A Linear Logical Framework

Iliano Cervesato and Frank Pfenning*
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
{*iliano*|*fp*}@*cs.cmu.edu*

Abstract

We present the linear type theory $\lambda^{\Pi \multimap \& \top}$ as the formal basis for *LLF*, a conservative extension of the logical framework *LF*. *LLF* combines the expressive power of dependent types with linear logic to permit the natural and concise representation of a whole new class of deductive systems, namely those dealing with state. As an example we encode a version of *Mini-ML* with mutable references including its type system and its operational semantics, and describe how to take practical advantage of the representation of its computations.

1 Introduction

A *logical framework* [Pfe96] is a formal meta-language specifically designed to represent and reason about programming languages, logics and other formalisms that can be described as deductive systems. These frameworks consist of a *meta-representation language* with desirable computational and representational properties, normally a logic or a type theory, and of a *meta-representation methodology* that suggests how to best take advantage of the underlying meta-language to encode a given formal system. The logical framework *LF* [HHP93] is among the most successful such proposals: it is based on the dependent type theory λ^{Π}, it relies on the *judgments-as-types* representation methodology, and it has been implemented as the higher-order constraint logic programming language *Elf* [Pfe91, Pfe94a]. *LF* and *Elf* have been widely used to study logical formalisms [Pfe95] and programming languages [MP91, Pfe92].

Unfortunately, many constructs and concepts needed in common programming practice cannot be represented in a satisfactory way in meta-languages based on intuitionistic logic and intuitionistic type theory, such as *LF*. In particular, constructs based on the notion of state as found in imperative languages often escape an elegant formalization by means of these tools.

* This work was supported by NSF Grant CCR-9303383. The second author was supported by the *Alexander-von-Humboldt-Stiftung* when working on this paper during a visit to the Department of Mathematics of the Technical University Darmstadt.

Similarly, logical systems that, by definition (e.g. substructural logics) or by presentation (e.g. Dyckhoff's contraction-free intuitionistic sequent calculus [Dyc92]), rely on destructive context operations require awkward encodings in an intuitionistic framework. Consequently the adequacy of the representation is difficult to prove and the formal meta-theory quickly becomes intractable.

Linear logic [Gir87] provides a view of context formulas as resources, which can be exploited to model the notion of state, as described for example in [Chi95, HM94, Mil94, Wad90]. The current proposals put the emphasis on the issue of *representing* imperative constructs and resource-based logics, but appear inadequate for *reasoning* effectively about these representations. On the other hand, intuitionistic type-theoretic frameworks such as *LF* make the representation of meta-reasoning easy, but do not have any notion of linearity built in.

In this paper, we propose a conservative extension of the logical framework *LF* that permits representing linear objects and reasoning about them. We call this formalism *Linear LF*, or *LLF* for short. The language underlying *LLF* is the dependent type theory $\lambda^{\Pi \multimap \& \top}$, which extends *LF*'s λ^{Π} with the linear connectives $\multimap$ (linear implication), & (additive conjunction) and $\top$ (additive truth), seen in this setting as type constructors. The language of objects of λ^{Π} is consequently extended with linear functional abstraction, additive pairs and unit, the corresponding destructors, and their equational theory. In order to keep the system simple we restrict the indices of type families to be linearly closed so that a type can depend only on intuitionistic assumptions, but not on linear variables. While at first this may appear to be a strong restriction, the expressive power of the resulting system does not seem to be hindered by this limitation.

The meta-representation methodology of *LLF* extends the judgments-as-types technique adopted in *LF* with a direct way to map state-related constructs and behaviors onto the linear operators of $\lambda^{\Pi \multimap \& \top}$. The resulting representations retain the elegance and immediacy that characterize *LF* encodings, and the ease of proving their adequacy. *LLF* has so far been used to encode the syntax of linear logic, sequent calculus and natural deduction presentations of its semantics, imperative programming languages and their operational behavior, and a number of state-based games. We have also applied *LLF* to formalize aspects of the meta-theory of these systems such as the proof of cut elimination for classical linear logic, translations between linear natural deduction and sequent calculus, and properties of imperative languages such as type preservation [Cer96].

The principal contributions of this paper are: (1) the definition of a uniform type theory admitting linear entities in conjunction with dependent types; (2) a thorough meta-theoretical investigation of this framework; and (3) the use of this system as a logical framework to represent and reason about problems that are not handled well by previous formalisms, either linear or intuitionistic. To our knowledge, $\lambda^{\Pi \multimap \& \top}$ is the first example in the literature of a *linear* type theory that goes beyond simple types, analyzed for example in [Bar96]. Subsequent work along the same lines of thought has been proposed by Ishtiaq and Pym in [IP96]. Both type theories were inspired by ideas in [MPP92].

The paper is organized as follows. Section 2 describes the linear type theory $\lambda^{\Pi \multimap \& \top}$ on which *LLF* is founded. It also presents major results in its meta-theory, such as the strong normalization theorem and the decidability of type-checking. Finally, it describes a canonical formulation of this language, which forms the basis for the meta-representation methodology adopted in *LLF*. Section 3 demonstrates the expressive power of *LLF* as a logical framework by providing an encoding of the syntax and the semantics of an imperative programming language, and by showing how to take practical advantage of the resulting representation of computations. Finally, Section 4 assesses the results and outlines future work. Further details about the work presented in this paper can be found in [Cer96].

2 The Linear Type Theory $\lambda^{\Pi\multimap\&\top}$

In this section, we define the linear type theory $\lambda^{\Pi\multimap\&\top}$ on which *LLF* is founded. More importantly, we present the major results in its meta-theory that justify adopting it as a meta-representation language. In order to facilitate the description of *LLF* in the available space, we must assume that the reader is familiar with both the logical framework *LF* [HHP93] and various presentations of linear logic [Gir87, Gir93] and linear λ-calculi [Bar96]. We will also take advantage of the natural extension of the Curry-Howard isomorphism to linear logic by viewing types as formulas. Due to space constraints, we limit our discussion to the main results in the meta-theory of *LLF* and, moreover, present only sketches of their proofs. The interested reader is invited to consult [Cer96] for further details about the presentation and the proofs in this section.

The discussion proceeds as follows: we first describe the syntax of $\lambda^{\Pi\multimap\&\top}$ in Section 2.1. Then, after a general presentation of a pre-canonical deductive system for *LLF* in Section 2.2, we focus our attention on the equational theory of this language in Section 2.3 and then present some basic properties in Section 2.4. We prove strong normalization for this system in Section 2.5 and then give an equivalent algorithmic presentation in Section 2.6 that we use to prove the decidability of type checking for *LLF* in Section 2.7. On the basis of these results, we devise a canonical system for $\lambda^{\Pi\multimap\&\top}$ in Section 2.8. Finally, in order to simplify the treatment of the case study in the next section, we extend the concrete syntax of *Elf*, the major implementation of *LF*, to the linear operators of $\lambda^{\Pi\multimap\&\top}$.

2.1 Language and Basic Operations

The linear type theory $\lambda^{\Pi\multimap\&\top}$ underlying *LLF* extends the language λ^{Π} of the logical framework *LF* with three connectives from linear logic, seen in this context as type constructors, namely *multiplicative implication* ($\multimap$), *additive conjunction* ($\&$) and *additive truth* ($\top$). The language of objects is augmented accordingly with the respective constructors and destructors. Linear types operate on *linear assumptions* which we represent as distinguished declarations of the form $x \,\hat{:}\, A$ in the context; we write $x : A$ for context elements à la λ^{Π} and call them *intuitionistic assumptions*. The syntax of $\lambda^{\Pi\multimap\&\top}$ is given by the following annotated grammar, where we have separated the constructs not present in λ^{Π} with a double bar $\|$:

$$
\begin{array}{rrcll}
\textit{Kinds:} & K & ::= & \mathrm{TYPE} & (\textit{Class of types}) \\
& & | & \Pi x{:}A.\,K & (\textit{Class of dependent type families}) \\
\textit{Types:} & A & ::= & P & (\textit{Base types}) \\
& & | & \Pi x{:}A_1.\,A_2 & (\textit{Intuitionistic function types}) \\
& & \| & A_1 \multimap A_2 & (\textit{Linear function types}) \\
& & | & A_1 \,\&\, A_2 & (\textit{Additive product types}) \\
& & | & \top & (\textit{Additive unit type}) \\
\textit{Type families:} & P & ::= & a & (\textit{Type family constants}) \\
& & | & P\,M & (\textit{Type family instantiation}) \\
\textit{Objects:} & M & ::= & c \mid x & (\textit{Object constants and variables}) \\
& & | & \lambda x{:}A.\,M \mid M_1\,M_2 & (\textit{Intuitionistic functions}) \\
& & \| & \hat{\lambda} x{:}A.\,M \mid M_1 \,\hat{}\, M_2 & (\textit{Linear functions}) \\
& & | & \langle M_1, M_2\rangle \mid \mathrm{FST}\,M \mid \mathrm{SND}\,M & (\textit{Additive pairing}) \\
& & | & \langle\rangle & (\textit{Additive unit element}) \\
\textit{Contexts:} & \Psi & ::= & \cdot & (\textit{Empty context}) \\
& & | & \Psi, x{:}A & (\textit{Intuitionistic assumption}) \\
& & \| & \Psi, x \mathrel{\hat{:}} A & (\textit{Linear assumption}) \\
\textit{Signatures:} & \Sigma & ::= & \cdot & (\textit{Empty signature}) \\
& & | & \Sigma, a{:}K & (\textit{Type family constant declaration}) \\
& & | & \Sigma, c{:}A & (\textit{Object constant declaration})
\end{array}
$$

Here x, c and a range over object-level variables, object constants and type family constants, respectively. We adopt the convention of denoting linear variables with the letter u, possibly subscripted; we will however continue to write x for generic variables. In particular, we write $x \mathrel{\hat{::}} A$ for a context assumption whose exact nature (linear — $x \mathrel{\hat{:}} A$ — or intuitionistic — $x : A$) is unimportant. In addition to the names displayed above, we will often use N and B to range over objects and types respectively. Moreover, we denote generic terms, i.e., objects, types or kinds, with the letters U and V, possibly subscripted. As usual, we write $A \to U$ for $\Pi x{:}A.\,U$ whenever x does not occur in the type or kind U.

The notions of free and bound variables are adapted from *LF* (notice the presence of a new binding construct: linear $\hat{\lambda}$-abstraction). We denote with $\mathrm{FV}(U)$ the free (linear or intuitionistic) variables of a term U. We extend this notation to contexts and write $\mathrm{FV}(\Psi)$ to denote the union of $\mathrm{FV}(A)$ for all $x \mathrel{\hat{::}} A$ in Ψ. As usual, we identify terms that differ only in the name of their bound variables and write $[M/x]U$ for the capture-avoiding substitution of M for x in the term U; note that x can be either linear or intuitionistic. We extend this notation to contexts and write $[M/x]\Psi$ for the result of substituting M for x in the type of every assumption in Ψ. Finally we require variables and constants to be declared at most once in a context and in a signature, respectively.

$$\frac{}{\cdot = \cdot \bowtie \cdot}\,\textbf{s\_dot} \qquad \frac{\Psi = \Psi' \bowtie \Psi''}{(\Psi, u \mathbin{\hat{:}} A) = (\Psi', u \mathbin{\hat{:}} A) \bowtie \Psi''}\,\textbf{s\_lin1}$$

$$\frac{\Psi = \Psi' \bowtie \Psi''}{(\Psi, x{:}A) = (\Psi', x{:}A) \bowtie (\Psi'', x{:}A)}\,\textbf{s\_int} \qquad \frac{\Psi = \Psi' \bowtie \Psi''}{(\Psi, u \mathbin{\hat{:}} A) = \Psi' \bowtie (\Psi'', u \mathbin{\hat{:}} A)}\,\textbf{s\_lin2}$$

Figure 1: Context Splitting

In the following discussion, we will as usual drop the leading *empty sequence* $(\cdot)$ from the representation of a context. Similarly, we overload the context constructor (,) and use it to denote *sequence concatenation* as well. We do not state or prove the usual properties of this operation. Whenever we concatenate two contexts Ψ_1 and Ψ_2 we assume they do not declare common variables so that the resulting context Ψ_1, Ψ_2 contains just one assumption for each declared variable. We denote with $\operatorname{dom}\Psi$ the domain of context Ψ, defined as the set of variables declared in it, and write $\Psi_{|\chi}$ for the restriction of Ψ to the variables appearing in χ.

Below we will often need to refer to the *intuitionistic part* of a context Ψ. Therefore, we introduce the function $\overline{\Psi}$, defined as follows:

$$\begin{cases} \overline{\cdot} & = & \cdot \\ \overline{\Psi, x{:}A} & = & \overline{\Psi}, x{:}A \\ \overline{\Psi, x \mathbin{\hat{:}} A} & = & \overline{\Psi} \end{cases}$$

We overload this notation and use $\overline{\Psi}$ to express the fact that the linear portion of the denoted context is constrained to be empty (e.g., in the all rules for type families in Figure 2).

Multiplicative connectives in linear logic require the context to be split among the premisses of a binary rule (or the contexts in the premisses to be merged in the conclusion, depending on the point of view). We rely on the *context splitting judgment* to specify that the linear assumptions in a context Ψ must be distributed in the contexts Ψ_1 and Ψ_2, while the intuitionistic assumptions should be shared. Whenever this is the case, then $\Psi = \Psi_1 \bowtie \Psi_2$ is derivable. The rules in Figure 1 define this judgment.

Notice that, whenever the judgment $\Psi = \Psi_1 \bowtie \Psi_2$ is derivable, Ψ_1 and Ψ_2 differ from Ψ only by missing linear assumptions. In particular, the relative order of the declarations still mentioned in these contexts corresponds to the order they occur in Ψ. We anticipate that assumptions, either intuitionistic or linear, cannot depend on linear variables in $\lambda^{\Pi\multimap\&\top}$. Therefore, splitting a context that is valid according to the specifications in the next subsection yields two valid contexts. Similarly, merging valid contexts having distinct names for their linear variables produces a valid context.

