A Definitional Implementation of the Lax Logical Framework $\mathsf{LLF}_{\mathscr{P}}$ in `Coq`, for Supporting Fast and Loose Reasoning

Fabio Alessi Alberto Ciaffaglione Pietro Di Gianantonio Furio Honsell Marina Lenisa

Department of Mathematics, Computer Science and Physics
University of Udine[*]
Udine, Italy

`name.surname@uniud.it`

The Lax Logical Framework, $\mathsf{LLF}_{\mathscr{P}}$, was introduced, by a team including the last two authors, to provide a conceptual framework for integrating different proof development tools, thus allowing for *external evidence* and for *postponing, delegating*, or *factoring-out* side conditions. In particular, $\mathsf{LLF}_{\mathscr{P}}$ allows for *reducing* the number of times a *proof-irrelevant* check is performed. In this paper we give a shallow, actually *definitional*, implementation of $\mathsf{LLF}_{\mathscr{P}}$ in `Coq`, *i.e.* we use `Coq` both as host framework and oracle for $\mathsf{LLF}_{\mathscr{P}}$. This illuminates the principles underpinning the mechanism of *Lock-types* and also suggests how to possibly extend `Coq` with the features of $\mathsf{LLF}_{\mathscr{P}}$. The derived proof editor is then put to use for developing case-studies on an emerging paradigm, both at logical and implementation level, which we call *fast and loose reasoning* following Danielsson et *alii* [6]. 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. Typical examples are branch-prediction in CPUs and optimistic concurrency control.

1 Introduction

The *Lax Logical Framework* $\mathsf{LLF}_{\mathscr{P}}$ is a conservative extension of LF. It was introduced in [11] with the goal of *factoring-out, postponing*, or *delegating* to external tools the verification of those time-consuming judgments, which are "morally" *proof-irrelevant*. This system was the final step of a series of papers stemming from [7, 8], aiming at integrating different sources of evidence in a unique Logical Framework. Evidence may derive more conveniently, in effect, from special-purpose external proof search tools, external oracles, or even alternative, non-apodictic, epistemic sources, *e.g.* explicit computations according to the Poincaré Principle [2], diagrams, or just physical analogies. The $\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\cdot]$ constructor was introduced as the appropriate type constructor for expressing *inhabitability up-to*. It turned out to be smoothly expressible as a monad, see [11], for details.

In this paper, we capitalize in particular on that feature of $\mathsf{LLF}_{\mathscr{P}}$ which allows for *postponing* the checking of *proof-irrelevant* side-conditions, in order to streamline formal reasoning according to an emerging paradigm both at logical and at implementation level. We call this paradigm, "fast and loose reasoning", following [6]. 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 *näive* Set Theory or in programming, based on conjecturing and introducing blanket assumptions, to be checked or formalized later, see *e.g.* [3, 9]. At the level of implementations natural examples of this paradigm occur both in *computer architecture* and *concurrency control*, *e.g.*

[*]Work supported by the Italian departmental research project "LambdaBridge" (D.R.N. 37 427/2018 of 03/08/2018, University of Udine).

D. Miller, I. Scagnetto (Eds.)
Logical Frameworks and Meta-Languages:
Theory and Practice 2019 (LFMTP 2019)
EPTCS 307, 2019, pp. 8–23, doi:10.4204/EPTCS.307.3

branch prediction in CPUs and *optimistic concurrency* in distributed systems [14]. 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.

The case studies in $\mathsf{LLF}_{\mathscr{P}}$, carried out in this paper, namely *call-by-value λ-calculus* and *branch prediction* for URM machines (see [5]) suggest natural extensions of $\mathsf{LLF}_{\mathscr{P}}$ itself, for expressing nested lock-types. This was already envisaged in [11]. Furthermore, when the predicate in the lock-type is decidable, the case-study on branch prediction suggests to consider the encoding of alternatives as a sort of *sum type*. We briefly sketch how to generalize these extensions of $\mathsf{LLF}_{\mathscr{P}}$ to a full algebra of predicates.

In order to prototype quickly an implementation of $\mathsf{LLF}_{\mathscr{P}}$ which supports mechanized proof search, we implement a *shallow* encoding of $\mathsf{LLF}_{\mathscr{P}}$ in the `Coq` proof assistant. "Shallow" in this context means that we delegate as much as possible the mechanics of $\mathsf{LLF}_{\mathscr{P}}$ to the metalanguage of the host system. Actually the lock-types are rendered by a `Coq Definition`. This is quite interesting in itself, both in exposing the principles underpinning lock-types as well as the bearing it has on proving that predicates are *well-behaved*, and conversely, by suggesting how to extend `Coq` with a Lock constructor.

The authors express their gratitude to Dr. Ivan Scagnetto for many inspiring discussions and comments. They also thank the anonymous referees for their helpful suggestions.

In Section 2 we recap $\mathsf{LLF}_{\mathscr{P}}$. In Section 3 we give the implementation in `Coq`. In the two following sections we briefly outline in $\mathsf{LLF}_{\mathscr{P}}$ paradigmatic applications: call-by-value λ-calculus and branch prediction for URM machines [5]. In Section 6 we outline possible extensions of the Lock constructor. We briefly discuss future directions in Section 7. The web appendix of the paper is online at [1].

