}$$

$$\epsilon^{\mathtt{exp}}_{\mathcal{X}}(n+m) \triangleq (\mathtt{+}\ \epsilon^{\mathtt{exp}}_{\mathcal{X}}(n)\, \epsilon^{\mathtt{exp}}_{\mathcal{X}}(m)) \qquad \epsilon^{\mathtt{exp}}_{\mathcal{X}}(n=m) \triangleq (\mathtt{=}\ \epsilon^{\mathtt{exp}}_{\mathcal{X}}(n)\, \epsilon^{\mathtt{exp}}_{\mathcal{X}}(m))$$

$$\epsilon^{\mathtt{exp}}_{\mathcal{X}}(\neg e) \triangleq (\mathtt{not}\ \epsilon^{\mathtt{exp}}_{\mathcal{X}}(e)) \qquad \epsilon^{\mathtt{exp}}_{\mathcal{X}}(e \wedge e') \triangleq (\mathtt{and}\ \epsilon^{\mathtt{exp}}_{\mathcal{X}}(e)\, \epsilon^{\mathtt{exp}}_{\mathcal{X}}(e'))$$

$$\epsilon^{\mathtt{exp}}_{\mathcal{X}}(e \supseteq e') \triangleq (\mathtt{imp}\ \epsilon^{\mathtt{exp}}_{\mathcal{X}}(e)\, \epsilon^{\mathtt{exp}}_{\mathcal{X}}(e')) \qquad \epsilon^{\mathtt{exp}}_{\mathcal{X}}(\forall x.\phi) \triangleq (\mathtt{forall}\ \lambda\mathtt{x{:}int.}\epsilon^{\mathtt{exp}}_{\mathcal{X}\cup\{x\}}(\phi))$$

where `x` in `(bang x)` denotes the encoding of the concrete memory location (*i.e.*, a constant of type `var`) representing the (free) source language identifier x; the other case represents the free variable x rendered as a $\mathsf{LLF}_{\mathcal{P}}$ metavariable `x` of type `int` in HOAS style. The syntax of imperative programs is defined as follows:

Definition 5.5 ($\mathsf{LLF}_{\mathcal{P}}$ signature Σ_{Imp} for Imp with command)**.**
We extend the signature of Definition 5.4 as follows:

```
  prog : Type
 Iskip : prog
  Iseq : prog -> prog -> prog
  Iset :  var ->  int -> prog
   Iif : Πe:bool.prog -> prog ->L^QF_{e,bool}[prog]
Iwhile : Πe:bool.prog -> L^QF_{e,bool}[prog]
```

where the predicate $\mathtt{QF}(\Gamma \vdash_{\Sigma_{\mathsf{Imp}}} \mathtt{e{:}bool})$ holds iff the formula `e` is *closed* and *quantifier free*, *i.e.*, it does not contain the `forall` constructor. We can look at `QF` as a "good formation" predicate, ruling out *bad programs with invalid boolean expressions* by means of stuck terms.

The encoding function $\epsilon^{\mathtt{prog}}_{\mathcal{X}}$ mapping programs with free variables in $\mathcal{X}$ of the source language Imp into the corresponding terms of $\mathsf{LLF}_{\mathcal{P}}$ is defined as follows:

$$\epsilon^{\mathtt{prog}}_{X}(skip) \triangleq \mathtt{Iskip}$$

$$\epsilon^{\mathtt{prog}}_{X}(x := e) \triangleq \mathtt{(Iset\ x}\ \epsilon^{\mathtt{exp}}_{X}(e)\mathtt{)}$$

$$\epsilon^{\mathtt{prog}}_{\mathcal{X}}(p\ ;\ p') \triangleq \mathtt{(Iseq}\ \epsilon^{\mathtt{prog}}_{\mathcal{X}}(p)\ \epsilon^{\mathtt{prog}}_{\mathcal{X}}(p')\mathtt{)}$$