2.2 Pre-Canonical Forms

The meaning of the syntactic entities of a language can be presented in various forms, the choice being dictated by the aspects we want to emphasize. In this section, we define the semantics of $\lambda^{\Pi\multimap\&\top}$ by means of a deductive system that we call *pre-canonical*. The aim of the present section is to study the main properties of the type theory underlying *LLF*, ultimately the decidability of type checking. Relying on a pre-canonical system is particularly convenient for this purpose since it cleanly separates the practical desideratum of having extensionality as part of our language, commonly expressed by means of η-conversion rules, from the crucial role of β-redices as the foundation of the equational theory of $\lambda^{\Pi\multimap\&\top}$.

The main properties of *LF* were originally stated and proved for a type theory whose notion of definitional equality was restricted to β-equivalence [HHP93]. However, the adequacy theorems that relate an object system to its *LF* encoding and the efficient implementation of this formalism as a logic programming language require considering λ^{Π} terms in canonical form. Therefore, the type theory that is used as a meta-representation language is based on $\beta\eta$-equivalence. This discrepancy was known to Harper, Honsell and Plotkin when they first presented *LF* in 1987 [HHP93], but they could not prove some of the necessary properties in the presence of the η-expansion rule (in particular the property that we call *confluence* in Section 2.3). A full treatment of the meta-theory of *LF* with $\beta\eta$-equivalence was successively devised by various authors [Coq91, Geu93, Sal90] and resulted in non-trivial complications.

The formulation of the semantics of *LLF* as a pre-canonical system has the advantage of forcing all derivable judgments to mention only terms in η-long form. Indeed, all the issues concerning η-conversion are hardwired into the system and do not require explicit treatment. Consequently, the terms that we will ultimately produce are exactly the $\beta\eta$-long terms needed in the adequacy theorems. That property allows us to focus on β-conversion and in particular to retain the simple techniques used in [HHP93], without the above mentioned anomaly. Our approach was inspired by Felty's *canonical LF* [Fel91].

The pre-canonical system for *LLF* is specified by means of a number of judgments. We first have seven judgments defining the notions of *pre-canonical* and *pre-atomic* terms and the natural extension of these concepts to contexts and signatures. The inference rules describing how to derive them are distributed over Figures 2 and 3. The shape of these expressions is reported here:

(i)	$\vdash \Sigma \Uparrow \mathit{Sig}$	(Σ *is a pre-canonical signature*)
(ii)	$\vdash_\Sigma \Psi \Uparrow \mathit{Ctx}$	(Ψ *is a pre-canonical context in* Σ)
(iii)	$\overline{\Psi} \vdash_\Sigma K \Uparrow \mathit{Kind}$	(K *is a pre-canonical kind in* $\overline{\Psi}$ *and* Σ)
(iv)	$\overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}$	(A *is a pre-canonical type in* $\overline{\Psi}$ *and* Σ)
(v)	$\overline{\Psi} \vdash_\Sigma P \downarrow K$	(P *is a pre-atomic type family of kind* K *in* $\overline{\Psi}$ *and* Σ)
(vi)	$\Psi \vdash_\Sigma M \Uparrow A$	(M *is a pre-canonical object of type* A *in* Ψ *and* Σ)
(vii)	$\Psi \vdash_\Sigma M \downarrow A$	(M *is a pre-atomic object of type* A *in* Ψ *and* Σ)

Note that the judgments referring to types and kinds operate on purely intuitionistic assump-

Signatures

$$\frac{}{\vdash \cdot \Uparrow \mathit{Sig}}\,\textbf{sp\_dot} \qquad \frac{\vdash \Sigma \Uparrow \mathit{Sig} \quad \cdot \vdash_\Sigma A \Uparrow \text{TYPE}}{\vdash \Sigma, c\!:\!A \Uparrow \mathit{Sig}}\,\textbf{sp\_obj} \qquad \frac{\vdash \Sigma \Uparrow \mathit{Sig} \quad \cdot \vdash_\Sigma K \Uparrow \mathit{Kind}}{\vdash \Sigma, a\!:\!K \Uparrow \mathit{Sig}}\,\textbf{sp\_fam}$$

Contexts

$$\frac{\vdash \Sigma \Uparrow \mathit{Sig}}{\vdash_\Sigma \cdot \Uparrow \mathit{Ctx}}\,\textbf{cp\_dot} \qquad \frac{\vdash_\Sigma \Psi \Uparrow \mathit{Ctx} \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE}}{\vdash_\Sigma \Psi, x\!:\!A \Uparrow \mathit{Ctx}}\,\textbf{cp\_int} \qquad \frac{\vdash_\Sigma \Psi \Uparrow \mathit{Ctx} \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE}}{\vdash_\Sigma \Psi, u \mathbin{\hat{:}} A \Uparrow \mathit{Ctx}}\,\textbf{cp\_lin}$$

Kinds

$$\frac{\vdash_\Sigma \overline{\Psi} \Uparrow \mathit{Ctx}}{\overline{\Psi} \vdash_\Sigma \text{TYPE} \Uparrow \mathit{Kind}}\,\textbf{kpc\_type} \qquad \frac{\overline{\Psi}, x\!:\!A \vdash_\Sigma K \Uparrow \mathit{Kind}}{\overline{\Psi} \vdash_\Sigma \Pi x\!:\!A.\, K \Uparrow \mathit{Kind}}\,\textbf{kpc\_dep}$$

Types/type families

$$\frac{\overline{\Psi} \vdash_\Sigma P \downarrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma P \Uparrow \text{TYPE}}\,\textbf{fpc\_a} \qquad \frac{\vdash_\Sigma \overline{\Psi} \Uparrow \mathit{Ctx}}{\overline{\Psi} \vdash_\Sigma \top \Uparrow \text{TYPE}}\,\textbf{fpc\_top} \qquad \frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE} \quad \overline{\Psi} \vdash_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma A \mathbin{\&} B \Uparrow \text{TYPE}}\,\textbf{fpc\_with}$$

$$\frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE} \quad \overline{\Psi} \vdash_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma A \multimap B \Uparrow \text{TYPE}}\,\textbf{fpc\_limp} \qquad \frac{\overline{\Psi}, x\!:\!A \vdash_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma \Pi x\!:\!A.\, B \Uparrow \text{TYPE}}\,\textbf{fpc\_dep}$$

(No **fpc_eq**, no **fpa_c**)

$$\frac{\overline{\Psi} \vdash_\Sigma P \downarrow K \quad K \equiv K' \quad \overline{\Psi} \vdash_\Sigma K' \Uparrow \mathit{Kind}}{\overline{\Psi} \vdash_\Sigma P \downarrow K'}\,\textbf{fpa\_eq}$$

$$\frac{\vdash_{\Sigma,a:K,\Sigma'} \overline{\Psi} \Uparrow \mathit{Ctx}}{\overline{\Psi} \vdash_{\Sigma,a:K,\Sigma'} a \downarrow K}\,\textbf{fpa\_con} \qquad \frac{\overline{\Psi} \vdash_\Sigma P \downarrow \Pi x\!:\!A.\, K \quad \overline{\Psi} \vdash_\Sigma N \Uparrow A}{\overline{\Psi} \vdash_\Sigma P\, N \downarrow [N/x]K}\,\textbf{fpa\_iapp}$$

Figure 2: Pre-Canonical Deduction System for $\lambda^{\Pi\multimap\&\top}$, Kinds and Types

tions, expressed by using the notation $\overline{\Psi}$ for their context. We will gain in clarity and conciseness in the following by relying on some abbreviations. Whenever a property holds uniformly for the judgments (*iii–vii*), we write $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ and then refer to the generic terms U and V if needed. Moreover, if two or more such expressions occur in a statement, we assume that the arrows of the actual judgments match, unless explicitly stated otherwise. We take the liberty of adopting this notation also in the case of kinds, even though *Kind* is not a term and there are no pre-atomic kinds. Whenever the judgment $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ has a derivation $\mathcal{P}$, a fact that we will sometimes write as $\mathcal{P} :: (\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V)$, we will often refer to U as the term being *validated* in this judgment, and call $\mathcal{P}$ a *validation* of U.

The notion of *definitional equality* we consider is β-equivalence and it operates at the level of objects, type families and kinds. Among the various possible presentations, we adopt *parallel nested reduction* ($\longrightarrow$), defined in Figure 4. We write $\longrightarrow^*$ and $\equiv$ for its transitive closure and the corresponding equivalence relation. We omit the obvious rules defining them. Cumulatively, we have the following nine judgments:

Objects

$$\frac{\Psi \vdash_\Sigma M \downarrow P}{\Psi \vdash_\Sigma M \Uparrow P}\ \textbf{opc\_a} \qquad \frac{\Psi \vdash_\Sigma M \Uparrow B \quad A \equiv B \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}}{\Psi \vdash_\Sigma M \Uparrow A}\ \textbf{opc\_eq}$$

$$\frac{\vdash_\Sigma \Psi \Uparrow Ctx}{\Psi \vdash_\Sigma \langle\rangle \Uparrow \top}\ \textbf{opc\_unit} \qquad \frac{\Psi \vdash_\Sigma M \Uparrow A \quad \Psi \vdash_\Sigma N \Uparrow B}{\Psi \vdash_\Sigma \langle M, N\rangle \Uparrow A \,\&\, B}\ \textbf{opc\_pair}$$

$$\frac{\Psi, u \hat{:} A \vdash_\Sigma M \Uparrow B}{\Psi \vdash_\Sigma \hat{\lambda} u{:}A.\, M \Uparrow A \multimap B}\ \textbf{opc\_llam} \qquad \frac{\Psi, x{:}A \vdash_\Sigma M \Uparrow B}{\Psi \vdash_\Sigma \lambda x{:}A.\, M \Uparrow \Pi x{:}A.\, B}\ \textbf{opc\_ilam}$$

$$\frac{\Psi \vdash_\Sigma M \Uparrow A}{\Psi \vdash_\Sigma M \downarrow A}\ \textbf{opa\_c} \qquad \frac{\Psi \vdash_\Sigma M \downarrow B \quad A \equiv B \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}}{\Psi \vdash_\Sigma M \downarrow A}\ \textbf{opa\_eq}$$

$$\frac{\vdash_{\Sigma, c:A, \Sigma'} \overline{\Psi} \Uparrow Ctx}{\overline{\Psi} \vdash_{\Sigma, c:A, \Sigma'} c \downarrow A}\ \textbf{opa\_con} \qquad \frac{\vdash_\Sigma \overline{\Psi}, u \hat{:} A, \overline{\Psi'} \Uparrow Ctx}{\overline{\Psi}, u \hat{:} A, \overline{\Psi'} \vdash_\Sigma u \downarrow A}\ \textbf{opa\_lvar} \qquad \frac{\vdash_\Sigma \overline{\Psi}, x{:}A, \overline{\Psi'} \Uparrow Ctx}{\overline{\Psi}, x{:}A, \overline{\Psi'} \vdash_\Sigma x \downarrow A}\ \textbf{opa\_ivar}$$

$$\text{(No rule for } \top) \qquad \frac{\Psi \vdash_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash_\Sigma \mathrm{FST}\, M \downarrow A}\ \textbf{opa\_fst} \qquad \frac{\Psi \vdash_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash_\Sigma \mathrm{SND}\, M \downarrow B}\ \textbf{opa\_snd}$$

$$\frac{\Psi' \vdash_\Sigma M \downarrow A \multimap B \quad \Psi'' \vdash_\Sigma N \Uparrow A \quad \Psi = \Psi' \bowtie \Psi''}{\Psi \vdash_\Sigma M\,\hat{}\,N \downarrow B}\ \textbf{opa\_lapp} \qquad \frac{\Psi \vdash_\Sigma M \downarrow \Pi x{:}A.\, B \quad \overline{\Psi} \vdash_\Sigma N \Uparrow A}{\Psi \vdash_\Sigma M\, N \downarrow [N/x]B}\ \textbf{opa\_iapp}$$

Figure 3: Pre-Canonical Deduction System for $\lambda^{\Pi\multimap\&\top}$, Objects

(*viii–x*)	$U \longrightarrow V$	(*U reduces to V*)
(*xi–xiii*)	$U \longrightarrow^* V$	(*U transitively reduces to V*)
(*xiv–xvi*)	$U \equiv V$	(*U is definitionally equal to V*)

We need one further ingredient to cope with the multiplicative type constructor $\multimap$, namely the context splitting judgment presented in the previous section and defined in Figure 1.

(*xvii*) $\quad \Psi = \Psi' \bowtie \Psi'' \qquad$ (*Ψ splits into Ψ' and Ψ''*).

We will now go through the rules defining $\lambda^{\Pi\multimap\&\top}$ and describe the main ideas behind this formulation of the semantics of our language. We first concentrate on the typing judgments in Figures 2–3, and then discuss the notion of definitional equality, founded on the reduction relation in Figure 4.