2 The $\mathsf{LLF}_{\mathscr{P}}$ logical framework

In this section, following the standard pattern and conventions of [7], we introduce the syntax and the rules of $\mathsf{LLF}_{\mathscr{P}}$, see [11] for more details. In Figure 1, we give the syntactic categories of $\mathsf{LLF}_{\mathscr{P}}$, namely signatures, contexts, kinds, families (*i.e.* types) and objects (*i.e.* terms). The language is essentially that of classical LF [7], to which we add the *lock types* constructor ($\mathscr{L}$) for building types of the shape $\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]$, where $\mathscr{P}$ is a predicate on typed judgments. Correspondingly, at the object level, we introduce the lock *constructor* ($\mathscr{L}$) and the unlock *destructor* ($\mathscr{U}$). The intended meaning of the $\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\cdot]$ constructor is that of a *logical filter* expressing inhabitability "up-to" the verification of $\mathscr{P}(N{:}\sigma)$.

The rules for the main one-step $\beta\mathscr{L}$-reduction, which combines the standard β-reduction with the novel $\mathscr{L}$-reduction (behaving as a lock-releasing mechanism, erasing the $\mathscr{U}$-$\mathscr{L}$ pair in a term of the form $\mathscr{U}^{\mathscr{P}}_{N,\sigma}[\mathscr{L}^{\mathscr{P}}_{N,\sigma}[M]]$) appear in Figure 2. The rules for one-step closure under context for kinds, families, objects are collected in Figures 3, 4, 5, respectively. We denote the reflexive and transitive closure of $\rightarrow_{\beta\mathscr{L}}$ by $\twoheadrightarrow_{\beta\mathscr{L}}$. Hence, $\beta\mathscr{L}$-definitional equality is defined in the standard way, as the reflexive, symmetric, and transitive closure of $\beta\mathscr{L}$-reduction on kinds, families, objects, as illustrated in Figure 6.

Following the standard specification paradigm of Constructive Type Theory, we define lock-types using *introduction*, *elimination*, and *equality rules*. Namely, see Figure 7, we introduce a lock-*constructor* for building objects $\mathscr{L}^{\mathscr{P}}_{N,\sigma}[M]$ of type $\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho]$, via the *introduction rule* $(O{\cdot}Lock)$. Correspondingly, we introduce an unlock-*destructor* $\mathscr{U}^{\mathscr{P}}_{N,\sigma}[M]$ 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 $\mathscr{L}^{\mathscr{P}}_{N,\sigma}[\cdot]$, we need to introduce in $\mathsf{LLF}_{\mathscr{P}}$ also the rule $(O{\cdot}Top{\cdot}Unlock)$, which allows for the elimination of the lock-type constructor if the predicate $\mathscr{P}$ is verified, possibly *externally*. Figure 7 shows the full

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

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

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

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

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

Figure 3: $\beta\mathscr{L}$-closure-under-context for kinds

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

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

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

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

Figure 4: $\beta\mathscr{L}$-closure-under-context for families

typing system of LLF$_{\mathscr{P}}$. All *type equality rules* of LLF$_{\mathscr{P}}$ use as notion of conversion $\beta\mathscr{L}$-definitional equality.

One may wonder why the rule $(O{\cdot}Top{\cdot}Unlock)$ is not enough and a $(O{\cdot}Guarded{\cdot}Unlock)$-rule is called for. First of all, releasing a locked term, *i.e.* checking a proof-irrelevant side condition is precisely what slows down a derivation. Ultimately we need an external evaluation or to query an external oracle (possibly more than once for the same property) obtaining a positive answer. Moreover, properties under lock are usually not essential to the main thrust of the proof, because they are *proof-irrelevant* and one would like to be free to proceed with the main argument, postponing, as much as possible, the verification of "details". This is precisely the spirit of the "fast and loose" reasoning paradigm [6]. Namely, when

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

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

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

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

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

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

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

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

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

we reach a given stage of a proof development where we are not able, or we do not want to waste time, to verify a side-condition, we may want to *postpone* such a task, unlock immediately the given term, and proceed with the proof. The $(O{\cdot}Guarded{\cdot}Unlock)$-rule allows us to realize exactly this. The external lock-type of the term 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 $\mathscr{P}$ affect reductions in $\mathsf{LLF}_{\mathscr{P}}$, 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 $\mathsf{LLF}_{\mathscr{P}}$. We introduce, therefore, the following crucial definition, where α is shorthand for the "conclusion" of a judgment.