$$\epsilon^{\mathtt{prog}}_{\mathcal{X}}(if\ e\ then\ p\ else\ p') \triangleq \mathcal{U}^{\mathtt{QF}}_{\epsilon^{\mathtt{exp}}_{\mathcal{X}}(e),\mathtt{bool}}[(\mathtt{Iif}\ \epsilon^{\mathtt{exp}}_{\mathcal{X}}(e)\ \epsilon^{\mathtt{prog}}_{\mathcal{X}}(p)\ \epsilon^{\mathtt{prog}}_{\mathcal{X}}(p'))]\ (*)$$

$$\epsilon^{\mathtt{prog}}_{\mathcal{X}}(while\ e\ \{p\}) \triangleq \mathcal{U}^{\mathtt{QF}}_{\epsilon^{\mathtt{exp}}_{\mathcal{X}}(e),\mathtt{bool}}[(\mathtt{Iwhile}\ \epsilon^{\mathtt{exp}}_{\mathcal{X}}(e)\ \epsilon^{\mathtt{prog}}_{\mathcal{X}}(p))]\ (*)$$

(*) if e is a quantifier-free formula. However, in the last two clauses, the terms on the right hand side cannot be directly expressed in general form (*i.e.*, for all expressions e) because if $\mathtt{QF}(\Gamma \vdash_{\Sigma_{\mathsf{Imp}}} \epsilon^{\mathtt{exp}}_{\mathcal{X}}(e) : \mathtt{bool})$ does not hold, we cannot use the unlock operator. Thus we could be left with two terms of type $\mathcal{L}^{\mathtt{QF}}_{\epsilon^{\mathtt{exp}}_{\mathcal{X}}(e),\mathtt{bool}}[\mathtt{prog}]$, instead of type `prog`. This is precisely the limit of the $\mathsf{LF}_{\mathcal{P}}$ encoding in [25]. Since a $\mathcal{U}$-term can only be introduced if the corresponding predicate holds, when we represent rules of Hoare Logic we are forced to consider only legal terms, and this ultimately amounts to restricting explicitly the object language in a way such that `QF` always returns true.

In $\mathsf{LLF}_{\mathcal{P}}$, instead, we can use naturally the following signature for representing Hoare's Logic, without assuming anything about the object language terms (given the predicate `true : bool -> Type` such that `(true e)` holds iff `e` is true):

Definition 5.6 ($\mathsf{LLF}_{\mathcal{P}}$ signature Σ_{HL} for Hoare Logics)**.**

```
       args : Type
      <_,_> :  var -> (int -> bool) -> args
      hoare : bool -> prog -> bool  -> Type
hoare_Iskip : Πe:bool.(hoare e Iskip e)
 hoare_Iset : Πt:int.Πx:var.Πe:int -> bool.
                          L^Pset_{⟨x,e⟩,args}[(hoare (e t) (Iset x t) (e (bang x))]
 hoare_Iseq : Πe,e',e'':bool.Πp,p':prog.(hoare e  p  e')  ->
                                        (hoare e' p' e'') ->
                                        (hoare e (Iseq p p') e'')
  hoare_Iif : Πe,e',b:bool.Πp,p':prog.(hoare (b and e) p e') ->
                                      (hoare ((not b) and e) p' e') ->
```

$$\mathcal{L}^{\mathtt{QF}}_{\mathtt{b,bool}}[(\mathtt{hoare\ e}\ \mathcal{U}^{\mathtt{QF}}_{\mathtt{b,bool}}[(\mathtt{Iif\ b\ p\ p'})]\ \mathtt{e'})]$$

$$\mathtt{hoare\_Iwhile} : \Pi \mathtt{e,b:bool}.\Pi \mathtt{p:prog}.\mathtt{(hoare\ (e\ and\ b)\ p\ e)} \to \mathcal{L}^{\mathtt{QF}}_{\mathtt{b,bool}}[(\mathtt{hoare\ e}\ \mathcal{U}^{\mathtt{QF}}_{\mathtt{b,bool}}[(\mathtt{Iwhile\ b\ p})]\ \mathtt{((not\ b)\ and\ e))}]$$