Consider first the rules displayed in the upper part of Figure 3. They validate pre-canonical objects M by deriving judgments of the form $\Psi \vdash_\Sigma M \Uparrow A$. Rules **opc_unit**, **opc_pair**, **opc_llam** and **opc_ilam** allow the construction of terms of the form $\langle\rangle$, $\langle M_1, M_2\rangle$, $\hat{\lambda} u : A.\, M$, and $\lambda x : A.\, M$, respectively. These four operators are the *object constructors* of our calculus. We call these inference patterns *introduction rules* since, if we focus our attention on their

Objects *Congruences*

$$\frac{}{c \longrightarrow c}\ \textbf{or\_con} \qquad \frac{}{x \longrightarrow x}\ \textbf{or\_var} \qquad \frac{}{\langle\rangle \longrightarrow \langle\rangle}\ \textbf{or\_unit}$$

$$\frac{M \longrightarrow M'}{\textsc{fst}\ M \longrightarrow \textsc{fst}\ M'}\ \textbf{or\_fst} \qquad \frac{M \longrightarrow M'}{\textsc{snd}\ M \longrightarrow \textsc{snd}\ M'}\ \textbf{or\_snd} \qquad \frac{M \longrightarrow M' \quad N \longrightarrow N'}{\langle M, N\rangle \longrightarrow \langle M', N'\rangle}\ \textbf{or\_pair}$$

$$\frac{M \longrightarrow M' \quad N \longrightarrow N'}{M\,\hat{}\,N \longrightarrow M'\,\hat{}\,N'}\ \textbf{or\_lapp} \qquad \frac{A \longrightarrow A' \quad M \longrightarrow M'}{\hat{\lambda} u\!:\!A.\,M \longrightarrow \hat{\lambda} u\!:\!A'.\,M'}\ \textbf{or\_llam}$$

$$\frac{M \longrightarrow M' \quad N \longrightarrow N'}{M\,N \longrightarrow M'\,N'}\ \textbf{or\_iapp} \qquad \frac{A \longrightarrow A' \quad M \longrightarrow M'}{\lambda x\!:\!A.\,M \longrightarrow \lambda x\!:\!A'.\,M'}\ \textbf{or\_ilam}$$

β−reductions

$$\frac{M \longrightarrow M'}{\textsc{fst}\ \langle M, N\rangle \longrightarrow M'}\ \textbf{or\_beta\_fst} \qquad \frac{N \longrightarrow N'}{\textsc{snd}\ \langle M, N\rangle \longrightarrow N'}\ \textbf{or\_beta\_snd}$$

$$\frac{M \longrightarrow M' \quad N \longrightarrow N'}{(\hat{\lambda} u\!:\!A.\,M)\,\hat{}\,N \longrightarrow [N'/u]M'}\ \textbf{or\_beta\_lin} \qquad \frac{M \longrightarrow M' \quad N \longrightarrow N'}{(\lambda x\!:\!A.\,M)\,N \longrightarrow [N'/x]M'}\ \textbf{or\_beta\_int}$$

Types/types families *Congruences*

$$\frac{}{a \longrightarrow a}\ \textbf{fr\_con} \qquad \frac{P \longrightarrow P' \quad N \longrightarrow N'}{P\,N \longrightarrow P'\,N'}\ \textbf{fr\_iapp}$$

$$\frac{}{\top \longrightarrow \top}\ \textbf{fr\_top} \qquad \frac{A \longrightarrow A' \quad B \longrightarrow B'}{A \,\&\, B \longrightarrow A' \,\&\, B'}\ \textbf{fr\_with}$$

$$\frac{A \longrightarrow A' \quad B \longrightarrow B'}{A \multimap B \longrightarrow A' \multimap B'}\ \textbf{fr\_limp} \qquad \frac{A \longrightarrow A' \quad B \longrightarrow B'}{\Pi x\!:\!A.\,B \longrightarrow \Pi x\!:\!A'.\,B'}\ \textbf{fr\_dep}$$

(No β-reductions for types and type families)

Kinds *Congruences*

$$\frac{}{\textsc{type} \longrightarrow \textsc{type}}\ \textbf{kr\_type} \qquad \frac{A \longrightarrow A' \quad K \longrightarrow K'}{\Pi x\!:\!A.\,K \longrightarrow \Pi x\!:\!A'.\,K'}\ \textbf{kr\_dep}$$

(No β-reductions for kinds)

Figure 4: Parallel Nested Reduction for $\lambda^{\Pi\multimap\&\top}$

type component, they introduce each of the type constructors of $\lambda^{\Pi\multimap\&\top}$ in their conclusion. The manner they handle their context is familiar from linear logic. Notice in particular that **opc_unit** is applicable with any context (as long as it is valid) and that the premisses of rule **opc_pair** share the same context, which also appears in its conclusion. Rules **opc_llam** and **opc_ilam** differ only by the nature of the assumption they add to the context in their premiss: linear in the case of the former, intuitionistic for the latter. The two remaining rules defining the object-level pre-canonical judgment leave the term in their central part unchanged. Rule **opc_eq** simply allows replacing the type component of a pre-canonical judgment with another type, under the condition that it is valid and definitionally equivalent to the original type. Rule **opc_a** is particularly interesting since it is the reason why all terms derivable in the pre-canonical system are in η-long form. Notice that this rule is applicable only at base types, indicated by writing P rather than A for the type in it.

The rules defining the pre-atomic judgment at the level of terms, $\Psi \vdash_\Sigma M \downarrow A$, are displayed in the lower part of Figure 3. They validate constants (rule **opa_con**) and linear and intuitionistic variables (rules **opa_lvar** and **opa_ivar**, respectively). They also allow the formation of the terms $M\,N$, $M\,\hat{}\,N$, $\text{FST}\,M$ and $\text{SND}\,M$ (rules **opa_iapp**, **opa_lapp**, **opa_fst** and **opa_snd**, respectively), whose main operators we call *destructors*. The latter four inference figures are called *elimination rules* since they permit taking apart each of the type constructors of $\lambda^{\Pi\multimap\&\top}$ from one of their premiss, with the exception of $\top$. The role played by linear assumptions in $\lambda^{\Pi\multimap\&\top}$ is particularly evident in these rules. Indeed, an axiom rule (**opa_con**, **opa_lvar** and **opa_ivar**) can be applied only if the linear part of its context is empty, or contains just the variable to be validated, with the proper type; this is expressed by using the $\overline{\cdots}$ notation. Linearity appears also in the elimination rule for $\multimap$, where we rely on the splitting judgment defined in Figure 1 to manage the context for this connective in rule **opa_lapp**. Observe also that the context of the argument part of an intuitionistic application, in rule **opa_iapp**, is constrained to be empty. Two remaining rules define pre-atomic derivability for the level of objects. The semantics of the equivalence rule **opa_eq** is similar to its pre-canonical counterpart. More interesting is rule **opa_c**, whose presence accounts for the possibility of validating terms containing β-redices.

The rules concerning linear objects in Figure 3 define the behavior of linear types. If we ignore the objects and the distinction between pre-canonical and pre-atomic judgments, they correspond to the specification of the familiar rules for the linear connectives $\top$, $\&$ and $\multimap$, presented in a *natural deduction* style. It is easy to prove the equivalence to the usual sequent formulation. The objects that appear on the left of these types record the structure of a natural deduction proof for the corresponding linear formulas. The dependent function type $\Pi x : A.\, B$ that $\lambda^{\Pi\multimap\&\top}$ inherits from λ^{Π} generalizes both intuitionistic implication $A \to B$ (customarily defined as $!A \multimap B$ in linear logic) and the universal quantifier $\forall x.\, B$, where A plays the role of the type of the (intuitionistic) variable x. With this interpretation, $\lambda^{\Pi\multimap\&\top}$ encompasses all the connectives and quantifiers of the freely generated fragment of the language of linear hereditary Harrop formulas, on which the programming language *Lolli* is based [HM94]. Additionally, $\lambda^{\Pi\multimap\&\top}$ offers the characteristic features of a type theory: higher-order functions, proof terms and type families indexed by arbitrary objects, possibly higher-order and linear.

We now turn to the judgments validating types, kinds, contexts and signatures, treated in Figure 2. The rules defining the pre-canonical judgment $\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE}$ simply specify that every valid type in the system should result from the combination of base types (rule **fpc_a**) by means of the type constructors Π, $\multimap$, $\&$ and $\top$ (rules **fpc_dep**, **fpc_limp**, **fpc_with** and **fpc_top**, respectively). Notice the differences in the rules concerning the two function type constructors: the validity of A in $\Pi x:A.\,B$ is implicitly ascertained when checking the validity of the context in the premiss; instead, the type A in $A \multimap B$ is to be validated explicitly since no assumption is inserted in the context. The rules for the pre-atomic type family judgment $\overline{\Psi} \vdash_{\Sigma} A \downarrow K$ simply verify that base types are syntactically well-formed and respect the kind declaration of their leading type family constant (rules **fpa_iapp** and **fpa_con**). Notice the presence of an equivalence rule: **fpa_eq**. Finally, the rules defining the pre-canonical kind judgment, $\overline{\Psi} \vdash_{\Sigma} K \Uparrow Kind$, check that every type appearing in K is valid. Note that this judgment is invoked only when validating a signature. The remaining rules in Figure 2 consider signatures and context. They specify that a signature is valid if the type or kind of every item declared in it is itself valid. Similarly, a context is valid if the type of each of its assumptions is valid.

In the rules in Figures 2–3, types and kinds are always checked using a purely intuitionistic context. This has the effect of preventing valid types from containing free linear variables. Therefore, although the indices of type families are in general linear objects, these terms can refer only to context variables that are intuitionistic. We say that indices are *linearly closed*. Loosening this restriction would require admitting linear dependent function types in our language, corresponding to linear quantifiers. Although this inclusion appears beneficial for certain applications, preliminary investigations seem to indicate that the consequent complications in the meta-theory of the language might outweigh the potential advantages.

We classify the rules in Figures 2–3 into *essential* and *non-essential* rules. We count among the latter class the equivalence rules (**opc_eq**, **opa_eq** and **fpa_eq**) and rule **opa_c**. All other rules are considered essential. A major task in this section of the paper will be to either eliminate or hide the non-essential rules to tailor the presentation of $\lambda^{\Pi\multimap\&\top}$ as a meta-language for the logical framework *LLF*.

We now turn to the reduction semantics of $\lambda^{\Pi\multimap\&\top}$, partially defined in Figure 4. The notion of definitional equality that we consider is the equivalence relation $\equiv$ constructed on the congruence relation $\longrightarrow$. The basis of this congruence consists of the following β-*reduction* rules:

$$\begin{array}{llcll}
\beta_{fst}: & \text{FST}\,\langle M,N\rangle \longrightarrow M & \qquad & \beta_{lapp}: & (\hat{\lambda} u:A.\,M)\,\hat{}\,N \longrightarrow [N/u]M \\
\beta_{snd}: & \text{SND}\,\langle M,N\rangle \longrightarrow N & & \beta_{iapp}: & (\lambda x:A.\,M)\;N \longrightarrow [N/x]M.
\end{array}$$

As usual, we call the expressions appearing on the left-hand side of the arrow *redices*. The only possible redex in λ^{Π} is β_{iapp}. We adopt the standard terminology and call term U that does not contain β-redices *normal*, or β-*normal*. This definition extends immediately to contexts and signatures. One of the tasks of this section will be to show that every valid entity in $\lambda^{\Pi\multimap\&\top}$ can be reduced to a normal form, and that this normal form is itself valid in *LLF*.

2.3 Equational Theory

We now attack the formal study of $\lambda^{\Pi\multimap\&\top}$. We aim at proving that this type theory has all the desirable properties that a formalism should possess in order to be a suitable meta-language for a logical framework. In particular, we will show that $\lambda^{\Pi\multimap\&\top}$ is strongly normalizing, admits unique normal forms, and that type-checking is decidable for this language. These properties rely on a large number of lemmas and it is a challenge of its own to organize the overall meta-theory in a linear sequence of results with simple dependencies. This organization relies crucially on the details of the formulation of the semantics of our type-theory. The adoption of a pre-canonical system that imposes extensionality, rather than just permitting it via additional rules, simplifies the development of the meta-theory of $\lambda^{\Pi\multimap\&\top}$ considerably. The principal consequence of this choice is that definitional equality can be based entirely on β-reduction and, more importantly, can be defined independently from typing, as shown in Figure 4. Therefore, the analysis of the equational theory of $\lambda^{\Pi\multimap\&\top}$ can be conducted in a totally self-contained manner. Indeed, the results that we will present in this subsection, in particular the Church-Rosser theorem, apply to arbitrary $\lambda^{\Pi\multimap\&\top}$ terms and not just those that are valid according to the typing rules of our language. This apparently unnecessary generality is the first step towards disentangling the meta-theory of $\lambda^{\Pi\multimap\&\top}$. Had we relaxed extensionality by considering an equational theory containing η-conversion rules, as normally done in the literature, we would have been forced to provide mutually recursive definitions for the typing and the definitional equality judgments. In particular, the equational theory of the resulting formalism could not be studied in isolation and most of its meta-theory would collapse in a dense theorem consisting of a discouraging number of mutually dependent properties. We will come back to this point at the end of this subsection.

As we already anticipated, the principal result of this section will be the Church-Rosser theorem for parallel nested reduction. We will rely on this property in order to prove the uniqueness of normal forms, and therefore the decidability of the equational theory of *LLF*.

The proofs of the results in this section adapt the technique originally devised by Tait and Martin-Löf for the traditional untyped λ-calculus [Bar80]. A very detailed presentation of that method, as well as its formalization in *Elf*, can be found in [Pfe99]. We deviate from this presentation in order to take into account all the entities of *LLF* that participate in the definition of parallel nested reduction. Specifically, we treat the linear constructs of our language and the new forms of β-reduction they introduce; we also need to consider types and kinds.

The parallel nested reduction strategy defined in Figure 4 is based on the four β-reduction rules **or_beta_fst**, **or_beta_snd**, **or_beta_lin** and **or_beta_int**. All the other rules are congruences that allow applying reductions to subterms. Notice that the β-reduction rules are directional: the expression on the left-hand side of the arrow is a β-redex, and we like to think of the expression on the right-hand side as "simpler", even if it may be larger, or contain more β-redices, than the term on the other side of the arrow.

A key result in the study of the definitional equality of $\lambda^{\Pi\multimap\&\top}$ is that substituting a variable in a reducible term maintains its reducibility. This property is formalized and generalized in the following lemma.

Lemma 2.1 (*Substitution*)

Assume that there exist a derivation for $N \longrightarrow N'$. *Then,*

i. if $\mathcal{R} :: U \longrightarrow U'$, *then* $[N/x]U \longrightarrow [N'/x]U'$;

ii. *if* $\mathcal{R} :: U \longrightarrow^{*} U'$, *then* $[N/x]U \longrightarrow^{*} [N'/x]U'$;

iii. *if* $\mathcal{R} :: U \equiv U'$, *then* $[N/x]U \equiv [N'/x]U'$.