Definition 1 (Well-behaved predicates, [10]) *A finite set of predicates* $\{\mathscr{P}_i\}_{i\in I}$ *is* well-behaved *if each* $\mathscr{P}$ *in the set satisfies the following conditions:*

1. ***Closure under signature and context weakening and permutation:***

Signature rules

$$\frac{}{\emptyset\ \mathsf{sig}}\ (S{\cdot}Empty)$$

$$\frac{\vdash_\Sigma K \quad a \notin \mathsf{Dom}(\Sigma)}{\Sigma, a{:}K\ \mathsf{sig}}\ (S{\cdot}Kind)$$

$$\frac{\vdash_\Sigma \sigma{:}\mathsf{Type} \quad c \notin \mathsf{Dom}(\Sigma)}{\Sigma, c{:}\sigma\ \mathsf{sig}}\ (S{\cdot}Type)$$

Context rules

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

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

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

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

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

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

$$\frac{\Gamma \vdash_\Sigma \rho : \mathsf{Type} \quad \Gamma \vdash_\Sigma N : \sigma}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho] : \mathsf{Type}}\ (F{\cdot}Lock)$$

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

Object rules

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

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

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

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

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

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

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

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

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

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

   (a) *If* Σ *and* Ω *are valid signatures such that* $\Sigma \subseteq \Omega$ *and* $\mathscr{P}(\Gamma \vdash_\Sigma \alpha)$*, then* $\mathscr{P}(\Gamma \vdash_\Omega \alpha)$.
   (b) *If* Γ *and* Δ *are valid contexts such that* $\Gamma \subseteq \Delta$ *and* $\mathscr{P}(\Gamma \vdash_\Sigma \alpha)$*, then* $\mathscr{P}(\Delta \vdash_\Sigma \alpha)$.

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

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

3 A definitional implementation of $\mathsf{LLF}_{\mathscr{P}}$ in `Coq`

An implementation, from scratch, of the logical framework $\mathsf{LLF}_{\mathscr{P}}$ in a functional language, would definitely be particularly efficient, and has indeed been attempted successfully as far as *proof checking*, by Vincent Michielini at ENS Lyon [15]. But in order to provide a rapid prototyping of a full-fledged *proof development environment* for $\mathsf{LLF}_{\mathscr{P}}$, we prefer to capitalize on the existing proof-assistant `Coq`. This could be done very easily, albeit indirectly, using `Coq` as a *logical metalanguage* by giving an encoding of $\mathsf{LLF}_{\mathscr{P}}$ in `Coq`. But we do not need a "deep" encoding of $\mathsf{LLF}_{\mathscr{P}}$'s syntactic categories and related judgments, since we are not interested in reasoning on $\mathsf{LLF}_{\mathscr{P}}$'s metatheory. Our encoding could be, actually "should be", as "shallow" as possible so that we may be able to delegate to `Coq`'s metalanguage not only all of $\mathsf{LLF}_{\mathscr{P}}$ metalanguage, but moreover, reduce *inhabitation-search* in $\mathsf{LLF}_{\mathscr{P}}$ to *proof-search* in `Coq`.

We achieve this by exploiting the fact that `Coq` is a conservative extension of the dependent constructive type theory of LF [7] which underpins the type system of $\mathsf{LLF}_{\mathscr{P}}$, [10]. We simulate/implement, therefore, in `Coq` the mechanism of lock-types, and use `Coq` both as the host system and as the oracle for external propositions. This yields a *definitional* encoding of $\mathsf{LLF}_{\mathscr{P}}$ 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.

In practice, therefore, $\mathsf{LLF}_{\mathscr{P}}$ *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`. We will explain below why it is convenient, although not necessary, to use both. Hence *type families* are rendered as `Coq` sets or propositions and *objects* as their inhabitants. Remarkably, we need to implement *only* the lock constructor for families, as follows:

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

Families are therefore typed by `Prop` and objects by families, with the exception of the family involved in the definition of the predicate $\mathscr{P}$, which is typed by `Set`. This is what makes possible, in using `Coq` as the oracle, to take full advantage, in defining the external predicates of $\mathsf{LLF}_{\mathscr{P}}$, of its logical strength in terms of (co)inductive datatypes and (co)recursive functions.

In a nutshell, the gist of the previous definition is to represent the locking of families in $\mathsf{LLF}_{\mathscr{P}}$ by the Π-type:

$$\ulcorner \mathscr{L}^{\mathscr{P}}_{N,\sigma}[\rho] \urcorner \qquad \rightsquigarrow \qquad \Pi_{x:\mathscr{P}(\ulcorner N \urcorner)} \ulcorner \rho \urcorner$$

This encoding might appear weak, but actually it permits us to develop formal proofs "under $\mathscr{P}$", just by "`unfold`"ing the `lockF` constructor when it appears in the goal. As a consequence, somewhat surprisingly, our `Definition` is sufficient to derive *all* the typing rules of $\mathsf{LLF}_{\mathscr{P}}$ that involve lock-types as `Coq`'s Lemmas:

- lock-introduction (see rule $(O{\cdot}Lock)$ in Fig. 7) 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. 7) 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. 7) may be rendered in several equivalent ways, which we have experimented with in our work. In the end, we have chosen to rephrase it in a way where the `lockF` constructor appears "`unfold`"ed in the conclusion of the rule, to support a more flexible management of proofs. Notice, in particular, how the rule is encoded 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)`. We will comment further on this rule in the following sections which deal with applications:

```
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.
```

In conclusion we have achieved an encoding of LLF$_{\mathscr{P}}$ through a simple `Definition` in `Coq`, see [1]. As pointed out earlier this does not support the full strength of LLF$_{\mathscr{P}}$, in that predicates are restricted to `Coq`-definable terms of some type which eventually maps into `Prop`. Apart from this restriction, however, since `Coq` is a conservative extension of LF, the implementation is obviously faithful with respect to all the rules of LLF$_{\mathscr{P}}$.

