Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 199 (2008) 129–147
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Meta Linear Logical Framework

Andrew McCreight and Carsten Schürmann[1] ,[2]

Yale University
New Haven, CT, USA

Abstract

Logical frameworks serve as meta languages to represent deductive systems, sometimes requiring special purpose meta logics to reason about the representations. In this work, we describe $\mathcal{L}_\omega^+$, a meta logic for the linear logical framework LLF [6] and illustrate its use via a proof of the admissibility of cut in the sequent calculus for the tensor fragment of linear logic. $\mathcal{L}_\omega^+$ is first-order, intuitionistic, and not linear. The soundness of $\mathcal{L}_\omega^+$ is shown.

Keywords: Logical framework, meta language, meta logic, $\mathcal{L}_\omega^+$, linear logical framework

1 Introduction

Logical frameworks are meta languages designed for representing various formal systems prevalent in programming language semantics, logics, and protocol design. By design, a logical framework is foundationally uncommitted, meaning that it is primarily concerned with the way formal systems are represented and not with reasoning about their properties. Logical frameworks have, in this spirit, undergone significant extensions, leaving the design of meta logics far behind. Modern logical frameworks incorporate linear types to model resource awareness (useful when designing programming languages with effects), ordered types (to model formal systems that access resources in a particular order), and even monadic types that capture concurrency.

By separating meta languages from meta logics, we get a quite substantial design space for special purpose meta logics. Each meta logic is tailored toward a particular

[1] This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519 and NSF grant CCR-0325808. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

[2] Email:{`aem,carsten`}`@cs.yale.edu`

doi:10.1016/j.entcs.2007.11.016

logical framework, responding to its requirements, expressiveness and idiosyncrasies, with the sole purpose of formalizing meta theoretic arguments about encodings in the logical framework. A logical framework together with a meta logic defines a meta logical framework. One example of a meta logic is $\mathcal{M}_\omega^+$ [17], designed specifically for the logical framework LF [10]. Conversely, McDowell and Miller [11], have chosen a fixed meta logic and to study how to encode and reason about various meta languages in their system. However, their design is also not immune to change. Non-standard extensions of their first-order meta logic with definitions and natural number induction have become necessary to facilitate reasoning about terms with open parameters [7].

The absence of well-understood meta logics has often been interpreted as a severe impediment to the deployment and acceptance of the technology among researchers and scientists as well as developers and industry. Consequently, the prevalent use of logical framework technology is as a representation language for one particular logic that is then used to describe and reason about the object systems in question. Higher-order logic is a popular candidate used in Isabelle/HOL [13] and Twelf/HOL [1] which have been instrumental in the formal study of programming languages, such as Java [12], hardware verification [9], and protocol verification [14], among other things. Higher-order logic is well-understood, clean, expressive, and when enriched with induction principles a good choice for many applications. However, it limits the ways in which deductive systems can be encoded, and therefore cannot take advantage of the advanced representation technology provided by modern logical frameworks.

In this work, we propose a special purpose meta logic for the linear logical framework LLF [6] which plays the role of a linear meta logical framework. LLF's distinguishing feature over LF is a set of linear operators capable of handling depletable resources. LLF has been successfully employed in representing and experimenting with a variety of security and authentication protocols [3]. Although the theory behind LLF is well-understood, our work is to our knowledge the first research towards a sound meta logic for LLF.

$\mathcal{L}_\omega^+$ extends the meta logic $\mathcal{M}_\omega^+$ for LF developed by the second author [17] into the LLF setting. $\mathcal{L}_\omega^+$ is first-order, intuitionistic, and not linear. Aside from $\top$, it does not define any logical constant symbols. It does however inherit proofs by induction over arbitrary higher-order types without the restrictive positivity condition, including those that take advantage of both linear and intuitionistic assumptions. Furthermore, it supports quantification over LLF contexts. $\mathcal{L}_\omega^+$ can be used to reason about meta properties of languages encoded in LLF. For instance, it should be possible to show type soundness of a simple programming language with references or cut elimination for a linear logic. $\mathcal{L}_\omega^+$ must be dependently typed to state these sorts of properties.

The paper is organized in the following way: in Section 2 we review the linear logical framework LLF and illustrate its representational expressiveness in terms of a sequent calculus for the tensor fragment of linear logic. In Section 3, we present a formal meta logic $\mathcal{L}_\omega^+$ that serves as the formalization of theorems as well as meta

$$\begin{array}{lll}
(\mathit{Kinds}) & K & ::= \text{type} \mid \Pi u{:}A.\ K \\
(\mathit{Types}) & A, B & ::= a \mid A\ M \mid \Pi u{:}A.\ B \mid A \multimap B \mid A \mathbin{\&} B \mid \top \\
(\mathit{Objects}) & M, N & ::= c \mid u \mid \lambda u{:}A.\ M \mid M\ N \mid \hat{\lambda} u{:}A.\ M \mid M\hat{\ }N \\
& & \quad \mid \langle M, N\rangle \mid \pi_1 M \mid \pi_2 M \mid \langle\rangle \\
(\mathit{Signatures}) & \Sigma & ::= \cdot \mid \Sigma, a : K \mid \Sigma, c : A \\
(\mathit{Contexts}) & \Gamma, \Delta & ::= \cdot \mid \Gamma, u : A \\
(\mathit{Substitution}) & \rho & ::= \cdot \mid \rho, M/u
\end{array}$$

Fig. 1. LLF syntax

theoretic proofs. We start by describing the interface between the meta logic and the logic, first by giving extensions to LLF, before describing the meta logic proper and its proof theory. Next, in Section 4, we use as an example the proof of the theorem that cuts are admissible in the previously defined sequent calculus encoding. Then $\mathcal{L}_\omega^+$'s soundness is shown in Section 5, before we conclude in Section 6 and assess results.

2 The Linear Logical Framework LLF

The linear logical framework LLF [6] extends the the logical framework LF [10] with linear resources that may be created, used, or modified. Its feature set supersedes that of LF, supporting dependent types. Every term in LLF reduces to a canonical form. LLF has established itself as an elegant tool for adequate encodings of judgments as types, derivations as objects, and hypothetical judgments as (linear) functions including an elegant treatment of depletable resources.

For example, the well-known derivability judgment for linear classical logic of the form $A_1, \ldots, A_n \Longrightarrow B_1, \ldots, B_n$ can be represented in LLF as a function of the form

$$\text{neg}\ A_1 \ldots \multimap \text{neg}\ A_n \multimap \text{pos}\ B_1 \ldots \multimap \text{pos}\ B_m \to \#.$$

neg and pos are families of types, representing assumptions to the left and right of the $\Longrightarrow$ symbol, respectively, while # is a type that stands for the empty sequent. Encoding lists of assumptions as linear functions instead of making them explicit as lists has several advantages, namely that lookup, consumption, and substitution are directly supported by LLF through variable names, linear application, and β-reduction, which renders encodings of resource oriented formal systems brief, concise, and readable.

LLF borrows its linear operators from linear logic [8] and uses $\beta\eta$ as the underlying notion of definitional equality [5]. Furthermore, it conservatively extends LF. LLF does not provide a dependent linear function space. The syntax for standard LLF [6] is given in Figure 1.