Proof.

We proceed by induction on $\mathcal{R}$ in each case. ☑

A term can contain several β-redices and the parallel reduction strategy can reduce any of them, possibly zero or more than one. Therefore, a term U can in general reduce to a number of distinct terms $U_1, \ldots, U_n$. However, a fundamental property of this strategy is that there always exist a common term V to which all these terms are reducible. This is known as the *Diamond Property* and it is stated below.

Lemma 2.2 (*Diamond property*)

If $\mathcal{R}' :: U \longrightarrow U'$ *and* $\mathcal{R}'' :: U \longrightarrow U''$, *then there is a term* V *such that* $U' \longrightarrow V$ *and* $U'' \longrightarrow V$.

Proof.

By induction on the structure of U and inversion on $\mathcal{R}'$ and $\mathcal{R}''$. Functional β-reduction is handled through the Substitution Lemma. ☑

Although parallel nested reduction has the possibility of reducing several β-redices, it is not sufficient in general to produce the normal form for a term, even when it exists. For example, the following judgment shows on the right-hand side the simplest term the expression on the left-hand side can be reduced to:

$$(\lambda x\!:\!(a \to a) \to (a \to a).\, x\ c)\ (\lambda y\!:\!a \to a.\, y) \ \longrightarrow\ (\lambda y\!:\!a \to a.\, y)\ c$$

Notice that one further step would suffice to obtain the normal form of that expression, which is c. Recall the above judgment is derivable regardless of whether the terms in it are typable.

In order to achieve normal forms when they exist, we need to chain parallel nested reductions by taking their transitive closure. *Confluence* extends the Diamond Property to $\longrightarrow^{*}$. The definitional equivalence relation is symmetric and therefore can reverse reduction steps. In general it can relate terms in which some subterm has been expanded while some other subterm has been reduced. It is however always possible to reduce equivalent entities back to a common term. This fundamental property is known as the *Church-Rosser theorem.* These two properties are stated below. They follow from the Diamond Property by virtue of general techniques [DJ90].

Theorem 2.3 (*Church-Rosser*)

Confluence: *If* $U \longrightarrow^{*} U'$ *and* $U \longrightarrow^{*} U''$, *then there is a term* V *such that* $U' \longrightarrow^{*} V$ *and* $U'' \longrightarrow^{*} V$.

Church-Rosser: *If* $U' \equiv U''$, *then there is a term* V *such that* $U' \longrightarrow^{*} V$ *and* $U'' \longrightarrow^{*} V$. □

The properties above apply to arbitrary terms, possibly ill-typed or in general invalid according to the pre-canonical system above (indeed, our example above contained the subterm $\lambda y : a \to a.\, y$ which is not η-expanded). Although definitional equality is always invoked with valid terms in the rules in Figures 2–3, intermediate terms participating in the equivalence derivation might not be pre-canonical. We will show that it is possible to limit the intermediate terms produced during a definitional equality test to entities that are valid.

Arbitrary terms do not have in general a normal form. A classical example is the term $(\lambda x\!:\!a.\, x\, x)\, (\lambda x\!:\!a.\, x\, x)$. This term reduces to itself and therefore it is not possible to eliminate the β-redex it contains by reduction. However, every valid term, i.e., every term that appears in a derivable typing judgment in our pre-canonical system admits a normal form. Furthermore, the strong normalization theorem proved below will show that the order in which β-redices are reduced is not important.

We conclude this subsection with a short discussion about notions of definitional equality that includes rules for η-conversion. If we were simply to add η-rules based on the following reductions

$$\begin{array}{ll}\eta_{pair}: & \langle \mathrm{FST}\, M, \mathrm{SND}\, M\rangle \longrightarrow M \\ \eta_{llam}: & \hat{\lambda} u\!:\!A.\, M\,\hat{}\,u \longrightarrow M \\ \eta_{ilam}: & \lambda x\!:\!A.\, M\; x \longrightarrow M\end{array}$$

with u and x not occurring free in M in the last two rules, then the Church-Rosser property would cease to hold: it is observed in [HHP93] that the term $\lambda x : A.\, (\lambda y : B.\, M)\, x$ reduces via η_{ilam} to $\lambda y\!:\!B.\, M$ and by β to $\lambda x : A.\, [x/y]M$, i.e. $\lambda y : A.\, M$; however, the two resulting terms can be equated only if A and B have a common reduct, which is not the case for certain ill-typed terms.

Furthermore, the simple η-conversion rules above are insufficient to capture the notion of definitional equality we are interested in. The situation is actually more serious than that: in the presence of a unit type, $\top$ in our case, the equational theory we need cannot be axiomatized by means of schematic reduction rules such as the above. Indeed, $\langle\rangle$ is the unique inhabitant of type $\top$ and therefore every object of type $\top$ is η-equivalent to $\langle\rangle$. However, adding the rule $M \longrightarrow \langle\rangle$ is clearly unsound since it does not take typing information into account. The correct approach to this problems requires typed definitional equality judgments. The specific inference rules that handle η-expansion are called *extensionality* rules and only vaguely resemble the η-conversions presented above. It seems that the η-rules can serve as a foundation of the definitional equality of type theories only in strongly restricted circumstances.

With a typed equivalence relation including extensionality, the meta-theory of the resulting system becomes tremendously complicated. Indeed, in this section we were able to study the meta-theory of our notion of definitional equality without referring to the typing judgments of the pre-canonical system. This is no longer possible with the judgments just presented. The properties of meta-theory become dependent on each other and must be stated and proved simultaneously.

2.4 Fundamental Properties

The purpose of this subsection is to illustrate the basic properties of the pre-canonical deduction system presented earlier. Many of these results are interesting by themselves since they provide insight about the type theory of *LLF* beyond what is apparent from the inference rules. Moreover, most of these properties will play a role in the development of the proof of the decidability of type checking for our language.

First, we turn our attention to the lowest level features of $\lambda^{\Pi\multimap\&\top}$. Handling variables is unpleasant since the definitions that concern these entities, the domain of a context or the free variables in a term for example, are set-theoretic. However, basic operations such as substitution rely on them and therefore it is not possible to avoid dealing with them. All constructions in this paper, e.g., terms and derivations, have a simple structural definition and admit identity modulo renaming of bound variables as their notion of syntactic equality. Therefore, they most proofs involving them rely on elegant structural inductions. A similar treatment of variables is much harder to achieve due to the necessity of taking into consideration the complex equational theory of sets.

Fortunately, we are able to factor out the principal properties of variables in our language in a single lemma, whose very long proof can be found in [Cer96]. Only a limited number of further results will mention free variables or domains explicitly. In reading this statement, recall that terms differing by the name of bound variables are considered syntactically equal and that the names of all variables declared in a context are distinct. Moreover, we remind the reader that $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ is an abbreviation for the pre-canonical or pre-atomic judgments at the level of objects, types and kinds.

Lemma 2.4 (*Free variables*)

i. *If* $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$, *then* $\mathrm{FV}(U) \subseteq \mathrm{dom}\,\Psi$ *and* $\mathrm{FV}(V) \subseteq \mathrm{dom}\,\overline{\Psi}$;

ii. *If* $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ *or* $\vdash_\Sigma \Psi \Uparrow Ctx$, *then* $\mathrm{FV}(\Psi) \subseteq \mathrm{dom}\,\overline{\Psi}$;

iii. *If* $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ *or* $\vdash_\Sigma \Psi \Uparrow Ctx$ *or* $\vdash \Sigma \Uparrow Sig$, *then* $\mathrm{FV}(\Sigma) = \varnothing$;

iv. *If* $\Psi, x \mathbin{\hat{:}} A, \Psi' \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ *or* $\vdash_\Sigma \Psi, x \mathbin{\hat{:}} A, \Psi' \Uparrow Ctx$, *then* $\mathrm{FV}(A) \cup \mathrm{FV}(\Psi) \subseteq \mathrm{dom}\,\overline{\Psi}$. □

This lemma provides some insight about how and where free variables can appear in a derivable judgment in $\lambda^{\Pi\multimap\&\top}$. By (*i*) and (*ii*), we have that all free variables occurring in a valid judgment must be declared in its context. Property (*iv*) specifies instead that the free variables in an assumption must have been declared in the part of the context that is to its left, i.e., assumptions can depend only on declarations made before them. Item (*iii*) in the lemma states instead that a signature cannot contain free variables. All these properties already hold in λ^{Π}. The peculiarities of our language appear when analyzing the role of free variables that are assumed linearly. Since the context of the judgments for types and kinds are strictly intuitionistic, (*i*) entails that free linear variables are permitted only in valid terms from the level of objects. Moreover, (*ii*) implies that no assumption in the context can depend on a linear variable. These strict constraints are a consequence of the property of our language of permitting only linearly closed expressions as indices to type families.

We now present a number of properties that can be seen as admissible rules of inference. First we have that assumptions in the context can be exchanged freely as long as they do not violate dependencies among them. More precisely, if $x \mathrel{\hat{::}} A$ immediately precedes $y \mathrel{\hat{::}} B$ and x does not occur free in B, then the relative order of these two assumptions can be exchanged. This idea is generalized in the lemma below. Permutation depends on weakening, which itself requires permutation in its proof. Therefore, we need to state and prove both properties at the same time. Notice that weakening forbids adding linear assumptions into the context.

Lemma 2.5 (*Structural properties of contexts*)

Permutation:

i. *If* $\mathcal{P}' :: (\Psi, \Psi', x \mathrel{\hat{::}} A, \Psi'' \vdash_\Sigma U \Uparrow\!\Downarrow V)$ *and* $\mathcal{P}'' :: (\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE})$, *then* $\Psi, x \mathrel{\hat{::}} A, \Psi', \Psi'' \vdash_\Sigma U \Uparrow\!\Downarrow V$*;*

ii. *If* $\mathcal{P}' :: (\vdash_\Sigma \Psi, \Psi', x \mathrel{\hat{::}} A, \Psi'' \Uparrow Ctx)$ *and* $\mathcal{P}'' :: (\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE})$, *then* $\vdash_\Sigma \Psi, x \mathrel{\hat{::}} A, \Psi', \Psi'' \Uparrow Ctx$*.*

Weakening:

If $\mathcal{P}' :: (\Psi \vdash_\Sigma U \Uparrow\!\Downarrow V)$ *and* $\mathcal{P}'' :: (\vdash_\Sigma \Psi, \overline{\Psi'} \Uparrow Ctx)$, *then* $\Psi, \overline{\Psi'} \vdash_\Sigma U \Uparrow\!\Downarrow V$*.*

Proof.

By simultaneous induction on $\mathcal{P}'$ and, in the case of permutation, on the length of Ψ''. We consider $\mathcal{P}''$ smaller than any subderivation of $\mathcal{P}'$ when applying the induction hypothesis. ☑

The permutation property has important consequences on the linear assumptions in the context. As we described earlier, no assumption in a context Ψ can depend on a linear variable. Therefore, the permutation lemma allows us to shift all these variables to end of Ψ. Let us write $\widehat{\Psi}$ for the linear assumptions of a context Ψ; notice that, because of possible dependencies on intuitionistic variables, $\widehat{\Psi}$ is not necessarily a valid context. We can then write Ψ as $\overline{\Psi}, \widehat{\Psi}$, or even $\overline{\Psi}; \widehat{\Psi}$ as in recent presentations of linear logic [Gir93, HM94, Pfe94c]. Furthermore, since the variables in the domain of $\widehat{\Psi}$ cannot occur in $\widehat{\Psi}$, we are free to permute the contents of this part of the context. Therefore, $\widehat{\Psi}$ can be treated as a multiset.

We could expect a further property, *strengthening*, to be part of the statement of the lemma above; the presence of this property would actually make that proof easier, but unfortunately we do not have yet the tools to prove it.

Strengthening states that, whenever a variable is declared in a context but does not occur free anywhere in a judgment, then it could be safely dropped and the judgment would still be provable. However, strengthening requires the strong normalization theorem, proved in Section 2.5. Even though a variable does not occur in a derivable judgment, it is possible that the application of one of the definitional equivalence rules produces a term containing it, so that not the judgment itself, but its derivation mentions it. These uses of equivalence are non-essential and can be removed from the derivation. However, we will be able to prove this only as a by-product of strong normalization.

Next, we present a technical result that, although of minor importance in itself, plays a fundamental role in the statement of the adequacy theorems for the representation of an object language. Specifically, when the meta-theory of the object language expects certain objects to be linear, an adequate encoding would require free linear variables in the indices of base types. This is not achievable in *LLF* since our language does not admit linear dependent types. We bypass the problem by encoding these linear entities as intuitionistic assumptions. Linearity conditions can be checked at an earlier stage of the computation. This technique permits us to give an effective representation to complex linear judgments without dealing with the complications of linear dependent types. Examples showing how this issue is handled in *LLF* can be found in [Cer96].

The lemma below states that whenever a derivable term mentions a linear variable, then we can safely make it intuitionistic. Intuitively, linear variables must be used once while intuitionistic variables can be used as many times as desired.

Lemma 2.6 (*Promotion*)

If $\mathcal{P} :: (\Psi, u \hat{:} A, \Psi' \vdash_\Sigma M \Uparrow\!\!\downarrow B)$, *then* $\Psi, u{:}A, \Psi' \vdash_\Sigma M \Uparrow\!\!\downarrow B$.

Proof.

By induction on the structure of $\mathcal{P}$. ☑

An important ingredient of the proofs of the theorems below are the lemmas that we call transitivity, following the terminology in [HHP93]. These results permit us to interpret assumptions as place-holders for unspecified derivations. Whenever a provable judgment depends on the assumption $x \,\hat{::}\, A$, then any derivation of a term N of type A satisfying certain context conditions can be substituted into the original derivation and still maintain its validity. Therefore, judgments containing assumptions can be thought of as parametric expressions. The transitivity lemmas specify how to instantiate these parameters.