We could give a slightly deeper implementation which, following [11], would yield a more perspicuous rendering of the monadic nature of Locks.

4 Call-by-value λ-calculus

In this section we test our implementation of LLF$_{\mathscr{P}}$ 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. In the literature there are many ways of encoding this system. We use the signature given in [11], because it illustrates the flexibility of LLF$_{\mathscr{P}}$ in capitalizing on Higher Order Abstract Syntax (HOAS) when considering *bound* variables, while retaining the ordinary way of referring to *free* variables. We proceed then to experiment with it in LLF$_{\mathscr{P}}$ using the `Coq` implementation introduced in Section 3.

The well-known abstract syntax of λ-calculus is given by: $M,N ::= x \mid M\ N \mid \lambda x.M$. We will model *free* variables in this object language as constants in LLF$_{\mathscr{P}}$. *Bound* variables will be 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 2 below). On the other hand, the λ-term $\lambda x.x$ (in which the variable is obviously bound) is encoded by $\vdash_\Sigma$ `(lam` λ`x:term.x)`.

We introduce therefore the following signature:

Definition 2 (LLF$_\mathcal{P}$ **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:

Definition 3 (Call-by-value equational theory)

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

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

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

where values are either variables or abstractions.

Accordingly, we extend the signature of Definition 2 as follows:

Definition 4 (LLF$_\mathcal{P}$ **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)
```

$\texttt{betav:}\quad \Pi\texttt{M:term} \to \texttt{term}.\ \Pi\texttt{N:term}.\ \mathcal{L}^{\mathrm{Val}}_{\texttt{N,term}}[\texttt{eq (app (lam M) N) (M N)}]$

$\texttt{csiv:}\quad \Pi\texttt{M,N:term} \to \texttt{term}.\ (\Pi\texttt{x:term}.\ \mathcal{L}^{\mathrm{Val}}_{\texttt{x,term}}[\texttt{eq (M x) (N x)}]) \to \texttt{eq (lam M) (lam N)}$

where the predicate $\mathit{Val}\,(\Gamma \vdash_{\Sigma_v} \texttt{N:term})$ *holds if and only if* `N` *is either an abstraction or a variable (i.e. a term of the shape* `(free i)`*).*

Notice how, in Definition 4, LLF$_\mathcal{P}$'s *lock-types* permit us to model the (β_v) and (ξ_v) rules: the former holds “up-to” the verification of $\mathit{Val}\,(\Gamma \vdash_{\Sigma_v} \texttt{N:term})$, while the latter depends, in turn, on a locked premise.

We now proceed to represent the above signature in the `Coq` editor for LLF$_\mathcal{P}$ presented in Section 3. Then we use such a formalization to carry out a simple interactive proof. The full code appears in the on-line appendix, see [1].

First, we declare the new kind of terms (typed by `Set`) and their “constructors”, by exploiting the built-in representation of natural numbers, which lives in `Set`:

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

Then, we model the predicate Val in `Coq`, since the oracle role is played by the host framework:

```
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 formalization of the intended concept, thereby giving evidence also, that the predicate originally used in the lock is well-behaved.

Finally, we encode the call-by-value equational theory, by means of a predicate (*i.e.* 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, as these would go beyond LLF$_{\mathscr{P}}$’s expressivity, but we rely on that part of Coq metalanguage which is shared with LLF$_{\mathscr{P}}$. We do not use Coq inductive types for encoding terms because we exploit full Higher Order Abstract Syntax (HOAS). We could have used *weak* HOAS to deal with variables but we prefer to stay minimal and avoid exotic terms.

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 LLF$_{\mathscr{P}}$ 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 point out the practical value of the Coq editor introduced in this paper, we conclude the section with the formal proof of the simple equation $\lambda x.\, z\, ((\lambda y.y)\, x) = \lambda x.\, z\, x$. 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. Please notice the power of the $(O{\cdot}Guarded{\cdot}Unlock)$ rule, which allows us to apply the rules of the Σ_{v} signature (in this case, the eq_app rule), “under Val”, *i.e.* the latter can handle even premises which are locked [1]:

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{z{:}term \vdash_{\Sigma_v} eq(z,z)}\,(\text{refl})
      \qquad
      \dfrac{}{x{:}term \vdash_{\Sigma_v} \mathscr{L}^{Val}_{x,term}[eq(app(lam(\lambda y{:}term.\,y),x),x)]}\,(\text{betav})
    }{z,x{:}term \vdash_{\Sigma_v} \mathscr{L}^{Val}_{x,term}[eq(app(z,app(lam(\lambda y{:}term.\,y),x)),app(z,x))]}\,(\text{eq\_app via } O{\cdot}G{\cdot}U,\ O{\cdot}L)
  }{z{:}term \vdash_{\Sigma_v} \forall x{:}term.\ \mathscr{L}^{Val}_{x,term}[eq(app(z,app(lam(\lambda y{:}term.\,y),x)),app(z,x))]}
}{z{:}term \vdash_{\Sigma_v} eq(\lambda x{:}term.\ app(z,app(lam(\lambda y{:}term.\,y),x)),\lambda x{:}term.\ app(z,x))}\,(\text{csiv})
$$

We conclude by remarking that using the Coq editor of LLF$_{\mathscr{P}}$ we may accomplish the above goal without having to exhibit the full proof term beforehand, as we had to in [11], because we can now build it interactively and incrementally, via Coq’s tactics.