Kinds can either be the kind for types or a dependent product. *Types* can either

$$\frac{}{\Psi; \Gamma; \cdot \rhd c : \Sigma(c)} \qquad \frac{}{\Psi; \Gamma; \cdot \rhd u : \Gamma(u)} \qquad \frac{}{\Psi; \Gamma; u : A \rhd u : A}$$

$$\frac{\Psi; \Gamma, u : A; \Delta \rhd M : B}{\Psi; \Gamma; \Delta \rhd \lambda u{:}A.\ M : \Pi u{:}A.\ B} \qquad \frac{\Psi; \Gamma; \Delta \rhd M : \Pi u{:}A.\ B \quad \Psi; \Gamma; \cdot \rhd N : A}{\Psi; \Gamma; \Delta \rhd M\ N : [\mathrm{id}_{\Gamma;\Delta}, N/u]B}$$

$$\frac{\Psi; \Gamma; \Delta, u : A \rhd M : B}{\Psi; \Gamma; \Delta \rhd \hat{\lambda} u{:}A.\ M : A \multimap B} \qquad \frac{\Psi; \Gamma; \Delta_1 \rhd M : A \multimap B \quad \Psi; \Gamma; \Delta_2 \rhd N : A}{\Psi; \Gamma; \Delta_1, \Delta_2 \rhd M\,\hat{}\,N : B}$$

$$\frac{\Psi; \Gamma; \Delta \rhd M : A \quad \Psi; \Gamma; \Delta \rhd N : B}{\Psi; \Gamma; \Delta \rhd \langle M, N \rangle : A \,\&\, B}$$

$$\frac{\Psi; \Gamma; \Delta \rhd M : A \,\&\, B}{\Psi; \Gamma; \Delta \rhd \pi_1 M : A} \qquad \frac{\Psi; \Gamma; \Delta \rhd M : A \,\&\, B}{\Psi; \Gamma; \Delta \rhd \pi_2 M : B} \qquad \frac{}{\Psi; \Gamma; \Delta \rhd \langle\rangle : \top}$$

Fig. 2. Typing rules of LLF.

$$\frac{}{A \Longrightarrow A}\,\mathsf{ax} \qquad \frac{\Gamma_1 \Longrightarrow C, \Delta_1 \quad \Gamma_2, C \Longrightarrow \Delta_2}{\Gamma_1, \Gamma_2 \Longrightarrow \Delta_1, \Delta_2}\,\mathsf{cut}$$

$$\frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A \otimes B \Longrightarrow \Delta}\,\otimes\mathsf{L} \qquad \frac{\Gamma_1 \Longrightarrow A, \Delta_1 \quad \Gamma_2 \Longrightarrow B, \Delta_2}{\Gamma_1, \Gamma_2 \Longrightarrow A \otimes B, \Delta_1, \Delta_2}\,\otimes\mathsf{R}$$

Fig. 3. Tensor fragment of linear logic

be a type constant, an application, a dependent function type, the linear function type, the additive product type, or the additive unit. *Objects* can either be an object constant, a variable, an intuitionistic function or application, a linear function or application, a linear additive pair or projection, or the constructor for the additive unit. A *signature* binds type and object constants. An LLF *context* is either empty, or a smaller context extended with an object binding.

We write LLF judgments using $\rhd$ to separate assumptions from the rest of the judgment. The meta context Ψ, yet unused, holds meta level assumptions, which we will discuss in Section 3. The form for the object typing judgment is $\Psi; \Gamma; \Delta \rhd M : A$, which states that under the meta assumptions in Ψ, the intuitionistic assumptions in Γ and the linear assumptions in Δ, the object M has type A. Figure 2 defines the static semantics of LLF. Kinds and types must be linearly closed, and thus the judgments that define their validity ($\Psi; \Gamma \rhd K : \mathrm{kind}$ and $\Psi; \Gamma \rhd A : K$) are declared without a linear context.

Throughout the paper we use simultaneous substitutions ρ that are defined simultaneously on the intuitionistic and linear variables. Out of notational convenience, we write $\mathrm{id}_{\Gamma;\Delta}$ for the identity substitution on $\Gamma; \Delta$.

As an example, consider the representation of the tensor fragment of classical

$$\begin{array}{lll} \text{ax} & : & \text{neg } A \multimap \text{pos } A \multimap \#. \\ \text{cut} & : & (\text{pos } C \multimap \#) \multimap (\text{neg } C \multimap \#) \multimap \#. \\ \text{tensorL} & : & (\text{neg } A \multimap \text{neg } B \multimap \#) \multimap (\text{neg } (A \otimes B) \multimap \#). \\ \text{tensorR} & : & (\text{pos } A \multimap \#) \multimap (\text{pos } B \multimap \#) \multimap (\text{pos } (A \otimes B) \multimap \#). \end{array}$$

Fig. 4. Encoding of Figure 3 in LLF

$$\begin{array}{lll} (\mathit{Objects}) & M, N & ::= \ldots \mid n[\rho] \mid \pi_{\mathrm{p}} m \\ (\mathit{Modules}) & m & ::= \alpha \mid \pi_{\mathrm{m}} m \\ (\mathit{Contexts}) & \Gamma, \Delta & ::= \ldots \mid \Gamma, \pi_{\mathrm{p}} m : A \mid \Gamma, \gamma \in \Phi \\ (\mathit{Substitution}) & \rho & ::= \ldots \mid \rho, M/\pi_{\mathrm{p}} m \end{array}$$

Fig. 5. LLF extensions

linear logic depicted in Figure 3. The rules cut and $\otimes$R illustrate how resources on either side of the sequent symbol are distributed as resources to either of the two premisses. A derivation can only then be closed by ax if the left and the right context contain a single formula A. Each inference rule is represented as a constant in LLF as shown in Figure 4. As usual, we omit the leading Π-quantifiers for inferable types. LLF's meta theory guarantees the existence of β-normal, η-long canonical forms [19] used in order to establish the adequacy of this encoding.

3 The Meta Logic $\mathcal{L}_\omega^+$

The meta logic $\mathcal{L}_\omega^+$ provides the syntactic and proof-theoretic means to express properties about encodings in LLF and their respective proofs, should they exist. Following the general philosophy underlying this and other meta logical frameworks [17,2], the elegance and scalability of our approach emerges from the clear distinction between the language of representation and the language for reasoning. The meta logic $\mathcal{L}_\omega^+$'s noteworthy properties include that it is first-order, i.e. only a universal and an existential quantifier are available, minimal, i.e. no other propositional constants but truth can be defined, and non-linear, i.e. $\mathcal{L}_\omega^+$ is an intuitionistic logic designed to reason about linearity.

We first present extensions to LLF that allow our meta logic $\mathcal{L}_\omega^+$ to express properties about LLF objects in Section 3.1. Next, we describe how the meta level deals with LLF contexts, and how the interface there works. The necessary vocabulary having been built, we then discuss the meta logic proper, starting with its syntax and semantics, then moving to the proof theory. The running example will be continued to illustrate the concepts in question.

3.1 *Extensions to LLF*

In a meta logic, we wish to reason abstractly about the existence and form of hypothetical LLF objects. LLF must be extended to allow the inclusion of these objects bound at the meta level. In a closed meta level context, any LLF objects will be standard, as described in the previous section. Figure 5 gives an exact account of these extensions, which are discussed in detail in the following paragraphs. All of LLF's fundamental properties, including conservative extension over LF, type soundness, and the existence of canonical forms remain unchanged under these extensions.