These results are of paramount importance in the suitability of $\lambda^{\Pi\multimap\&\top}$ as the meta-language of the logical framework *LLF*. They are the formal justification of the representation of the hypothetical and parametric judgments, so common in formal systems, as simple and dependent function types, respectively. The transitivity lemmas, modulo the Inversion Lemma below, postulate that these operators have a semantics that mimics the behavior of those forms of judgment. *LLF* extends this correspondence, already present in *LF*, to capture hypothetical judgments where the hypotheses are linear.

The transitivity lemmas are tightly connected to several results in logic and type theory. The interpretation depends on which part of the judgments we focus our attention on. From the point of view of the λ-calculus embedded in our language, these lemmas can be seen as substitution principles since they describe how variables can be substituted into valid terms while preserving validity. In this, they are closely related to the notion of subject reduction for functional objects.

From the logical perspective, under the interpretation of types as formulas, the transitivity lemmas state the admissibility of the cut rule for intuitionistic and linear formulas. Whenever a formula B relies on an assumption A, any evidence of the validity of A, possibly on the basis of further assumptions, can be included directly in an equivalent proof of B that does not mention A among its hypotheses.

Linear and intuitionistic assumptions need to be treated separately since they require a different structuring of the context. Therefore, we distinguish two transitivity lemmas. This corresponds to differentiating two substitution principles or to having a linear and an intuitionistic cut rule (see [Hod94, Pfe94b]).

Lemma 2.7 (*Intuitionistic transitivity*)

i. *If* $\overline{\Psi} \vdash_\Sigma N \Uparrow A$ *and* $\mathcal{P} :: (\Psi, x\!:\!A, \Psi' \vdash_\Sigma U \Uparrow\!\!\Downarrow V)$, *then* $\Psi, [N/x]\Psi' \vdash_\Sigma [N/x]U \Uparrow\!\!\Downarrow [N/x]V$;

ii. *If* $\overline{\Psi} \vdash_\Sigma N \Uparrow A$ *and* $\mathcal{P} :: (\vdash_\Sigma \Psi, x\!:\!A, \Psi' \Uparrow \mathit{Ctx})$, *then* $\vdash_\Sigma \Psi, [N/x]\Psi' \Uparrow \mathit{Ctx}$.

Proof.

By induction on the structure of $\mathcal{P}$. ☑

We now state the linear transitivity lemma. Notice that, differently from the intuitionistic case, the context in the resulting judgment can be larger than the contexts mentioned in either premiss.

Lemma 2.8 (*Linear transitivity*)

If $\Psi \vdash_\Sigma N \Uparrow A$ *and* $\mathcal{P} :: (\overline{\Psi}, u \mathrel{\hat{:}} A, \Psi' \vdash_\Sigma M \Uparrow\!\!\Downarrow B)$, *then* $\Psi, \Psi' \vdash_\Sigma [N/u]M \Uparrow\!\!\Downarrow B$.

Proof.

We proceed by induction on the structure of $\mathcal{P}$. Weakening is required in this proof. ☑

The cumulative validity lemma below is important in its own. It says that whenever a judgment is derivable, all entities mentioned in it are themselves valid, i.e., have derivations validating them.

Lemma 2.9 (*Consistency*)

i. *If* $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow\!\!\Downarrow A$, *then* $\overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}$;

ii. *If* $\mathcal{P} :: \overline{\Psi} \vdash_\Sigma A \Uparrow\!\!\Downarrow K$, *then* $\overline{\Psi} \vdash_\Sigma K \Uparrow \mathit{Kind}$;

iii. *If* $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$, *then* $\vdash_\Sigma \Psi \Uparrow \mathit{Ctx}$;

iv. *If* $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ *or* $\mathcal{P} :: \vdash_\Sigma \Psi \Uparrow \mathit{Ctx}$, *then* $\vdash \Sigma \Uparrow \mathit{Sig}$.

Proof.

By induction on the structure of $\mathcal{P}$. We need intuitionistic transitivity in order to handle the dependent function type constructor. ☑

The traditional Curry-Howard interpretation associates types to formulas and terms to proofs of these formulas. Clearly, a single formula can have more than one proof, expressed in type theory by admitting several objects of the same type, possibly infinitely many. This is also consistent with the view of types as sets and terms as their elements. In the logical interpretation, we expect every proof to be the proof of a single formula. This property might not be desirable as stated in set theory (an element belongs to all the supersets of the singleton set containing it) or in type theory (e.g., if we consider theories admitting a hierarchy of types ruled by a subtyping relation, as in [CDCV81, Pfe93]). However, this property holds in the case of languages such as *LF* and *LLF*. Uniqueness, in these frameworks, is considered modulo definitional equality. Therefore, every derivable $\lambda^{\Pi\multimap\&\top}$ object level term has a unique type. The validity of this property, which extends naturally to kinds, is essential in order to prove the decidability of type checking in our language. It is formally stated in the following lemma.

Lemma 2.10 (*Uniqueness of types and kinds*)

i. If $\mathcal{P}' :: \Psi' \vdash_\Sigma M \Uparrow\!\!\Downarrow A'$ *and* $\mathcal{P}'' :: \Psi'' \vdash_\Sigma M \Uparrow\!\!\Downarrow A''$ *with* $\Psi'_{|\mathrm{FV}(M)} = \Psi''_{|\mathrm{FV}(M)}$ *and where the arrows do not need to match, then* $A' \equiv A''$*;*

ii. If $\mathcal{P}' :: \overline{\Psi} \vdash_\Sigma A \Uparrow\!\!\Downarrow K'$ *and* $\mathcal{P}'' :: \overline{\Psi} \vdash_\Sigma A \Uparrow\!\!\Downarrow K''$ *where the arrows do not need to match, then* $K' \equiv K''$*.*

Proof.

By induction on the structure of $\mathcal{P}'$ and $\mathcal{P}''$. The idea is to unfold these derivations until an introduction or an elimination rule is exposed, then we apply the induction hypothesis on the subderivations. ☑

A fundamental proof technique when dealing with deductive systems is *inversion*. Given a particular instance of a judgment, it allows identifying a limited number of inference rules whose conclusion matches this judgment. Each matching rule constitutes an alternative case and the judgments obtained by instantiation of its premisses can be used in order to draw further inferences. Notice however that a conclusion of these inferences can be accepted as valid with respect to the original judgment only if it is supported by all the matching rules. This technique is general and can be applied in our system. Deductive systems having the property that every rule of inference is fully determined by the shape of a particular term in its conclusion are particularly useful since matching this term yields a single rule. Therefore, further inferences can be drawn on the basis of its premisses, without having to consider alternatives. The essential rules in the pre-canonical deductive system for $\lambda^{\Pi\multimap\&\top}$ have this property with respect to the term they validate. However, the equivalences and the rules that bridge the pre-atomic and pre-canonical judgments do not change the derived term and can therefore be seen as filters or pipelines that connect these essential rules, from the standpoint of this term. A detailed analysis of these rules shows that we can indirectly recover the stronger form of inversion. This desirable property is expressed by the following lemma.

Lemma 2.11 (*Inversion*)

i. *If* $\mathcal{P} :: (\Psi \vdash_\Sigma \langle M, N\rangle \Uparrow\!\!\Downarrow A \,\&\, B)$*, then* $\Psi \vdash_\Sigma M \Uparrow\!\!\Downarrow A$ *and* $\Psi \vdash_\Sigma N \Uparrow\!\!\Downarrow B$*;*

ii. *If* $\mathcal{P} :: (\Psi \vdash_\Sigma \hat{\lambda} u\!:\!A.\, M \Uparrow\!\!\Downarrow A' \multimap B)$*, then* $\Psi, u \,\hat{:}\, A \vdash_\Sigma M \Uparrow\!\!\Downarrow B$ *and* $A \equiv A'$*;*

iii. *If* $\mathcal{P} :: (\Psi \vdash_\Sigma \lambda x\!:\!A.\, M \Uparrow\!\!\Downarrow \Pi x\!:\!A'.\, B)$*, then* $\Psi, x\!:\!A \vdash_\Sigma M \Uparrow\!\!\Downarrow B$ *and* $A \equiv A'$*.*

Proof.

By induction on the structure of $\mathcal{P}$. All these results apply the same technique: the derivation is unfolded until an introduction or an elimination appears as its last inference rule. ☑

The last property we present in this section is extensionality. It confirms that all valid terms according to the pre-canonical judgment of the object level are indeed pre-canonical, i.e., in η-long form. It forbids, for example, a constant c of compound type A to be derivable by means of a judgment of the form $\Psi \vdash_\Sigma c \Uparrow A$, whatever the signature Σ and the context Ψ are.

Lemma 2.12 (*Extensionality*)

i. *If* $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow \top$*, then* $M = \langle\rangle$*;*

ii. *If* $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow A \,\&\, B$*, then* $M = \langle M_1, M_2\rangle$*;*

iii. *If* $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow A \multimap B$*, then* $M = \hat{\lambda} u\!:\!A'.\, M_1$ *with* $A \equiv A'$*;*

iv. *If* $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow \Pi x\!:\!A.\, B$*, then* $M = \lambda x\!:\!A'.\, M_1$ *with* $A \equiv A'$*.*

Proof.

By induction on the structure of $\mathcal{P}$. ☑

2.5 Strong Normalization

We showed in Section 2.3 that there exist terms that do not have a normal form. The problem of whether a term can be normalized is undecidable in general. Indeed, functional programming languages adopt reduction as their operational semantics. Therefore, normalization is equivalent to the *halting problem* for these languages. However, as soon as a strong typing discipline, in the style of λ^Π for example, is imposed, all well-formed terms are normalizing. The motto "well-typed programs cannot go wrong", often used in a slightly different context, summarizes the situation well.

The aim of this subsection is to show that this result applies to $\lambda^{\Pi\multimap\&\top}$ as well. We will do more and prove the strong normalization theorem for our language. This property implies that every valid $\lambda^{\Pi\multimap\&\top}$ term U has a normal form $\mathrm{NF}(U)$, that this normal form is unique, and that it can be obtained by performing β-reductions in arbitrary order in U.

In order to prove this property, we adapt the technique originally proposed for *LF* in [HHP93]. We will present a translation of $\lambda^{\Pi\multimap\&\top}$ into the simply typed λ-calculus with pairs $\lambda^{\times\rightarrow}$. The effect of this encoding will be to eliminate dependencies and linearity, considerably simplifying the treatment of the calculus. This translation will have two fundamental properties: first it will maintain well-typedness, so that valid terms in $\lambda^{\Pi\multimap\&\top}$ will be mapped to terms that are valid in $\lambda^{\times\rightarrow}$. Second, it will preserve reductions so that every reduction sequence in $\lambda^{\Pi\multimap\&\top}$ will correspond to a reduction sequence in $\lambda^{\times\rightarrow}$. The strong normalization theorem for $\lambda^{\Pi\multimap\&\top}$ will then be a consequence of the same property of $\lambda^{\times\rightarrow}$.

The first step towards proving the strong normalization theorem is given by the following lemma. It states that derivability is closed under reduction, i.e., if a term U is valid in $\lambda^{\Pi\multimap\&\top}$, then every term U' that differs from U only by the application of a β-reduction step is also valid. This property is known as *subject reduction*. We write $U \longrightarrow^{+} V$ if $U \longrightarrow^{*} V$ and $U \neq V$. Notice that the symmetric property considering β-expansion does not hold in general.

Lemma 2.13 (*Subject reduction*)

If $\mathcal{P} :: \Psi \vdash_{\Sigma} U \Updownarrow V$ *and* $\mathcal{R} :: U \longrightarrow^{+} U'$, *then* $\Psi \vdash_{\Sigma} U' \Updownarrow V$.

Proof.

By induction on the structure of $\mathcal{P}$ and inversion on $\mathcal{R}$. ☑

We now present the simply typed λ-calculus $\lambda^{\times\rightarrow}$ which we will use in order to prove the strong normalization theorem for *LLF*. With a careful encoding, the validity of this property in our language will be reduced to the fact that it holds in $\lambda^{\times\rightarrow}$, which we assume from the literature.

The language $\lambda^{\times\rightarrow}$ is an extension of Church's simply typed λ-calculus [Chu40] with product types of the form $\sigma_1 \times \sigma_2$ and the corresponding object level entities, namely pairs and the two projections. Product types are needed to encode the additive type constructor & of $\lambda^{\Pi\multimap\&\top}$. As a matter of convenience, we distinguish between constants and variables. The former are declared in a signature, while the latter are assumed in the context. The choice of using a calculus à la Church or in the style of Curry is arbitrary. Harper, Honsell and Plotkin adopted the latter solution in the original *LF* paper [HHP93]. We rely on the former only because it makes the types of the bound variables in the encoding below directly available.

The language of $\lambda^{\times\rightarrow}$ is specified by the following grammar:

$$\begin{array}{rrcl}
\textit{Objects:} & M & ::= & c \mid x \mid \langle M_1, M_2\rangle \mid \textsf{FST}\,M \mid \textsf{SND}\,M \mid \lambda x\!:\!\sigma.\,M \mid M_1\,M_2 \\
\textit{Types:} & \sigma & ::= & b \mid \sigma_1 \times \sigma_2 \mid \sigma_1 \rightarrow \sigma_2 \\
\textit{Contexts:} & \Gamma & ::= & \cdot \mid \Gamma, x\!:\!\sigma \\
\textit{Signatures:} & \Sigma & ::= & \cdot \mid \Sigma, c\!:\!\sigma
\end{array}$$

We have overloaded some of the operators of $\lambda^{\Pi\multimap\&\top}$ to indicate analogous symbols of $\lambda^{\times\rightarrow}$. However, the distinction should be clear from the context where the different languages are used. The symbols b, c and x range over base types, constants and variables, respectively. For our purposes, we will need a single base type, that we denote as ω. The definitions of free and bound variables are given similarly to $\lambda^{\Pi\multimap\&\top}$. As usual, we identify terms that differ only in the name of their bound variables, and write $[N/x]M$ for the capture avoiding substitution of N for x in M. Finally, we require variables and constants to be declared at most once in a context and in a signature, respectively; contexts and signatures will be treated as multisets.