[1]Note that in the following proof tree we shorten $(O{\cdot}Guarded{\cdot}Unlock)$ to $(O{\cdot}G{\cdot}U)$ and $(O{\cdot}Lock)$ to $(O{\cdot}L)$ for saving space.

5 Branch prediction

In computer architecture, a *branch predictor* is a construct that tries to guess which branch the control will exit, *e.g.* in an `if-then-else`, before the result of the test is actually known. Such a construct is convenient when the evaluation of the test is so much more time demanding w.r.t. executing the other instructions, that the time lost, when having to backtrack because the guess was wrong, is significantly compensated by the speed-up which is achieved, when the guess is correct.

In this section we model the behavior of such a structure in $\mathsf{LLF}_{\mathcal{P}}$ in the case of the *Unlimited Register Machine (URM)*, a simple universal model of computation popularized by Cutland [5].

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:

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

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

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

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:

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

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 [4] for a coinductive approach to diverging computations).

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

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

To introduce an $\mathsf{LLF}_{\mathcal{P}}$ signature, for the URM machine, we need first to encode infinite stores and non-structured programs. Both datatypes are handled by mimicking lists.

Definition 5 (LLF$_{\mathcal{P}}$ **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). As far as stores, we use the nil-like `zeros` constructor which represents an infinite sequence of `0` values. Stores may be updated on demand via the cons-like `cs` constructor. We encode programs, similarly, as lists of instructions in `ins`, with the addition of `Ht`, which represents the $Halt$ instruction motivated above.

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. LLF$_{\mathcal{P}}$'s *lock-types* allow us to model faithfully also the execution of a "branch prediction" version of this semantics, by postponing the double access to the store and test required by J, which is a slow instruction. Lock-types permit to carry out the double 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.

Definition 6 (LLF$_{\mathcal{P}}$ **signature for 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 (Z i) → zero s i t → step P n s (S n) t
```

$$\begin{array}{lll}
\texttt{sJt} & : & \Pi\texttt{P:pgm}.\,\Pi\texttt{n,i,j,k:nat}.\,\Pi\texttt{s:store}.\\
& & \texttt{fetch P n (J i j k)} \to \mathscr{L}^{\texttt{Eq}}_{\langle\texttt{s,i,j}\rangle,\texttt{T}}[\texttt{step P n s k s}]\\
\texttt{sJf} & : & \Pi\texttt{P:pgm}.\,\Pi\texttt{n,i,j,k:nat}.\,\Pi\texttt{s:store}.\\
& & \texttt{fetch P n (J i j k)} \to \mathscr{L}^{\texttt{Neq}}_{\langle\texttt{s,i,j}\rangle,\texttt{T}}[\texttt{step P n s (S n) s}]\\
\texttt{sTr} & : & \Pi\texttt{P:pgm}.\,\Pi\texttt{n,m,q:nat}.\,\Pi\texttt{s,t,u:store}.\\
& & \texttt{step P n s m t} \to \texttt{step P m t q u} \to \texttt{step P n s q u}\\
\texttt{e0} & : & \Pi\texttt{P:pgm}.\,\Pi\texttt{n:nat}.\,\Pi\texttt{s:store}.\,\texttt{fetch P n halt} \to \texttt{eval P n s s}\\
\texttt{e1} & : & \Pi\texttt{P:pgm}.\,\Pi\texttt{n,m:nat}.\,\Pi\texttt{s,t:store}.\\
& & \texttt{step P n s m t} \to \texttt{fetch P m halt} \to \texttt{eval P n s t}
\end{array}$$

where $\texttt{Eq}(\Gamma \vdash_\Sigma \langle\texttt{s,i,j}\rangle : \texttt{T})$ *holds iff* $s(i){=}s(j)$, *and* $\texttt{Neq}(\Gamma \vdash_\Sigma \langle\texttt{s,i,j}\rangle : \texttt{T})$ *iff* $s(i){\neq}s(j)$.

We now handle this second case study via the Coq editor introduced in Section 3. We take advantage of built-in natural numbers and lists to define stores, instructions, and programs (all typed by Set), namely:

```
Definition store: Set := list nat.
Parameter ins: Set. Parameter Ht: ins. ...
Definition pgm: Set := list ins.
```

The input to the oracle, *i.e.* a store and a pair of locations, is defined as an inductive type T of triples and corresponding projection functions. Memory access is realized 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 formalized in Coq by using these datatypes, as follows:

```
Inductive T: Set := triple: store -> nat -> nat -> T.
Definition pr1 (x:T): store := match x with triple s i j => s end. ...
Definition s_nth (s:store) (n:nat): nat := nth n s 0.
Definition Eq := 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 6, it is available at [1]):

```
Parameter step: pgm -> nat -> store -> nat -> 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 eval: pgm -> nat -> store -> store -> Prop. ...
```

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 $s \triangleq 1{:}1{:}zeros$ and $t \triangleq 0{:}1{:}zeros$. Then we have the fragment derivation[2]:

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

[2]In the present and the next derivations we display $\mathsf{LLF}_{\mathscr{P}}$'s types without the proof terms because these are synthesized by the editor.