Meta variables

LLF objects may refer to other hypothetical LLF objects whose existence is postulated by the meta logic, usually in form of a universally quantified variable. Those *meta variables*, denoted by n, are bound on the meta level and visible from within LLF terms.

Since meta variables are bound outside of any LLF context, they are given an explicit fixed context (of linear and intuitionistic variables). Consequently, each occurrence of a meta variable n requires an explicit mediating substitution ρ that casts an occurrence of n into the appropriate ambient context. This combination of meta variable and explicit substitution is written as $n[\rho]$. [3]

Context variables

To control the flow of resources inside a meta theoretic proof, the meta level has to communicate to LLF how many resources are available, how many are to be consumed, and which hypothetical objects are consuming which resources. Context variables γ that are declared as part of LLF contexts in Figure 5 communicate this information and stand for slices of LLF contexts (including the intuitionistic and linear part). Within LLF, context variables are virtually invisible. For example, they can neither be consumed, substituted into, nor can they occur inside LLF objects or types. In fact, the only places where they may occur are in the contexts of other hypothetical objects, characterized by the previously described meta variables. Context variables are declared in the context Ψ that is part of the LLF typing judgment described in Figure 2.

Module variables

Meta variables and context variables form the basic interface between LLF and the meta level. This would be sufficient if we only wanted to reason about closed LLF terms. But the goal of the paper is significantly more ambitious than this, i.e. to reason about all higher-order LLF encodings, including those that may very well be open. The meta theoretic view of openness inevitably impacts the LLF level. For reasons that have not been discussed so far (but will be in the next section),

[3] Our extension of LLF with meta variables is similar to a system developed for a different purpose [16], from which we take the syntax for meta variable binders.

$$\begin{array}{lrl}
(\mathit{Module\ Kinds}) & k & ::= \mathrm{sig} \mid \Pi u\!:\!A.\ k \\
(\mathit{Module\ Sigs}) & s & ::= \epsilon \mid \exists u\!:\!A.\ s \mid \lambda u\!:\!A.s \\
(\mathit{Worlds}) & \Phi & ::= s \mid \Phi^* \mid \Phi_1 + \Phi_2 \\
(\mathit{Module\ Contexts}) & \chi & ::= \cdot \mid \chi_1, \chi_2 \mid \gamma \in \Phi \mid \alpha : s \\
(\mathit{Meta\ Contexts}) & \Psi & ::= \cdot \mid \Psi, n :: (\chi \triangleright A) \mid \Psi, \gamma \in (\chi \triangleright \Phi) \mid \Psi, \alpha :: (\chi \triangleright s)
\end{array}$$

Fig. 6. Module context syntax

$$\begin{array}{ll}
\llbracket \cdot \rrbracket & = \cdot \\
\llbracket \chi, \chi' \rrbracket & = \llbracket \chi \rrbracket, \llbracket \chi' \rrbracket \\
\llbracket \gamma \in \Phi \rrbracket & = \gamma \in \Phi \\
\llbracket m : \epsilon \rrbracket & = \cdot \\
\llbracket m : \exists u\!:\!A.\ s \rrbracket & = \pi_{\mathrm{p}} m : A,\ \llbracket \pi_{\mathrm{m}} m : [\pi_{\mathrm{p}} m/u] s \rrbracket
\end{array}$$

Fig. 7. Flattening

the open parameters are grouped into modules, made visible to LLF in the form of *module projections* (such as $\pi_{\mathrm{p}}(\alpha)$, $\pi_{\mathrm{p}}(\pi_{\mathrm{m}}(\alpha))$, and $\pi_{\mathrm{p}}(\pi_{\mathrm{m}}(\pi_{\mathrm{m}}(\alpha)))$) of module variables α. These projections behave like any other LLF variables, and are thus subject to declaration in an LLF context and to instantiation by a substitution, as described in Figure 5.

3.2 Module contexts and worlds

We have thus far discussed the required extensions to LLF from the point of view of LLF. For the remainder of this section, we switch our point of view to that of the meta level. In the full generality of higher-order encodings, inductive arguments often require reasoning under λ-binders, which is tantamount to reasoning about open objects. The argument often calls for more than one hypothesis that seem unrelated at first glance. It is the simultaneous presence of these hypotheses that make a base case go through, or justify the application of a previously proved lemma.

Thus, instead of dealing with individual parameters, the meta level deals with collections of related LF parameters called *modules*. Modules are classified by *module signatures* s. A module is either empty (classified by the signature ϵ) or a pair, where the first element is an LLF parameter and the second element is another module (classified by the signature $\exists u : A.\ s$). The final possible classification, $\lambda u\!:\!A.s$, denotes a module of signature s parameterized by an LLF object of type A. Module kinds k are used to keep track of whether a module is fully instantiated (sig) or parameterized ($\Pi u\!:\!A.\ k$).

Modules themselves remain abstract, so no concrete module constructors are needed. Instead, a module can consist of a variable (α), or the second element of

$$\begin{aligned}
\lfloor \cdot \rfloor_A^P &= \cdot \\
\lfloor \Gamma, u : B \rfloor_A^P &= \begin{cases} \lfloor \Gamma \rfloor_A^P, u : B & \text{if } P(B, A) \\ \lfloor \Gamma \rfloor_A^P & \text{otherwise} \end{cases} \\
\lfloor \Gamma, \pi_{\mathrm{p}} m : B \rfloor_A^P &= \begin{cases} \lfloor \Gamma \rfloor_A^P, \pi_{\mathrm{p}} m : B & \text{if } P(B, A) \\ \lfloor \Gamma \rfloor_A^P & \text{otherwise} \end{cases} \\
\lfloor \Gamma, \gamma \in \Phi \rfloor_A^P &= \lfloor \Gamma \rfloor_A^P, \gamma \in \lfloor \Phi \rfloor_A^P \\
\lfloor \epsilon \rfloor_A^P &= \epsilon \\
\lfloor \exists u{:}B.\ s \rfloor_A^P &= \exists u{:}B.\ \lfloor s \rfloor_A^P \ \text{ if } P(B, A) \\
\lfloor \exists u{:}B.\ s \rfloor_A^P &= \lfloor s \rfloor_A^P \ \text{ if not } P(B, A) \\
\lfloor \lambda u : A.w \rfloor_A^P &= \lambda u : A. \lfloor w \rfloor_A^P \\
\lfloor \Phi^* \rfloor_A^P &= (\lfloor \Phi \rfloor_A^P)^* \\
\lfloor \Phi_1 + \Phi_2 \rfloor_A^P &= \lfloor \Phi_1 \rfloor_A^P + \lfloor \Phi_2 \rfloor_A^P
\end{aligned}$$

Fig. 8. Filtering modulo P

some other module m ($\pi_{\mathrm{m}} m$), with the m subscript indicating this is the module subcomponent. The typing rules for modules are standard, as they are simply an instance of dot notation [4].

The meta logic's view of LLF (intuitionistic and linear) contexts $\Gamma; \Delta$ is called a *module context*, defined in Figure 6 by the syntactic category χ. Informally, the meta level does not distinguish between the intuitionistic and linear contexts, it merely stipulates the existence of particular modules α (of module signature s), or slices γ whose linear part is known to be consumed by a quantified LLF object (expressed as a meta variable).

