LF$^+$ in Coq for fast-and-loose reasoning

FABIO ALESSI, ALBERTO CIAFFAGLIONE, PIETRO DI GIANANTONIO, FURIO HONSELL, MARINA LENISA, and IVAN SCAGNETTO
Department of Mathematics, Computer Science and Physics, University of Udine[1], Italy

We develop the metatheory and the implementation, in Coq, of the novel logical framework LF$^+$ and discuss several of its applications. LF$^+$ generalises research work, carried out by the authors over more than a decade, on *Logical Frameworks* conservatively extending LF and featuring *lock*-type constructors $\mathcal{L}^{\mathcal{P}(N:\sigma)}[\![\cdot]\!]$. Lock-types capture monadically the concept of *inhabitability up-to*. They were originally introduced for *factoring-out, postponing*, or *delegating* to external tools the verification of time-consuming judgments, which are morally *proof-irrelevant*, thus allowing for integrating different sources of epistemic evidence in a unique Logical Framework. Besides introducing LF$^+$ and its "shallow" implementation in Coq, the main novelty of the paper is to show that lock-types are also a very flexible tool for expressing in Type Theory several diverse cognitive attitudes and mental strategies used in ordinary reasoning, which essentially amount to *reasoning up-to*, as in *e.g. Typical Ambiguity* provisos or *co-inductive Coq proofs*. In particular we address the encoding of the emerging paradigm of *fast-and-loose* reasoning, which trades off efficiency for correctness. This paradigm, implicitly used normally in *naïve* Set Theory, is producing considerable impact also in *computer architecture* and *distributed systems*, when *branch prediction* and *optimistic concurrency control* are implemented.

1. INTRODUCTION

The *Logical Framework* LF$^+$ is a conservative extension of LF. It builds on [HLMS17], but in fact it stems from a research line on conservative extensions of the *Edinburgh Logical Framework* [HHP93] started more than a decade ago by the authors, together with L. Liquori, P. Maksimović, and V. Michielini, see [HLL07, HLLS08, HLL$^+$12, Hon13, HLS14, HLMS15, HLS$^+$16, HLLS16, HLMS18]. These systems have been introduced prudentially and incrementally with two goals in mind:

— *integrating*, in a *unique* Logical Framework, different epistemic sources of evidence deriving from special-purpose tools and oracles, or even non-apodictic sources such as, *e.g.* explicit computations according to the Poincaré Principle [BB02], deduction up-to, diagrams, physical analogies;

— *factoring-out, postponing, delegating*, or *running in parallel* the verification of morally proof-irrelevant and time-consuming judgments and side conditions.

In the course of the research we came to the conclusion that all these issues could be dealt with conveniently using a notion of *lock-type constructor*, $\mathcal{L}^{\mathcal{P}(N:\sigma)}[\![\cdot]\!]$, which can be naturally construed as a monad, see Section 3.

Recent experiments (see [ACDG$^+$19b]) show that lock-types are a very flexible tool also for expressing in Type Theory a wide gamut of diverse mental strategies and cognitive attitudes used in ordinary reasoning, which generalise the above goals, and which can be generally referred to as *reasoning up-to*. One can say, indeed,

[1]Work supported by the Italian Departmental Research Project "LambdaBridge" (D.R.N. 37 427/2018 of 03/08/2018, University of Udine).

that the $\mathcal{L}^{\mathcal{P}(N:\sigma)}[\![\cdot]\!]$ type constructor expresses a very general notion of *inhabitability up-to*.

LF^+ extends the expressive power of the lock-type systems available of the previous frameworks, while streamlining and simplifying the rule-system and notation. The present paper includes an, almost self-contained, introduction to lock-types using the novel system LF^+. We discuss in detail the monadic nature of locks and give a rather complete and visionary overview of existing and possible applications of lock-types.

The main novelty of this paper is that of using LF^+ to address the formalisation of the emerging paradigm of *fast-and-loose* reasoning, [DHJG06]. This paradigm trades off efficiency for correctness and amounts to postponing, or running in parallel, tedious or computationally demanding checks, until we are really sure that the intended goal can be achieved. At logical level this paradigm amounts to the ordinary practice in everyday Mathematics based on *naïve* Set Theory or Category Theory, when potentially large objects are freely mentioned, see [HLLS16], or when conjecturing or introducing blanket assumptions to be checked or formalised later. The latter is often carried out also in program development, see *e.g.* [DHJG06, HLLS16]. At the level of implementations natural examples of this paradigm occur in *Computer Architecture* and *Concurrency Control*, *i.e. branch prediction* in CPUs [JKL00] and *optimistic concurrency control* in distributed systems [KR81]. In both cases efficiency is improved by forgetting, *i.e.* running in parallel, time-demanding tests which otherwise would significantly slow down the computation, if carried out sequentially. Of course in the event that the outcome of the test is negative there might be an extra cost for backtracking and restoring the original context. But the trade-off in speed when this does not occur compensates significantly this drawback.

Another important contribution of this paper is a prototype implementation of LF^+, which supports also mechanised proof search. This is crucial if we want to explore the above paradigms using a logical framework. The implementation is achieved through a careful *shallow* encoding of LF^+ in the `Coq` proof assistant. "Shallow" in this context means that we delegate as much as possible the mechanics of LF^+ to the metalanguage of the host system. Actually lock-types are rendered as a `Coq Definition`. This is quite interesting in itself, both in exposing the principles underpinning lock-types as well as for the bearing it has on proving that predicates are *well-behaved*. Of course, everything relies on an appropriate *adequacy result* (see Theorem 5.1).

In this paper we carry out three detailed case studies. The first is the paradigmatic *call-by-value* λ*-calculus*, the other two are the completely novel case studies pertaining to the fast-and-loose reasoning paradigm, namely *branch prediction* for URM machines (see [Cut80]) and *optimistic concurrency control* for a simple programming language, which we call *STL* (*i.e.* Simple Transaction Language) [CDK02].

Many intriguing issues arise in connection with the latter examples: *e.g.* can we short-cut reasoning on different nested locks? What is the correct way of expressing *adequacy* when dealing with reasoning up-to? How do we express the adequacy of a *branch prediction protocol*? We outline possible answers to some of these questions.

In Section 2 we introduce LF^+ and we discuss its differences w.r.t. $\mathsf{LLF}_{\mathcal{P}}$. In

Section 3 we explain the monadic nature of locks. In Section 4 we present a gamut of logical contexts and attitudes which have benefitted or would benefit by an analysis in terms of locks. In Section 5 we give a shallow definitional implementation in Coq of LF$^+$ and CLLF$_{\mathcal{P}?}$. The paradigmatic case-study of call-by-value λ-calculus is implemented in Section 6. Section 7 is a brief discussion of the fast-and-loose reasoning paradigm and serves as an introduction to the following two sections. In these we present two massive case-studies, namely: branch prediction for URM machines [Cut80], and optimistic concurrency control. We briefly discuss future directions in Section 10. The web appendix of the paper is online at [ACDG$^+$19b].

The authors express their gratitude to F. Rabe and to the anonymous referees of this paper and of a preliminary version presented at LFMTP 2019 [ACDG$^+$19a] for their helpful suggestions.

2. THE LF$^+$ LOGICAL FRAMEWORK

The reader familiar with [HLMS17] can read only Subsection 2.3, while the reader familiar with Type Theory in general can skip both Subsections 2.2 and 2.3.

2.1 The System and its Intuition

In this section, following the standard pattern and conventions of [HHP93], we introduce the syntax and the rules of LF$^+$. In Figure 1, we give the syntactic categories of LF$^+$, namely signatures, contexts, kinds, families (*i.e.* types), and objects (*i.e.* terms). The language is essentially that of classical LF [HHP93], to which we add the *lock-types* constructor ($\mathcal{L}$) for building types of the shape $\mathcal{L}^{\mathcal{C}}[\![\rho]\!]$, where $\mathcal{C} \equiv \mathcal{P}(U : V)$ is a predicate $\mathcal{P}$ on the entity U classified by V, which can either be a term M and its type σ, or a type σ and its kind K. The predicate $\mathcal{P}$ can, and usually will, access the whole environment Γ. Correspondingly, at the object level, we introduce the lock *constructor* ($\mathcal{L}$) and the unlock *destructor* ($\mathcal{U}$). The intended meaning of the $\mathcal{L}^{\mathcal{P}(U:V)}[\![\cdot]\!]$ constructor is that of a *logical filter* expressing inhabitability up-to the verification of $\Gamma \vdash \mathcal{P}(U{:}V)$.

Notice that the syntactic category of *lock conditions* in Figure 1 is merely an abbreviation, in order to obtain a smoother and lighter notation in the rest of the paper (in particular, in the type rules of Figure 8). Hence, in the following we will switch from $\mathcal{C}$ to $\mathcal{P}(U : V)$ or to the specific subcases $P(M : \sigma)$, $\mathcal{P}(\sigma : K)$, according to our needs.

The rules for the main one-step $\beta\mathcal{L}$-reduction, which combines the standard β-reduction with the novel $\mathcal{L}$-reduction (behaving as a lock-releasing mechanism, erasing the $\mathcal{U}$-$\mathcal{L}$ pair in a term of the form $\mathcal{U}[\![\mathcal{L}^{\mathcal{C}}[\![M]\!]]\!]$) appear in Figure 2. The rules for one-step closure under context for kinds, families, objects, and lock conditions are collected in Figures 3, 4, 5, 6 respectively. 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, objects, lock conditions, as illustrated in Figure 7.

Following the standard specification paradigm of Constructive Type Theory, we define lock-types using *formation*, *introduction*, *elimination*, and *equality rules*. Namely, see Figure 8, we introduce a lock-*constructor* for building objects $\mathcal{L}^{\mathcal{C}}[\![M]\!]$ of type $\mathcal{L}^{\mathcal{C}}[\![\rho]\!]$, via the *introduction rule* ($O{\cdot}Lock$). Correspondingly, we introduce

$$
\begin{array}{rcll}
\Sigma & \in & \textit{Signatures} & \Sigma ::= \emptyset \mid \Sigma, a{:}K \mid \Sigma, c{:}\sigma \\
\Gamma & \in & \textit{Contexts} & \Gamma ::= \emptyset \mid \Gamma, x{:}\sigma \\
K & \in & \textit{Kinds} & K ::= \mathsf{Type} \mid \Pi x{:}\sigma.K \\
\sigma, \tau, \rho & \in & \textit{Families (Types)} & \sigma ::= a \mid \Pi x{:}\sigma.\tau \mid \sigma\ N \mid \mathcal{L}^{\mathcal{C}}[\![\rho]\!] \\
M, N & \in & \textit{Objects} & M ::= c \mid x \mid \lambda x{:}\sigma.M \mid M\ N \mid \mathcal{L}^{\mathcal{C}}[\![M]\!] \mid \mathcal{U}[\![M]\!] \\
\mathcal{C} & \in & \textit{Lock Conditions} & \mathcal{C} ::= \mathcal{P}(N : \sigma) \mid \mathcal{P}(\sigma : K) \\
 & & & \text{where the } \mathcal{P} \text{ denotes an external predicate symbol} \\
 & & & \text{from a fixed set of identifiers.}
\end{array}
$$

Fig. 1. The pseudo-syntax of LF^+

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

Fig. 2. Main one-step-$\beta\mathcal{L}$-reduction rules

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

Fig. 3. $\beta\mathcal{L}$-closure-under-context for kinds

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

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

$$\frac{\mathcal{C} \to_{\beta\mathcal{L}} \mathcal{C}'}{\mathcal{L}^{\mathcal{C}}[\![\rho]\!] \to_{\beta\mathcal{L}} \mathcal{L}^{\mathcal{C}'}[\![\rho]\!]}\ (F{\cdot}\mathcal{L}_1{\cdot}\beta\mathcal{L}) \qquad \frac{\rho \to_{\beta\mathcal{L}} \rho'}{\mathcal{L}^{\mathcal{C}}[\![\rho]\!] \to_{\beta\mathcal{L}} \mathcal{L}^{\mathcal{C}}[\![\rho']\!]}\ (F{\cdot}\mathcal{L}_2{\cdot}\beta\mathcal{L})$$

Fig. 4. $\beta\mathcal{L}$-closure-under-context for families

an unlock-*destructor* $\mathcal{U}[\![M]\!]$ via the *elimination rule* $(O{\cdot}Guarded{\cdot}Unlock)$, which is reminiscent in its shape of a Gentzen-style *left-introduction* rule.

In order to provide the intended meaning of $\mathcal{L}^{\mathcal{C}}[\![\cdot]\!]$, we need to introduce in LF^+ also the rule $(O{\cdot}Top{\cdot}Unlock)$, which allows for the elimination of the lock-type constructor if the condition $\mathcal{C}$ is verified, possibly *externally*. Figure 8 shows the full type system of LF^+. All *type equality rules* of LF^+ use as notion of conversion $\beta\mathcal{L}$-definitional equality.

The reader familiar with previous notations for lock-types may notice that we introduced a simplification in the notation of the unlock operator ($\mathcal{U}$), dropping the predicate, since the latter is automatically implied by the type of the term argument M.

One may wonder why the rule $(O{\cdot}Top{\cdot}Unlock)$ is not enough and $(F{\cdot}Guarded{\cdot})$ and $(O{\cdot}Guarded{\cdot})$ unlock-rules are called for. In Section 3, we will show that these two rules are crucial for providing the *monadic interpretation* of locks, while $(O{\cdot}Top{\cdot}Unlock)$ is merely a non-standard rule for exiting our monads. But the

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

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

$$\frac{\mathcal{C} \to_{\beta\mathcal{L}} \mathcal{C}'}{\mathcal{L}^{\mathcal{C}}[\![M]\!] \to_{\beta\mathcal{L}} \mathcal{L}^{\mathcal{C}'}[\![M]\!]}\ (O{\cdot}\mathcal{L}_1{\cdot}\beta\mathcal{L}) \qquad \frac{M \to_{\beta\mathcal{L}} M'}{\mathcal{L}^{\mathcal{C}}[\![M]\!] \to_{\beta\mathcal{L}} \mathcal{L}^{\mathcal{C}}[\![M']\!]}\ (O{\cdot}\mathcal{L}_2{\cdot}\beta\mathcal{L})$$

$$\frac{M \to_{\beta\mathcal{L}} M'}{\mathcal{U}[\![M]\!] \to_{\beta\mathcal{L}} \mathcal{U}[\![M']\!]}\ (O{\cdot}\mathcal{U}{\cdot}\beta\mathcal{L})$$

Fig. 5. $\beta\mathcal{L}$-closure-under-context for objects

$$\frac{M \to_{\beta\mathcal{L}} M'}{\mathcal{P}(M:\sigma) \to_{\beta\mathcal{L}} \mathcal{P}(M':\sigma)}\ (O{\cdot}\mathcal{C}_1{\cdot}\beta\mathcal{L}) \qquad \frac{\sigma \to_{\beta\mathcal{L}} \sigma'}{\mathcal{P}(M:\sigma) \to_{\beta\mathcal{L}} \mathcal{P}(M:\sigma')}\ (O{\cdot}\mathcal{C}_2{\cdot}\beta\mathcal{L})$$

$$\frac{\sigma \to_{\beta\mathcal{L}} \sigma'}{\mathcal{P}(\sigma:K) \to_{\beta\mathcal{L}} \mathcal{P}(\sigma':K)}\ (O{\cdot}\mathcal{C}_3{\cdot}\beta\mathcal{L}) \qquad \frac{K \to_{\beta\mathcal{L}} K'}{\mathcal{P}(\sigma:K) \to_{\beta\mathcal{L}} \mathcal{P}(\sigma:K')}\ (O{\cdot}\mathcal{C}_4{\cdot}\beta\mathcal{L})$$

Fig. 6. $\beta\mathcal{L}$-closure-under-context for lock conditions

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

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

Fig. 7. $\beta\mathcal{L}$-definitional equality

guarded-unlock rules are crucial also pragmatically. It is precisely them that make it possible to reason hypothetically, *i.e.* to concatenate reasoning steps under assumptions, thus making locks flexible for modeling the fast-and-loose reasoning paradigm. Linking deduction steps under assumptions is actually a kind of Kleisli composition. If we were to always release a locked term before accessing the encapsulated judgement this would mean that we would always have to check a proof-irrelevant side condition, before using that judgement. And this is precisely what would slow down the main flow of a derivation. Properties occurring in locks are usually not essential to the main thrust of the proof, because they are usually *proof-irrelevant*. The hard stuff goes on within the lock. In practice, one wants to be free to proceed with the main argument, postponing, as much as possible, the verification of details. In reaching in a proof development a stage, where we are