In this proof tree there is a limited amount of parallelism, 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 left-hand J reduction:

$$\dfrac{\dfrac{\dfrac{P(1){=}J(0,1,0)}{\mathscr{L}^{Eq}_{\langle s,0,1\rangle,T}[\langle 1,s\rangle \rightsquigarrow^P \langle 0,s\rangle]} \quad \dfrac{P(0){=}Z(0)}{\langle 0,s\rangle \rightsquigarrow^P \langle 1,t\rangle}}{\mathscr{L}^{Eq}_{\langle s,0,1\rangle,T}[\langle 1,s\rangle \rightsquigarrow^P \langle 1,t\rangle]}\text{(sTr via O·Guarded·Unlock)} \qquad Eq(\langle s,0,1\rangle)}{\langle 1,s\rangle \rightsquigarrow^P \langle 1,t\rangle}\text{(O·Top)}$$

The $Eq(\langle s,0,1\rangle)$ check can now be delayed, and carried out independently w.r.t. the main reduction. The $(O{\cdot}Guarded{\cdot}Unlock)$ 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 LLF$_{\mathscr{P}}$ 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 predictions, we can define introduction and elimination rules of the following shape:

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

$$\dfrac{\mathscr{L}^{\mathscr{P}_1;\mathscr{P}_2}_{\langle \vec{x_1}\rangle;\langle \vec{x_2}\rangle,T}[\langle n,s\rangle \rightsquigarrow^P \langle m,t\rangle] \quad \mathscr{P}_1(\vec{x_1})}{\mathscr{L}^{\mathscr{P}_2}_{\langle \vec{x_2}\rangle,T}[\langle n,s\rangle \rightsquigarrow^P \langle m,t\rangle]}\ (\mathscr{P}_{-1}) \qquad \dfrac{\mathscr{L}^{\mathscr{P}_1;\mathscr{P}_2}_{\langle \vec{x_1}\rangle;\langle \vec{x_2}\rangle,T}[\langle n,s\rangle \rightsquigarrow^P \langle m,t\rangle] \quad \mathscr{P}_2(\vec{x_2})}{\mathscr{L}^{\mathscr{P}_1}_{\langle \vec{x_1}\rangle,T}[\langle n,s\rangle \rightsquigarrow^P \langle m,t\rangle]}\ (\mathscr{P}_{-2})$$

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

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

We will focus on these envisaged extensions and corresponding encodings in the following Section 6. We anticipate here that the “composition” of predicates can be dealt with via lock nesting, that is, we manage elimination rules in the form $\mathscr{P}_-$ by means of the $(O{\cdot}Top{\cdot}Unlock)$ rule (*i.e.* `Coq`’s `top_unlock` lemma), and we manage introduction rules such as $\mathscr{P}_+$ by “`unfold`”ing the `lockF` constructor.

In conclusion, in this section, we have shown how LLF$_{\mathscr{P}}$ can naturally accommodate computations running in parallel asynchronously, as it happens when performing branch prediction.

6 Towards an algebra of locks

In the previous section we have informally argued about possible extensions of $\mathsf{LLF}_{\mathscr{P}}$ in order to accommodate logical combinations of predicates in locks. In fact, the branch prediction case study has pointed out on the one hand the need of "conjunctions" of lock predicates (in order to augment the parallelism of execution), on the other hand the possibility of managing "disjunctions" of lock predicates (to represent in a compact way pairs of mutually exclusive computations). Therefore, we would like to handle both conjunctions and disjunctions of lock predicates, according to the following introduction rules:

$$\frac{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}_1}_{N_1,\sigma_1}[M] : \mathscr{L}^{\mathscr{P}_1}_{N_1,\sigma_1}[\rho] \qquad \Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}_2}_{N_2,\sigma_2}[M] : \mathscr{L}^{\mathscr{P}_2}_{N_2,\sigma_2}[\rho]}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}_1 \wedge \mathscr{P}_2}_{\langle N_1,N_2\rangle,\langle\sigma_1,\sigma_2\rangle}[M] : \mathscr{L}^{\mathscr{P}_1 \wedge \mathscr{P}_2}_{\langle N_1,N_2\rangle,\langle\sigma_1,\sigma_2\rangle}[\rho]} \; (O{\cdot}Lock{\cdot}\wedge)$$

$$\frac{\Gamma \vdash_\Sigma M_1 : \rho_1 \qquad \Gamma \vdash_\Sigma M_2 : \rho_2}{\Gamma \vdash_\Sigma \mathscr{L}^{\mathscr{P}_1 \oplus \mathscr{P}_2}_{N,\sigma}[[M_1,M_2]] : \mathscr{L}^{\mathscr{P}_1 \oplus \mathscr{P}_2}_{N,\sigma}[\rho_1 \oplus \rho_2]} \; (O{\cdot}Lock{\cdot}\oplus)$$