Module contexts χ must not be thought of as a collection of meta level bindings of γ and α variables, but rather as an abstract description of LLF level bindings. The actual meta level binding takes place in meta contexts Ψ (Figure 6), that we have already used (however not defined) in Figure 2. Meta variables, context variables, and module variables are declared in Ψ, and each declaration is indexed by a module context (denoted by the leading $\chi \rhd$) describing its free variables.

The colorful collection of α's and γ's fully describes a hypothetical pair of valid LLF contexts. The precise relation between the two is discussed in the next subsection. It is important to note, however, that the particular order of declarations in χ is irrelevant and does not reflect the order or declarations within $\Gamma; \Delta$. For example, the module context γ, γ' stands for an arbitrary valid interleaving of two valid contexts $\Gamma; \Delta$ and $\Gamma'; \Delta'$.

The type of a module context is defined by *world* Φ, that, intuitively speaking, describes the shape of a context in the form of a regular expression built from module signatures, repetition and alternation. Worlds have been extensively studied in prior work by the second author [18]. Module contexts may contain only modules valid in Φ. We write $\Psi; \chi' \vdash \chi : \Phi$ for the judgment that decides when (χ', χ) is a valid module context, and χ is in world Φ. For space reasons, the definition is omitted here.

3.3 Context conversion

Module contexts χ, while useful at the meta level, cannot directly be used by LLF. For instance, the aggregation of parameters into modules complicates the splitting of contexts required to type linear application $(M \,\hat{}\, N)$. Additionally, we want to be able to relate the intuitionistic and linear LLF contexts, so we must derive them both from a single χ.

A module context χ is converted to an LLF context Γ in a two step process. First, χ is *flattened* into an LLF context $\llbracket \chi \rrbracket$, as defined in Figure 7. This process simply breaks apart each module m in χ into its individual parameters.

Flattening keeps all parameters, which leads to unwanted parameter duplication if used to produce both the Γ and the Δ from a single χ. Furthermore, in the case of the linear context, we must cull extra variables that may occur in χ that simply cannot occur in an LLF object of a certain type.

We solve both of these problems by *filtering*. Filtering, given by $\lfloor \Gamma \rfloor_A^P$, eliminates from an LLF context any variables of type B that do not match the binary predicate $P(B, A)$. It is defined in Figure 8. Similarly, in the case of context variables, we apply filtering to the world annotation $\lfloor \Phi \rfloor_A^P$ and remove all references to module projections that do not match the predicate, creating a narrower view of γ. Our notion of filtering is very general because we permit two seemingly unrelated predicates to transform χ into the intuitionistic and linear context. We require that the resulting $\Gamma; \Delta$ always forms a valid LLF context.

A good choice for each P is one based on the subordination relation [20]. In LLF, all types must be linearly closed. Therefore for the linear context, we use the predicate $A \mathrel{\hat{\prec}} B$, which holds when objects of type A can occur in objects of type B, but not at the type level. For the intuitionistic context, we use the predicate $A \prec: B$, which holds if some object of type A can occur in an object of type B, possibly at the type level. This pair of predicates makes as many things as possible linear. If instead the predicate used for the intuitionistic context holds for all pairs of LLF types and the predicate for the linear context holds for none, $\mathcal{L}_\omega^+$ reduces to a meta logic of the logical framework LF [10].

We write the composition of filtering with flattening as $\llbracket \chi \rrbracket_A^P$. This composition is used any time we are transitioning from the meta logic level to the logic level.

3.4 LLF typing rules revisited

The additional typing rules of our extension to LLF in Figure 9 can now be explained in detail. The two bottom rules for the intuitionistic and linear use of module parameters follow the axiom rule of LLF. The top rule in that figure is the typing rule

$$\frac{\Psi(n) = (\chi \rhd A) \quad \Psi;\Gamma;\Delta \rhd \rho : \llbracket\chi\rrbracket_A^{\prec:} ; \llbracket\chi\rrbracket_A^{\hat{\prec}}}{\Psi;\Gamma;\Delta \rhd n[\rho] : [\rho]A}$$

$$\frac{}{\Psi;\Gamma;\cdot \rhd \pi_{\mathrm{p}} m : \Gamma(\pi_{\mathrm{p}} m)} \qquad \frac{}{\Psi;\Gamma;\pi_{\mathrm{p}} m : A \rhd \pi_{\mathrm{p}} m : A}$$

Fig. 9. Typing rules of extended LLF.

$$\begin{array}{lll}
(\mathit{Formulas}) & F & ::= \forall n :: (\chi \rhd A).\ F \mid \forall \gamma \in (\chi \rhd \Phi).\ F \mid \exists n :: (\chi \rhd A).\ F \mid \top\!\!\!\top \\
(\mathit{Programs}) & P & ::= \Lambda n :: (\chi \rhd A).\ P \mid \Lambda \gamma \in (\chi \rhd \Phi).\ P \mid P\ M \mid P\ \chi \\
 & & \quad \mid \langle\!\langle \chi \rhd M; P \rangle\!\rangle \mid \langle\!\langle\rangle\!\rangle \mid x \mid \text{case}\ \Omega \mid \mu\ x \in F.\ P \\
 & & \quad \mid \nu\ \alpha :: (\chi \rhd s).\ P \\
(\mathit{Cases}) & \Omega & ::= \cdot \mid \Omega, (\Psi \vdash \sigma \mapsto P) \\
(\mathit{Meta\ Contexts}) & \Psi & ::= \ldots \mid \Psi, x \in F \\
(\mathit{Substitutions}) & \sigma & ::= \cdot \mid \sigma, M/n \mid \sigma, \chi/\gamma \mid \sigma, P/x \mid \sigma, \alpha/\alpha
\end{array}$$

Fig. 10. $\mathcal{L}_\omega^+$ syntax

for meta variables n of type A in context χ declared in Ψ. The judgment $\Psi;\Gamma';\Delta' \rhd \rho : \Gamma;\Delta$, which we omit the definition of, ensures that the substitution ρ will, when applied to an object well-typed under $\Psi;\Gamma;\Delta$, produce an object well-typed under $\Psi;\Gamma';\Delta'$. The second premiss of the typing rule for meta variables therefore checks that the substitution associated with the meta variable will correctly map an object substituted for n into the ambient context.

3.5 Formulas and their semantics

$\mathcal{L}_\omega^+$ itself is a first-order meta logic custom designed for LLF. Similar to $\mathcal{M}_\omega^+$ [17] its syntactic categories consist of formulas, programs, and cases, given in Figure 10.

The universal quantifiers of $\mathcal{L}_\omega^+$ range over meta variables n and context variables γ, where χ is the aforementioned module context that describes all free variables of the term in question. There are no quantifiers for module variables α. We do not include existential quantification over module contexts because it does not seem to serve any useful purpose, as opposed to universal quantification, which is required for induction. $\top\!\!\!\top$ stands for the only propositional constant truth expressible in $\mathcal{L}_\omega^+$.