Signature rules

$$\frac{}{\emptyset\ \mathsf{sig}}\ (S{\cdot}Empty) \qquad \frac{\Gamma \vdash_\Sigma \sigma : \Pi x{:}\tau.K \quad \Gamma \vdash_\Sigma N : \tau}{\Gamma \vdash_\Sigma \sigma\, N : K[N/x]}\ (F{\cdot}App)$$

$$\frac{\vdash_\Sigma K \quad a \notin \mathsf{Dom}(\Sigma)}{\Sigma, a{:}K\ \mathsf{sig}}\ (S{\cdot}Kind) \qquad \frac{\Gamma \vdash_\Sigma \rho : \mathsf{Type} \quad \Gamma \vdash_\Sigma U : V \quad \mathcal{C} \equiv \mathcal{P}(U:V)}{\Gamma \vdash_\Sigma \mathcal{L}^{\mathcal{C}}[\![\rho]\!] : \mathsf{Type}}\ (F{\cdot}Lock)$$

$$\frac{\vdash_\Sigma \sigma{:}\mathsf{Type} \quad c \notin \mathsf{Dom}(\Sigma)}{\Sigma, c{:}\sigma\ \mathsf{sig}}\ (S{\cdot}Type) \qquad \frac{\Gamma \vdash_\Sigma \sigma : K \quad \Gamma \vdash_\Sigma K' \quad K =_{\beta\mathcal{L}} K'}{\Gamma \vdash_\Sigma \sigma : K'}\ (F{\cdot}Conv)$$

Context rules

$$\frac{\Sigma\ \mathsf{sig}}{\vdash_\Sigma \emptyset}\ (C{\cdot}Empty)$$

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

Kind rules

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \mathsf{Type}}\ (K{\cdot}Type)$$

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

Family rules

$$\frac{\vdash_\Sigma \Gamma \quad a{:}K \in \Sigma}{\Gamma \vdash_\Sigma a : K}\ (F{\cdot}Const)$$

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

Object rules

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

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

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma M : \tau}{\Gamma \vdash_\Sigma \lambda x{:}\sigma.M : \Pi x{:}\sigma.\tau}\ (O{\cdot}Abs)$$

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

$$\frac{\Gamma \vdash_\Sigma M : \sigma \quad \Gamma \vdash_\Sigma \tau : \mathsf{Type} \quad \sigma =_{\beta\mathcal{L}} \tau}{\Gamma \vdash_\Sigma M : \tau}\ (O{\cdot}Conv)$$

$$\frac{\Gamma \vdash_\Sigma M : \rho \quad \Gamma \vdash_\Sigma U : V \quad \mathcal{C} \equiv \mathcal{P}(U:V)}{\Gamma \vdash_\Sigma \mathcal{L}^{\mathcal{C}}[\![M]\!] : \mathcal{L}^{\mathcal{C}}[\![\rho]\!]}\ (O{\cdot}Lock)$$

$$\frac{\begin{array}{c}\Gamma, x : \tau \vdash_\Sigma M : \rho \quad \Gamma \vdash_\Sigma N : \mathcal{L}^{\mathcal{C}}[\![\tau]\!] \\ \mathcal{P}(\Gamma \vdash_\Sigma U : V) \quad C \equiv \mathcal{P}(U:V) \quad C =_{\beta\mathcal{L}} C'\end{array}}{\Gamma \vdash_\Sigma M[\mathcal{U}[\![N]\!]/x] : \rho[\mathcal{U}[\![N]\!]/x]}\ (O{\cdot}Top{\cdot}Unlock)$$

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

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

Fig. 8. The LF^+ Type System

not able, or we do not want to waste time, to verify a side-condition, we want to *postpone* such a task and use immediately, albeit in a nested way, the given term, thus proceeding with the main argument of the proof. The $(F{\cdot}Guarded{\cdot}Unlock)$ and the $(O{\cdot}Guarded{\cdot}Unlock)$ rules allows us to realise precisely this. The external lock-type of the type within which we release the unlocked term will preserve safety, keeping track that the verification has to be carried out at least once, sooner or later.

We conclude this section by recalling that, since external predicates $\mathcal{P}$ affect reductions in LF^+, they must be *well-behaved* in order to preserve subject reduction. This property is necessary for achieving *decidability*, relative to an oracle, which is

essential to any proof-checker such as LF$^+$. We introduce, therefore, the following crucial definition, where α is shorthand for the conclusion of a judgment.

Definition 2.1 Well-behaved predicates, [HLS$^+$16]. 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 $\mathcal{P}(\Gamma, x{:}V', \Gamma' \vdash_\Sigma U : V)$ and $\Gamma \vdash_\Sigma U' : V'$, then $\mathcal{P}(\Gamma, \Gamma'[U'/x] \vdash_\Sigma U[U'/x] : V[U'/x])$.

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

2.2 The metatheory of LF$^+$

For the sake of completeness, we will briefly sketch the metatheory of LF$^+$. There are two ways to prove the standard properties (*e.g.* strong normalisation, confluence and subject reduction): either directly or exploiting a *compositional embedding function* mapping entities and judgments of LF$^+$ into another principled logical framework, *e.g.* LF. This approach has been pursued in [HLMS17] for LLF$_\mathcal{P}$. In this section we simply notice that, since in Section 5 we give a *shallow* and *adequate* encoding of LF$^+$ in Coq, we inherit all the main metatheoretic properties from the Calculus of (Co)Inductive Constructions (the underlying type theory of Coq). This argument is robust enough to support also all the extensions which we make in LF$^+$ to the original theory of LLF$_\mathcal{P}$.

Theorem 2.2 (Confluence of LF$^+$). $\beta\mathcal{L}$-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 $M'' \twoheadrightarrow_{\beta\mathcal{L}} M'''$.

We recall here that, as it is often the case for systems without η-like conversions, confluence can alternatively be proved directly on *raw terms* as in [HHP93, HLS$^+$16]. Namely using *Newman's Lemma* ([Bar84], Chapter 3), and showing that the reduction on raw terms is *locally confluent*.

Theorem 2.3 (Strong normalisation of LF$^+$). (1) If $\Gamma \vdash_\Sigma K$, then K is $\to_{\beta\mathcal{L}}$-strongly normalising.

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

(3) if $\Gamma \vdash_\Sigma M : \sigma$, then M is $\to_{\beta\mathcal{L}}$-strongly normalising.

Alternatively, strong normalisation can be proved following the same pattern used in [HLS+16], relying on the strong normalisation of LF, as proven in [HHP93].

Well-behavedness of predicates is not necessary when we prove metatheoretic results directly on raw terms, as in the theorems above. On the other hand, it is necessary when we prove metatheoretic results about typings, as in the following theorems.

If the predicates are well-behaved, we have also the following:

Theorem 2.4 (Subject reduction of LF^+). 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 $\Gamma \vdash_\Sigma M' : \sigma$.

Since we implement predicates in Coq, all the predicates in this paper are well-behaved. Other useful standard metatheoretic results are the following ones:

Proposition 2.5 (Weakening and permutation). If predicates are closed under signature/context weakening and permutation, then:

(1) If Σ and Ω are valid signatures, and $\Sigma \subseteq \Omega$, 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 $\Gamma \subseteq \Delta$, then $\Gamma \vdash_\Sigma \alpha$ implies $\Delta \vdash_\Sigma \alpha$.

Proposition 2.6 (Substitution). 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 $\Gamma, \Gamma'[N/x] \vdash_\Sigma \alpha[N/x]$.

2.3 Comparison with the system $\mathsf{LLF}_\mathcal{P}$

A more detailed comparison of LF^+ with the earlier system $\mathsf{LLF}_\mathcal{P}$, of [HLMS17], is in order.

— In the language definition of LF^+ we have introduced the new syntactic class $\mathcal{C}$ of *lock conditions*. This is essentially a definitional modification which simplifies the notations and the rules.
— We have dropped the reference to $\mathcal{P}(U : V)$ in *unlock* constructors. This is safe since the index can be directly recovered from the type of argument to the unlock.
— The judgements considered in predicates are extended to families and kinds, *e.g.* $\mathcal{P}(\sigma, Type)$, thus permitting us to consider *inhabitability* of types as a predicate.
— The first premise of the rule ($O{\cdot}Guarded{\cdot}Unlock$) is relaxed in that the subject of that premise no longer needs to be explicitly locked. Safety is preserved because the type has an external lock. We recall that, if subject reduction is to hold, in this rule some extra complexity is needed to take care of the parameters occurring in the predicate.
— Rule ($O{\cdot}Top{\cdot}Unlock$) is rephrased so as to make it similar in shape to the rule ($O{\cdot}Guarded{\cdot}Unlock$).

3. THE MONADIC NATURE OF LF$^+$

In this Section we illustrate in full detail how locks can be understood as monads. This Section can be skipped by readers only interested in the applications.

The type rules of the $\mathcal{L}$ and $\mathcal{U}$ constructors allow for an understanding of $\mathcal{L}^{\mathcal{C}}[\![\cdot]\!]$ as a *monad*. Domain-theoretically this monad behaves as the Identity Monad, creating a copy of all objects. This *look-alike* or *doppelgänger* monad creates a double, parallel, hypothetical, possibly counterfactual world.

The introduction rules ($F{\cdot}Lock$) and ($O{\cdot}Lock$) of lock-types and lock-terms correspond immediately to the introduction rules of monadic types and monadic terms in the *Metalanguage* of [Mog91].

The elimination rules, on the other hand, require more discussion.

— The ($O{\cdot}Top{\cdot}Unlock$) rule is specific to this monad. In general, there is no standard way to exit the encapsulation produced by a monad.

— Eliminations need to be performed also at the level of families by *(F·Guarded·Unlock)*, but we do not need explicit unlock-operators at that level.

— Both rules *(F·Guarded·Unlock)* and *(O·Guarded·Unlock)* need to take into account also *definitional equality* to preserve subject reduction.

— We do not use the standard $\mathtt{let}_T$ construct as elimination construct, in favour of the $\mathcal{U}$-constructor. We could have introduced it, *e.g.*

$$\frac{\Gamma, x : \tau \vdash_\Sigma \mathcal{L}^{\mathcal{C}}[\![\rho]\!] : \mathsf{Type} \quad \Gamma \vdash_\Sigma N : \mathcal{L}^{\mathcal{C}'}[\![\tau]\!] \quad C =_{\beta\mathcal{L}} C'}{\Gamma \vdash_\Sigma \mathtt{let}\ x = N\ \mathtt{in}\ \mathcal{L}^{\mathcal{C}'}[\![\rho]\!] : \mathsf{Type}.}\ (F{\cdot}Guarded{\cdot}Unlock)'$$

But the use of `let` would introduce unnecessary terms in our language and make the ($O{\cdot}Top{\cdot}Unlock$) rule more awkward to express.

— We can define in LF$^+$the function $\mathbf{let}_{\mathcal{C}} : (\sigma \to \mathcal{L}^{\mathcal{C}}[\![\tau]\!]) \to \mathcal{L}^{\mathcal{C}}[\![\sigma]\!] \to \mathcal{L}^{\mathcal{C}}[\![\tau]\!]$ by

$$\lambda x : \sigma \to \mathcal{L}^{\mathcal{C}}[\![\tau]\!].\ \lambda y : \mathcal{L}^{\mathcal{C}}[\![\sigma]\!].\ x(\mathcal{U}[\![y]\!]) : (\sigma \to \mathcal{L}^{\mathcal{C}}[\![\tau]\!]) \to \mathcal{L}^{\mathcal{C}}[\![\sigma]\!] \to \mathcal{L}^{\mathcal{C}}[\![\tau]\!].$$

It evaluates as the traditional $\mathtt{let}_T$ construct apart from the case when the first argument is a constant function. Erasing the dependency is wrong if we use the $\mathtt{let}_T$ construct to model Kleisli composition in the general setting. But we are essentially interested in *inhabitability of types*, rather than encapsulating evaluations, and such judgements do not produce effects. In our context the case of the constant functions amounts simply to an instance of the *strengthening* structural rule.

In order to define formally the monad induced by the $\mathcal{L}^{\mathcal{P}(U:V)}[\![\cdot]\!]$ construct, for a given a predicate $\mathcal{P}$ and a judgement $\Gamma \vdash_\Sigma U : V$, we need to endow the term model of LF$^+$ with the structure of a category. In fact this should be done in the more general setting of *categories with families* [Dyb96], due to the fact that we use dependent types. We will not develop all the details of this construction here, but give only the crucial insights.

The objects and morphisms of our category will be classes of LF$^+$ terms up to expansion to $\beta\eta$-lnf. The notion of $\beta\eta$-lnf is given by the following definitions (see Definition 5.8 and Definition 5.9 in [HLS$^+$16]):

Definition 3.1. An occurrence ξ of a constant or a variable in a term of a LF^+ judgement is *fully applied and unlocked* with respect to its type or kind $\Pi\vec{x}_1{:}\vec{\sigma}_1.\vec{\mathcal{L}}_1[\ldots\Pi\vec{x}_n{:}\vec{\sigma}_n.\vec{\mathcal{L}}_n[\alpha]\ldots]$, where $\vec{\mathcal{L}}_1,\ldots,\vec{\mathcal{L}}_n$ are vectors of locks, if ξ appears in subobjects of the form $\vec{\mathcal{U}}_n[(\ldots(\vec{\mathcal{U}}_1[\xi\vec{M}_1])\ldots)\vec{M}_n]$, where $\vec{M}_1,\ldots,\vec{M}_n$, $\vec{\mathcal{U}}_1,\ldots,\vec{\mathcal{U}}_n$ have the same arities of the corresponding vectors of Π's and locks.

Definition 3.2 (Judgements in $\beta\eta$-long normal form).

(1) A term T in a judgement is in *$\beta\eta$-lnf* if T is in normal form and every constant and variable occurrence in T is fully applied and unlocked w.r.t. its classifier in the judgement.
(2) A judgement is in *$\beta\eta$-lnf* if all terms appearing in it are in $\beta\eta$-lnf.

For the rest of this section, by *semantics of an object* in the term model we will mean implicitly its $\beta\eta$-lnf. In order to avoid an excessive burden in the notation, we will define the monad $(\mathcal{L}^{\mathcal{C}}[\![\cdot]\!],\eta,\mu)$ by giving the corresponding concrete instances of the natural transformations η and μ as $\beta\eta$-lnf-classes of terms in LF^+. By abuse of notation, we will use the same symbols both on the syntactic level and on the semantic level.

In particular, we have that $\eta_\rho \triangleq \lambda x{:}\rho.\mathcal{L}^{\mathcal{C}}[\![x]\!]$ and $\mu_\rho \triangleq \lambda x{:}\mathcal{L}^{\mathcal{C}}[\![\mathcal{L}^{\mathcal{P}(U:V)}[\![\rho]\!]]\!].\,\mathcal{U}[\![x]\!]$. Indeed, if $\mathcal{C}\equiv\mathcal{P}(U:V)$ and $\Gamma,x{:}\rho\vdash_\Sigma U:V$ 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:

$$\dfrac{\dfrac{\Gamma,x{:}\rho\vdash_\Sigma x:\rho \quad \Gamma,x{:}\rho\vdash_\Sigma U:V \quad \mathcal{C}\equiv\mathcal{P}(U:V)}{\Gamma,x{:}\rho\vdash_\Sigma \mathcal{L}^{\mathcal{C}}[\![x]\!]:\mathcal{L}^{\mathcal{C}}[\![\rho]\!]}}{\Gamma\vdash_\Sigma \lambda x{:}\rho.\mathcal{L}^{\mathcal{C}}[\![x]\!]:\Pi x{:}\rho.\mathcal{L}^{\mathcal{C}}[\![\rho]\!]}$$

The term for μ, if $\Gamma\vdash_\Sigma U:V$ is derivable, is obtained by applying weakening and the rules $(O{\cdot}Var)$ and $(O{\cdot}Guarded{\cdot}Unlock)$ and finally $(O{\cdot}Abs)$:

$$\dfrac{\dfrac{\Gamma,x{:}\mathcal{L}^{\mathcal{C}}[\![\mathcal{L}^{\mathcal{C}}[\![\rho]\!]]\!],z{:}\mathcal{L}^{\mathcal{C}}[\![\rho]\!]\vdash z{:}\mathcal{L}^{\mathcal{C}}[\![\rho]\!] \qquad \Gamma,x{:}\mathcal{L}^{\mathcal{C}}[\![\mathcal{L}^{\mathcal{C}}[\![\rho]\!]]\!]\vdash_\Sigma x{:}\mathcal{L}^{\mathcal{C}}[\![\mathcal{L}^{\mathcal{C}}[\![\rho]\!]]\!]}{\Gamma,x{:}\mathcal{L}^{\mathcal{C}}[\![\mathcal{L}^{\mathcal{C}}[\![\rho]\!]]\!]\vdash_\Sigma \mathcal{U}[\![x]\!]:\mathcal{L}^{\mathcal{C}}[\![\rho]\!]}}{\Gamma\vdash \lambda x{:}\mathcal{L}^{\mathcal{C}}[\![\mathcal{L}^{\mathcal{C}}[\![\rho]\!]]\!].\,\mathcal{U}[\![x]\!].}$$

Locks yield *strong monads*, whose *tensorial strength* is given, in our context, by a function:

$$t_{\sigma,\tau}:\Pi x{:}\sigma.\Pi y{:}\mathcal{L}^{\mathcal{C}}[\![\tau]\!].\mathcal{L}^{\mathcal{C}}[\![\rho]\!]\longrightarrow\Pi x{:}\mathcal{L}^{\mathcal{C}}[\![\sigma]\!].\Pi y{:}\mathcal{L}^{\mathcal{C}}[\![\tau]\!].\mathcal{L}^{\mathcal{C}}[\![\rho]\!]$$

namely:

$$t_{\sigma,\tau}\triangleq\ \lambda f:(\Pi x{:}\sigma.\Pi y{:}\mathcal{L}^{\mathcal{C}}[\![\tau]\!].\mathcal{L}^{\mathcal{C}}[\![\rho]\!]).\lambda x:(\mathcal{L}^{\mathcal{C}}[\![\sigma]\!]).\lambda y{:}(\mathcal{L}^{\mathcal{C}}[\![\tau]\!]).f(\mathcal{U}[\![x]\!])y.$$