where $[M_1,M_2]$ denotes the "bookkeeping" of the terms M_1 and M_2 of types ρ_1 and ρ_2, respectively, into a new *binary record* structure. Indeed, $\rho_1 \oplus \rho_2$ represents the record type whose components are of types ρ_1 and ρ_2, and $\mathscr{P}_1$ and $\mathscr{P}_2$ are two *mutually exclusive* predicates. The $\oplus$ type is eliminated as follows:

$$\frac{\Gamma \vdash_\Sigma M : \mathscr{L}^{\mathscr{P}_1 \oplus \mathscr{P}_2}_{N,\sigma}[\rho] \qquad \mathscr{P}_1(\Gamma \vdash_\Sigma N : \sigma) \qquad \mathscr{P}_1 \text{ and } \mathscr{P}_2 \text{ are mutually exclusive}}{\Gamma \vdash_\Sigma (\mathscr{U}^{\mathscr{P}_1 \oplus \mathscr{P}_2}_{N,\sigma}[M])_l : (\rho)_l} \; (O{\cdot}Lock{\cdot}\oplus_l)$$

$$\frac{\Gamma \vdash_\Sigma M : \mathscr{L}^{\mathscr{P}_1 \oplus \mathscr{P}_2}_{N,\sigma}[\rho] \qquad \mathscr{P}_2(\Gamma \vdash_\Sigma N : \sigma) \qquad \mathscr{P}_1 \text{ and } \mathscr{P}_2 \text{ are mutually exclusive}}{\Gamma \vdash_\Sigma (\mathscr{U}^{\mathscr{P}_1 \oplus \mathscr{P}_2}_{N,\sigma}[M])_r : (\rho)_r} \; (O{\cdot}Lock{\cdot}\oplus_r)$$

where $(M)_l$, respectively $(M)_r$, represents the left, respectively right, component of the binary record term M, and $(\rho)_l$, respectively $(\rho)_r$, represents the left, respectively right, component of the binary record type ρ. Due to lack of space, we omit the obvious elimination rules for the conjunction of lock predicates, and their nested equivalents.

Given our shallow encoding of $\mathsf{LLF}_{\mathscr{P}}$ in `Coq`, such derived rules can be rendered very easily introducing two new definitions:

```
Definition lockF_and :=
  fun s1: Set => fun N1: s1 => fun P1: s1 -> Prop =>
  fun s2: Set => fun N2: s2 => fun P2: s2 -> Prop =>
  fun r: Prop => forall x: P1 N1, forall y: P2 N2, r.

Definition lockF_xor :=
  fun s: Set => fun N: s => fun P1: s -> Prop => fun P2: s -> Prop =>
  fun r1 r2: Prop => xor (P1 N) (P2 N) ->
  xor (forall x: P1 N, r1) (forall y: P2 N, r2).
```

where mutual exclusion is encoded as follows:

```
Definition xor := fun A B:Prop => (A /\ not B) \/ (not A /\ B).
```

So doing, we can formally prove the following lemmata:

```
Lemma lock_and: forall s1: Set, forall N1: s1, forall P1: s1 -> Prop,
  forall s2: Set, forall N2: s2, forall P2: s2 -> Prop,
  forall r: Prop, forall x: P1 N1, forall y: P2 N2,
  lockF_and s1 N1 P1 s2 N2 P2 r <-> lockF s1 N1 P1 (lockF s2 N2 P2 r).

Lemma lock_xor: forall s: Set, forall N: s,
  forall P1: s -> Prop, forall P2: s -> Prop,
  forall r1 r2: Prop, forall x: P1 N, forall y: P2 N,
  xor (P1 N) (P2 N) ->
  (lockF_xor s N P1 P2 r1 r2 <-> xor (lockF s N P1 r1) (lockF s N P2 r2)).
```

In other words, `lockF_and` is syntactic sugar for lock nesting, while `lockF_xor` reduces to an exclusive disjunction between two `lockF` judgments.

We remark that alternative approaches to the development of an algebra of locks may be pursued.

7 Conclusion

This paper provides two contributions to the development of the Lax Logical Framework LLF$_{\mathscr{P}}$, introduced in [11]. The first contribution is a very "shallow", actually definitional, implementation of LLF$_{\mathscr{P}}$ in Coq. This produces immediately a proof development environment, supporting mechanized proof search, for a version of LLF$_{\mathscr{P}}$ in which the predicates used in locks are Coq-definable. The second contribution shows how the feature of LLF$_{\mathscr{P}}$, which allows for postponing the evaluation of an ultimately proof-irrelevant side-condition, can model naturally instances of the emerging paradigm of "fast and loose" reasoning, [6]. Actually, we can say that the philosophy of locks amounts to applying such a paradigm at a metatheoretic level. Both contributions are essential in the development of the case study reported in the paper concerning *branch prediction*, which is a form of "fast and loose" evaluation, of the URM machine.