The semantic entailment for $\mathcal{L}_\omega^+$ is written in terms of $\models$, a relation that is defined as follows (in terms of the flattening and filtering operation $\llbracket\chi\rrbracket_A^P$ described in Section 3.3):

$$
\begin{aligned}
&\models \forall\gamma \in (\chi \triangleright \Phi).\ F \quad \text{iff} \quad \models [\chi'/\gamma]F \text{ for all } \cdot\,; \chi \vdash \chi' : \Phi \\
&\models \forall n :: (\chi \triangleright A).\ F \quad \text{iff} \quad \models [M/x]F \text{ for all } \cdot; \llbracket\chi\rrbracket_A^{\prec:}; \llbracket\chi\rrbracket_A^{\hat{\prec}} \triangleright M : A \\
&\models \exists n :: (\chi \triangleright A).\ F \quad \text{iff} \quad \models [M/x]F \text{ for some } \cdot; \llbracket\chi\rrbracket_A^{\prec:}; \llbracket\chi\rrbracket_A^{\hat{\prec}} \triangleright M : A \\
&\models \mathbb{T}
\end{aligned}
$$

The existential is the dual to the universal quantifier, and true is always valid.

3.6 Programs

The semantics of $\mathcal{L}_\omega^+$ portrays its intended use as a meta logic to reason about LLF encodings. Any proof within this meta logic should convince a critical observer of the validity of the statement, lemma, or theorem. It is almost certainly possible to give a categorical or model theoretic explanation of proof. We have instead chosen to view proofs as *total programs* via a realizability interpretation. A proof hence acts as a transformation in between LLF encodings. Its input/output behavior is fixed by the formula, its type.

Figure 10 describes the syntactic category for *programs*. $\Lambda n :: (\chi \triangleright A).\ P$ and $\Lambda\gamma \in (\chi \triangleright \Phi).\ P$ are the two binding constructs of $\mathcal{L}_\omega^+$ for LLF objects n and module contexts γ, respectively. Symmetrically, two forms of application $P\ M$ and $P\ \chi$ serve as the respective elimination forms. $\langle\!\langle \chi \triangleright M; P \rangle\!\rangle$ is a proof term for an existential formula, pairing an LLF term with a program. Next, the Figure shows the familiar unit $\langle\!\langle\rangle\!\rangle$ and program variables x and three more constructs that we will explain next: the case construct with cases Ω, the recursion operator μ, and finally the new operator ν.

Case and recursion are necessary to express inductive proofs as programs. The formulation of case (case Ω), the elimination form for LLF objects, looks peculiar, but is in fact quite natural. There is no explicit case subject, because implicitly, case matches against the ambient context in which a "case" may occur. This choice will prove useful in the meta theoretic investigation in Section 5, because dependencies render matching a non-local operation. Each individual case in $\Omega, (\Psi \vdash \sigma \mapsto P)$, consists of a substitution σ that serves as the pattern for that particular case. Each free variable that occurs in a pattern must be declared in Ψ and the body P may not refer to any other variables other than the ones declared in Ψ. The fixed point operator $\mu\ x \in F.\ P$ provides the most general form of the induction hypotheses.

Unbounded recursion and case with an empty Ω illustrate that without further side condition $\mathcal{L}_\omega^+$ programs may be partial and hence non-total. The following three side conditions to case Ω and $\mu\ x \in F.\ P$, respectively, remedy that problem and enforce totality.

Strictness. Each $x \in \Psi$ must have at least one occurrence in the pattern that leads to an unambiguous solution of the higher-order matching algorithm to be used.

Coverage. For all patterns σ within Ω, and or all ambient environments η, there exists a new ambient environment η', such that $[\eta']\sigma = \eta$.

$$\frac{\Psi; \cdot \vdash \chi : \Phi \quad \Psi; [\![\chi]\!]^{\prec:}_{A} \rhd A : \text{type} \quad \Psi, n :: (\chi \rhd A) \vdash P \in F}{\Psi \vdash \Lambda n :: (\chi \rhd A).\ P \in \forall n :: (\chi \rhd A).\ F}$$

$$\frac{\Psi \vdash P \in \forall n :: (\chi \rhd A).\ F \quad \Psi; [\![\chi]\!]^{\prec:}_{A}; [\![\chi]\!]^{\hat{\prec}}_{A} \rhd M : A}{\Psi \vdash P\ M \in [\text{id}_{\Psi}, M/n]F}$$

$$\frac{\Psi; \cdot \vdash \chi : \Phi \quad \Psi, \gamma \in (\chi \rhd \Phi) \vdash P \in F}{\Psi \vdash \Lambda \gamma \in (\chi \rhd \Phi).\ P \in \forall \gamma \in (\chi \rhd \Phi).\ F} \qquad \frac{\Psi \vdash P \in \forall \gamma \in (\chi \rhd \Phi).\ F \quad \Psi; \chi \vdash \chi' : \Phi}{\Psi \vdash P\ \chi' \in [\text{id}_{\Psi}, \chi'/\gamma]F}$$

$$\frac{\Psi; \cdot \vdash \chi : \Phi \quad \Psi \vdash P \in [\text{id}_{\Psi}, M/n]F \quad \Psi; [\![\chi]\!]^{\prec:}_{A}; [\![\chi]\!]^{\hat{\prec}}_{A} \rhd M : A}{\Psi \vdash \langle\!\langle \chi \rhd M; P \rangle\!\rangle \in \exists n :: (\chi \rhd A).\ F}$$

$$\frac{}{\Psi \vdash \langle\!\langle\rangle\!\rangle \in \mathbb{T}} \qquad \frac{\Psi \vdash \Omega \in F}{\Psi \vdash \text{case } \Omega \in F} \qquad \frac{}{\Psi \vdash x \in \Psi(x)} \qquad \frac{\Psi, x \in F \vdash P \in F}{\Psi \vdash \mu\ x \in F.\ P \in F}\ (**)$$

$$\frac{\Psi; \cdot \vdash \chi : \Phi \quad \Psi; [\![\chi]\!] \rhd s : \text{sig} \quad \Psi, \alpha :: (\chi \rhd s) \vdash P \in F \quad \Psi \vdash F \text{ ok}}{\Psi \vdash \nu\ \alpha :: (\chi \rhd s).\ P \in F}$$

. . . .

$$\frac{}{\Psi \vdash \cdot \in F} \qquad \frac{\Psi \vdash \Omega \in F \quad \Psi' \rhd \sigma : \Psi \quad \Psi' \vdash P \in [\sigma]F}{\Psi \vdash \Omega, (\Psi' \vdash \sigma \mapsto P) \in F}\ (*)$$

Fig. 11. Derivability in $\mathcal{L}^{+}_{\omega}$

Termination. For all arguments $M_1 \ldots M_n$ to P it holds that for all x that occur in P and arguments $N_1 \ldots N_n$ to x, it holds that $(N_1 \ldots N_n) < (M_1 \ldots M_n)$ with respect to some well-founded order $<$.

Finally, $\nu\ \alpha :: (\chi \rhd s).\ P$ introduces a new module variable during runtime. Often, for proofs about higher-order encodings the corresponding program has to recurse under an LLF λ binder, be it linear or intuitionistic. Afterwards modules can always be discharged via the mediating substitutions attached to meta variables. There are no other elimination forms for modules.