We are left to check that the monad equalities hold. This we will do by showing that the *unity* and *associativity* axioms of the *Kleisli triple* $(T,\eta,-^*)$, corresponding to the monad $\mathcal{L}^{\mathcal{C}}[\![\cdot]\!]$, hold on the corresponding classes of the term model. Recall that (see, *e.g.*, [Mog91] for details), given a monad (T,η,μ), we can consider the corresponding Kleisli triple $(T,\eta,-^*)$, *i.e.* the category of computations induced by the endofunctor $T:\mathcal{C}\longrightarrow\mathcal{C}$. In our case the endofunctor is parameterised by a predicate $\mathcal{P}$ and a type judgment $\Gamma\vdash_\Sigma U:V$. The natural transformation $-^*$ is

defined on $f : \sigma \longrightarrow \mathcal{L}^{\mathcal{C}}[\![\tau]\!]$ by:

$$f^* : \mathcal{L}^{\mathcal{C}}[\![\sigma]\!] \longrightarrow \mathcal{L}^{\mathcal{C}}[\![\tau]\!]$$

where:

$$f^* \triangleq \lambda x : \mathcal{L}^{\mathcal{C}}[\![\sigma]\!].f(\mathcal{U}[\![x]\!]).$$

The unity and associativity axioms for the Kleisli category are the following:

(1) $f; \eta^*_\tau = f$ for $f : \sigma \longrightarrow \mathcal{L}^{\mathcal{C}}[\![\tau]\!]$
(2) $\eta_\sigma; f^* = f$ for $f : \sigma \longrightarrow \mathcal{L}^{\mathcal{C}}[\![\tau]\!]$
(3) $(f; g^*); h^* = f; (g; h^*)^*$ for $f : \sigma \longrightarrow \mathcal{L}^{\mathcal{C}}[\![\tau]\!]$, $g : \tau \longrightarrow \mathcal{L}^{\mathcal{C}}[\![\rho]\!]$, and $h : \rho \longrightarrow \mathcal{L}^{\mathcal{C}}[\![\xi]\!]$

These axioms hold in the term model of LF$^+$. Indeed, for point 1), we have that $f; \eta^*_\tau = \lambda x{:}\sigma.\mathcal{L}^{\mathcal{C}}[\![\mathcal{U}[\![f(x)]\!]]\!]$ which is the $\beta\eta$-lnf form of f (whose type is $\sigma \longrightarrow \mathcal{L}^{\mathcal{C}}[\![\tau]\!]$).

Similarly, for point 2), we have that $\eta_\sigma; f^* = \lambda x{:}\sigma.f(\mathcal{U}[\![\mathcal{L}^{\mathcal{C}}[\![x]\!]]\!])$, which, according to $(\mathcal{L}{\cdot}O{\cdot}Main)$, reduces to $\lambda x{:}\sigma.f(x)$, which in turn corresponds to $\lambda x{:}\sigma.\mathcal{L}^{\mathcal{C}}[\![\mathcal{U}[\![f(x)]\!]]\!]$ in our term model, *i.e.* the $\beta\eta$-lnf of f.

The verification of point 3) is simpler, since both sides of the equation are equal to $\lambda x{:}\sigma.h(\mathcal{U}[\![g(\mathcal{U}[\![f(x)]\!])]\!])$, which corresponds in our term model to the $\beta\eta$-lnf form $\lambda x{:}\sigma.\mathcal{L}^{\mathcal{C}}[\![\mathcal{U}[\![h(\mathcal{U}[\![g(\mathcal{U}[\![f(x)]\!])]\!])]\!]]\!]$.

For the sake of simplicity, in all the above cases we considered σ, τ, and ρ as constant types, otherwise we would have to further unravel their structures to comply with the notion of $\beta\eta$-lnf.

It is worth noting that all the previous Kleisli axioms would be for free if the term model were quotiented by the notion of congruence induced by $\beta\mathcal{L}$-reduction, standard η-reduction, and the following notion of $(\mathcal{L}{\cdot}O{\cdot}\eta)$-reduction:

$$\mathcal{L}^{\mathcal{C}}[\![\mathcal{U}[\![M]\!]]\!] \to M \qquad (\mathcal{L}{\cdot}O{\cdot}\eta)$$

The reason why we do not add η-rules it that those rules would make the language theory of LF$^+$ more difficult, because CR does not hold on raw terms, while η-expansion is always safe. Moreover, adequacy statements need only to mention $\beta\eta$-long normal forms [HHP93, AHMP92].

Finally, to provide the intended meaning of $\mathcal{L}^{\mathcal{P}(U:V)}[\![\cdot]\!]$, we need to introduce in LF$^+$ 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. Thus, it amounts to the possibility of "exiting" from the monad, if certain special conditions are fulfilled. We will come back to this issue in Section 7.

Concluding the monadic account of *locks*, we remark the moral of this section: *reasoning under assumptions*, via the *Guarded*$\cdot$*Unlock* rules, essentially corresponds to *Kleisli composition*.

4. A REPERTOIRE OF LOCKS

This section is a cursory review of existing and potential applications of locks and can be skipped without losing the main flow of the paper.

In the following we list various reasoning logical systems and proof strategies where locks have been, or could be, used fruitfully to provide a faithful and transparent encoding in a Logical Framework. Implicitly this will show also that such reasoning attitudes can be viewed as monads. Namely the concatenation of deduction steps in these patterns essentially amounts to carrying out Kleisli composition.

A first kind of examples where locks come in handy arises when we want to factor out, possibly proof-irrelevant, *side-conditions*. The only critical issue is to ensure that the predicates involved in locks are well-behaved. Here is a non-exhaustive list:

— *modal* logics. The syntactic side condition is that a proof term is either *closed* w.r.t. variables whose type is an object-logic judgment, or the free variables of object-logic judgement type occur in subterms of a particular shape, [HLL$^+$12, HLS$^+$16];

— *substructural* logics: *e.g. affine elementary linear* logic, *non-commutative linear logic*. The side condition is that object-logic judgment typed variables are constrained appropriately in proof terms, [HLL$^+$12, Hon13, HLS14, HLMS15, HLS$^+$16, HLMS18];

— *Hoare's* logic. *Quantifier-free* formulæ, and *non-interference* predicates are defined using syntactic constraints on formulæ, [HLL$^+$12, Hon13, HLS14, HLMS15, HLS$^+$16, HLMS17, HLMS18];

— *Fitch-Prawitz* Set Theory. The side condition is that proof terms are *normalisable* [HLMS15, HLLS16, HLMS17, HLMS18]. We elaborate more on this in Section 7.

Proof attitudes and concepts which can be naturally represented in LF^+ using lock-types, and hence be construed as monads, are for example:

— systems which separate *deduction* and *computation*. For instance in *Deduction Modulo*, the implication-up-to rule:

$$\frac{C \qquad A \supset B \qquad A \equiv C}{B}$$

can be expressed as:

$$\Pi A, B, C : o.True(A \supset B) \to True(C) \to \mathcal{L}^{\mathcal{P}(A \equiv C : o)}[\![True(B)]\!];$$

— systems which support reasoning and programming *up-to equivalence relations*, or *up-to computations* according to *Poincaré*'s principle, see [BB02], where terms are taken to be *definitionally equal* even if only computationally equal;

— the logical ambient in [DHJG06], where the very paradigm of fast-and-loose reasoning was coined in order to reason under the assumption that objects are *total*;

— the *squash, bracket, or (-1)-truncation* type constructor, (see [Uni13], page 152) can be expressed using locks by taking $||\sigma|| \triangleq \mathcal{L}^{\mathcal{I}nh(\sigma : Type)}[\![\sigma]\!]$, where $\mathcal{I}nh$ is the inhabitability predicate. Using guarded-unlock rules, the introduction rule, and both the recursive and inductive elimination rules for squash types (see [Uni13],

ex.3.17) can be derived in LF$^+$:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathcal{L}^{\mathcal{I}nh(\sigma:Type)}[\![M]\!] : \mathcal{L}^{\mathcal{I}nh(\sigma:Type)}[\![\sigma]\!]}\ Intro$$

$$\frac{\Gamma, x : \sigma \vdash M\ : \tau \qquad \text{all occurrences of } x \text{ in } \tau \text{ appear in a } \mathcal{L}^{\mathcal{I}nh(\sigma:Type)}[\![\cdot]\!] \text{ context}}{\Gamma, x : \mathcal{L}^{\mathcal{I}nh(\sigma:Type)}[\![\sigma]\!] \vdash M : \tau}\ Rec{-}elim$$

$$\frac{\begin{array}{l}\Gamma, x : \mathcal{L}^{\mathcal{I}nh(\sigma:Type)}[\![\sigma]\!] \vdash \rho : Type \\ \Gamma, y : \sigma \vdash N : \rho(\mathcal{L}^{\mathcal{I}(\sigma:Type)}[\![y]\!]) \qquad \text{all occurrences of } y \text{ in } N \text{ appear within a } \mathcal{L}^{\mathcal{I}nh(\sigma:Type)}[\![\cdot]\!]\end{array}}{\Gamma, x : \mathcal{L}^{\mathcal{I}(\sigma:Type)}[\![\sigma]\!] \vdash N[\mathcal{U}[\![\sigma]\!]/y] : \rho x}\ Ind\ .$$

In this context the condition that "$\tau(x)$ is a *mere proposition*" in [Uni13] implies that all occurrences of x in τ appear in a $\mathcal{L}^{\mathcal{I}nh(\sigma:Type)}[\![\cdot]\!]$ context;

— *generalised propositions* such as $x \neq 0 \supset x^{-1} \neq 0$ were explained by Martin-Löf in [MLS84] using the following rule for $A \supset B$:

$$\frac{\begin{array}{ll} & A\ \mathbf{True} \\ A\ \mathbf{Prop} & B\ \mathbf{Prop}\end{array}}{A \supset B\ \mathbf{Prop}}$$

Using locks in LF$^+$ we can express this by:

$$\frac{A\ \ \mathbf{Prop} \qquad \mathcal{L}^{\mathcal{T}rue(A:\mathbf{Prop})}[\![B\ \mathbf{Prop}]\!]}{A \supset B\ \mathbf{Prop}}\ .$$

— Finally *typical ambiguity* as tacitly assumed in [Uni13], *i.e.* $\Phi(\mathcal{U} \in \mathcal{U})$, can be expressed using locks in LF$^+$ by:

$$\mathcal{L}^{\mathcal{S}tratifiable(\Phi:Type)}[\![\Phi]\!]$$

5. A DEFINITIONAL IMPLEMENTATION OF LF$^+$ IN COQ

An implementation of the logical framework LLF$_{\mathcal{P}}$, from scratch, in a functional language has been attempted successfully, as far as *proof checking*, by Vincent Michielini at ENS Lyon [Mic16]. Florian Rabe has given an implementation of LLF$_{\mathcal{P}}$ in the meta-metalanguage MMT [MR19]. However, extending any of the two to a full-fledged *proof development environment* for LF$^+$ would be a major task. To achieve this we capitalise on the existing proof-assistant Coq, [Coq18].

5.1 The Coq encoding

We do not use Coq as a *logical metalanguage* to give a deep encoding of LF$^+$ in Coq, since we are not interested in reasoning on LF$^+$'s metatheory. On the contrary, we provide in this section an encoding as *shallow* as possible, so as to delegate to Coq's metalanguage not only all of LF$^+$'s metalanguage, but also to reduce *inhabitation-search* in LF$^+$ to *proof-search* in Coq.

We exploit the fact that Coq is a conservative extension of the dependent constructive type theory of LF [HHP93], which underpins the type system of LF$^+$, [HLS$^+$16]. Then we simulate/implement in Coq the mechanism of lock-types. Thus we can use Coq both as the host system and as the oracle for external propositions. This yields

a *definitional* encoding of LF^+ in `Coq`. It restricts us, of course, to dealing only with total `Coq`-definable predicates, but this is enough for illustrating our approach and moreover has the advantage of enforcing automatically the *well-behavedness* of the external predicates, provided their `Coq`-encoding is adequate.

Therefore, LF^+ *signatures* and *contexts* are not modeled via structured datatypes, such as *e.g.* lists, but are represented by `Coq`'s contexts and made available as assumptions. The *kind* `Type` is represented directly via `Coq`'s sorts `Set` and `Prop`. The choices are consistent with the `Coq` tradition: datatypes are modeled in `Set`, because these feature strong recursion principles, while `Prop`s are used when introducing specifications. Hence LF^+'s *families* are rendered as `Coq` sets or propositions and *objects* as their inhabitants. Remarkably, we need to implement *only the lock constructor for families*, which we do twice, once for predicates on terms, and once for predicates on families:

```
Definition lockF := fun sigma: Set => fun N: sigma => fun P: sigma->Prop =>
                    fun rho: Prop => forall x: P N, rho.
```

and

```
Definition lockK := fun kappa: Type => fun sigma: kappa => fun P: kappa->Prop =>
                    fun rho: Prop => forall x: P sigma, rho.
```

In the first case families, represented by `rho`, are typed by `Prop` and objects by families, except for the family `sigma` in the definition of $\mathcal{P}$, which is typed by `Set`. In the second case families, represented by `rho`, are again typed by `Prop` and objects by families, while kinds in the definition of $\mathcal{P}$ are typed by `Coq`'s `Type`.

External predicates of LF^+ are therefore encoded as `Prop`-valued functions in `Coq`. In doing this we can take full advantage of `Coq`'s logical strength, *e.g.* Leibniz equality, natural numbers, inductive datatypes, and recursive functions. This is what makes it possible to use `Coq` also as an oracle.

In a nutshell, the gist of the `lockF` definition is to represent the locking of families by a predicate on terms in LF^+ by the Π-type:

$$\ulcorner \mathcal{L}^{\mathcal{P}(N:\sigma)}[\![\rho]\!] \urcorner \quad \rightsquigarrow \quad \Pi x : \ulcorner \mathcal{P} \urcorner (\ulcorner N \urcorner).\ulcorner \rho \urcorner.$$

while the gist of the `lockK` definition is to represent the locking of families by a predicate on families in LF^+ by the Π-type:

$$\ulcorner \mathcal{L}^{\mathcal{P}(\sigma:\kappa)}[\![\rho]\!] \urcorner \quad \rightsquigarrow \quad \Pi x : \ulcorner \mathcal{P} \urcorner (\ulcorner \sigma \urcorner).\ulcorner \rho \urcorner.$$

For example:

$$\ulcorner \mathcal{L}^{\mathcal{C}losed(N:\sigma)}[\![\rho]\!] \urcorner \quad \rightsquigarrow \quad \Pi_{x:\texttt{Closed}(\ulcorner N \urcorner)}.\ulcorner \rho \urcorner,$$

$$\ulcorner \mathcal{L}^{\mathcal{I}nh(\sigma:Type)}[\![\sigma]\!] \urcorner \quad \rightsquigarrow \quad \Pi_{x:\texttt{Inh}(\ulcorner \sigma \urcorner)}.\ulcorner \sigma \urcorner,$$

where `Closed` and `Inh` are `Prop`-valued functions in `Coq` which check *closedness* and *inhabitation* conditions.

Finally locked objects will be represented in `Coq` either using `LockF` or `LockK`, according as to whether the predicate is on terms or on families, as follows (we only give the case for a predicate on terms):

$$\ulcorner \mathcal{L}^{\mathcal{P}(N:\sigma)}[\![M]\!] \urcorner \quad \rightsquigarrow \quad \lambda x : \ulcorner \mathcal{P} \urcorner (\ulcorner N \urcorner) \ulcorner M \urcorner.$$

For example, the necessitation box-introduction rule in Hilbert-style S_4 (see [HLS$^+$16]) is given by:

$$NEC : \Pi A : o.\Pi x : True\ A.\mathcal{L}^{Closed(x:True\ A)}[\![True(\Box A)]\!]$$

whence, an occurrence of $NEC\ A\ x$ can be rendered by:

$$\Gamma \vdash \lambda y : \texttt{Closed}(x).\ulcorner NEC\ A\ x\urcorner : \Pi y : \texttt{Closed}(x).\ulcorner True(\Box A)\urcorner$$

where `Closed` is a `Prop`-valued function in `Coq` which checks the *closedness* condition, *i.e.* there are no variables of type $True(\,\cdot\,)$.

Our encoding might appear weak, but actually it permits us to develop formal proofs up-to $\mathcal{P}$. As a consequence, somewhat surprisingly, our `Definition` is sufficient to derive as `Coq` Lemmas *all* the LF$^+$'s typing rules that involve lock-types. We only give the cases involving families, the other cases for kinds are dealt with similarly. In the following, to avoid wasting space (in `Coq` code), we abbreviate the names of variables denoting types as follows: `s` stands for sigma (σ), `r` stands for rho (ρ), and `t` stands for tau (τ).