We do not provide a formal adequacy theorem for the branch prediction case study. We are currently working on it as well as other "fast and loose" reasoning patterns. This is problematic however, since they are not fully spelled out in the literature. We believe that adequacy would be very significant because it would provide a thorough understanding of the heuristics underpinning such paradigms.

Both contributions appear to be rather fruitful. The definitional implementation suggests how to rapidly prototype editors for other calculi such as CLLF$_{\mathscr{P}?}$, see [11], or extensions of LLF$_{\mathscr{P}}$ which support an algebraic structure of locks as outlined in Section 6.

More case studies need to be developed. For lack of space, we could not even outline here another seminal case-study, namely that on *optimistic concurrency control*, which is another important example of the "fast and loose" paradigm applied to non-interference issues. Another important case study related to the "fast and loose" philosophy which we intend to develop is that of Fitch-Prawitz consistent Set Theory, [9]. This is the natural counterpart of the naïve Set Theory used in developing ordinary mathematics.

It would be interesting to address the issue of extending full-fledged locks to Coq itself.

Finally, we intend to explore how to prototype an alternate editor for LLF$_{\mathscr{P}}$ using the MMT UniFormal Framework of F. Rabe, [17].

References

[1] Fabio Alessi et alii (2019): *The Web appendix of this paper*. Available at `https://users.dimi.uniud.it/~alberto.ciaffaglione/LLFP/LFMTP-19.tar.gz`.

[2] Henk Barendregt & Erik Barendsen (2002): *Autarkic Computations in Formal Proofs*. J. Autom. Reasoning 28(3), pp. 321–336, doi:10.1023/A:1015761529444.

[3] Chris Casinghino, Vilhelm Sjöberg & Stephanie Weirich (2014): *Combining proofs and programs in a dependently typed language*. In Jagannathan & Sewell [13], pp. 33–46, doi:10.1145/2535838.2535883.

[4] Alberto Ciaffaglione (2011): *A coinductive semantics of the Unlimited Register Machine*. In Yu & Wang [18], pp. 49–63, doi:10.4204/EPTCS.73.7.

[5] Nigel Cutland (1980): *Computability - An introduction to recursive function theory*, doi:10.1017/CBO9781139171496. Cambridge University Press.

[6] Nils Anders Danielsson, John Hughes, Patrik Jansson & Jeremy Gibbons (2006): *Fast and loose reasoning is morally correct*. In Morrisett & Peyton Jones [16], pp. 206–217, doi:10.1145/1111037.1111056.

[7] Robert Harper, Furio Honsell & Gordon D. Plotkin (1993): *A Framework for Defining Logics*. J. ACM 40(1), pp. 143–184, doi:10.1145/138027.138060. Preliminary version in Proc. of LICS'87.

[8] Furio Honsell, Marina Lenisa & Luigi Liquori (2007): *A Framework for Defining Logical Frameworks*. Volume in Honor of G. Plotkin, Electr. Notes Theor. Comput. Sci. 172, pp. 399–436, doi:10.1016/j.entcs.2007.02.014.

[9] Furio Honsell, Marina Lenisa, Luigi Liquori & Ivan Scagnetto (2016): *Implementing Cantor's Paradise*. In Igarashi [12], pp. 229–250, doi:10.1007/978-3-319-47958-3_13.

[10] Furio Honsell, Marina Lenisa, Ivan Scagnetto, Luigi Liquori & Petar Maksimovic (2016): *An open logical framework*. J. Log. Comput. 26(1), pp. 293–335, doi:10.1093/logcom/ext028.

[11] Furio Honsell, Luigi Liquori, Petar Maksimovic & Ivan Scagnetto (2017): $\mathsf{LLF}_{\mathscr{P}}$*: a logical framework for modeling external evidence, side conditions, and proof irrelevance using monads*. Logical Methods in Computer Science 13(3), doi:10.23638/LMCS-13(3:2)2017.

[12] Atsushi Igarashi, editor (2016): *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*. Lecture Notes in Computer Science 10017, doi:10.1007/978-3-319-47958-3.

[13] Suresh Jagannathan & Peter Sewell, editors (2014): *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM. Available at `http://dl.acm.org/citation.cfm?id=2535838`.

[14] H. T. Kung & John T. Robinson (1981): *On Optimistic Methods for Concurrency Control*. ACM Trans. Database Syst. 6(2), pp. 213–226, doi:10.1145/319566.319567.

[15] Vincent Michielini (2016): $\mathsf{LLF}_{\mathscr{P}}$ *type checker*. Available at `https://github.com/francescodellamorte/llfp-type-checker`.

[16] J. Gregory Morrisett & Simon L. Peyton Jones, editors (2006): *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. ACM. Available at `http://dl.acm.org/citation.cfm?id=1111037`.

[17] Florian Rabe & Michael Kohlhase (2013): *A scalable module system*. Inf. Comput. 230, pp. 1–54, doi:10.1016/j.ic.2013.06.001.

[18] Fang Yu & Chao Wang, editors (2011): *Proceedings 13th International Workshop on Verification of Infinite-State Systems, INFINITY 2011, Taipei, Taiwan, 10th October 2011*. EPTCS 73, doi:10.4204/EPTCS.73.