3.7 Proof theory for $\mathcal{L}^{+}_{\omega}$

$\mathcal{L}^{+}_{\omega}$'s design is based on the realizability interpretation of total programs as proof. The type system for programs that is described in this section plays the role of a meta logic, whose soundness is shown in Section 5. Our approach to developing the meta logic follows closely [17] and differs significantly from [7], who show the soundness of their design by a cut-elimination argument.

$\mathcal{L}^{+}_{\omega}$'s type theory is defined in Figure 11 in terms of two mutually dependent typing judgments: $\Psi \vdash P \in F$ for programs and $\Psi \vdash \Omega \in F$ for cases, using three

$$\frac{\Psi' \rhd \sigma : \Psi \quad \Psi'; \llbracket[\sigma]\chi\rrbracket^{\prec:}_{[\sigma]A}; \llbracket[\sigma]\chi\rrbracket^{\hat{\prec}}_{[\sigma]A} \rhd M : [\sigma]A}{\Psi' \rhd (\sigma, M/n) : (\Psi, n :: (\chi \rhd A))}$$

$$\frac{}{\Psi \rhd \cdot : \cdot} \qquad \frac{\Psi' \rhd \sigma : \Psi \quad \Psi'; [\sigma]\chi \vdash \chi' : \Phi}{\Psi' \rhd (\sigma, \chi'/\gamma) : (\Psi, \gamma \in (\chi \rhd \Phi))}$$

$$\frac{\Psi' \rhd \sigma : \Psi \quad \Psi' \vdash P \in [\sigma]F}{\Psi' \rhd (\sigma, P/x) : \Psi, x \in F} \quad \frac{\Psi' \rhd \sigma : \Psi \quad \alpha \in \Psi'}{\Psi' \rhd (\sigma, \alpha/\alpha) : \Psi, \alpha}$$

Fig. 12. Typing rules for patterns

auxiliary judgments. Two of those judgments $\Psi; \Gamma \rhd s : \mathrm{sig}$ and $\Psi \vdash F : \mathrm{ok}$ ensure the respective validity of module signatures and formulas but do not contribute much to the understanding of the rules. The other judgment $\Psi' \rhd \sigma : \Psi$ ensures the validity of the substitutions that play the role of patterns. Patterns are important for the understanding $\mathcal{L}^+_\omega$ and given in Figure 12. The first rule exhibits the need for flattening and filtering when instantiating meta variables.

All of the rules in Figure 11 lend themselves to two complementary interpretations. Type-theoretically speaking, the first four rules account for well-typed abstractions and applications, and logically speaking, they are merely introduction and elimination rules for the universal quantifiers, albeit ones using the flattening and filtering operation $\llbracket\chi\rrbracket^P_A$ described in Section 3.3. The fifth rule is the typing rule for pairs, and simultaneously an introduction rule for the existential quantifier. The corresponding elimination rule is subsumed by the case rules defined in below the dotted line [18], and thus need not be considered separately. The typing rule for unit is standard. Ω, the argument to case, is a list of all of the cases (which must all have the same type). The type of a variable x can be inferred from the meta context, and recursion is standard. The rule for ν extends Ψ by a new module constant. The typing rule ensures that α does not escape during evaluation by requiring that the type of the body not contain α.

4 Example

We have considered a few examples from programming language and logic design to exercise and experiment with the meta logic $\mathcal{L}^+_\omega$ for LLF. Our case studies include stateful computations (we managed to represent all meta theoretic proofs about Mini-ML with references in the original LLF paper [6]), linear lambda calculi and of linear logic itself. We found that the proof of the admissibility of cut for the tensor fragment of linear logic (see Figure 3), illustrates $\mathcal{L}^+_\omega$'s unique characteristics the best. Of course, there is a certain risk of confusing the reader with two conceptually different yet linear logics.

Theorem 4.1 (Admissibility of cut) *If* $\mathcal{P} :: \Gamma_1 \Longrightarrow C, \Delta_1$ *and* $\mathcal{Q} :: \Gamma_2, C \Longrightarrow \Delta_2$ *then* $\Gamma_1, \Gamma_2 \Longrightarrow \Delta_1, \Delta_2$.

Proof. By lexicographic structural induction on the subformula A and simultaneously on $\mathcal{P}$ and $\mathcal{Q}$ [15]. We show only the essential case between $\otimes$R and $\otimes$L.

$\mathcal{P} :: \Gamma_1', \Gamma_1'' \Longrightarrow A \otimes B, \Delta_1', \Delta_1''$	(by assumption)
$\mathcal{P}_1 :: \Gamma_1' \Longrightarrow A, \Delta_1'$	(by assumption)
$\mathcal{P}_2 :: \Gamma_1'' \Longrightarrow B, \Delta_1''$	(by assumption)
$\mathcal{Q} :: \Gamma_2, A \otimes B \Longrightarrow \Delta_2$	(by assumption)
$\mathcal{Q}_1 :: \Gamma_2, A, B \Longrightarrow \Delta_2$	(by assumption)
$\mathcal{R}_1 :: \Gamma_1', \Gamma_2, B \Longrightarrow \Delta_1', \Delta_2$	(by ind. hyp. on $\mathcal{P}_1, \mathcal{Q}_1$)
$\mathcal{R} :: \Gamma_1', \Gamma_1'', \Gamma_2 \Longrightarrow \Delta_1', \Delta_1'', \Delta_2$	(by ind. hyp. on $\mathcal{P}_2, \mathcal{R}_1$)

□

Theorem 4.1 corresponds to the following formula in $\mathcal{L}_\omega^+$.

$$(1) \quad \begin{aligned} &\forall \gamma_1 \in (\cdot \triangleright \Phi).\, \forall \gamma_2 \in (\gamma_1 \triangleright \Phi).\, \forall C : (\cdot \triangleright \mathrm{o}).\\ &\quad \forall P : (\gamma_1 \triangleright \mathrm{pos}\ C \to \#).\, \forall Q : (\gamma_2 \triangleright \mathrm{neg}\ C \to \#).\\ &\quad \exists R : (\gamma_1, \gamma_2 \triangleright \#).\, \top \end{aligned}$$

where

$$(2) \quad \begin{aligned} \Phi = &((\lambda A : \mathrm{o}.\, \exists n : \mathrm{neg}\ A.\, \epsilon)\\ &\quad + (\lambda A : \mathrm{o}.\, \exists p : \mathrm{pos}\ A.\, \epsilon))^*. \end{aligned}$$

The first two quantifiers in (1) range over module contexts γ_1 (valid in the empty context $\cdot$) and γ_2 (valid in γ_1). γ_1 represents the list of hypotheses of both Γ_1 and Δ_1, while γ_2 represents Γ_2 and Δ_2. Φ is the world of these contexts, ensuring that γ_1 and γ_2 only contain assumptions of the form "pos A" and "neg A". For example,

$$p_1 : \mathrm{pos}\ A_1, p_2 : \mathrm{pos}\ A_2, n_3 : \mathrm{neg}\ A_3 \in \Phi.$$

In (1), C ranges over closed formulas, P over sequent derivations in γ_1 with formula C on the left, and Q over sequent derivations in γ_2 with formula C on the right. R stands for the result derivation, necessarily valid in the union of γ_1 and γ_2.