— Lock-introduction (see rule ($O{\cdot}Lock$) in Fig. 8) is rendered by Π-introduction:

```
Lemma lock: forall s: Set, forall N: s, forall P: s -> Prop,
            forall r: Prop, forall M:r, lockF s N P r.
intros; unfold lockF; intro; assumption.
Qed.
```

— Unlocking at top level (see rule ($O{\cdot}Top{\cdot}Unlock$) in Fig. 8) is rendered by means of Π-elimination:

```
Lemma top_unlock: forall s: Set, forall N: s, forall P: s -> Prop,
                  forall r: Prop, forall M:lockF s N P r, forall x: P N, r.
intros; exact (M x).
Qed.
```

— Finally, guarded-unlocking (see rule ($O{\cdot}Guarded{\cdot}Unlock$) in Fig. 8) is rendered by an interplay of dependencies, namely that of the unlocked inner term `(N x)` on the externally bound variable of the outer lock `x`, and that of the outer locked typed `(r (N x))` on the unlocked inner term `(N x)`:

```
Lemma guarded_unlock: forall s: Set, forall S: s, forall P: s -> Prop,
                   forall t: Prop, forall r: t -> Prop,
                   forall M: forall y:t, lockF s S P (r y),
                   forall N: lockF s S P t,
                   forall x: P S, r (N x).
intros; unfold lockF; unfold lockF in M; intros; apply M; auto.
Qed.
```

The encoding of this last rule is slightly problematic, because it is necessary to deal with the *unlock constructor*. Namely, we need a witness $x : \ulcorner\mathcal{P}\urcorner(\ulcorner S\urcorner, \ulcorner s\urcorner)$ such that:

$$\ulcorner\mathcal{U}[\![N]\!]\urcorner \qquad \rightsquigarrow \qquad \ulcorner N\urcorner\ x.$$

This is a technical and subtle point, because the witness x is bound in the encoding of the locked type. We have therefore rephrased the guarded-unlock rule

so that the definition in Coq of the lockF constructor, in the conclusion of the rule, appears already unfolded (*i.e.* δ-reduced). Thus the proof witness which needs to be supplied to N is directly available.
However, this trick is transparent to the user, because we have provided him/her with the following *user-defined tactic*[2], which allows one to apply the (*O·Guarded ·Unlock*) rule to a `lockF _ _ _ _` goal by supplying as witness only the proposition `t` (represented below by the formal parameter `z`):

```
Ltac Guarded_unlock z :=
  match goal with
  [ |- lockF _ _ _ ?A ] =>
  unfold lockF at 1;
  apply guarded_unlock with (r := (fun w: z => A));
  [> intro; apply lock | idtac ]
  end.
```

In the following sections on applications we will illustrate further the use of this rule.

In conclusion, we have given a shallow encoding of LF^+ in Coq via a subtle, but ultimately natural, Definition, see [ACDG$^+$19b], which expands the proof-technique of [HLMS17]. As pointed out earlier this does not support the full strength of LF^+, in that predicates are restricted to Coq-definable terms of some type which eventually maps into Prop.

5.2 Adequacy of the encoding of locks in Coq

As to the adequacy of our encoding of locks in Coq, we can state the following result:

Theorem 5.1. For every Γ, M, and A, if $\Gamma \vdash^{\mathsf{LF}^+} M : A$, then $\ulcorner\Gamma\urcorner \vdash^{\mathtt{Coq}} \ulcorner M\urcorner : \ulcorner A\urcorner$. On the other hand, if $\ulcorner\Gamma\urcorner \vdash^{\mathtt{Coq}} N : \ulcorner A\urcorner$, then there exists M such that $\Gamma \vdash^{\mathsf{LF}^+} M : A$, provided that N does not contain variables of type $P(\cdot)$ (corresponding to predicates $\mathcal{P}$ in LF^+) in operator positions.

The variable occurrence constraint in the theorem above is crucial. Otherwise, for every σ, the translated type $\ulcorner\mathcal{L}^{Inh(\sigma:Type)}[\![\sigma]\!]\urcorner \equiv \Pi x : \ulcorner\sigma\urcorner.\ulcorner\sigma\urcorner$ would be always inhabited in Coq, since it would amount to $A \longrightarrow A$.

However, we cannot avoid that, from $\Pi x : \ulcorner A\urcorner.\Pi y : \ulcorner\mathcal{P}\urcorner(\ulcorner U\urcorner).\ulcorner B\urcorner$, *i.e.* the Coq encoding of $A \longrightarrow \mathcal{L}^{\mathcal{P}(U:V)}[\![B]\!]$, we can derive $\Pi y : \ulcorner\mathcal{P}\urcorner(\ulcorner U\urcorner).\Pi x : \ulcorner A\urcorner.\ulcorner B\urcorner$, *i.e.* the Coq encoding of $\mathcal{L}^{\mathcal{P}(U:V)}[\![A \longrightarrow B]\!]$. While in LF^+ we cannot get from a derivation of type $A \longrightarrow \mathcal{L}^{\mathcal{P}(U:V)}[\![B]\!]$ a derivation of type $\mathcal{L}^{\mathcal{P}(U:V)}[\![A \longrightarrow B]\!]$.

In order to be able to achieve this, we would have to extend LF^+ with the axiom that $(A \longrightarrow \mathcal{L}^{\mathcal{P}(U:V)}[\![B]\!]) \longrightarrow \mathcal{L}^{\mathcal{P}(U:V)}[\![A \longrightarrow B]\!]$ is inhabited. Notice that this kind of *scoping enlargement* of locks can happen only when the abstracted variables pushed within the lock do not appear in $\ulcorner\mathcal{P}\urcorner(\ulcorner U\urcorner)$.

[2] L_{tac} is the tactic language available in Coq, allowing one to define proof search procedures [Coq18].

5.3 Implementation of CLLF$_{\mathcal{P}?}$ in Coq

In this subsection, to illustrate the generality of our approach to implementing locks in Coq, we discuss a system introduced in [HLMS15, HLMS18]. The reader can skip this subsection, if not interested in that system, without losing the flow of the paper.

In [HLMS15, HLMS18] we introduced, using the *canonical format*, the system CLLF$_{\mathcal{P}?}$. It generalises the lock mechanism of LLF$_{\mathcal{P}}$ allowing for the external tool to *synthesise a witness*. The $\mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![\cdot]\!]$ becomes a *binder* which binds the variable x of type σ in the locked term/type. In this section we outline a possible implementation in Coq of CLLF$_{\mathcal{P}?}$ along the lines of what we did for LF$^+$.

For the sake of brevity, we will discuss only the main rules involving locks in CLLF$_{\mathcal{P}?}$. But, since the system introduced in CLLF$_{\mathcal{P}?}$ was given in canonical form, for the sake of clarity, we will rephrase them in standard form. The *lock-introduction rule* at the level of terms is straightforward:

$$\frac{\Gamma, x:\sigma \vdash M : \tau}{\Gamma \vdash \mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![M]\!] : \mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![\tau]\!]}\ O{\cdot}? \cdot Lock$$

The lock-operator, graphically highlighted by the question mark (?) before the bound variable x, is a binder. Its bound variable will be replaced by the witness produced in establishing the predicate, according to the following *lock-elimination rule*:

$$\frac{\Gamma \vdash \mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![M]\!] : \mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![\tau]\!] \qquad \mathcal{P}(\Gamma \vdash N : \sigma)}{\Gamma \vdash M[N/x] : \tau[N/x]}\ O{\cdot}? \cdot Top \cdot Unlock$$

Please notice the difference with the ($O{\cdot}Top{\cdot}Unlock$)-rule. In CLLF$_{\mathcal{P}?}$ the unlock provides in fact a witness which needs to be replaced in the term.

Otherwise, if we are not interested in verifying immediately the predicate and computing the witness, we can postpone the task by the following *guarded-unlock rule*:

$$\frac{\Gamma, y{:}\tau \vdash M : \mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![\tau]\!] \qquad \Gamma \vdash N : \mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![']\!]\tau \qquad \sigma =_{\beta\mathcal{L}} \sigma'}{\Gamma \vdash \mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![']\!]\mathcal{U}^{\mathcal{P}(x:\sigma')}[\![M[\mathcal{U}^{\mathcal{P}(x:\sigma)}[\![']\!]N/y]\!] : \mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![']\!]\tau[\mathcal{U}^{\mathcal{P}(x:\sigma)}[\![']\!]N/y]}\ O{\cdot}? \cdot Guarded \cdot Unlock$$

It is worth noticing that, differently from the rule ($O{\cdot}Guarded{\cdot}Unlock$) of LF$^+$, appearing in Figure 8, in this case the subject of the type judgment in the conclusion must be lock-expanded, in order to bind the variable x in $\mathcal{U}^{\mathcal{P}(x:\sigma)}[\![N]\!]$. Indeed this allows for postponing the verification of predicate $\mathcal{P}$ and hence also the computation of the witness.

The previous encoding can be accommodated in Coq as follows:

$$\ulcorner \mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![\rho]\!] \urcorner \qquad \rightsquigarrow \qquad \Pi x : \ulcorner \sigma \urcorner \Pi y : \ulcorner \mathcal{P} \urcorner (x, \ulcorner \sigma \urcorner).\ulcorner \rho \urcorner$$

The binding nature of $\mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![\cdot]\!]$ is rendered in the encoding by the outer Π, which acts on the variable x of type $\ulcorner \sigma \urcorner$.

Clearly also the generalised notion of lock, $\mathcal{L}^{\mathcal{P}(?x:\sigma)}[\![\cdot]\!]$, discussed in this subsection, can have a monadic-ful interpretation, which we shall not discuss here.

6. CALL-BY-VALUE λ-CALCULUS

In this section we use the implementation of LF^+ introduced in Section 5 on a standard benchmark-encoding for Logical Frameworks, namely untyped λ-calculus with a call-by-value equational theory, *i.e.* the λ_v-calculus, see [Plo75]. In the literature there are many ways of encoding this system starting from [AHMP92]. We use the signature given in [HLMS17], because it illustrates the flexibility of LF^+ in capitalising on Higher Order Abstract Syntax (HOAS) when considering *bound* variables, while retaining the ordinary way of referring to *free* variables. We carry out a simple, but meaningful derivation, which provides a paradigm of how to reason with locked assumptions. The full code appears in the online appendix, see [ACDG$^+$19b].

The well-known abstract syntax of λ-calculus is given by: $M ::= x \mid M\ M \mid \lambda x.M$. We model *free* variables in this object language as constants in LF^+. *Bound* variables are modeled by variables of the metalanguage, thus exploiting HOAS in delegating *α-conversion* and *capture-avoiding substitution* to the metalanguage. For instance, the λ-term x (in which the variable is free) is encoded by the term $\vdash_\Sigma$`(free n):term` for a suitable (encoding of a) natural number `n` (see Definition 6.1 below). On the other hand, the λ-term $\lambda x.x$ (in which the variable is obviously bound) is encoded by $\vdash_\Sigma$ `(lam λx:term.x)`.

We introduce therefore the following signature:

Definition 6.1 (LF^+ SIGNATURE Σ_λ FOR UNTYPED λ-CALCULUS).

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

We use natural numbers as standard abbreviations for repeated applications of `S` to `0`. Standard call-by-value conversion is given by the following rules:

Definition 6.2 (CALL-BY-VALUE EQUATIONAL THEORY).

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

$$\frac{\Gamma \vdash_{CBV} M = N \quad \Gamma \vdash_{CBV} N = P}{\Gamma \vdash_{CBV} M = P}\ (\mathsf{trans}) \qquad \frac{\Gamma \vdash_{CBV} M = N \quad \Gamma \vdash_{CBV} M' = N'}{\Gamma \vdash_{CBV} MM' = NN'}\ (\mathsf{app})$$

$$\frac{v \text{ is a free variable or an abstraction}}{\Gamma \vdash_{CBV} (\lambda x.M)v = M[v/x]}\ (\beta_\mathsf{v}) \qquad \frac{\Gamma \vdash_{CBV} M = N,\ x \notin FV(\Gamma)}{\Gamma \vdash_{CBV} \lambda x.M = \lambda x.N}\ (\xi_\mathsf{v})$$

where the contexts Γ represent lists of equality assumptions.

Accordingly, we extend the signature of Definition 6.1 as follows:

Definition 6.3 (LF^+ SIGNATURE Σ_v FOR λ_v-CALCULUS).

```
eq:      term → term → Type
refl:    ΠM:term. eq M M
symm:    ΠM,N:term. eq M N → eq N M
trans:   ΠM,N,P:term. eq M N → eq N P → eq M P
eq_app:  ΠM,N,P,Q:term. eq M N → eq P Q → eq (app M P) (app N Q)
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*(M : term) holds if and only if M is either an abstraction or a variable (*i.e.* a term of the shape (free n)) up-to reduction. Notice that the encoding of the (ξ_v)-rule explicitly enforces the fact that the bound variable ranges over values.

Notice how, in Definition 6.3, LF$^+$'s *lock-types* permit us to model the (β_v) and (ξ_v) rules: the former holds "up-to" the verification of *Val*(N : term), while the latter depends, in turn, on a locked premise.

6.1 Adequacy

The adequacy of the signature for the λ_v-calculus can be stated as follows, see [HLS$^+$16] for more details. Given an enumeration $\{x_i\}_{i\in\mathbb{N}\setminus\{0\}}$ of the variables in the untyped λ_v-calculus, we put:

$$\begin{aligned}\epsilon_{\mathcal{X}}(x_i) &= \begin{cases} \texttt{x\_i}, \text{ if } x_i \in \mathcal{X} \\ (\texttt{free i}), \text{ if } x_i \notin \mathcal{X}\end{cases}\\ \epsilon_{\mathcal{X}}(MN) &= (\texttt{app}\ \epsilon_{\mathcal{X}}(M)\,\epsilon_{\mathcal{X}}(N))\\ \epsilon_{\mathcal{X}}(\lambda x.M) &= (\texttt{lam}\ \lambda \texttt{x:term}.\epsilon_{\mathcal{X}\cup\{\texttt{x}\}}(M)),\end{aligned}$$

where $\mathcal{X}$ is the set of *bindable* variables (*i.e.* not free variables), and hence in the latter clause, $x \notin \mathcal{X}$. The expression x_i denotes a Coq variable.

Theorem 6.4 (Adequacy of syntax). Let $\{x_i\}_{i\in\mathbb{N}\setminus\{0\}}$ be an enumeration of the variables in the λ_v-calculus. Then, the encoding function $\epsilon_{\mathcal{X}}$ is a bijection between the λ_v-calculus terms with bindable variables in $\mathcal{X}$ and the terms M derivable in judgements $\Gamma \vdash_{\Sigma_\lambda} M : \texttt{term}$ in $\beta\eta$-lnf, where $\Gamma = \{\texttt{x} : \texttt{term} \mid x \in \mathcal{X}\}$. Moreover, the encoding is compositional, *i.e.* for a term M, with bindable variables in $\mathcal{X} = \{x_1, \ldots, x_k\}$, and $N_1, \ldots, N_k$, with bindable variables in $\mathcal{Y}$, the following holds:

$$\epsilon_{\mathcal{X}}(M[N_1, \ldots, N_k/x_1, \ldots, x_k]) = \epsilon_{\mathcal{X}}(M)[\epsilon_{\mathcal{Y}}(N_1), \ldots, \epsilon_{\mathcal{Y}}(N_k)/x_1, \ldots, x_k].$$

Theorem 6.5 (Adequacy of equational theory). Given an enumeration $\{x_i\}_{i\in\mathbb{N}\setminus\{0\}}$ of the variables in the λ_v-calculus, there is a bijection between derivations of the judgment $\vdash_{CBV} M = N$ on terms with no bindable variables in the CBV λ_v-calculus and proof terms h, such that $\vdash_{\Sigma_{CBV}} \texttt{h} : (\texttt{eq}\ \epsilon_\emptyset(M)\ \epsilon_\emptyset(N))$ is in $\beta\eta$-lnf.

6.2 Formalisation in Coq

We now proceed to represent the above signature in the Coq editor for LF$^+$ presented in Section 5.

First, we declare the syntactic category of terms (typed by Set) and its constructors, by exploiting the built-in representation of natural numbers, which is available in Set:

```
Parameter term: Set.
Parameter free: nat -> term.
Parameter app : term -> term -> term.
Parameter lam : (term -> term) -> term.
```

Then, we model the *Val* predicate in `Coq`, since `Coq` can play in this case also the oracle role:

```
Definition Val := fun N:term => (exists n, N = (free n)) \/
                                (exists M, N = (lam M)).
```

One can easily, albeit *not formally*, check that the above `Coq`-encoding of "being a value" is an adequate formalisation of the intended concept, thereby giving evidence also that the predicate originally used in the lock is well-behaved in the sense of Definition 2.1. All `Coq` definable predicates are obviously well-behaved.

Finally, we encode the call-by-value equational theory, by means of a predicate `eq` typed by `Prop`:

```
Parameter eq: term -> term -> Prop.
Parameter refl:   forall M:term, eq M M.
Parameter symm:   forall M N:term, eq M N -> eq N M.
Parameter trans:  forall M N P:term, eq M N -> eq N P -> eq M P.
Parameter eq_app: forall M N P Q:term, eq M N -> eq P Q ->
                  eq (app M P) (app N Q).
Parameter betav:  forall M:term->term, forall N:term,
                  lockF term N Val (eq (app (lam M) N) (M N)).
Parameter csiv:   forall M N:term->term,
                  (forall x:term, lockF term x Val (eq (M x) (N x))) ->
                  eq (lam M) (lam N).
```

Notice that, in defining `term` and `eq`, we do not use `Coq`'s inductive types. This would go beyond the expressivity of LF^+ and would not support *full* Higher Order Abstract Syntax (HOAS). Inductive types would be compatible, in dealing with variables, with *weak* HOAS, [DFH95] but *exotic terms* would arise, see [DH94].

The use of lock-types in expressing the (ξ_v)-rule, although natural, might appear to be unmanageable in applications, since the variable in the premise is not immediately free or bound, but only *bindable*. But, as it will become apparent in the following example, the $(O{\cdot}Guarded{\cdot}Unlock)$ rule in LF^+ accommodates precisely this issue. Namely, the necessary verification is pushed at the outermost level, where it is discharged by the application of the (ξ_v)-rule.

To illustrate the `Coq` editor of LF^+ in action, we conclude the section with the formal proof of the simple equation $\lambda x.\ z\ ((\lambda y.y)\ x) = \lambda x.\ z\ x$. As pointed out earlier, this is a paradigm example of how to reason with locks. The crucial step is the application of the $(O{\cdot}Guarded{\cdot}Unlock)$ rule; the first premise is given by the application of the $(O{\cdot}Lock)$ rule to the conclusion of the `eq_app` rule, while the second premise is the conclusion of the `betav` rule[3]:

$$\dfrac{\dfrac{\dfrac{\nabla \qquad \dfrac{}{x{:}t \vdash_{\Sigma_v} \mathcal{L}^{Val(x:t)}[\![eq(app(lam(\lambda y{:}t.\ y), x), x)]\!]}\,(\text{betav})}{z, x{:}t \vdash_{\Sigma_v} \mathcal{L}^{Val(x:t)}[\![eq(app(z, app(lam(\lambda y{:}t.\ y), x)), app(z, x))]\!]}\,(\text{O}{\cdot}\text{G}{\cdot}\text{U})}{z{:}t \vdash_{\Sigma_v} \forall x{:}t.\ \mathcal{L}^{Val(x:t)}[\![eq(app(z, app(lam(\lambda y{:}t.\ y), x)), app(z, x))]\!]}}{z{:}t \vdash_{\Sigma_v} eq(\lambda x{:}t.\ app(z, app(lam(\lambda y{:}t.\ y), x)), \lambda x{:}t.\ app(z, x))}\,(\text{csiv})$$

[3]In the following proof tree we use the abbreviation t for *term* and $(O{\cdot}G{\cdot}U)$ for $(O{\cdot}Guarded{\cdot}Unlock)$. We apply it via the `Guarded_unlock` tactic introduced in Section 5.

"∇" represents the following derivation, where $u \triangleq eq(app(lam(\lambda y{:}t.\, y), x), x)$ is the witness that we supply to the `Guarded_unlock` tactic (see Section 5):

$$\dfrac{\dfrac{\dfrac{}{z{:}t \vdash eq(z,z)}\,(\text{refl}) \quad \dfrac{}{x{:}t,\ w{:}u \vdash eq(app(lam(\lambda y{:}t.\, y), x), x)}\,(\text{Hyp})}{z, x{:}t,\ w{:}u \vdash eq(app(z, app(lam(\lambda y{:}t.\, y), x)), app(z, x))}\,(\text{eq\_app})}{z, x{:}t,\ w{:}u \vdash \mathcal{L}^{Val(x:t)}[\![eq(app(z, app(lam(\lambda y{:}t.\, y), x)), app(z, x))]\!]}\,(\text{O·Lock})$$

We remark the role of the (*O·Guarded·Unlock*) rule, which allows us to apply the rules of the Σ_{v} signature (in this case, the `eq_app` rule) *under Val*, *i.e.* it permits to handle even premises which are locked.

We finally observe that using the `Coq` editor of LF$^+$ we may accomplish the above goal without having to exhibit the full proof term beforehand, as we had to do using the ad-hoc type checker written by Michielini [Mic16] or the embedding of LLF$_{\mathcal{P}}$ in MMT [MR19], because we can now build it interactively and incrementally, via `Coq`'s tactics.

7. FAST-AND-LOOSE REASONING PARADIGMS

The present section is an introduction to the reasoning paradigms underpinning the massive case studies presented in the two following sections, concerning the management of *branch prediction* and of *optimistic concurrency control*.

In Section 4 we outlined already many logical situations which could benefit if analysed using locks. Here, we focus on a family of emerging paradigms which we have termed *fast-and-loose reasoning paradigms* borrowing the phrase from [DHJG06].

It is often the case, when carrying out informal, or rather pre-formal, reasoning in program development [DHJG06] or in ordinary mathematics [HLLS16], that we trade off efficiency against correctness and postpone tedious or computationally demanding checks, until we are really sure that the intended goal can be achieved. Only then it is worth "dotting the i's and crossing the t's". At machine level similar policies are implemented to achieve efficiency in concurrency control or branch prediction. In these cases the checks are not postponed, but rather *run in parallel*, and strict protocols are implemented to recover from possibly counterfactual situations.

In this section we briefly speculate on the novel idea of using *Computer Assisted Formal Reasoning* (CAFR) tools for fast-and-loose *informal reasoning*. The gist of the idea is to *encapsulate*, using locks, the arguments which are informal for the time being. It is thus apparent how monads come into play.

We list below a number of situations which, in our view, come under the fast-and-loose reasoning paradigm. In the following two sections the same ideas will be used to model the instances of this paradigm in Computer Architecture.

7.1 *A historical example: the* Regula Falsi *for solving linear equations*

The paramount example of fast-and-loose reasoning attitudes are *trial and error* procedures. The *Regula Falsi* is perhaps the oldest one. It is a technique for solving linear equations, widespread over continents and along centuries, see [CWB$^+$12]. It first appeared in the ancient Egyptian *Rhind Papyrus* dating from 1650 BC; then as the *ying bu zu shu* (rule of too much and not enough) in Chapter 7 of the *Jiuzhang Suanshu* (Computational Prescriptions in Nine Chapters) dating from 200 BC; the

Arab mathematician Abū Kāmil used it extensively in the 9th century in the *Kitāb al Khāṭa'ayn* (the book of the two errors); the Indian mathematician Bhāshkara in his book *Līlāvati* dating from the 12th century called it *false supposition*, and finally Fibonacci in the 13th century in his *Liber Abaci* called it *Regula Falsi*.

The *Regula* is the following: when presented with a linear equation guess a value for the indeterminate and compute the value of the linear terms in the equation, using your guess. Next compute the fraction of the result corresponding to the constant term in the equation. The root of the equation will be the same fraction of your guess. In the general case the guess is iterated as in a dichotomic search, hence the name "too much and not enough". Using locks the simple case can be expressed as an alternate *top-unlock-rule*, namely:

$$\frac{\mathcal{L}^{\{M(a)=N\}(a:\mathbb{Q})}[\![M(a)=N]\!] \qquad kM(a)=N}{M(ka)=N}$$

provided M is a rational linear homogenous polynomial, and N is a constant term.

Notice that the lock in this case plays the role of a counterfactual conditional.

7.2 Fitch-Prawitz Set Theory FP

Fitch-Prawitz Set Theory, [Fit52, Pra65, HLLS16], is a non standard set-theory which comes closest among consistent set-theories to approximating the contradictory *naïve unrestricted comprehension axiom*, namely the axiom asserting the existence of the set $\{x \mid \phi(x)\}$ for any property ϕ. The theory is a natural deduction First Order Logic w.r.t. a signature with only one term constructor $\lambda x.\phi(x)$ and only one binary predicate, *i.e.* membership $\in$. These are introduced and eliminated by the rules:

$$\frac{\phi(t)}{t \in \lambda x.\phi(x)}\ \lambda - Intro \qquad \frac{t \in \lambda x.\phi(x)}{\phi(t)}\ \lambda - Elim$$

Since FP does not rely on any hierarchy of sets, Russell's paradox is immediately derivable if no further restrictions are enforced. Fitch introduced various conditions, but it was Prawitz who isolated the crucial property, namely that the derivation be *normal*. In [HLMS15] we have extended this condition to *normalisability*. In our view this system is a paradigm example of a system supporting the fast-and-loose reasoning paradigm. We can pretend to reason in *Cantor's paradise* and freely use the un-restricted set-theoretic comprehension principle. These arguments however yield only *pseudo-proofs*. At some stage we need to check that these pseudo-proofs are indeed normalisable. One could check this proviso only at the very end on the whole proof. Lock types [HLMS17, HLMS18] permit to highlight incrementally the crucial points in the proofs where such checks need to be carried out, *e.g.* each time introduction rules are applied, possibly removing the locks at one's convenience,

Clearly the *Type Ambiguity proviso* in Type Theory, discussed in Section 4, or the *small categories* blanket assumption, normally made by the working category theorist, are other examples of the fast-and-loose reasoning paradigm when dealing with large objects in the context of hierarchies.

For the sake of completeness we mention that similar issues arise also in dealing with highly self-descriptive concepts such as *truth*, see [Kri75].

7.3 Co-inductive proofs in Coq

Coq itself offers an immediate example of the fast-and-loose reasoning paradigm in the way by which it handles *co-inductive proofs* and the Cofix-*tactic*, [Coq18]. Indeed in proof development one can freely apply the coinductive hypothesis, namely assume the goal, without checking that it is guarded by some constructor. The editor checks this condition either at the end or whenever the user invokes the Guarded command. The Coq editor keeps track *implicitly* of the guardedness condition, whereas locks propagate such proviso explicitly.

7.4 "Fast-and-Loose Reasoning is Morally Correct" [DHJG06]

In the seminal paper [DHJG06] the authors introduced the concept of "Fast-and-Loose Reasoning" to denote those program transformations and program equality derivations, in *possibly non-terminating* functional programming languages, which are strictly speaking sound only if the *data are total and finite*. The authors prove that this kind of informal program development is morally correct in the sense that if two programs are proved to be equal in the world of sets they are equivalent (w.r.t. a certain notion based on PER's) also in the world of domains. Hence correctly terminating programs, proved equal assuming that the inputs are total, are not transformed in looping ones.

The reasoning attitude of [DHJG06] can be naturally expressed using locks. For example $reverse \circ reverse \;=\; id$ is sound if the type of $reverse$ is taken to be $\Pi x \in List[Nat].\mathcal{L}^{\{x \neq \bot\}(x:List[Nat])}[\![List[Nat]]\!]$. Locks automate the burden of propagating preconditions across the proof tree.

7.5 A visionary example

Quantum Computing is really a kind of analogical computing. Somewhat futuristically we can imagine to use Quantum Computers for deriving propositions, either using massively parallel case analyses or even in *counterfactual computing*, *i.e.* in computing without executing or in truth-without-proof. Locks could be used to make apparent where such non-apodictic sources of evidence are invoked.

7.6 The issue of Adequacy

Fast-and-loose reasoning paradigms, especially in computer architecture, see Section 8, usually come with some *error-recovery protocol* or *backtracking strategy*. In LF$^+$ predicates in Locks are *implicitly* checked in parallel. How do we express and capture *adequately* such paradigms/protocols? In Section 8.3 we give a possible solution in Coq, but more experimenting is necessary. This issue is related to the problem of *exiting* from a monad, when the predicate is false, possibly backtracking to a previous "safe" state, or even an entirely new state. In general the natural exit from a monad (T, η, μ) can be expressed by $\exists x.N = \eta(x) \;\rightarrow$

$$\mathcal{L}^{\exists x.N=\eta(x)(N:\mathcal{A})}[\![\texttt{let}_T\ y = N\ \texttt{in}\ M = M[\eta^{-1}(N)/y]]\!] \rightarrow (\texttt{let}_T\ y = N\texttt{in}\ M) = M[\eta^{-1}(N)/y]$$

but we could also speculatively imagine that, in case $\neg\exists x.N = \eta(x)$, then some kind of judgement

$$\mathcal{L}^{\exists x.N=\eta(x)(N:\mathcal{A})}[\![(\texttt{let}_T\ x = N\ \texttt{in}\ M) = M[\eta^{-1}(N)/y]]\!] \rightarrow (\texttt{let}_T\ x = N\texttt{in}\ M) = M'$$

holds for a suitable M'.

8. BRANCH PREDICTION

In computer architecture, a *branch predictor* is a digital circuit which tries to guess which branch will exit a *conditional control*, *e.g.* an `if-then-else`, before the result of the test is actually known [JKL00]. The purpose of a branch predictor circuit is to speed-up the flow in the instruction pipeline. Namely, the time needed for the branch instruction, to pass the execution stage, is not wasted idly, but the computation proceeds by guessing whether the conditional branch is more likely to be taken or not taken. The branch which is guessed to be the most likely is then fetched and speculatively executed; if later it is detected that the guess was wrong, the speculatively executed or partially executed instructions are discarded and the pipeline starts over with the correct branch, incurring a delay.

In this section we model the behavior of conditional branching in LF^+ by considering the *conditional jump* construct of the *Unlimited Register Machine (URM)*, a simple universal model of computation popularised by N. Cutland [Cut80]. Our modeling of branch prediction is rather coarse-grained w.r.t. real scenarios in CPUs, but this helps to focus on the main issues involved. As a first step, we give the semantics of URM using lock-types; then, in the second part of the section, we deal with the issue of handling *misprediction*. Finally we address the adequacy of the whole approach.

An URM has an infinite number of registers $R_0, R_1, \ldots$ containing natural numbers $r_0, r_1, \ldots$ which may be mutated by instructions. Sequences of instructions form programs:

s	$::= \langle \iota \mapsto r_\iota \rangle^{\iota \in [0..\infty]}$		Store
I	$::= Z(i) \mid S(i) \mid T(i,j) \mid J(i,j,k)$	$i,j,k \in \mathbb{N}$	Instruction
P	$::= (\iota \mapsto I_\iota)^{\iota \in [1..m]}$	$m \in \mathbb{N}$	Program

The four kinds of instructions Zero, Successor, Transfer, Jump have the following intended meanings ($r \to R$ stands for loading the natural r into the register R):

$$
\begin{array}{ll}
Z(i) & \triangleq\ 0 \to R_i \\
S(i) & \triangleq\ r_i + 1 \to R_i \\
T(i,j) & \triangleq\ r_i \to R_j \\
J(i,j,k) & \triangleq\ \text{if } r_i{=}r_j \text{ then execute as next instruction } I_k \text{ else the following one}
\end{array}
$$

When given a program P, a program counter n, and a store s, an URM executes the program starting from the n-th instruction in P and carries out the instructions sequentially (unless a "positive" J instruction is encountered), mutating at each step the contents of the store as prescribed by the instructions. The evaluation of a program may be described, therefore, as follows:

$$
E(P,n,s) \;=\; \begin{cases}
s & \text{if } fetch(P,n){=}Halt \\
E(P, n{+}1, zero(s,i)) & \text{if } fetch(P,n){=}Z(i) \\
\ldots & \ldots \\
E(P,k,s) & \text{if } fetch(P,n){=}J(i,j,k) \text{ and } s(i){=}s(j) \\
E(P,n{+}1,s) & \text{if } fetch(P,n){=}J(i,j,k) \text{ and } s(i){\neq}s(j)
\end{cases}
$$

We use the *zero* function for updating the store according to the Z instruction (similar updating functions *succ* for S and *move* for T are omitted) and the $fetch$ function for recovering the instruction pointed to by the program counter. The

Halt instruction is added to make the function $fetch$ total. A computation stops if and only if $fetch$ fetches $Halt$. On the other hand, due to the looping back via the J instruction, there are non-terminating computations. In our case study we consider only terminating computations (the interested reader may refer to [Cia11] for a coinductive approach to diverging computations).

The functions introduced in order to formalise evaluation are defined as follows:

$$\begin{array}{ll} fetch(P,n) & \triangleq \text{ if } n{>}length(P) \text{ then } Halt \text{ else } I_n \\ zero(s,i) & \triangleq \lambda\iota{\in}\mathbb{N}.\text{ if } \iota{=}i \text{ then } 0 \text{ else } s(\iota) \\ succ(s,i) & \triangleq \lambda\iota{\in}\mathbb{N}.\text{ if } \iota{=}i \text{ then } s(\iota){+}1 \text{ else } s(\iota) \\ move(s,i,j) & \triangleq \lambda\iota{\in}\mathbb{N}.\text{ if } \iota{=}j \text{ then } s(i) \text{ else } s(\iota) \end{array}$$

To introduce an LF$^+$ signature for the URM machine, we need first to encode infinite stores and non-structured programs. Both datatypes are handled by mimicking lists.

Definition 8.1 (LF$^+$ signature for Stores and Programs).