$$\mathtt{hoare\_Icons} : \Pi \mathtt{e,e',f,f':bool}.\Pi \mathtt{p:prog}.\mathtt{(true\ (imp\ e'\ e))} \to \mathtt{(hoare\ e\ \ p\ f)\ \ \ } \to \mathtt{(true\ (imp\ f\ f'))} \to \mathtt{(hoare\ e'\ p\ f')}$$

where $\mathtt{P^{set}}(\Gamma \vdash_{\Sigma_{\mathsf{HL}}} \langle \mathtt{x,e} \rangle : \mathtt{args})$ holds iff `e` is closed[4] and the location (*i.e.*, constant) `x` does not occur in `e`. Such requirements amount to formalizing that no assignment made to the location denoted by `x` affects the meaning or value of `e` (*non-interference* property). The intuitive idea here is that

> if $\mathtt{e}{=}\epsilon^{\mathtt{exp}}_{\mathcal{X}}(E)$, and $\mathtt{p}{=}\epsilon^{\mathtt{prog}}_{\mathcal{X}}(P)$, and $\mathtt{e'}{=}\epsilon^{\mathtt{exp}}_{\mathcal{X}}(E')$ hold,
> then `(hoare e p e')` holds iff the Hoare's triple $\{E\}P\{E'\}$ holds.

The advantage w.r.t. previous encodings (see, *e.g.*, [4]), is that in $\mathsf{LLF}_{\mathcal{P}}$ we can delegate to the external predicates `QF` and $\mathtt{P^{set}}$ all the complicated and tedious checks concerning *non-interference* of variables and good formation clauses for guards in the conditional and looping statements. Thus, the use of lock-types, which are subject to the verification of such conditions, allows to legally derive $\Gamma \vdash_{\Sigma_{\mathsf{HL}}} \mathtt{m} : \mathtt{(hoare\ e\ p\ e')}$ only according to the Hoare semantics.

Moreover, the (*O·Guarded·Unlock*) rule allows also to "postpone" the verification that $\mathtt{QF}(\Gamma \vdash_{\Sigma} \mathtt{e} : \mathtt{bool})$ holds (*i.e.*, that the formula `e` is *quantifier free*).

5.3. **Fitch Set Theory *à la* Prawitz.** In this section we present the encoding of a logical system of remarkable logical, as well as historical, significance, namely the system of consistent *Naive* Set Theory, `F` due to Fitch [16]. This system, was first presented in Natural Deduction style by Prawitz [36]. Naive Set Theory, being inconsistent, in order to prevent the derivation of inconsistencies from the unrestricted *abstraction* rule, in the system `F` only normalizable *deductions* are allowed. Of course this side condition in the rule is extremely difficult to capture using traditional tools.

In the present context, instead, we can put to use the machinery of $\mathsf{LLF}_{\mathcal{P}}$ to provide an appropriate encoding of `F` where the *global* normalization constraint is enforced *locally* by checking the proof-object. This system is a beautiful example for illustrating the *bag of tricks* of $\mathsf{LLF}_{\mathcal{P}}$. Checking that a proof term is normalizable would be the obvious predicate to use in the corresponding lock type, but this would not be a well-behaved predicate if free variables, *i.e.* assumptions, are not sterilized, because predicates need to be well-behaved. To this end we introduce a distinction between *generic* judgements, which cannot be directly utilized in arguments, but which can be assumed and *apodictic* judgements, which are directly involved in proof rules. In order to make use of generic judgements, one has to downgrade them to an apodictic one. This is achieved by a suitable coercion function.