The proof Formula (1), on the other hand is a total program that maps contexts Δ_1, Δ_2, and LLF objects C, P and Q such that $\cdot; \cdot \vdash C : \mathrm{o}$, $\cdot; \Delta_1 \vdash P : \mathrm{pos}\ C \to \#$, and $\cdot; \Delta_2 \vdash Q : \mathrm{neg}\ C \to \#$ into an LLF object R such that $\cdot; \Delta_1, \Delta_2 \vdash R : \#$. In the interest of clarity, the surface language used in Figure 13 that depicts only the essential case of the proof above, making use of a significant amount of syntactic sugar.

fun defines the recursive program "ca" by cases. "ca" expects five arguments, including two contexts, all in the form of patterns. $\Gamma_1', \Gamma_1'', \Gamma_2, A, B, P_1, P_2$ and Q_2 occur free in the pattern and in the body of that case. Thus **fun** is a shorthand for a leading μ, followed by several Λ binders and a case expression. For uniformity reasons, we write **new**... **in** ... **end** for $\nu\ \alpha :: (\chi \triangleright s).\ P$. And finally, the **let** ... **in** ... **end** is the standard local binding construct that can be directly expressed using $\mathcal{L}_\omega^+$ programs by combining nested program application with implicit case analysis.

Figure 13 illustrates the novel and distinct features of $\mathcal{L}_\omega^+$ including pattern-matching against linear patterns, hypothetical reasoning, and context splitting. We describe the program in greater detail in the rest of this section, in the context of

```
fun ca $(\Gamma_1', \Gamma_1'')\ \Gamma_2\ (A \otimes B)$
        $(\hat{\lambda} p{:}\mathrm{pos}\ (A \otimes B).\, \mathrm{tensorR}\,\hat{}\,(\Gamma_1' \triangleright P_1)\,\hat{}\,(\Gamma_1'' \triangleright P_2)\,\hat{}\,p)$
        $(\hat{\lambda} n{:}\mathrm{neg}\ (A \otimes B).\, \mathrm{tensorL}\,\hat{}$
            $(\Gamma_2 \triangleright (\hat{\lambda} n_1{:}\mathrm{neg}\ A.\, \hat{\lambda} n_2{:}\mathrm{neg}\ B.\, Q_1\,\hat{}\,n_1\,\hat{}\,n_2))\,\hat{}\,n) =$
    new $\alpha :: (\Gamma_1', \Gamma_2 \triangleright \exists n : \mathrm{neg}\ B.\, \epsilon)$ in
      let
        val $\langle R_1, \langle\rangle\rangle =$ ca $\Gamma_1'\ (\Gamma_2, \alpha : \exists n : \mathrm{neg}\ B.\, \epsilon)\ A\ P_1$
                                        $(\hat{\lambda} n_1{:}\mathrm{neg}\ A.\, Q_1\,\hat{}\,n_1\,\hat{}\,\pi_{\mathrm{p}}(\alpha)])$
        val $\langle R, \langle\rangle\rangle =$ ca $\Gamma_1''\ (\Gamma_1', \Gamma_2)\ B\ P_2\ (\hat{\lambda} n{:}\mathrm{neg}\ B.\, R_1[n/\pi_{\mathrm{p}}(\alpha)])$
      in
        $\langle R, \langle\rangle\rangle$
      end
    end
```

Fig. 13. Admissibility of cut, essential case

an analysis of ca's properties regarding strictness, coverage, and termination.

Strictness.

Upon application, matching will always instantiate all free variables in the pattern of "ca". The claim follows directly for Γ_2, A, B, P_1, P_2, and Q_1, which leaves Γ_1' and Γ_1'' to be explained. In $\mathcal{L}_\omega^+$, every object carries its own context, which means that any instantiation of P_1 and P_2 decides the instantiations for Γ_1' and Γ_1'', respective, rendering matching a deterministic operation.

Coverage.

The first two arguments to "ca" are the context patterns (γ_1', γ_1'') and γ_2. How the context is split into γ_1' and γ_1'' is determined by how the two contexts are used. This is fixed by ascribing context information to the two variables P_1 and P_2 bound in the fourth argument to "ca": $(\hat{\lambda} p{:}\mathrm{pos}\ (A \otimes B).\, \mathrm{tensorR}\,\hat{}\,(\gamma_1' \triangleright P_1)\,\hat{}\,(\gamma_1'' \triangleright P_2)\,\hat{}\,p)$. Context and type ascription are features of the syntax we have chosen to present proofs in $\mathcal{L}_\omega^+$ in, with counterparts in the formal development of $\mathcal{L}_\omega^+$ in Section 3.

The challenge is to verify that "ca" covers all cases. Canonical forms are patterns and in the interest of completeness, two additional cases related to "tensorR" must be considered, depending on if p is consumed in P_1 or P_2.

Termination.

"ca" must be total in order to be considered a proof. Therefore any evaluation of "ca", independent of what arguments are applied, must terminate. Consider the body of "ca" in Figure 13. The two recursive calls to "ca" correspond to appeals to the induction hypothesis in the proof of Theorem 4.1, yielding result objects R_1 and R, respectively.

The first instruction is the **new** instruction that introduces a new hypotheses of type neg B. Recall from the proof of Theorem 4.1 that $\mathcal{R}_1$ is the result of the induction hypothesis applied to $\mathcal{P}_1$ and $\mathcal{Q}_1$, which is parametric in B. Since hypothetical arguments are encoded via higher-order functions, "ca" can only execute a recursive call after traversing the binder $(\hat{\lambda} n_2 : \text{neg } B)$. In general one can only do this by applying it to a new parameter n_2 : neg B, in form of the module declaration

$$\alpha :: (\gamma_1', \gamma_2 \triangleright \exists n : \text{neg } B.\, \epsilon). \tag{3}$$

α is a new variable, that ranges over groups of new parameters, and is similar to $\underline{x}$ in [18]. Intuitively, one can think of a module as a temporary list of new constant symbols that act as placeholders within the body of **new**. The γ_1', γ_2 resolve all ambiguities related to the naming of α. We write π_{p} to project the head of the list, and π_{m} for the tail. $\pi_{\mathrm{p}}(\alpha)$, for example, is a new name for the newly introduced parameter, and should be used instead of n_2.

The first recursive call cuts P_1 and Q_1 with cut-formula A. Eventually, the computation will finish and the resulting derivation R_1 will use all resources of the set $\gamma_1', \gamma_2, \alpha : \exists n : \text{neg } B.\, \epsilon$, which corresponds directly to the informal proof. Recall that γ_1' represents assumption lists Γ_1' and Δ_1', γ_2 the assumption lists Γ_2 and Δ_2, and α to the additional hypothesis B that occurs to the left of the sequence arrow.

The other recursive call for cutting P_2 and R_1 is similar to the first except that this time the cut formula is B. R_1 is parametric in $\pi_{\mathrm{p}}(\alpha)$, which is subsequently replaced by a linear variable n *before* the second recursive call is invoked. Replacements of this kind are supported in $\mathcal{L}_\omega^+$, expressed by substituting n for $\pi_{\mathrm{p}}(\alpha)$. The resulting R is valid in $\gamma_1', \gamma_1'', \gamma_2$, and does therefore not depend on α. Hence, it can safely escape the scope of **new**.