```
nat: Type   0: nat    S: nat → nat
store: Type zeros: store    cs: nat → store → store
ins: Type   Ht: ins    Zr: nat → ins    ...    Jp: nat → nat → nat → ins
pgm: Type   void: pgm    cp: ins → pgm → pgm
```

Natural numbers `nat` are extensively used in the URM-signature: actually, we make them play also the role of store locations, *e.g.* in `Zr` (encoding Z), and program counters, in `Jp` (encoding J). Stores are modeled as "LF$^+$-lists". We introduce therefore the `nil`-like `zeros` object constant and the cons-like `cs` constructor, whose intended meanings are given by the constants in Definition 8.2. We encode programs as lists of instructions in `ins`, where `Ht` represents the $Halt$ instruction.

To structure the evaluation of URM programs, we introduce the two small-step relations $\rightsquigarrow \subseteq\ pgm \times nat \times store \times nat \times store$ and $\Rightarrow \subseteq\ pgm \times nat \times store \times store$, as follows:

$$\frac{fetch(P,n){=}Z(i)}{\langle n,s\rangle \rightsquigarrow^P \langle n{+}1, zero(s,i)\rangle}\ (\text{eZ}) \qquad \frac{fetch(P,n){=}S(i)}{\langle n,s\rangle \rightsquigarrow^P \langle n{+}1, succ(s,i)\rangle}\ (\text{eS})$$

$$\frac{fetch(P,n){=}T(i,j)}{\langle n,s\rangle \rightsquigarrow^P \langle n{+}1, move(s,i,j)\rangle}\ (\text{eT}) \qquad \frac{\langle n,s\rangle \rightsquigarrow^P \langle m,t\rangle \quad \langle m,t\rangle \rightsquigarrow^P \langle q,u\rangle}{\langle n,s\rangle \rightsquigarrow^P \langle q,u\rangle}\ (\text{trans})$$

$$\frac{fetch(P,n){=}J(i,j,k) \quad s(i){=}s(j)}{\langle n,s\rangle \rightsquigarrow^P \langle k,s\rangle}\ (\text{Jt}) \qquad \frac{fetch(P,n){=}J(i,j,k) \quad s(i){\neq}s(j)}{\langle n,s\rangle \rightsquigarrow^P \langle n{+}1,s\rangle}\ (\text{Jf})$$

$$\frac{fetch(P,n){=}Halt}{\langle n,s\rangle \Rightarrow^P s}\ (\text{empty}) \qquad \frac{\langle n,s\rangle \rightsquigarrow^P \langle m,t\rangle \quad fetch(P,m){=}Halt}{\langle n,s\rangle \Rightarrow^P t}\ (\text{stop})$$

Now we come to the crucial issue. LF$^+$'s *lock-types* allow us to model faithfully also the execution of a "branch prediction" version of this semantics, by postponing the store access and test required by J, which is a slow instruction. Lock-types permit to carry out the store access and equality check concurrently and asynchronously w.r.t. the main computation, in the spirit of the fast-and-loose philosophy. We omit

for simplicity in the following definition the encoding of the S and T instructions. In the following definition some of the constants are self-explicatory, constants `step` and `eval` are the encodings of $\rightsquigarrow$ and $\Rightarrow$ respectively, the remaining constants reify the behaviour of the auxiliary functions and judgements.

Definition 8.2 (LF^+ SIGNATURE Σ_B FOR URM EVALUATION).

```
T      : Type
fetch  : pgm → nat → ins → Type
zero   : store → nat → store → Type
step   : prg → nat → store → nat → store → Type
eval   : prg → nat → store → store → Type
⟨_,_,_⟩ : store → nat → nat → T
fvn    : Πn:nat. fetch void n Ht
fc0    : ΠI:ins. ΠQ:prg. fetch (cp I Q) 0 I
fcn    : ΠI,L:ins. ΠQ:prg. Πn:nat. fetch Q n L → fetch (cp I Q) (S n) L
zvn    : Πn:nat. zero zeros n zeros
zc0    : Πv:nat. Πs:store. zero (cs v s) 0 (cs 0 s)
zcn    : Πv,n:nat. Πs,t:store. zero s n t → zero (cs v s) (S n) (cs v t)
sZ     : ΠP:pgm. Πn,i:nat. Πs,t:store.
         fetch P n (Zr i) → zero s i t → step P n s (S n) t
sJt    : ΠP:pgm. Πn,i,j,k:nat. Πs:store.
         fetch P n (Jp i j k) → L^Eq(⟨s,i,j⟩:T)[[step P n s k s]]
sJf    : ΠP:pgm. Πn,i,j,k:nat. Πs:store.
         fetch P n (Jp i j k) → L^Neq(⟨s,i,j⟩:T)[[step P n s (S n) s]]
sTr    : ΠP:pgm. Πn,m,q:nat. Πs,t,u:store.
         step P n s m t → step P m t q u → step P n s q u
e0     : ΠP:pgm. Πn:nat. Πs:store. fetch P n Ht → eval P n s s
e1     : ΠP:pgm. Πn,m:nat. Πs,t:store.
         step P n s m t → fetch P m Ht → eval P n s t
```

where $\mathtt{Eq}(\langle \mathtt{s}, \mathtt{i}, \mathtt{j}\rangle, \mathtt{T})$ holds if and only if $s(i){=}s(j)$, and $\mathtt{Neq}(\langle \mathtt{s}, \mathtt{i}, \mathtt{j}\rangle, \mathtt{T})$ iff $s(i){\neq}s(j)$.

The above signature is adequate in the sense of Claim 8.6.

8.1 Formalisation in Coq

In this subsection we illustrate how to encode the signature Σ_B in the implementation of LF^+ in `Coq`, given in Section 5. Accordingly, we express the external predicates Eq and Neq in `Coq` thus using it as the *oracle*. In so doing we take advantage also of `Coq`'s built-in notion of Leibniz equality, natural numbers, and lists, to define stores, locations, program counters, instructions, and programs (all typed by `Set`), namely:

```
Definition store: Set := list nat.
Definition loc: Set := nat. Definition pc: Set := nat.
Parameter ins: Set. Parameter Ht: ins.
Parameter Zr: loc -> ins. ... Parameter Jp: loc -> loc -> pc -> ins.
Definition pgm: Set := list ins.
```

We define also the type of the input to the oracle, *i.e.* a store and a pair of locations, as an *inductive type* `T` of triples, to gain efficient projection functions. Memory access is realised through the built-in total function `nth`, which returns the `0` value when the end of a list-store is reached.

The oracle predicates can then be formalised in Coq using these datatypes, as follows:

```
Inductive T: Set := triple: store -> loc -> loc -> T.
Definition pr1 (x:T): store := match x with triple s i j => s end. ...
Definition s_nth (s:store) (n:loc): nat := nth n s 0.
Definition Eq := fun x:T => s_nth (pr1 x) (pr2 x) = s_nth (pr1 x) (pr3 x).
Definition Neq := fun x:T => s_nth (pr1 x) (pr2 x) <> s_nth (pr1 x) (pr3 x).
```

The evaluation semantics is finally encoded as a predicate, via suitable auxiliary functions that update the store. We omit for lack of space such functions and most of the Coq translation of Definition 8.2, this is available at [ACDG$^+$19b].

```
Parameter step: pgm -> pc -> store -> pc -> store -> Prop.
Parameter sJt: forall P n i j k s, fetch P n = (Jp i j k) ->
               lockF T (triple s i j) Eq (step P n s k s).
Parameter sJf: forall P n i j k s, fetch P n = (Jp i j k) ->
               lockF T (triple s i j) Neq (step P n s (S n) s). ...
Parameter eval: pgm -> pc -> store -> store -> Prop. ...
```

In principle we could have used directly the signature Σ_B of Definition 8.2 in defining the external predicates, but this would have required to spell out in an extended signature all the *recursion* and *inversion* principles necessary to manipulate the data-types. But this would have been a time-consuming task. Exploiting Coq's expressive power is another feature of our implementation.

8.2 Examples

In order to appreciate the encoding at work, let us consider the simple program $P \triangleq [Z(0), J(0,1,0)]$ and the stores $s \triangleq [1,1]$ and $t \triangleq [0,1]$.

Example 8.3 (First proof of $\langle 1,s\rangle \rightsquigarrow^P \langle 1,t\rangle$). [4]

$$\dfrac{\dfrac{\dfrac{P(1){=}J(0,1,0)}{\mathcal{L}^{Eq(\langle s,0,1\rangle:T)}[\![\langle 1,s\rangle \rightsquigarrow^P \langle 0,s\rangle]\!]}\,(\text{sJt}) \qquad Eq(\langle s,0,1\rangle)}{\langle 1,s\rangle \rightsquigarrow^P \langle 0,s\rangle}\,(\text{O}{\cdot}\text{Top}) \qquad \dfrac{P(0){=}Z(0)}{\langle 0,s\rangle \rightsquigarrow^P \langle 1,t\rangle}\,(\text{sZ})}{\langle 1,s\rangle \rightsquigarrow^P \langle 1,t\rangle}\,(\text{sTr})$$

In this proof tree there is a limited amount of parallelism between the equality check of the contents of store locations and the main computation, because we wait until the verification of $Eq(\langle s,0,1\rangle)$ is accomplished, before channeling the reductions via the transitivity (sTr) rule. The parallelism may be increased by exploiting the ($O{\cdot}Guarded{\cdot}Unlock$) rule, which handles arguments within a lock-type, and allows us to apply the (sTr) rule even in the presence of a locked J reduction.

[4] As already remarked in Section 5, in the present and the next derivations we display LF$^+$'s types without the proof terms because these are synthesised by the editor.

Example 8.4 (Second proof of $\langle 1,s\rangle \rightsquigarrow^P \langle 1,t\rangle$). [5]

$$\dfrac{\dfrac{\dfrac{P(1){=}J(0,1,0)}{\mathcal{L}^{Eq(\langle s,0,1\rangle:T)}[\![\langle 1,s\rangle \rightsquigarrow^P \langle 0,s\rangle]\!]} \quad \dfrac{P(0){=}Z(0)}{\langle 0,s\rangle \rightsquigarrow^P \langle 1,t\rangle}\ (\text{sTr via O·G·U})}{\mathcal{L}^{Eq(\langle s,0,1\rangle:T)}[\![\langle 1,s\rangle \rightsquigarrow^P \langle 1,t\rangle]\!]} \quad Eq(\langle s,0,1\rangle)}{\langle 1,s\rangle \rightsquigarrow^P \langle 1,t\rangle}\ (\text{O·Top})$$

The $Eq(\langle s,0,1\rangle)$ check can now be delayed (actually, it now appears closer to the conclusion) and carried out independently w.r.t. the main reduction. The (*O·Guarded·Unlock*) rule allows for more proof trees for the same judgment. This is precisely what accommodates the *branch prediction* philosophy.

An even higher degree of parallelism could be achieved in LF^+ if a mechanism to "compose" pieces of reductions within *different* lock-types were available. This would give us the opportunity to apply the transitivity rule "under" *pairs* of Jump instructions. If, for instance, we want to manage a maximum of 2 branch constructs, we can define introduction and elimination rules of the following shape:

$$\dfrac{\mathcal{L}^{\mathcal{P}_1(\langle \vec{x_1}\rangle:T)}[\![\langle n,s\rangle \rightsquigarrow^P \langle m,t\rangle]\!] \qquad \mathcal{L}^{\mathcal{P}_2(\langle \vec{x_2}\rangle:T)}[\![\langle m,t\rangle \rightsquigarrow^P \langle q,u\rangle]\!]}{\mathcal{L}^{\mathcal{P}_1(\langle \vec{x_1},T\rangle);\mathcal{P}_2(\langle \vec{x_2}\rangle:T)}[\![\langle n,s\rangle \rightsquigarrow^P \langle q,u\rangle]\!]}\ (\mathcal{P}_+)$$

$$\dfrac{\mathcal{L}^{\mathcal{P}_1(\langle \vec{x_1},T\rangle);\mathcal{P}_2(\langle \vec{x_2}\rangle:T)}[\![\langle n,s\rangle \rightsquigarrow^P \langle m,t\rangle]\!] \qquad \mathcal{P}_1(\vec{x_1})}{\mathcal{L}^{\mathcal{P}_2(\langle \vec{x_2}\rangle:T)}[\![\langle n,s\rangle \rightsquigarrow^P \langle m,t\rangle]\!]}\ (\mathcal{P}_{-1})$$

$$\dfrac{\mathcal{L}^{\mathcal{P}_1(\langle \vec{x_1},T\rangle);\mathcal{P}_2(\langle \vec{x_2}\rangle:T)}[\![\langle n,s\rangle \rightsquigarrow^P \langle m,t\rangle]\!] \qquad \mathcal{P}_2(\vec{x_2})}{\mathcal{L}^{\mathcal{P}_1(\langle \vec{x_1}\rangle:T)}[\![\langle n,s\rangle \rightsquigarrow^P \langle m,t\rangle]\!]}\ (\mathcal{P}_{-2})$$

where $\mathcal{P}_\iota$ stands for Eq or Neq, and $\vec{x_\iota} \equiv \langle x_\iota, i_\iota, j_\iota\rangle$, for all $\iota{\in}\{1,2\}$. We could then delay even more the access to pairs of memory locations to check for (in)equality of their contents, as follows.

Example 8.5 (Proof of $\langle 1,s\rangle \rightsquigarrow^P \langle 2,t\rangle$).

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathcal{L}^{Eq(\langle s,0,1\rangle:T)}[\![\langle 1,s\rangle \rightsquigarrow^P \langle 1,t\rangle]\!]} \quad \dfrac{P(1){=}J(0,1,0)}{\mathcal{L}^{Neq(\langle t,0,1\rangle:T)}[\![\langle 1,t\rangle \rightsquigarrow^P \langle 2,t\rangle]\!]}\ (\mathcal{P}_+)}{\mathcal{L}^{Eq(\langle s,0,1\rangle,T);Neq(\langle t,0,1\rangle:T)}[\![\langle 1,s\rangle \rightsquigarrow^P \langle 2,t\rangle]\!]} \quad Eq(\langle s,0,1\rangle)}{\mathcal{L}^{Neq(\langle t,0,1\rangle:T)}[\![\langle 1,s\rangle \rightsquigarrow^P \langle 2,t\rangle]\!]}\ (\mathcal{P}_{-1}) \quad Neq(\langle t,0,1\rangle)}{\langle 1,s\rangle \rightsquigarrow^P \langle 2,t\rangle}$$

It turns out that such a "composition" of predicates can be dealt with via lock nesting: that is, we manage elimination rules in the form $\mathcal{P}_-$ by means of the (*O·Top·Unlock*) rule (*i.e.* `Coq`'s `top_unlock` lemma) and introduction rules such as $\mathcal{P}_+$ via the (*O·Guarded·Unlock*) rule (*i.e.* `Coq`'s `guarded_unlock` lemma and `Guarded_unlock` tactic, see Section 5). Moreover, the order of predicates does not matter, as the commutativity property holds.

We have thus shown how LF^+ can naturally accommodate computations running in parallel asynchronously, as it happens when performing branch prediction.

[5] We omit here the details of the full application of (*O·G·U*).

8.3 Adequacy: prediction and misprediction

It is apparent that the LF$^+$'s signature Σ_{B} introduced so far models all possible *outcomes* of a branch predictor circuit, within a "soup" of provable judgments, many of which are in fact counterfactual. Namely, given a URM program P, a store s and a program counter n, by working with our machinery we are able to mimic every terminating computation that a branch predictor would perform in transforming s into a final store t. We could formalise all this by a suitable *adequacy* result.

If we want to formalise more perspicuously a branch predictor circuit, we need to make more precise assumptions on its behaviour. Recall, however, that our formal model of branch prediction is coarse-grained, being in fact without pipelining, differently from real-world branch predictor circuits. In the following claim we state the adequacy of the signature Σ_{B} w.r.t. a branch predictor which predicts at most one Jump instruction and carries out at most two instructions ahead of the fully checked, definitive execution flow.

We introduce the notation $\{A\} \longrightarrow_P^* \{B\}$ to denote the computation carried out by such a branch predictor circuit from the state A to the state B under the program P.

The claim can be proved by induction on the structure of the derivation (soundness) and by induction on the length of the computation (completeness).

Claim 8.6 (Adequacy of Σ_{B} signature). For every program $P \in pgm$, program counters $m, n \in nat$, stores $s, t \in store$, and $\mathcal{P} \in \{Eq, Neq\}$:

Soundness: *If* $\Gamma \vdash_{\Sigma_{\mathsf{B}}} \texttt{step P n s m t}$ *then* $\{n, s\} \longrightarrow_P^* \{m, t\}$

Completeness: *1) If* $\{n, s\} \longrightarrow_P^* \{m, t\}$ *without active mispredictions then* $\Gamma \vdash_{\Sigma_{\mathsf{B}}} \texttt{step P n s m t}$

2) If $\{n, s\} \longrightarrow_P^* \{m, t\}$ *with one active misprediction then* $\exists u, i, j.\ \Gamma \vdash_{\Sigma_{\mathsf{B}}} \mathcal{L}^{\mathcal{P}(\langle u,i,j\rangle:T)}[\![\texttt{step P n s m t}]\!]$,

The way branch predictor computations are represented in our setting is not fully satisfactory. In fact, we do not model either *prediction* or *misprediction*, but allow for taking always the "correct" branch. In the practice of computer architecture, the selection of which strategy to take, may be either dynamic (via information collected at runtime) or static, *i.e.* decided in advance. When a misprediction occurs, the execution of the incorrect branch is discarded and the computation is resumed from the correct branch.

In this section, we discuss two different URM semantics, which implement two kinds of *static* prediction that we name *NEVER* protocol and *BACKWARDS* protocol. When a Jump instruction is encountered, in the former case we guess the result of the test to be false and arrange the computation to proceed from the following instruction; in the latter scenario, the target of the Jump is checked at runtime and it is taken only if it is less than the current program counter.

Suppose that $P_n{=}fetch(P, n){=}J(i, j, k)$, then the intended *prediction* of the Jump instruction for the NEVER and the BACKWARDS protocol are, respec-

tively:

$J(i,j,k) \triangleq_{\mathsf{N}}$ execute as the next instruction the one immediately following, *i.e.* P_{n+1}
$J(i,j,k) \triangleq_{\mathsf{W}}$ if $k<n$ then execute as next instruction P_k else P_{n+1}

The LF^+ signature for the NEVER protocol is almost the same as that of Definition 8.2, but for the `sJt` rule, which disappears, and for `sJf`, which is updated to:

```
sJnever : ΠP:pgm. Πn,i,j,k:nat. Πs:store.
          fetch P n (Jp i j k) → L^Neq(⟨s,i,j,P,n,k⟩:U)⟦step P n s (S n) s⟧
```

On the other hand, in the LF^+ signature for the BACKWARDS protocol one replaces both rules `sJt` and `sJf` of Definition 8.2, respectively by:

```
sJback  : ΠP:pgm. Πn,i,j,k:nat. Πs:store.
          fetch P n (Jp i j k) → k < n → L^Eq(⟨s,i,j,P,n,k⟩:U)⟦step P n s k s⟧
sJahead : ΠP:pgm. Πn,i,j,k:nat. Πs:store.
          fetch P n (Jp i j k) → k >= n → L^Neq(⟨s,i,j,P,n,k⟩:U)⟦step P n s (S n) s⟧
```

In both protocols the predicate given as input to the oracle contains now three extra parameters (thus its type becomes U), namely the program P, the address n of the Jump instruction involved, and its potential target k, *i.e.* the information to be used to backtrack the computation in the case of misprediction. Therefore, a novel mechanism to handle such an event is required. In the NEVER scenario we have to add just one rule, which completes the Σ_{N} signature, namely:

```
backtrack_neq : ΠP:pgm. Πn,i,j,q,k,m:nat. Πs,t,u:store.
                L^Neq(⟨u,i,j,P,q,k⟩:U)⟦step P n s m t⟧ → u(i)=u(j) →
                step P n s k u
```

The BACKWARDS case demands for a second rule, which gives the Σ_{W} signature:

```
backtrack_eq : ΠP:pgm. Πn,i,j,q,k,m:nat. Πs,t,u:store.
               L^Eq(⟨u,i,j,P,q,k⟩:U)⟦step P n s m t⟧ → u(i)≠u(j) →
               step P n s (S q) u
```

These new rules provide the mechanism which allows us to discard a misprediction, performed with the store being u, and resume the computation from the correct branch, that is, from u itself and the k-th instruction, provided we are able to prove that $u(i)=u(j)$, in the first case, and from the q+1-th instruction, provided that $u(i)\neq u(j)$ holds, in the second one. Notice that the tests, $u(i)=u(j)$ and $u(i)\neq u(j)$, in the backtracking rules are computationally demanding and of the same nature as the predicate in the lock. In our encoding philosophy we must conceive them as being carried out at the level of the Oracle. We will discuss further this issue at the end of the section.

Claim 8.7 (Adequacy of Σ_{N} for the "never" protocol). For every program $P \in pgm$, program counters $m, n \in nat$, stores $s, t \in store$:

Soundness: *If* $\Gamma \vdash_{\Sigma_{\mathsf{N}}} \texttt{step P n s m t}$ *then* $\{n, s\} \longrightarrow^*_{PN} \{m, t\}$
Completeness: *1) If* $\{n, s\} \longrightarrow^*_{PN} \{m, t\}$ *without active mispredictions then* $\Gamma \vdash_{\Sigma_{\mathsf{N}}} \texttt{step P n s m t}$
2) If $\{n, s\} \longrightarrow^*_{PN} \{m, t\}$ *with one active misprediction then* $\exists u, i, j, q, k.\ \Gamma \vdash_{\Sigma_{\mathsf{N}}} \mathcal{L}^{Neq(\langle u,i,j,P,q,k\rangle:U)}[\![\texttt{step P n s m t}]\!]$

A similar Claim may be made for Σ_{W} w.r.t. the BACKWARDS protocol.

Wrapping up, the modifications of the Coq code, to formalise the NEVER protocol, concern the type of (the input to) the oracle, the *Neq* predicate, the semantics of the Jump instruction, and the extra rule to backtrack from a misprediction:

```
Inductive U: Set := sestet: store -> loc -> loc -> pgm -> pc -> pc -> U.
Definition Neq := fun x:U => store_nth (pr1 x) (pr2 x) <>
                             store_nth (pr1 x) (pr3 x) /\
                             fetch (pr4 x) (pr5 x) =
                             Jp (pr2 x) (pr3 x) (pr6 x).
Parameter sJnever: forall P n i j k s, fetch P n = (Jp i j k) ->
                   lockF U (sestet s i j P n k) Neq (step P n s (S n) s).
Parameter backtrack_neq: forall u i j q k P n s m t,
                   lockF U (sestet u i j P q k) Neq (step P n s m t) ->
                   s_nth u i = s_nth u j -> step P n s k u.
```

The proof trees performed with the Σ_{B} signature in the previous section are clearly affected by these additions. In the case of the NEVER protocol, in Example 8.3 the subgoal $\langle 1, s\rangle \rightsquigarrow^P \langle 0, s\rangle$ cannot be proved via (O·Top), because the *Eq* predicate is not defined in this protocol: hence, one has to apply `backtrack_neq` and discard the $\mathcal{L}^{Neq(\langle s,0,1,P,1,0\rangle:T)}[\![\langle 1, s\rangle \rightsquigarrow^P \langle 2, s\rangle]\!]$ misprediction. And this is precisely what would happen if the protocol were implemented. For the same reason the proof carried out in Example 8.4 is no longer available, while that in Example 8.5 may be accomplished alternatively. On the other hand, proofs involving only the *Neq* predicate remain unchanged.

Corresponding remarks apply to the BACKWARDS protocol, see [ACDG$^+$19b].

A few remarks are in order regarding the rules `backtrack_eq` and `backtrack_neq`. As we pointed out earlier these rules have a *hybrid* nature, mixing, in fact, both Coq and LF$^+$ judgements. They could be viewed as further additions to LF$^+$, formalising exits out of the lock monad, alternative to the (*O·Top·Unlock*) rule, *e.g.*

$$\frac{\Gamma \vdash M : \mathcal{L}^{\texttt{Neq}(\langle \texttt{u,i,j,P,q,k}\rangle:\texttt{U})}[\![\texttt{step P n s m t}]\!] \quad \texttt{u(i)=u(j)} \quad \Gamma \vdash \texttt{step P n s m t} : \mathit{Type}}{\Gamma \vdash \mathcal{E}^{\texttt{Neq}(\langle \texttt{u,i,j,P,q,k}\rangle:\texttt{U})}[M] : \texttt{step P n s (S q) u}}$$

for a suitable new object level constructor $\mathcal{E}^{\cdot(\cdot:\cdot)}[\cdot]$ This reading however would break the nature of LF$^+$ as a "framework" because such rules are object-language specific. We are currently exploring how they could be subsumed conveniently under a general scheme for handling mechanisms for exiting monads. See also Section 7.6.

9. OPTIMISTIC CONCURRENCY CONTROL

In transactional systems, *concurrency control* (CC) ensures that concurrent operations generate correct results, still taking advantage of parallelism in speeding up the management of possibly remote operations. The *optimistic* approach (OCC), in particular, assumes that *transactions* do not interfere frequently; hence, transactions are allowed to access *resources* without implementing lock mechanisms.

When a transaction A is completed and "commits" the shared resources trying to make permanent the modifications it has carried out, the concurrency control manager checks that no other transaction B, which has committed after A was "activated", has already modified the data that A has used, *i.e.* read or written. If the check reveals *interference* , A "rolls back" *i.e.* it is restarted, otherwise it is allowed to commit its modifications, which are therefore made permanent.

In this section, we approach formally OCC in LF^+ by formalising the semantics of the Simple Transaction Language (STL) [CDK02], a language which is usually adopted by textbooks addressing concurrency control of transactional systems.

Throughout the section we assume that transactions modify resources only *locally* before committing. A generic transaction i may perform the following *actions*:

$$
\begin{array}{ll}
start(i) & \triangleq \text{ activation (first action of the transaction)} \\
read(i,j) & \triangleq \text{ reading on resource } j \\
write(i,j) & \triangleq \text{ writing to resource } j \\
check(i) & \triangleq \text{ checking against interference (last action): commit vs roll back}
\end{array}
$$

We represent both transactions and resources by means of natural numbers; hence, *schedules* are finite sequences of actions, carried out by the former on the latter:

$$
\begin{array}{lcll}
T & \triangleq & \mathbb{N} & \text{Transaction} \\
R & \triangleq & \mathbb{N} & \text{Resource} \\
A & ::= & start(i) \mid check(i) \mid \quad i{\in}T & \text{Action} \\
 & & read(i,j) \mid write(i,j) \quad j{\in}R & \\
S & ::= & (\iota{\mapsto}A_\iota)^{\iota\in[1..m]} \quad m{\in}\mathbb{N} & \text{Schedule}
\end{array}
$$

To design a possible semantics for OCC, we use the following data structures:

$$
\begin{array}{lll}
ctr & \triangleq stack(A) & \text{activation control (only } start \text{ and } check \text{ actions)} \\
seq & \triangleq T \rightharpoonup queue(A) & \text{actions by transactions, in sequencial order} \\
usr & \triangleq T \rightharpoonup list(R) & \text{used resources (by transactions)} \\
wrt & \triangleq R \rightharpoonup list(T) & \text{writing transactions (to resources),}
\end{array}
$$

We use these datatypes[6] to implement concurrency and diagnose interference as follows. When transactions start (via *start*), we push on top of the stack *ctr* such an information. Then, we keep recording the resources used by transactions (via *read* and *write*) in *usr*, and the transactions writing to them (via *write*) in *wrt*; correspondingly, we collect all the actions carried out by transactions, in sequential order, in *seq*. Note that if a $write(i,j)$ action takes place, we intend that the same resource j has been also read by the transaction i.

[6]The type of *wrt* goes in the opposite direction, just for convenience in implementing checks.

The crucial (potentially conflicting) action is $check(i)$, when two mutually exclusive alternatives arise: either there is no transaction k which has committed after the activation of i (*i.e.* we do not find any $check(k)$ on top of $start(i)$ in ctr), or at least one of such transactions does exist.

In the first case no interference occurs, and therefore we may both register the successful conclusion of i (by pushing $check(i)$ on the top of ctr) and consider permanent its actions, just by leaving unchanged the usr and wrt datatypes.

In the second alternative scenario (when one or more occurrences of $check(k)$ are on top of $start(i)$ in ctr) a deeper validation process has to be undertaken, by computing the intersection of the resources used by i with those written by the potentially interfering transactions k. In case the intersection set is empty we are in the previous non-interference scenario. If at least one resource used by i and written by k does exist, we remove the (activation of the) transaction i (*i.e.* $start(i)$) from ctr and erase i from usr and wrt; and we evaluate sequentially i from its beginning by appending its actions, collected in seq, on top of the current schedule.

Some remarks are in order. First, it is apparent that the readings and writings of successfully committing transactions must not be deleted, because their actions are needed to be checked against potential future interferences. Second, coherently w.r.t. the toy-languages usually introduced by textbooks that address concurrency control, *e.g.* [CDK02], we give an *abstract semantics*, *i.e.* we do not consider a *store* structure to record the values "read from" and "written to" resources. We could easily do so, but this would pointlessly obscure matters.

To formalise the OCC that we have conceived, we introduce a *small-step* semantics, whose intended meaning is to process list-like schedules $S \in sch$, till they are emptied, by transforming the initial *state* $\langle [\,], \uparrow, \uparrow, \uparrow \rangle \in \mathcal{M} \equiv ctr \times seq \times usr \times wrt$ into the final one. Hence, we define the relations $\mapsto \subseteq\ sch \times \mathcal{M} \times sch \times \mathcal{M}$ and $\Longrightarrow \subseteq\ sch \times \mathcal{M} \times \mathcal{M}$, namely[7]:

$$\frac{}{\{[\,], N\} \Longrightarrow N}\,(\text{empty}) \qquad \frac{\{S, N\} \mapsto \{[\,], N'\}}{\{S, N\} \Longrightarrow N'}\,(\text{stop})$$

$$\frac{\{S, N\} \mapsto \{S', N'\} \quad \{S', N'\} \mapsto \{S'', N''\}}{\{S, N\} \mapsto \{S'', N''\}}\,(\text{trans})$$

$$\frac{}{\{s(i)::S, M\} \mapsto \{S, \langle push(s(i), c), enqueue(s(i), s_i), u, w\rangle\}}\,(\text{start})$$

$$\frac{}{\{r(i,j)::S, M\} \mapsto \{S, \langle c, enqueue(r(i,j), s_i), cons(j, u_i), w\rangle\}}\,(\text{read})$$

$$\frac{}{\{w(i,j)::S, M\} \mapsto \{S, \langle c, enqueue(w(i,j), s_i), cons(j, u_i), cons(i, w_j)\rangle\}}\,(\text{write})$$

$$\frac{}{\mathcal{L}^{\mathrm{Opt}(\langle i, M\rangle : C)}[\![\{c(i)::S, M\} \mapsto \{S, \langle push(c(i), c), enqueue(c(i), s_i), u, w\rangle\}]\!]}\,(\text{opt})$$

$$\frac{}{\mathcal{L}^{\mathrm{Itf}(\langle i, M\rangle : C)}[\![\{c(i)::S, M\} \mapsto \{S', \langle remove(s(i), c), empty(s_i), empty(u_i), delete(i, w\rangle\}]\!]}\,(\text{itf})$$

[7] To save space, we shorten the notation for actions to their contracted versions (*e.g.* $start(i)$ becomes $s(i)$) and we use subscripts for the components of the state (*e.g.* $s(i) \in seq$ is written s_i).

where we make use of the following notations: $M = \langle c, s, u, w\rangle$; $S' = append($ $enqueue(check(i), s_i), S)$ in the last rule; $C = T \times \mathcal{M}$ in the last two rules. The (non-)interference conditions, that we encode via lock-types (one is the negation of the other, as in Section 8) are defined by:

$$\begin{array}{lll} \mathrm{Itf}(\langle i, M\rangle, C) & \triangleq & \exists k{\in}T.\ \exists h{\in}R.\ check(k) \gg_c start(i) \wedge h{\in}u_i \wedge k{\in}w_h \\ \mathrm{Opt}(\langle i, M\rangle, C) & \triangleq & \forall k{\in}T.\ \neg check(k) \gg_c start(i) \vee \\ & & \quad check(k) \gg_c start(i) \Rightarrow (\forall h{\in}u_i.\ k{\notin}w_h) \end{array}$$

where the notation "$\gg_c$" represents that the lefthand side *precedes*, *i.e.* "is on the top of", the righthand side in the stack c.

The non-self evident functions adopted in the semantics are defined as follows[8]:

$$\begin{array}{lll} remove(a, c) & \triangleq & \text{if } top(c){=}a \text{ then } pop(c) \text{ else } push(top(c), remove(a, pop(c))) \\ delete(i, w) & \triangleq & erase(i, w_j)\ \forall write(i, j){\in}s_i \\ erase(i, w_j) & \triangleq & \text{if } w_j{=}[\,] \text{ then } [\,] \text{ else if } w_j{=}h{::}l \text{ then} \\ & & \quad \text{if } h{=}i \text{ then } erase(i, l) \text{ else } h{::}erase(i, l) \end{array}$$

In our semantics, the intended meaning of $\{S, M\} \mapsto \{[\,], N\}$ is that the schedule S in the state M completes its processes, by transforming M into N.

9.1 Formalisation in Coq

All these structures could be promptly defined in LF^+ and then encoded in Coq as we did in Section 8. For the sake of brevity, we skip such a phase, and work directly in Coq, since the role and behaviour of lock-types is not affected by this choice. Thus we take advantage of Coq's native inductive features and use the built-in notion of Leibniz equality, natural numbers for defining transactions and resources, and lists to define schedules. It is convenient to model also actions as an *inductive type*, because in the semantics we need decidability of equality on actions, which has to proved by (double) induction on the actions themselves. Hence we put:

```
Definition trs: Set := nat.
Definition res: Set := nat.
Inductive act: Set := start: trs -> act | read: trs -> res -> act
                    | write: trs -> res -> act | check: trs -> act.
Definition sch: Set := list act.
Lemma eq_dec_act: forall a b: act, {a = b} + {a <> b}. ...
```

To gain efficient projection functions, as in Section 8 we encode via inductive types both the type `T` of the input to the oracle, *i.e.* a transaction and a state, and the `state` type itself, which is formed by a cartesian product of four components:

```
Definition ctr: Set := list act.
Definition seq: Set := list (list act).
Definition usr: Set := list (list res).
Definition wrt: Set := list (list trs).
Inductive state: Set := quad: ctr -> seq -> usr -> wrt -> state.
Definition pr1 (x:state): ctr := match x with quad c s u w => c end. ...
Inductive T: Set := pair: trs -> state -> T.
Definition left (y:T): trs := match y with pair i M => i end. ...
```

[8] Notice that, in rule (itf), $delete(i, w)$ has to be computed before emptying s_i through $empty(s_i)$.

We have rendered also the `state` components as lists. Some remarks are in order:

— the activation control `ctr`, being a *stack* datatype, is naturally modeled by a list;

— sequential actions `seq`, used resources `usr`, writing transactions `wrt`, being all partial functions, are lists of lists, where the n-th list represents the application of the corresponding function to n, the empty list representing undefined values;

— the queue `seq` is also represented, albeit inefficiently, by a list.

The oracle predicates are considerably more involved w.r.t. the ones in previous sections. We need to formalise the "$\gg_c$" predicate (`on_top_of`) and a polymorphic function which selects the n-th component of a list of lists, namely:

```
Fixpoint on_top_of (k i:trs)(L:list act){struct L}: Prop := ...
Fixpoint select (i:nat)(A:Set)(LL:list(list A)) {struct i}: list A := ...
Definition Itf := fun z:T => exists k:trs, exists h:res,
                  on_top_of k (left z) (pr1 (right z)) /\
                  In h (select (left z) trs (pr3 (right z))) /\
                  In k (select h trs (pr4 (right z))).
Definition Opt := fun z:T => forall k:trs,
                  (not (on_top_of k (left z) (pr1 (right z)))) \/
                  (on_top_of k (left z) (pr1 (right z)) ->
                  forall h:res, In h (select (left z) trs (pr3 (right z))) ->
                  not (In k (select h trs (pr4 (right z))))).
```

Full details appear in the web-appendix [ACDG$^+$19b].

The small-step semantics is finally encoded as a predicate, via suitable auxiliary functions that update the state (the full code is available at [ACDG$^+$19b]):

```
Parameter step: sch -> state -> sch -> state -> Prop. ...
Parameter eval: sch -> state -> state -> Prop. ...
```

9.2 Example: lost update

The formalisation we have introduced allows us to master typical correctness problems, a.k.a. *anomalies*, that may occur in transactional systems, such as e.g. *lost update, inconsistent read, ghost update*. To manage, for example, the first kind of anomaly, let us consider the following schedule, where two transactions a and b operate concurrently on a resource x that initially contains the n value:

Transaction a	Transaction b
$start(a),\ read(a,x)$	
	$start(b),\ read(b,x),\ write(b,x),\ check(b)$
$write(a,x),\ check(a)$	

We assume, for instance, that both transactions calculate the successor of x and then write its new value. It is apparent that $check(b)$ succeeds whereas $check(a)$ does not, *i.e.* the transaction a must be rolled back, otherwise the final value stored in x would be incorrectly $n{+}1$, being thus lost the effect of the b transaction.

We outline some of the formal steps in the execution of the above schedule. First we define:

$$S = [start(a), read(a,x), start(b), read(b,x), write(b,x), check(b), write(a,x), check(a)]$$
$$T = [start(a), read(a,x), write(a,x), check(a)]$$

An interference is diagnosed by our formal system when a prompts to commit, because $check(b)$ is on top of $start(a)$ in the stack (first line below), and b has written to the resource x (fifth line), in turn read by a (first list of the fourth line):

$$\begin{aligned}\{S, \langle [\,], \uparrow, \uparrow, \uparrow \rangle\} \mapsto \{[check(a)], \langle & [check(b), start(b), start(a)], \\ & [[start(a), read(a,x), write(a,x)], \\ & \ [start(b), read(b,x), write(b,x), check(b)]], \\ & [[x,x],[x,x]], \\ & [[a,b]]\rangle\}\end{aligned}$$

Therefore, the transaction a is rolled back, with the following immediate effect:

$$\begin{aligned}\mapsto \{T, \langle & [check(b), start(b)], \\ & [[\,], \\ & \ [start(b), read(b,x), write(b,x), check(b)]], \\ & [[\,],[x,x]], \\ & [[b]]\rangle\}\end{aligned}$$

Afterwards, a can be processed successfully, hence the schedule is eventually emptied:

$$\begin{aligned}\{S, \langle [\,], \uparrow, \uparrow, \uparrow \rangle\} \mapsto \{[\,], \langle & [check(a), start(a), check(b), start(b)], \\ & [[start(a), read(a,x), write(a,x), check(a)], \\ & \ [start(b), read(b,x), write(b,x), check(b)]], \\ & [[x,x],[x,x]], \\ & [[a,b]]\rangle\}\end{aligned}$$

The goal is accomplished via `Coq`'s top-down editor similarly to what was done in the previous case study on branch prediction in Section 8. Apart from requiring to deal with more involved datatypes and semantics, the application presented in this section does not demand for the development of new specifications or proof techniques w.r.t. Section 8, see [ACDG+19b].

9.3 Adequacy

The optimistic concurrency control system, which we have modeled in this section, has quite a different purpose from the branch prediction application. The latter tries to speedup the computation within CPUs, whereas OCC handles transactions which may be remote w.r.t. each other and w.r.t. the resources they use.

Again, as for branch prediction, we claim that our formalisation models all possible outcomes of an OCC system, simply because it is always possible to take the "correct" management action, and use the top-unlock rule to get rid of the lock. Hence it is adequate in the sense of Claim 8.6.

Claim 9.1 (Adequacy of `Coq` signature for OCC). Let be $\{A\} \longrightarrow^* \{B\}$ the representation of a concurrent transaction carried out by an OCC system. Given schedules $S, T \in sch$, states $s, t \in state$, and $\mathcal{P} \in \{Opt, Itf\}$:

$$\begin{array}{ll}\text{Soundness:} & \textit{If } \Gamma \vdash \texttt{step S s T t} \textit{ then } \{S,s\} \longrightarrow^* \{T,t\} \\ \text{Completeness:} & \textit{If } \{S,s\} \longrightarrow^* \{T,t\} \textit{ then } \Gamma \vdash \texttt{step S s T t}\end{array}$$

Modeling optimistic concurrency control requires only the use of the top-unlock rule. But the logical setting that we have defined in `Coq` in this section is quite gen-

eral and could allow us to model also other concurrency control strategies, different from the optimistic one. We do not elaborate on these.

10. CONCLUSION

This paper is a significant enhancement and expanded version of [ACDG$^+$19a]. We present LF$^+$, a generalisation of the Lax Logical Framework LLF$_\mathcal{P}$, introduced in [HLMS17]. The novel features of this framework w.r.t. previous ones are a more expressive lock-type system, as well as a streamlined and simplified rule-system and notation. We show, furthermore, how to use Coq as a proof development environment, supporting mechanised proof search, for a version of LF$^+$ in which the predicates used in locks are Coq-definable. This is achieved by giving a *shallow*, actually definitional, implementation of LF$^+$ in Coq. A specific *adequacy result*, see Theorem 5.1, puts the encoding on firm ground.

In this paper we discuss in detail the monadic nature of locks and give a repertoire of logical situations and attitudes which we believe can benefit from an analysis in terms of lock-types. These range from the historically significant example of the *regula falsi* to *squash types*. An emerging family of such proof attitudes we term fast-and-loose reasoning paradigms, following [DHJG06]. These arise in situations where boring and cumbersome checks are postponed until they are worthwhile, but go well beyond these examples, leading to seemingly unrelated programming techniques such as *branch prediction circuits* and *concurrency control systems*. The main pragmatic contributions of this paper are the two case studies which illustrate how locks in LF$^+$ can model adequately such techniques. More work needs to be done in order to streamline adequacy results for the various protocols used in such contexts.

One may legitimately ask why should a user go through the overhead of learning LF$^+$ rather than utilise directly the Coq encoding. First, there are pragmatic motivations. When a proof is expressed using LF$^+$ we can use in principle also external tools, and LF$^+$ shows how to encapsulate them. Encoding predicates in Coq might be a cumbersome task or might even not be possible. There are also cognitive motivations. As all logical frameworks, also LF$^+$ is *normative*. The very process of formalising a reasoning pattern, highlighting where *up-to* reasoning steps can arise using LF$^+$, renders the argument conceptually easier to grasp and hence to formalise in a Logical Framework. Factoring out patterns introduces a uniform perspective which can be more easily transferred than an *ad hoc* approach. Moreover an *ad hoc* encoding in Coq is more prone to being inadequate.

The very idea of *reasoning formally* on *fast-and-loose reasoning patterns* has not been considered before, to our knowledge, in the context of Type Theory, while it arose naturally in the LF$^+$ perspective. We believe that *reasoning up-to* is an almost ubiquitous (albeit under diverse formats) irreducible cognitive pattern. To increase the transparency of a formalisation, it deserves therefore to be dealt with by a primitive notion such as that of lock-type.

However more work needs to be done to understand fully the connections between *lock-types*, proving and programming *up-to-equivalence*, *monads*, and *fast-and-loose reasoning*. We feel, however, that in this paper we have provided clear indications of the conceptual relationship relating these notions.

Our definitional implementation suggests how to rapidly prototype editors for other calculi such as $\mathsf{CLLF}_{\mathcal{P}?}$, see [HLMS17], or extensions of LF^+ which support an algebraic structure of locks.

There are many questions still to be asked about lock-types. The most natural one is: what are the logical limitations, if any, in having considered only typing judgements as predicates in locks? Since our predicates can access the whole environment, *everything*, in principle, can be represented via a predicate on a judgement. Of course opaque encodings might arise in doing this, so there could be a point in introducing predicates on other kinds of judgements, such as equality judgements, or in making the logical structure of predicates more visible and thus accessible to the user. Many possibilities are open, even considering judgements in different contexts, provided predicates are well-behaved.

Besides the many suggestions briefly sketched in the paper, an important case study related to the fast-and-loose philosophy which we intend to develop is that of Fitch-Prawitz consistent Set Theory, [HLLS16]. This is a natural proof theoretic counterpart of the naïve Set Theory used in developing ordinary mathematics (see also [Gir98]).

It would be interesting to address the issue of extending full-fledged locks to `Coq` itself. To this end connections with the paper [CSW14] should be explored, for its approach in combining different proof systems.

Finally, we intend to explore how to prototype an alternate editor for LF^+ using the MMT UniFormal Framework of F. Rabe, [RK13]. Indeed, there exists already an implementation of $\mathsf{LLF}_{\mathcal{P}}$ in MMT (see, *e.g.*, [MR19]); hence, it should be intriguing to develop and compare a new encoding, taking into account the new alternative lock-rules of LF^+ presented in Section 2.

References

[ACDG+19a] F. Alessi, A. Ciaffaglione, P. Di Gianantonio, F. Honsell, and M. Lenisa. A definitional implementation of the lax logical framework LLFP in coq, for supporting fast and loose reasoning. In D. Miller and I. Scagnetto, editors, *Proceedings of the Fourteenth Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@LICS 2019, Vancouver, Canada, 22nd June 2019*, volume 307 of *EPTCS*, pages 8–23, 2019. doi:10.4204/EPTCS.307.3.

[ACDG+19b] F. Alessi, A. Ciaffaglione, P. Di Gianantonio, F. Honsell, M. Lenisa, and I. Scagnetto. *The Web appendix of this paper.* `https://users.dimi.uniud.it/~alberto.ciaffaglione/LLFP/JFR-19.tar.gz`, 2019.

[AHMP92] 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.

[Bar84] H. Barendregt. *Lambda Calculus: its Syntax and Semantics*. North Holland, 1984.

[BB02] H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. *J. Autom. Reasoning*, 28(3):321–336, 2002. doi:10.1023/A:1015761529444.

[CDK02] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems - concepts and designs (3. ed.)*. International computer science series. Addison-Wesley-Longman, 2002.

[Cia11] A. Ciaffaglione. A coinductive semantics of the unlimited register machine. In F. Yu and C. Wang, editors, *Proceedings 13th International Workshop on Verification of Infinite-State Systems, INFINITY 2011, Taipei, Taiwan, 10th October 2011.*, volume 73 of *EPTCS*, pages 49–63, 2011. doi:10.4204/EPTCS.73.7.

[Coq18] The Coq Development Team. *The Coq Reference Manual - Release 8.8.2*. `https://coq.inria.fr`, 2018.

[CSW14] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 33–46. ACM, 2014. doi:10.1145/2535838.2535883.

[Cut80] N. Cutland. *Computability - An introduction to recursive function theory*. Cambridge University Press, 1980.

[CWB$^+$12] J. Chabert, C. Weeks, E. Barbin, J. Borowczyk, J. Chabert, M. Guillemot, A. Michel-Pajus, A. Djebbar, and J. Martzloff. *A History of Algorithms: From the Pebble to the Microchip*. Springer Berlin Heidelberg, 2012. URL `https://books.google.it/books?id=XcDqCAAAQBAJ`.

[DFH95] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 124–138, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[DH94] J. Despeyroux and A. Hirschowitz. Higher-order abstract syntax with induction in coq. In F. Pfenning, editor, *Logic Programming and Automated Reasoning*, pages 159–173, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[DHJG06] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In J. G. Morrisett and S. L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 206–217. ACM, 2006. doi:10.1145/1111037.1111056.

[Dyb96] P. Dybjer. Internal type theory. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs*, pages 120–134, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[Fit52] F. B. Fitch. *Symbolic logic*. New York, 1952.

[Gir98] J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2):175 – 204, 1998. doi:https://doi.org/10.1006/inco.1998.2700.

[HHP93] R. Harper, F. Honsell, and G. D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993. doi:10.1145/138027.138060. Preliminary version in Proc. of LICS'87.

[HLL07] F. Honsell, M. Lenisa, and L. Liquori. A framework for defining logical frameworks. *Volume in Honor of G. Plotkin, Electr. Notes Theor. Comput. Sci.*, 172:399–436, 2007. doi:10.1016/j.entcs.2007.02.014.

[HLL+12] F. Honsell, M. Lenisa, L. Liquori, P. Maksimović, and I. Scagnetto. Lfp: A logical framework with external predicates. In *Proceedings of the Seventh International Workshop on Logical Frameworks and Meta-languages, Theory and Practice*, LFMTP '12, pages 13–22, New York, NY, USA, 2012. ACM. doi:10.1145/2364406.2364409.

[HLLS08] 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.

[HLLS16] F. Honsell, M. Lenisa, L. Liquori, and I. Scagnetto. Implementing cantor's paradise. In A. Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 229–250, 2016. doi:10.1007/978-3-319-47958-3_13.

[HLMS15] F. Honsell, L. Liquori, P. Maksimović, and I. Scagnetto. Gluing together proof environments: Canonical extensions of LF type theories featuring locks. In I. Cervesato and K. Chaudhuri, editors, *Proceedings Tenth International Workshop on Logical Frameworks and Meta Languages: Theory and Practice, LFMTP 2015, Berlin, Germany, 1 August 2015.*, volume 185 of *EPTCS*, pages 3–17, 2015. doi:10.4204/EPTCS.185.1.

[HLMS17] F. Honsell, L. Liquori, P. Maksimović, and I. Scagnetto. $\mathsf{LLF}_{\mathcal{P}}$: a logical framework for modeling external evidence, side conditions, and proof irrelevance using monads. *Logical Methods in Computer Science*, 13(3), 2017. doi:10.23638/LMCS-13(3:2)2017.

[HLMS18] F. Honsell, L. Liquori, P. Maksimović, and I. Scagnetto. Plugging-in proof development environments using Locks in LF. *Mathematical Structures in Computer Science*, 28(9):1578–1605, 2018. doi:10.1017/S0960129518000105.

[HLS14] F. Honsell, L. Liquori, and I. Scagnetto. $\mathsf{LF}^{\times}$: 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.

[HLS+16] F. Honsell, M. Lenisa, I. Scagnetto, L. Liquori, and P. Maksimović. An open logical framework. *J. Log. Comput.*, 26(1):293–335, 2016. doi:10.1093/logcom/ext028.

[Hon13] F. Honsell. 25 years of formal proof cultures: Some problems, some philosophy, bright future. In *Proceedings of the Eighth ACM SIG-*

PLAN International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP'13, pages 37–42, New York, NY, USA, 2013. ACM. doi:10.1145/2503887.2503896.

[JKL00] D. A. Jiménez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In A. Wolfe and M. S. Schlansker, editors, *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 33, Monterey, California, USA, December 10-13, 2000*, pages 67–76. ACM/IEEE Computer Society, 2000. doi:10.1109/MICRO.2000.898059.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981. doi:10.1145/319566.319567.

[Kri75] S. Kripke. Outline of a theory of truth. *The journal of philosophy*, 72(19):690–716, 1975.

[Mic16] V. Michielini. LLF$_\mathcal{P}$ *type checker*. `https://github.com/francescodellamorte/llfp-type-checker`, 2016.

[MLS84] P. Martin-Löf and G. Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. doi:https://doi.org/10.1016/0890-5401(91)90052-4. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[MR19] D. Müller and F. Rabe. Rapid Prototyping Formal Systems in MMT: 5 Case Studies. In *LFMTP 2019 Logical Frameworks and Meta-Languages: Theory and Practice 2019*, Vancouver, Canada, June 2019. URL `https://hal.archives-ouvertes.fr/hal-02150167`.

[Plo75] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125 – 159, 1975. doi:https://doi.org/10.1016/0304-3975(75)90017-1.

[Pra65] D. Prawitz. *Natural Deduction. A Proof Theoretical Study*. Almqvist Wiksell, Stockholm, 1965.

[RK13] F. Rabe and M. Kohlhase. A scalable module system. *Inf. Comput.*, 230:1–54, 2013. doi:10.1016/j.ic.2013.06.001.

[Uni13] T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.