Definition 5.7 ($\mathsf{LLF}_{\mathcal{P}}$ signature Σ_{FPST} for Fitch Prawitz Set Theory)**.** The following constants are introduced:

[4]Otherwise, the predicate P would not be well-behaved, see Definition 2.1.

```
o   : Type
ι   : Type
T   : o -> Type
V   : o -> Type
lam : (ι -> o)-> ι
∀   : (ι -> o)-> o
ϵ   : ι -> (ι -> o ) -> o
⊃   : o -> o -> o
δ       : ΠA:o.  (V(A) -> T(A))
⊃_intro : ΠA,B:o.(V(A) -> T(B)) -> (T(A ⊃B))
⊃_elim  : ΠA,B:o.Πx:T(A).Πy:T(A⊃B) -> L^Fitch_⟨x,y⟩,T(A)×V(A)->T(B)[T(B)]
```

where `o` is the type of propositions, $\supset$ is the syntactic constructor for propositions together with the "membership" predicate ϵ and `lam` is the "abstraction" operator for building "sets", `T` is the apodictic judgement, while `V` s the generic judgement, and $< \mathtt{x}, \mathtt{y} >$ denotes the encoding of pairs, whose type is denoted by $\sigma \mathtt{X} \tau$, *e.g.*

$$\lambda \mathtt{u}{:}\sigma \to \tau \to \rho.\ \mathtt{u}\ \mathtt{x}\ \mathtt{y} : (\sigma \to \tau \to \rho) \to \rho$$

The predicate in the lock is defined as follows:

$$\mathtt{Fitch}(\Gamma \vdash_{\mathsf{FPST}} < \mathtt{x}, \mathtt{y} > : \mathtt{T(A)XV(A)} \to \mathtt{T(B)})$$

it holds iff the proof derived by combining `x` and `y` is normalizable and all occurrences of free variables of judgement type occur within the scope of a δ.

For lack of space we do not spell out the rules concerning the other logical operators, because they are all straightforward provided we use only the apodictic judgement $\mathtt{T}(\cdot)$. But a few remarks are mandatory. The notion of *normalizable proof* is the standard notion of normal proof used in natural deduction. The predicate `Fitch` is well-behaved because free judgement variables cannot be replaced by any sensible object. Adequacy for this signature can be achieved in the general formulation of [25], namely:

Theorem 5.8 (Adequacy for Fitch-Prawitz Naive Set Theory)**.** $A_1 \dots A_n \vdash_{\mathsf{FPST}} A$ *iff there exists a normalizable* $\mathtt{M}$ *such that* $\mathtt{x_1}{:}\mathtt{V}(\mathtt{A_1}), \dots, \mathtt{x_n}{:}\mathtt{V}(\mathtt{A_n}) \vdash_{\mathsf{FPST}} \mathtt{M} : \mathtt{T}(\mathtt{A})$.

6. Concluding remarks: from Predicates to Functions and beyond

We have shown how to extend LF with a class of monads which capture the effect of delegating to an external oracle the task of providing part of the necessary evidence for establishing a judgement. Thus we have extended with an additional clause the LF paradigm for encoding a logic, namely: *external evidence as monads*. This class of monads is very well-behaved and so it permits to simplify the equational theory of the system. In principle we could have used the let_T destructor, together with its equational theory as in Moggi's general approach [31], but we think that our approach greatly simplifies the theory, since it does away with permutative reductions.

The technique for proving the metatheoretic properties of $\mathsf{LLF}_{\mathcal{P}}$ that we used in this paper is rather powerful and novel. It generalizes the technique that was traditionally used to prove normalization properties for systems with dependent types, by reducing such languages to the corresponding dependency-less systems, see [7]. In this paper we have actually managed to reduce the whole proof system $\mathsf{LLF}_{\mathcal{P}}$ to LF as well as to translate it back, thereby transferring nearly all metatheoretic properties of LF to our new system.

We have presented $\mathsf{LLF}_{\mathcal{P}}$ in the standard style, but as future work we want to move to the *canonical* style of [20] in vision of a future prototype implementation.