"ca" terminates because the arguments that correspond to derivations $\mathcal{P}$ and $\mathcal{Q}$ are smaller with respect to a well-founded lexicographical order on the cut formula and simultaneously on $\mathcal{P}$ and $\mathcal{Q}$. In this work, we consider only lexicographic and simultaneous extensions of the subterm ordering. In particular the first recursive call terminates because A and B are subterms of $A \otimes B$.

5 Meta Theory of $\mathcal{L}_\omega^+$

The totality of every program in $\mathcal{L}_\omega^+$ is a sufficient and necessary condition for the soundness of $\mathcal{L}_\omega^+$. The argument relies on a small-step operational semantics which we omit here. We define a evaluation meta context E to be a meta context Ψ binding only module variables α. For the purposes of the operational semantics, we extend the set of programs with a closure $\{\sigma; P\}$, in which σ is a substitution that maps P from whatever meta context it is well-typed under into the outer meta context. The evaluation judgment $E \vdash P \to P'$ relates a program P to the outcome of a single evaluation step P'. For a sequence of zero or more evaluation steps, we write $E \vdash P \to^* P'$. The set of values is V.

$$V ::= \Lambda n :: (\chi \triangleright A).\ P \mid \Lambda \gamma \in (\chi \triangleright \Phi).\ P \mid \langle\!\langle \chi \triangleright M; V \rangle\!\rangle \mid \langle\!\langle\rangle\!\rangle$$

For functions, applications, existentials and fixed points, evaluation proceeds in the standard fashion. The evaluation of a closure $\{\sigma; P\}$ is essentially carrying out a single step of lazily applying the substitution σ to P. This is done because eager substitution is not sound in the presence of case. Evaluation of (case $\Omega, (\Psi \vdash \sigma' \mapsto P)$) in a closure proceeds by attempting to generate a substitution σ'' that, when composed with σ', is equivalent to the σ of the closure. If one is found, then evaluation of P continues in a closure under σ''. The evaluation of $\nu\ \alpha :: (\chi \triangleright s).\ P$ proceeds by evaluating P until it becomes a value. When it finally becomes a value, the ν binding is pushed into any non-values (as occur in a function) that may exist in the value. In addition to the usual weakening and exchange lemmas, the following properties hold:

Lemma 5.1 (LLF module variable strengthening) *If* $\Psi; \cdot \vdash \chi : \Phi$ *and* $\Psi, \alpha :: (\chi' \triangleright s); [\![\chi]\!]_A^{\prec:}; [\![\chi]\!]_A^{\hat{\prec}} \triangleright M : A$ *then* $\Psi; [\![\chi]\!]_A^{\prec:}; [\![\chi]\!]_A^{\hat{\prec}} \triangleright M : A$.

Proof. If χ does not contain α, then the LLF context produced by flattening and filtering χ cannot contain any projections from α, and thus neither can M. □ □

Theorem 5.2 (Type preservation) *If* $E \vdash P \in F$ *and* $E \vdash P \to P'$ *then* $E \vdash P' \in F$.

Proof. By induction on the structure of the evaluation relation. The cases for ν rely on the fact that the type of the body of the ν must not use the bound module variable, and on Lemma 5.1. This allows the ν to be pushed inward while preserving the type. The substitution cases rely on the soundness of the substitution of σ into χ, A, M and x. □ □

Theorem 5.3 (Progress) *If* $E \vdash P \in F$ *then either* P *is a value or* $E \vdash P \to P'$.

Proof. By induction on the structure of the typing derivation. The progress proof uses the fact that E binds only module variables, and on the usual canonical forms lemma. It also relies on the coverage condition holding, which ensures that the program (case $\cdot$) is never evaluated. □ □

Theorem 5.4 (Termination) *If* $E \vdash P \in F$ *then* $E \vdash P \to^* V$.

Proof. By induction on the typing derivation, keeping track of the instantiations of the values bound by reductions of μ, using the termination condition. □ □

Theorem 5.5 (Soundness) *If* $\cdot \vdash P \in F$ *then* $\models F$.

Proof. By induction on F, using Theorems 5.2, 5.3 and 5.4. □ □

6 Conclusion

We have described the meta logic $\mathcal{L}_\omega^+$ for the linear logical framework LLF. LLF is useful for the representation of formal systems that rely on a notion of deletable resource. Surprisingly, many such systems can be represented in LLF, among them programming languages with effects, state transition systems, such as the infamous blocks world often used in AI, and of course resource oriented logics such as linear logic itself.

The meta logic $\mathcal{L}_\omega^+$ is custom-made for LLF, which means that it incorporates knowledge about linear assumptions, how they are consumed, split in the multiplicative, and duplicated in the additive fragment. It enables the formalization of meta theoretic properties, the mechanization of reasoning about LLF encodings, and leads to relatively short proof terms. The soundness of $\mathcal{L}_\omega^+$ follows from a realizability argument that shows that every function in $\mathcal{L}_\omega^+$ is total, i.e. it terminates and covers all cases.

In future work, we plan to implement a proof checker and an automated theorem prover for $\mathcal{L}_\omega^+$, and consider extensions to the ordered logical framework and the concurrent logical framework.

References

[1] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, Boston, USA, June 2001.

[2] David Basin, Manuel Clavel, and Jos Meseguer. Rewriting logic as a metalogical framework. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 55–80. Springer-Verlag LNCS 1974, 2000.

[3] Iliano Cervesato, Nancy Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A Meta-Notation for Protocol Analysis. In *12th Computer Security Foundations Workshop — CSFW-12*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

[4] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In *Proc. Programming Concepts and Methods*, pages 479–504. North Holland, 1990.

[5] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[6] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[7] Alwen Fernanto Tiu Dale Miller. A proof theory for generic judgments. In *Proceedings of 18th IEEE Symposium on Logic in Computer Science (LICS 2003)*, pages 118–227, Ottawa, Canada, 2003. IEEE Computer Society.

[8] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[9] John Harrison. Floating point verification in HOL Light: The exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[10] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[11] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax: An extended abstract. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, pages 434–445, Warsaw, Poland, June 1997.

[12] Tobias Nipkow and David von Oheimb. Java-light is type-safe — definitely. In L. Cardelli, editor, *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*, pages 161–170, San Diego, California, January 1998. ACM Press.

[13] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag LNCS 828, 1994.

[14] Lawrence C. Paulson. Proving properties of security protocols by induction. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 70–83. IEEE Computer Society Press, June 1997.

[15] Frank Pfenning. A structural proof of cut elimination and its representation in a logical framework. Technical Report CMU-CS-94-218, Department of Computer Science, Carnegie Mellon University, November 1994.

[16] Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In *CADE-19*, pages 473–487, Miami Beach, Florida, July 2003.

[17] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems.* PhD thesis, Carnegie Mellon University, 2000. CMU-CS-00-146.

[18] Carsten Schürmann. Recursion for higher-order encodings. In Laurent Fribourg, editor, *Proceedings of the Conference on Computer Science Logic (CSL 2001)*, pages 585–599, Paris, France, August 2001. Springer Verlag LNCS 2142.

[19] Joseph C. Vanderwaart and Karl Crary. A simplified account of the metatheory of linear lf. *Electronic Notes in Theoretical Computer Science*, 70(2), 2002.

[20] Roberto Virga. *Higher-Order Rewriting with Dependent Types.* PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999. Forthcoming.