Typing rules

$$\frac{}{\Gamma \vdash_{\Sigma, c:\sigma} c : \sigma}\,\textbf{st\_con} \qquad \frac{}{\Gamma, x{:}\sigma \vdash_{\Sigma} x : \sigma}\,\textbf{st\_var}$$

$$\frac{\Gamma \vdash_\Sigma M_1 : \sigma_1 \quad \Gamma \vdash_\Sigma M_2 : \sigma_2}{\Gamma \vdash_\Sigma \langle M_1, M_2\rangle : \sigma_1 \times \sigma_2}\,\textbf{st\_pair} \qquad \frac{\Gamma \vdash_\Sigma M : \sigma \times \sigma'}{\Gamma \vdash_\Sigma \textsc{fst}\, M : \sigma}\,\textbf{st\_fst} \qquad \frac{\Gamma \vdash_\Sigma M : \sigma' \times \sigma}{\Gamma \vdash_\Sigma \textsc{snd}\, M : \sigma}\,\textbf{st\_snd}$$

$$\frac{\Gamma, x{:}\sigma_1 \vdash_\Sigma M : \sigma_2}{\Gamma \vdash_\Sigma \lambda x{:}\sigma_1.\, M : \sigma_1 \to \sigma_2}\,\textbf{st\_lam} \qquad \frac{\Gamma \vdash_\Sigma M_1 : \sigma_2 \to \sigma_1 \quad \Gamma \vdash_\Sigma M_2 : \sigma_2}{\Gamma \vdash_\Sigma M_1\, M_2 : \sigma_1}\,\textbf{st\_app}$$

One−step reduction *Congruences and reductions*

$$\frac{M \longrightarrow_1 M'}{\textsc{fst}\, M \longrightarrow_1 \textsc{fst}\, M'}\,\textbf{str1\_fst} \qquad \frac{M \longrightarrow_1 M'}{\textsc{snd}\, M \longrightarrow_1 \textsc{snd}\, M'}\,\textbf{str1\_snd}$$

$$\frac{M \longrightarrow_1 M'}{\langle M, N\rangle \longrightarrow_1 \langle M', N\rangle}\,\textbf{str1\_pair1} \qquad \frac{N \longrightarrow_1 N'}{\langle M, N\rangle \longrightarrow_1 \langle M, N'\rangle}\,\textbf{str1\_pair2}$$

$$\frac{M \longrightarrow_1 M'}{M\, N \longrightarrow_1 M'\, N}\,\textbf{str1\_app1} \qquad \frac{N \longrightarrow_1 N'}{M\, N \longrightarrow_1 M\, N'}\,\textbf{str1\_iapp2}$$

$$\frac{M \longrightarrow_1 M'}{\lambda x{:}\sigma.\, M \longrightarrow_1 \lambda x{:}\sigma.\, M'}\,\textbf{str1\_lam2} \qquad \frac{}{(\lambda x{:}\sigma.\, M)\, N \longrightarrow_1 [N/x]M}\,\textbf{str1\_beta}$$

$$\frac{}{\textsc{fst}\, \langle M, N\rangle \longrightarrow_1 M}\,\textbf{str1\_beta\_fst} \qquad \frac{}{\textsc{snd}\, \langle M, N\rangle \longrightarrow_1 N}\,\textbf{str1\_beta\_snd}$$

Transitive closure

$$\frac{M \longrightarrow_1 M'}{M \longrightarrow_1^+ M'}\,\textbf{st1*\_r} \qquad \frac{M \longrightarrow_1^+ M' \quad M' \longrightarrow_1^+ M''}{M \longrightarrow_1^+ M''}\,\textbf{st1*\_trans}$$

Figure 5: The Simply Typed λ-calculus with Pairs $\lambda^{\times\to}$

The typing rules for $\lambda^{\times\to}$ are presented in Figure 5. We use the same notation for the corresponding judgments, again relying on the surrounding context for distinguishing them. We base the equational theory of $\lambda^{\times\to}$ on the one-step reduction relation $\longrightarrow_1$, which is slightly more appropriate for our purposes than parallel nested reduction. We write $\longrightarrow_1^+$ for its transitive closure. We will rely on some basic properties for these judgments. We do not state or prove them formally since they resemble similar properties of *LLF* and are well-known from the literature. A further property that $\lambda^{\times\to}$ enjoys is strong normalization: if $\Gamma \vdash_\Sigma M : \sigma$, then every reduction sequence on M terminates and, by confluence, yields a unique normal form for this term. Proofs of this and stronger properties for extensions of this language can be found in the literature [Gan80, Tro86].

The encoding we propose transforms *LLF* judgments $\Psi \vdash_\Sigma U \Updownarrow V$ into $\lambda^{\times\to}$ judgments of the form $\Gamma \vdash_{\Sigma'} M : \sigma$. It maps the generic term U to an object M in $\lambda^{\times\rightarrow}$, V to a simple type σ, Σ to a signature Σ', and Ψ to a context Γ. We now present the four parts that constitute this translation.

We use the function $\tau(\_)$ to denote the translation of a term that appears on the right-hand side of the arrow of an *LLF* judgment. These terms can be either types, kinds or the symbol *Kind*, which we map to ω. Given a type or kind U, $\tau(U)$ is a simple type of $\lambda^{\times\rightarrow}$ that maintains the structure of U, but forgets dependencies and linearity. Type families are mapped to the base type ω, the additive product type constructor & of *LLF* is encoded into the (intuitionistic) product types $\times$ of $\lambda^{\times\rightarrow}$, and both function type constructors $\multimap$ and Π of our language are represented by the unique arrow of that calculus. Kinds are treated similarly. Specifically, we have the following definition for $\tau(\_)$.

Types:		*Kinds:*	
$\tau(P)$	$= \omega$	$\tau(\text{TYPE})$	$= \omega$
$\tau(\top)$	$= \omega$	$\tau(\Pi x\!:\!A.\,K)$	$= \tau(A) \rightarrow \tau(K)$
$\tau(A \,\&\, B)$	$= \tau(A) \times \tau(B)$		
$\tau(A \multimap B)$	$= \tau(A) \rightarrow \tau(B)$		
$\tau(\Pi x\!:\!A.\,B)$	$= \tau(A) \rightarrow \tau(B)$		

As an example, the $\lambda^{\Pi\multimap\&\top}$ type

$$A \;=\; a \multimap ((\Pi x\!:\!a \multimap a.\, a) \;\;\&\;\; (a \multimap \Pi y\!:\!a.\, a))$$

encountered earlier has the following encoding in $\lambda^{\times\rightarrow}$:

$$\tau(A) \;=\; \omega \rightarrow (((\omega \rightarrow \omega) \rightarrow \omega) \;\;\times\;\; (\omega \rightarrow \omega \rightarrow \omega)).$$

A term U appearing immediately to the left of the arrow of an *LLF* judgment $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ is mapped to a $\lambda^{\times\rightarrow}$ object by means of the function $|\_|$. U can be an object, a type, a type family or a kind.

The encoding of objects maps variables to variables, constants to constants, and constructors and destructors of $\lambda^{\Pi\multimap\&\top}$ to the corresponding operator of $\lambda^{\times\rightarrow}$. The two forms of λ-abstraction of *LLF* require a slightly more complex mapping in order to represent in $\lambda^{\times\rightarrow}$ the terms appearing in the type of their bound variable. We cope with this issue by encoding $\hat{\lambda} x\!:\!A.\,M$, for example, as $(\lambda y\!:\!\omega.\, \lambda x\!:\!\tau(A).\, |M|)\; |A|$. The expected translation of that term is $\lambda x\!:\!\tau(A).\, |M|$. We wrap around it the β-redex $(\lambda y : \omega.\, \_)\; |A|$ in order to account for possible reductions performed in the $\lambda^{\Pi\multimap\&\top}$ type A. This redex is vacuous since y is a fresh variable not appearing in A or M.

The encoding of types and kinds translates each $\lambda^{\Pi\multimap\&\top}$ operator as a constant in $\lambda^{\times\rightarrow}$, which is applied to the encoding of the arguments. The dependent type and kind constructor Π require a functional second argument since its semantics introduces assumptions in the context.

We have the following definition for this encoding function, where π, possibly subscripted, denotes constants in $\lambda^{\times\rightarrow}$:

Objects:

$$\begin{array}{rcl}
|x| &=& x \\
|c| &=& \pi_c \\
|\langle\rangle| &=& \pi_{\langle\rangle} \\
|\langle M, N\rangle| &=& \langle |M|, |N| \rangle \\
|\text{FST}\, M| &=& \text{FST}\, |M| \\
|\text{SND}\, M| &=& \text{SND}\, |M| \\
|\hat{\lambda} u\!:\!A.\, M| &=& (\lambda y\!:\!\omega.\, \lambda u\!:\!\tau(A).\, |M|)\; |A| \\
|M\,\hat{}\,N| &=& |M|\; |N| \\
|\lambda x\!:\!A.\, M| &=& (\lambda y\!:\!\omega.\, \lambda x\!:\!\tau(A).\, |M|)\; |A| \\
|M\; N| &=& |M|\; |N|
\end{array}$$

Types and kinds:

$$\begin{array}{rcl}
|a| &=& \pi_a \\
|P\; N| &=& |P|\; |N| \\
|\top| &=& \pi_\top \\
|A \,\&\, B| &=& \pi_\&\; |A|\; |B| \\
|A \multimap B| &=& \pi_\multimap\; |A|\; |B| \\
|\Pi x\!:\!A.\, B| &=& \pi_{\tau(A)}\; |A|\; (\lambda x\!:\!\tau(A).\, |B|) \\
\\
|\text{TYPE}| &=& \pi_{\text{TYPE}} \\
|\Pi x\!:\!A.\, K| &=& \pi_{\tau(A)}\; |A|\; (\lambda x\!:\!\tau(A).\, |K|)
\end{array}$$

with y not appearing in either A or M. For example, the translation of the functional identity

$$M \;=\; \lambda x\!:\!(\Pi z\!:\!a.\, a).\, x$$

is

$$|M| \;=\; (\lambda y\!:\!\omega.\, \lambda x\!:\!\omega \rightarrow \omega.\, x)(\pi_\omega\; \pi_a\; (\lambda z\!:\!\omega.\, \pi_a))\,.$$

The encoding of contexts inductively extends $\tau(\_)$, blurring the distinction between intuitionistic and linear assumptions. The encoding $\tau(\Sigma)$ of an *LLF* signature Σ consists of two parts: a variable part defining the type of every declaration $v\!:\!U$ in Σ as $\pi_v\!:\!\tau(U)$, and a fixed part declaring the type of the constants needed to represent the type and kind operators of our language. The treatment of Π yields an infinite family of declarations, one for each simple type in $\lambda^{\times\rightarrow}$. We have the following definitions:

Context:

$$\begin{array}{lcl}
\tau(\cdot) &=& \cdot \\
\tau(\Psi, u\,\hat{:}\,A) &=& \tau(\Psi), u\!:\!\tau(A) \\
\tau(\Psi, x\!:\!A) &=& \tau(\Psi), x\!:\!\tau(A)
\end{array}$$

Signature:

$$\begin{array}{rcll}
\tau(\Sigma) = \pi_c &:& \tau(A), & \textit{for } c\!:\!A \textit{ in } \Sigma \\
\pi_a &:& \tau(K), & \textit{for } a\!:\!K \textit{ in } \Sigma \\
\pi_\sigma &:& \omega \rightarrow (\sigma \rightarrow \omega) \rightarrow \omega, & \\
\pi_{\langle\rangle} &:& \omega, & \\
\pi_\top &:& \omega, & \\
\pi_\& &:& \omega \rightarrow \omega \rightarrow \omega, & \\
\pi_\multimap &:& \omega \rightarrow \omega \rightarrow \omega, & \\
\pi_{\text{TYPE}} &:& \omega &
\end{array}$$

The encoding just presented preserve well-typedness: Whenever a term U is valid in $\lambda^{\Pi\multimap\&\top}$, then the object $|U|$ is valid in $\lambda^{\times\rightarrow}$. This property is formally stated in the following lemma.

Lemma 2.14 (*Adequacy of the translation*) *i. If* $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow\!\Downarrow A$, *then* $\tau(\Psi) \vdash_{\tau(\Sigma)} |M| : \tau(A)$*;*

ii. If $\mathcal{P} :: \overline{\Psi} \vdash_\Sigma A \Uparrow\!\Downarrow K$, *then* $\tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} |A| : \tau(K)$*;*

iii. If $\mathcal{P} :: \overline{\Psi} \vdash_\Sigma K \Uparrow Kind$, *then* $\tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} |K| : \omega$*.*

Proof.

We proceed by induction on the structure of $\mathcal{P}$. ☑

The proposed encoding has the further property of preserving reductions. Therefore, whenever a term U reduces to U' in $\lambda^{\Pi\multimap\&\top}$, then, $|U|$ reduces to $|U'|$ in $\lambda^{\times\rightarrow}$ in at least as many steps. The extra β-redex in the representation of λ-abstraction causes individual reductions in our language to be mapped to multi-step reductions in the target language, in general.

Lemma 2.15 (*Preservation of reduction sequences*)

If $\mathcal{R} :: U \longrightarrow^+ V$, *then* $|U| \xrightarrow{s}{}^+_1 |V|$.

Proof.

By induction on the structure of $\mathcal{R}$. ☑

We now have all the ingredients to prove the strong normalization theorem for $\lambda^{\Pi\multimap\&\top}$. A term U is *normalizing* if there exist a term U' in normal form such that $U \longrightarrow^* U'$. U is *strongly normalizing* if every reduction sequence yields a normal term. For example, the term

$$(\lambda y\!:\!a.\, c)\ ((\lambda x\!:\!a.\, x\ x)\ (\lambda x\!:\!a.\, x\ x))$$

is normalizing (its normal form is c), but not strongly normalizing. In order to normalize this term, we must guess the correct reduction sequence. This form of non-determinism does not arise with strongly normalizing terms since the choice of the β-redex to reduce at every step is arbitrary.