In this paper we consider the verification of predicates in locks as purely *atomic actions*, *i.e.* each predicate *per se*. But of course predicates have a logical structure which can be reflected onto locks. *E.g.* we can consistently extend $\mathsf{LLF}_{\mathcal{P}}$ by assuming that locks *commute, combine*, and *entail*, *i.e.* that the following types are inhabited:

$$\mathcal{L}^{\mathcal{P}}_{x,\sigma}[\tau] \to \mathcal{L}^{\mathcal{Q}}_{x,\sigma}[\tau] \text{ if}$$
$$\mathcal{P}(\Gamma \vdash_\Sigma x : \sigma) \to \mathcal{Q}(\Gamma \vdash_\Sigma x : \sigma), \text{ and}$$
$$\mathcal{L}^{\mathcal{P}}_{x,\sigma}[\mathcal{L}^{\mathcal{Q}}_{x,\sigma}[M]] \to \mathcal{L}^{\mathcal{P}\&\mathcal{Q}}_{x,\sigma}[M], \text{ and}$$
$$\mathcal{L}^{\mathcal{P}}_{x,\sigma}[\mathcal{L}^{\mathcal{Q}}_{y,\tau}[M]] \to \mathcal{L}^{\mathcal{Q}}_{y,\tau}[\mathcal{L}^{\mathcal{P}}_{x,\sigma}[M]].$$

We encoded call-by-value λ-calculus with Plotkin's classical notion of *value*. But the encoding remains the same, apart from what is delegated to the lock, if we consider other notions of value *e.g. closed normal forms* only for K-redexes [23]. The example of Fitch's system suggests further generalization, and illustrates how monads handle side-conditions uniformly.

Yet, as a near future work, another interesting direction will be to extend $\mathsf{LLF}_{\mathcal{P}}$ to oracle-calls which produce an *output*. Namely, rather then just having external predicates which check that a judgement satisfies a given property, we could give the oracle a *query* and let it provide the *witness*. More precisely the lock operator $\mathcal{L}$ could bind a particular variable `x` in `M` that needs to be instantiated. The predicate in the lock would then become a sort of query on `x`, which could be fed to the oracle. If successful, the unlock operator could provide then this witness. The Unlock/Lock reduction would amount to replacing `x` by the witness. Suitable *compilation* and *decompilation* functions between LF and the language of the oracle should allow for the correct expression of the witness.

References

[1] H. Albert. *Traktat über kritische Vernunft*. J.C.B. Mohr (Paul Siebeck), Tübingen, 1991.

[2] N. Alechina, M. Mendler, V. De Paiva, and E. Ritter. Categorical and kripke semantics for constructive s4 modal logic. In *Computer Science Logic*, pages 292–307. Springer, 2001.

[3] Archimede. *Metodo. Nel laboratorio di un genio*. Bollati Boringhieri, 2013.

[4] A. Avron, F. Honsell, I. Mason, and R. Pollack. Using Typed Lambda Calculus to Implement Formal Systems on a Machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992.

[5] A. Badiou. *La relation énigmatique entre philosophie et politique*. Germina, 2011.

[6] H. Barendregt. *Lambda Calculus: its Syntax and Semantics*. North Holland, 1984.

[7] H. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, volume II, pages 118–310. Oxford University Press, 1992.

[8] H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28:321–336, 2002.

[9] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Pattern Type Systems. In *POPL'03*, pages 250–261. The ACM Press, 2003.

[10] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in hol. In *Theorem Provers in Circuit Design, TPCD*, pages 129–156. North-Holland, 1992.

[11] L. Carroll. What the Tortoise Said to Achilles. *Mind*, 4:278–280, 1895.

[12] D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In *Typed Lambda Calculi and Applications, TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer-Verlag, 2007.