The strong normalization theorem states that every derivable term is strongly normalizing. This property holds in $\lambda^{\times\rightarrow}$, as proved for example in [Gan80, Tro86], and we use this fact to prove that it is valid also for $\lambda^{\Pi\multimap\&\top}$.

Theorem 2.16 (*Strong normalization*)

If $\Psi \vdash_\Sigma U \Uparrow\!\Downarrow V$, *then* U *is strongly normalizing.*

Proof.

By the adequacy of the translation (Lemma 2.14), we have that $\tau(\Psi) \vdash_{\tau(\Sigma)} |U| : \tau(V)$ is derivable in $\lambda^{\times\rightarrow}$.

Assume we have a (possibly infinite) reduction sequence in $\lambda^{\Pi\multimap\&\top}$ starting from U

$$U = U_0 \longrightarrow^+ U_1 \longrightarrow^+ \ldots$$

By reduction preservation (Lemma 2.15) there is a corresponding reduction sequence

$$|U_0| \xrightarrow{s}{}^+_1 |U_1| \xrightarrow{s}{}^+_1 \ldots$$

in $\lambda^{\times\to}$. Since the latter must be finite, the former will also be finite. ☑

The validity of strong normalization permits the derivation of a number of further properties for our language. A first result is that the normal form of a derivable term is unique, stated below.

Corollary 2.17 (*Uniqueness of normal forms*)

If $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$, $U \longrightarrow^* U'$ *and* $U \longrightarrow^* U''$ *with both* U' *and* U'' *in normal form, then* $U' = U''$.

Proof.

By confluence, there exist a term V such that $U' \longrightarrow^* V$ and $U' \longrightarrow^* V$. However, since U' and U'' are normal, they do not contain β-redices, and therefore $U' = V = U''$. ☑

This property allows us to define a function NF(_) in order to denote *the* normal form of a valid term U. NF(U) is computed from U by applying β-reductions in this term until a normal form is eventually reached. Strong normalization guarantees that this normal form arises after a finite number of steps, and the lemma above ensures that the resulting term is unique.

A further consequence of the strong normalization theorem is that the equational theory of *LLF* is decidable, i.e. it can be effectively decided whether there exists a derivation for the judgment $U \equiv U'$, for U and U' valid terms. The idea is to check that NF(U) and NF(U') are identical.

Corollary 2.18 (*Decidability of the equational theory*)

If $\Psi \vdash_\Sigma U' \Uparrow\!\!\Downarrow V$ *and* $\Psi \vdash_\Sigma U'' \Uparrow\!\!\Downarrow V$, *then it can be recursively decided whether* $U' \equiv U''$ *is derivable.*

Proof.

By the Church-Rosser property, U' and U'' have a common reduct U. By subject reduction, U is valid. Therefore, by uniqueness (Corollary 2.17), U', U and U'' share the same normal form NF(U).

By the strong normalization theorem, every sequence of reduction on U' and U'' eventually produces NF(U) after a finite number of steps. Therefore, a possible decision procedure for definitional equality is as follows: compute the normal forms of U' and U'' and then check whether they are equal. If they are, then U' is definitionally equal to U''. Otherwise, they are not equivalent. ☑

Yet another consequence of the strong normalization theorem is that every derivable $\lambda^{\Pi\multimap\&\top}$ judgment can be constrained to mention only objects in normal form. Although not strictly needed, it is a common practice to write *LLF* signatures in normal form.

Corollary 2.19 (*Normal forms*)

i. If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$, *then* $\mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(U) \Uparrow\!\!\Downarrow \mathrm{NF}(V)$;

ii. If $\mathcal{P} :: \vdash_\Sigma \Psi \Uparrow Ctx$, *then* $\vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(\Psi) \Uparrow Ctx$

iii. If $\mathcal{P} :: \vdash \Sigma \Uparrow Sig$, *then* $\vdash \mathrm{NF}(\Sigma) \Uparrow Sig$.

Proof.

We first reduce U to normal form in (i) by means of the previous corollary, and then proceed by induction on the structure of $\mathcal{P}$. ☑

In an implementation of the language, converting terms to normal form as soon as β-redices appear as the result of substitutions is not always necessary. It is usually more efficient to work with *weak head-normal forms*.

A final consequence of the strong normalization theorem is that rule **opa_c** can be dropped as soon as we are only interested in valid normal terms. As we briefly motivated earlier, only the presence of this rule permits the formation of β-redices in valid terms (i.e. in the terms immediately to the left of the arrow in a pre-canonical or pre-atomic judgment). Eliminating this rule is beneficial since it prevents non-normal terms from being validated without losing any valid normal term.

2.6 Algorithmic System

A proof of the decidability of type checking for *LLF* is difficult to achieve directly in the pre-canonical system in Figures 2–3. Indeed, it is not possible to predict the size of a derivation for a judgment since the rules that we called "non-essential" in Section 2.1 (the equivalence rules and **opa_c**) can be chained arbitrarily. The strong normalization theorem and the admissibility of **opa_c** limit the need for these rules drastically. In this section, we will embed the first of these results in a deductive system for $\lambda^{\Pi\multimap\&\top}$ having the characteristic that every derivable judgment has a derivation whose maximum size is a function of the terms constituting this judgment; we will come back to this aspect in Section 2.7. Following the terminology of [HHP93], we call this system, somewhat improperly, *algorithmic*. The properties of this system will also permit us to eventually prove the validity of strengthening for our language.

The algorithmic system for $\lambda^{\Pi\multimap\&\top}$ consists of judgments similar to the pre-canonical presentation; indeed, we use the same expressions, only annotating the turnstile symbol with the letter a, and adopting arrows of a different shape. The notion of definitional equality is the same, but we do not access the judgments defining the equational theory directly. We rely instead on the normalization function $\mathrm{NF}(\_)$, that is known to exist from the previous section. We remind the reader that this function is defined only for valid terms, and it will be easy to check that whenever it is used in the algorithmic system, its argument is valid.

The inference rules defining the behavior of the algorithmic system are displayed in Figure 6. This deductive system shares with the pre-canonical system of Section 2.2 the property that every derivable term is in η-long form. This aspect will be a consequence of the soundness theorem below. However, the algorithmic system has the further characteristic that all terms mentioned in any well-formed derivation are themselves valid. As we said, ill-formed terms could appear

Signatures

$$\frac{}{\vdash^a \cdot \Uparrow \mathit{Sig}}\,\textbf{sa\_dot} \qquad \frac{\vdash^a \Sigma \Uparrow \mathit{Sig} \quad \cdot \vdash^a_\Sigma A \Uparrow \textsc{type}}{\vdash^a \Sigma, c{:}A \Uparrow \mathit{Sig}}\,\textbf{sa\_obj} \qquad \frac{\vdash^a \Sigma \Uparrow \mathit{Sig} \quad \cdot \vdash^a_\Sigma K \Uparrow \mathit{Kind}}{\vdash^a \Sigma, a{:}K \Uparrow \mathit{Sig}}\,\textbf{sa\_fam}$$

Contexts

$$\frac{\vdash^a \Sigma \Uparrow \mathit{Sig}}{\vdash^a_\Sigma \cdot \Uparrow \mathit{Ctx}}\,\textbf{ca\_dot} \qquad \frac{\vdash^a_\Sigma \Psi \Uparrow \mathit{Ctx} \quad \overline{\Psi} \vdash^a_\Sigma A \Uparrow \textsc{type}}{\vdash^a_\Sigma \Psi, x{:}A \Uparrow \mathit{Ctx}}\,\textbf{ca\_int} \qquad \frac{\vdash^a_\Sigma \Psi \Uparrow \mathit{Ctx} \quad \overline{\Psi} \vdash^a_\Sigma A \Uparrow \textsc{type}}{\vdash^a_\Sigma \Psi, u\,\hat{:}\,A \Uparrow \mathit{Ctx}}\,\textbf{ca\_lin}$$

Kinds

$$\frac{\vdash^a_\Sigma \overline{\Psi} \Uparrow \mathit{Ctx}}{\overline{\Psi} \vdash^a_\Sigma \textsc{type} \Uparrow \mathit{Kind}}\,\textbf{kca\_type} \qquad \frac{\overline{\Psi}, x{:}A \vdash^a_\Sigma K \Uparrow \mathit{Kind}}{\overline{\Psi} \vdash^a_\Sigma \Pi x{:}A.\, K \Uparrow \mathit{Kind}}\,\textbf{kca\_dep}$$

Types/type families

$$\frac{\overline{\Psi} \vdash^a_\Sigma P \downarrow \textsc{type}}{\overline{\Psi} \vdash^a_\Sigma P \Uparrow \textsc{type}}\,\textbf{fca\_a} \qquad \frac{\vdash^a_\Sigma \overline{\Psi} \Uparrow \mathit{Ctx}}{\overline{\Psi} \vdash^a_\Sigma \top \Uparrow \textsc{type}}\,\textbf{fca\_top} \qquad \frac{\overline{\Psi} \vdash^a_\Sigma A \Uparrow \textsc{type} \quad \overline{\Psi} \vdash^a_\Sigma B \Uparrow \textsc{type}}{\overline{\Psi} \vdash^a_\Sigma A \,\&\, B \Uparrow \textsc{type}}\,\textbf{fca\_with}$$

$$\frac{\overline{\Psi} \vdash^a_\Sigma A \Uparrow \textsc{type} \quad \overline{\Psi} \vdash^a_\Sigma B \Uparrow \textsc{type}}{\overline{\Psi} \vdash^a_\Sigma A \multimap B \Uparrow \textsc{type}}\,\textbf{fca\_limp} \qquad \frac{\overline{\Psi}, x{:}A \vdash^a_\Sigma B \Uparrow \textsc{type}}{\overline{\Psi} \vdash^a_\Sigma \Pi x{:}A.\, B \Uparrow \textsc{type}}\,\textbf{fca\_dep}$$

$$\frac{\vdash^a_{\Sigma,a:K,\Sigma'} \overline{\Psi} \Uparrow \mathit{Ctx}}{\overline{\Psi} \vdash^a_{\Sigma,a:K,\Sigma'} a \downarrow \mathrm{NF}(K)}\,\textbf{faa\_con} \qquad \frac{\overline{\Psi} \vdash^a_\Sigma P \downarrow \Pi x{:}A.\, K \quad \overline{\Psi} \vdash^a_\Sigma N \Uparrow A}{\overline{\Psi} \vdash^a_\Sigma P\, N \downarrow \mathrm{NF}([N/x]K)}\,\textbf{faa\_iapp}$$

Objects

$$\frac{\Psi \vdash^a_\Sigma M \downarrow P}{\Psi \vdash^a_\Sigma M \Uparrow P}\,\textbf{oca\_a} \qquad \frac{\vdash^a_\Sigma \Psi \Uparrow \mathit{Ctx}}{\Psi \vdash^a_\Sigma \langle\rangle \Uparrow \top}\,\textbf{oca\_unit} \qquad \frac{\Psi \vdash^a_\Sigma M \Uparrow A \quad \Psi \vdash^a_\Sigma N \Uparrow B}{\Psi \vdash^a_\Sigma \langle M, N\rangle \Uparrow A \,\&\, B}\,\textbf{oca\_pair}$$

$$\frac{\Psi, u\,\hat{:}\,A \vdash^a_\Sigma M \Uparrow B}{\Psi \vdash^a_\Sigma \hat{\lambda} u{:}A.\, M \Uparrow \mathrm{NF}(A) \multimap B}\,\textbf{oca\_llam} \qquad \frac{\Psi, x{:}A \vdash^a_\Sigma M \Uparrow B}{\Psi \vdash^a_\Sigma \lambda x{:}A.\, M \Uparrow \Pi x{:}\mathrm{NF}(A).\, B}\,\textbf{oca\_ilam}$$

$$\frac{\Psi \vdash^a_\Sigma M \Uparrow A}{\Psi \vdash^a_\Sigma M \downarrow A}\,\textbf{oaa\_c} \qquad \frac{\vdash^a_{\Sigma,c:A,\Sigma'} \overline{\Psi} \Uparrow \mathit{Ctx}}{\overline{\Psi} \vdash^a_{\Sigma,c:A,\Sigma'} c \downarrow \mathrm{NF}(A)}\,\textbf{oaa\_con}$$

$$\frac{\vdash^a_\Sigma \overline{\Psi}, u\,\hat{:}\,A, \overline{\Psi'} \Uparrow \mathit{Ctx}}{\overline{\Psi}, u\,\hat{:}\,A, \overline{\Psi'} \vdash^a_\Sigma u \downarrow \mathrm{NF}(A)}\,\textbf{oaa\_lvar} \qquad \frac{\vdash^a_\Sigma \overline{\Psi}, x{:}A, \overline{\Psi'} \Uparrow \mathit{Ctx}}{\overline{\Psi}, x{:}A, \overline{\Psi'} \vdash^a_\Sigma x \downarrow \mathrm{NF}(A)}\,\textbf{oaa\_ivar}$$

(No rule for $\top$)

$$\frac{\Psi \vdash^a_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash^a_\Sigma \textsc{fst}\, M \downarrow A}\,\textbf{oaa\_fst} \qquad \frac{\Psi \vdash^a_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash^a_\Sigma \textsc{snd}\, M \downarrow B}\,\textbf{oaa\_snd}$$

$$\frac{\Psi' \vdash^a_\Sigma M \downarrow A \multimap B \quad \Psi'' \vdash^a_\Sigma N \Uparrow A \quad \Psi = \Psi' \bowtie \Psi''}{\Psi \vdash^a_\Sigma M \,\hat{}\, N \downarrow B}\,\textbf{oaa\_lapp} \qquad \frac{\Psi \vdash^a_\Sigma M \downarrow \Pi x{:}A.\, B \quad \overline{\Psi} \vdash^a_\Sigma N \Uparrow A}{\Psi \vdash^a_\Sigma M\, N \downarrow \mathrm{NF}([N/x]B)}\,\textbf{oaa\_iapp}$$

Figure 6: Algorithmic Deduction System for *LLF*

in equivalence subderivations by using β-reductions in the reverse direction. In the algorithmic system, the equivalence relation $\equiv$ has been eliminated in favor of pointwise applications of the normalization function in the rules introducing the dependent type or kind constructor Π (**faa_iapp** and **oaa_iapp**), which involve the application of a substitution. The algorithmic system has also the property that the terms appearing on the right of the arrow are always in canonical form. We achieve this effect by normalizing types and kinds when fetching them from the signature or the context (rules **faa_con**, **oaa_con**, **oaa_lvar**, **oaa_ivar**, **oca_llam** and **oca_ilam**).