[13] M. Fairtlough and M. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, 1997.
[14] M. Fairtlough, M. Mendler, and X. Cheng. Abstraction and refinement in higher order logic. In *Theorem Proving in Higher Order Logics*, pages 201–216. Springer, 2001.
[15] M. Fairtlough, M. Mendler, and M. Walton. First-order lax logic as a framework for constraint logic programming. Technical report, 1997.
[16] F. B. Fitch. *Symbolic logic*. New York, 1952.
[17] D. Garg and M. C. Tschantz. From indexed lax logic to intuitionistic logic. Technical report, DTIC Document, 2008.
[18] J.-Y. Girard. *The Blind Spot: lectures on logic*. European Mathematical Society, 2011.
[19] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1993. Preliminary version in Proc. of LICS'87.
[20] R. Harper and D. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17:613–673, 2007.
[21] D. Hirschkoff. Bisimulation proofs for the π-calculus in the Calculus of Constructions. In *Proc. TPHOL'97*, number 1275 in LNCS. Springer-Verlag, 1997.
[22] F. Honsell. 25 years of formal proof cultures: Some problems, some philosophy, bright future. In *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-languages: Theory and Practice*, LFMTP'13, pages 37–42, New York, NY, USA, 2013. ACM.
[23] F. Honsell and M. Lenisa. Semantical analysis of perpetual strategies in λ-calculus. *Theoretical Computer Science*, 212(1):183–209, 1999.
[24] F. Honsell, M. Lenisa, and L. Liquori. A Framework for Defining Logical Frameworks. *Volume in Honor of G. Plotkin, ENTCS*, 172:399–436, 2007.
[25] F. Honsell, M. Lenisa, L. Liquori, P. Maksimovic, and I. Scagnetto. An Open Logical Framework. *Journal of Logic and Computation*, Oct. 2013.
[26] F. Honsell, M. Lenisa, L. Liquori, and I. Scagnetto. A conditional logical framework. In *LPAR'08*, volume 5330 of *LNCS*, pages 143–157. Springer-Verlag, 2008.
[27] F. Honsell, L. Liquori, and I. Scagnetto. $\mathsf{L}^{\mathsf{x}}\mathsf{F}$: Side Conditions and External Evidence as Monads. In *Proc. of MFCS 2014 (39th International Symposium on Mathematical Foundations of Computer Science), Part I*, volume 8634 of *Lecture Notes in Computer Science*, pages 327–339, Budapest, Hungary, August 2014. Springer.
[28] F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (Co)Inductive Type Theories. *Theoretical Computer Science*, 253(2):239–285, 2001.
[29] M. Mendler. Constrained proofs: A logic for dealing with behavioural constraints in formal hardware verification. In *Designing Correct Circuits*, pages 1–28. Springer-Verlag, 1991.
[30] E. Moggi. *The partial lambda calculus*. PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics, 1988.
[31] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science (LICS 1989)*, pages 14–23. IEEE Computer Society Press, June 1989.
[32] A. Nanevski, F. Pfenning, and B. Pientka. Contextual Modal Type Theory. *ACM Transactions on Computational Logic*, 9(3), 2008.
[33] F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Proc. of CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer-Verlag, 1999.
[34] B. Pientka and J. Dunfield. Programming with proofs and explicit contexts. In *PPDP'08*, pages 163–173. ACM, 2008.
[35] B. Pientka and J. Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21. Springer-Verlag, 2010.
[36] D. Prawitz. *Natural Deduction. A Proof Theoretical Study*. Almqvist Wiksell, Stockholm, 1965.
[37] A. Schopenhauer. *The World as Will and Representation*, volume 2. Dover edition, 1966.
[38] P. Schroeder-Heister. Paradoxes and Structural Rules. In C. D. Novaes and O. T. Hjortland, editors, *Insolubles and consequences : essays in honour of Stephen Read*, pages 203–211. College Publications, London, 2012.

[39] P. Schroeder-Heister. Proof-theoretic semantics, self-contradiction, and the format of deductive reasoning. *Topoi*, 31(1):77–85, 2012.

[40] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A Concurrent Logical Framework I: Judgments and Properties. Tech. Rep. CMU-CS-02-101, CMU, 2002.