The correspondence between the algorithmic and the pre-canonical systems is formalized by means of the following soundness and completeness theorems. First, every valid term in the algorithmic system is also valid in the pre-canonical formulation.

Theorem 2.20 (*Soundness of the algorithmic system*)

i. If $\mathcal{A} :: \Psi \vdash^a_\Sigma U \Uparrow\!\!\Downarrow V$*, then* $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ *and* V *is in normal form;*

ii. If $\mathcal{A} :: \vdash^a_\Sigma \Psi \Uparrow Ctx$*, then* $\vdash_\Sigma \Psi \Uparrow Ctx$*;*

iii. If $\mathcal{A} :: \vdash^a \Sigma \Uparrow Sig$*, then* $\vdash \Sigma \Uparrow Sig$*.*

Proof.

We proceed by induction on the structure of $\mathcal{A}$. ☑

The completeness theorem states that every judgment having a derivation in the pre-canonical system is also derivable in the system presented in this section. Therefore, no valid term is lost by moving to the algorithmic system. Notice however that the type or kind appearing on the right-hand side of the main judgments of the pre-canonical system must be normalized in the algorithmic system.

Theorem 2.21 (*Completeness of the algorithmic system*)

i. If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$*, then* $\Psi \vdash^a_\Sigma U \Uparrow\!\!\Downarrow \mathrm{NF}(V)$*;*

ii. If $\mathcal{P} :: \vdash_\Sigma \Psi \Uparrow Ctx$*, then* $\vdash^a_\Sigma \Psi \Uparrow Ctx$*;*

iii. If $\mathcal{P} :: \vdash \Sigma \Uparrow Sig$*, then* $\vdash^a \Sigma \Uparrow Sig$*.*

Proof.

By induction on the structure of $\mathcal{P}$. ☑

The proofs of these results are constructive and therefore specify an effective transformation procedure.

The strict access to definitional equality, and in particular the impossibility of using it for β-expansions permits a direct proof of the strengthening lemma in the algorithmic system. The soundness and completeness theorems above entitle us to bring this result back to the pre-canonical system.

Lemma 2.22 (*Strengthening*)

i. *If* $\Psi, x \mathbin{\hat{::}} A, \Psi' \vdash_\Sigma U \Uparrow\!\Downarrow V$ *and* $x \notin \mathrm{FV}(\Psi') \cup \mathrm{FV}(U) \cup \mathrm{FV}(V)$, *then* $\Psi, \Psi' \vdash_\Sigma U \Uparrow\!\Downarrow V$;

ii. *If* $\vdash_\Sigma \Psi, x \mathbin{\hat{::}} A, \Psi' \Uparrow Ctx$ *and* $x \notin \mathrm{FV}(\Psi')$, *then* $\vdash_\Sigma \Psi, \Psi' \Uparrow Ctx$.

Proof.

We first prove the analogous lemma for the algorithmic system by induction on a derivation of the given derivations, and then use the above soundness and completeness results to transfer it to the pre-canonical setting. ☑

2.7 Decidability

The absence of explicit equivalences in the algorithmic system limits the choices of the inference rules that can be used at every step of a derivation considerably. Every well-formed judgment matches the conclusion of at most one rule, with the only exception of judgments of the form $\Psi \vdash_\Sigma M \downarrow A$, for which **oaa_c** is always available. Moreover, the terms appearing in the central part of a validity judgment become smaller when going from the conclusion to the premisses in all rules in Figure 6 except **fca_a**, **oca_a** and **oaa_c**, for which they remain the same. Notice that possible cycles generated by the last two can be easily detected and removed.

In this section, we take advantage of these characteristics in order to prove that the type checking and type synthesis problems are decidable for *LLF*. Both issues need to be faced simultaneously in our language. The proofs of these results yields a decision procedure able to check whether a fully specified $\lambda^{\Pi\multimap\&\top}$ judgment is derivable or not. This procedure is also able to compute a type or a kind for a judgment whose rightmost term is left unspecified, or to declare that no such term exists.

In order to achieve this goal, a preliminary step consists of defining a complexity measure for an algorithmic judgment. This number yields an upper bound for the size of at least one of its derivations. For this purpose, we rely on a family of *size functions* that we denote uniformly as $\|\_\|$. We first define the size of terms as follows.

Objects:

$$\begin{array}{rcl}
\|x\| & = & 1 \\
\|c\| & = & 1 \\
\|\langle\rangle\| & = & 1 \\
\|\langle M, N\rangle\| & = & 1 + \|M\| + \|N\| \\
\|\mathrm{FST}\, M\| & = & 1 + \|M\| \\
\|\mathrm{SND}\, M\| & = & 1 + \|M\| \\
\|\hat{\lambda} u : A.\, M\| & = & 2 + \|A\| + \|M\| \\
\|M \,\hat{}\, N\| & = & 1 + \|M\| + \|N\| \\
\|\lambda x : A.\, M\| & = & 2 + \|A\| + \|M\| \\
\|M\, N\| & = & 1 + \|M\| + \|N\|
\end{array}$$

Types and kinds:

$$\begin{array}{rcl}
\|a\| & = & 1 \\
\|P\, N\| & = & 1 + \|P\| + \|N\| \\
\|\top\| & = & 1 \\
\|A \,\&\, B\| & = & 1 + \|A\| + \|B\| \\
\|A \multimap B\| & = & 1 + \|A\| + \|B\| \\
\|\Pi x : A.\, B\| & = & 2 + \|A\| + \|B\| \\
\\
\|\mathrm{TYPE}\| & = & 1 \\
\|\Pi x : A.\, K\| & = & 2 + \|A\| + \|K\|
\end{array}$$

At this point, we can extend this definition to contexts and signatures:

Contexts:

$$\begin{array}{rcl}
\|\cdot\| & = & 1 \\
\|\Psi, u \hat{:} A\| & = & 1 + \|\Psi\| + \|A\| \\
\|\Psi, x : A\| & = & 1 + \|\Psi\| + \|A\|
\end{array}$$

Signatures:

$$\begin{array}{rcl}
\|\cdot\| & = & 1 \\
\|\Sigma, c : A\| & = & 1 + \|\Sigma\| + \|A\| \\
\|\Sigma, a : K\| & = & 1 + \|\Sigma\| + \|K\|
\end{array}$$

Finally, we can combine the size of terms, contexts and signature in order to define the size of all the judgments participating in the algorithmic system. We have the following definition:

Judgments:

$$\begin{array}{rcl}
\| \vdash^{a} \Sigma \Uparrow \mathit{Sig}\| & = & \|\Sigma\| \\
\| \vdash^{a}_{\Sigma} \Psi \Uparrow \mathit{Ctx}\| & = & \|\Sigma\| + \|\Psi\| \\
\|\overline{\Psi} \vdash^{a}_{\Sigma} K \Uparrow \mathit{Kind}\| & = & \|\Sigma\| + \|\Psi\| + \|K\| \\
\|\overline{\Psi} \vdash^{a}_{\Sigma} A \Uparrow \mathrm{TYPE}\| & = & \|\Sigma\| + \|\Psi\| + \|A\| \\
\|\overline{\Psi} \vdash^{a}_{\Sigma} A \downarrow K\| & = & \|\Sigma\| + \|\Psi\| + \|A\| \\
\|\Psi \vdash^{a}_{\Sigma} M \Uparrow A\| & = & \|\Sigma\| + \|\Psi\| + \|M\| \\
\|\Psi \vdash^{a}_{\Sigma} M \downarrow A\| & = & \|\Sigma\| + \|\Psi\| + \|M\| \\
\|\Psi = \Psi' \bowtie \Psi''\| & = & \|\Psi\|
\end{array}$$

Notice that the size of a judgment does not refer to the term appearing to the right of the arrow. This is necessary for our purposes since the size of this term in the premisses of the elimination rules can in general be larger than in the conclusion.

We designed the functions above so that the size of a judgment is an upper bound on the height of at least one of its derivations. This property is expressed by the following lemma. It is proved in two steps: first we eliminate from the given derivation all sequences of rules consisting only of the alternation of **oca_a** and **oaa_c**; second, we show that the size of the premisses of an introduction or elimination rule is always smaller than the size of the conclusion. In devising the constants in the functions above, we used this property as a guideline, and a spreadsheet as a tool to partially automate checking that these weights respect it.

Lemma 2.23 (*Upper bound on the size of a derivation*)

i. *Let* $h = \|\Psi \vdash^a_\Sigma U \Uparrow\!\!\Downarrow V\|$. *If* $\mathcal{A} :: \Psi \vdash^a_\Sigma U \Uparrow\!\!\Downarrow V$, *then* $\mathcal{A}' :: \Psi \vdash^a_\Sigma U \Uparrow\!\!\Downarrow V$,

ii. *Let* $h = \| \vdash^a_\Sigma \Psi \Uparrow Ctx\|$. *If* $\mathcal{A} :: \vdash^a_\Sigma \Psi \Uparrow Ctx$, *then* $\mathcal{A}' :: \vdash^a_\Sigma \Psi \Uparrow Ctx$,

iii. *Let* $h = \| \vdash^a \Sigma \Uparrow Sig\|$. *If* $\mathcal{A} :: \vdash^a \Sigma \Uparrow Sig$, *then* $\mathcal{A}' :: \vdash^a \Sigma \Uparrow Sig$,

where $\mathcal{A}'$ *has height less than* $2h$ *and contains at most* 3^{2h} *nodes.* □

We have now all the necessary ingredients to prove that the type checking problem is decidable in *LLF*. Given a judgment whose validity we want to decide, a first naive idea is to match it against the conclusion of the inference rules defining the algorithmic system. If none of these rules apply, then the judgment is not derivable, otherwise, we check recursively that the instantiated premisses of the viable rules are derivable. The lemma above provides an upper bound on the number of rules that need to be considered.

Unfortunately, this simple bottom-up strategy is not correct. The elimination rules in Figure 6 mention in their premisses types on the right-hand side of the arrow that do not appear in their conclusion. Therefore, checking recursively that the premisses of these rules are derivable requires *guessing* these types. Since we can build arbitrarily big terms in $\lambda^{\Pi\multimap\&\top}$, a blind generate-and-test technique could at most be a semi-decision procedure since, if no derivation exists, it would keep on generating bigger and bigger types and attempting to prove the judgments in the premisses with them.

The solution to this problem is to combine type checking and type synthesis. Whenever we need to guess a type, we run the type synthesis procedure in order to infer it, and then proceed as above. Proving the decidability of type synthesis requires type checking in order to validate contexts. The decidability theorem can be stated as follows:

Theorem 2.24 (*Decidability of type checking*)

i. (*Type checking*)
It can be recursively decided whether $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$, $\vdash_\Sigma \Psi \Uparrow Ctx$ *and* $\vdash \Sigma \Uparrow Sig$ *are derivable.*

ii. (*Type synthesis*)
Given a signature Σ, *a context* Ψ *and a term* U, *there is a recursive procedure that computes a term* V *such that the judgment* $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$ *is derivable, or determines that no such* V *exists.*

Proof.

We prove the analogous property for the algorithmic system and rely the constructive aspects of the soundness and completeness theorems above to transfer it to the pre-canonical setting. The idea, in order to prove the algorithmic formulation of this result, is to apply inference rules that match U (and V for type checking) until either a derivation is produced, or no rule is applicable, or the upper bound on the size of the derivation at hand has been reached. ☑

The mutually recursive parts of this theorem yield effective procedures for type checking and type synthesis. Once this result has been proved, type checking can be conveniently reduced to type synthesis: In order to check whether $\Psi \vdash_\Sigma M \Uparrow\!\!\Downarrow A$ is derivable, it suffices to check that A is valid, infer a type A' such that $\Psi \vdash_\Sigma M \Uparrow\!\!\Downarrow A'$ is derivable, and check whether $A \equiv A'$ holds, which, by Corollary 2.18, is decidable. An application of the equivalence rules yields $\Psi \vdash_\Sigma M \Uparrow\!\!\Downarrow A$. The subproblem of checking whether A is valid is also reduced to finding a kind for it, without indirections this time.

The decidability of type checking is a fundamental property for using a formalism as a meta-representation language. Similarly to *LF*, *LLF* encodes judgments from an object language as types and their derivations as object level terms. The decidability of type checking for $\lambda^{\Pi\multimap\&\top}$ permits determining effectively whether a given object is the representation of a valid derivation from a (potentially linear) object formalism.

2.8 Canonical Forms

Similarly to the pre-canonical case, the algorithmic system presented in Section 2.6 prevents terms that are not in η-long form from being validated. The only exception concerns the process of constructing a term U by means of the judgments $\Psi \vdash^a_\Sigma U \downarrow V$, where U will not necessarily be η-expanded, in general. In this section, we will present a proof system that forces all entities appearing in a derivable judgment to be also in normal form. Therefore, all valid terms will be canonical, i.e. in η-long form and without β-redices. We will achieve this property by removing rule **oaa_c**, that, as we saw, permits the formation of β-redices in derivable terms. In this system, we will also customize the rules of the algorithmic system to make them more suited for automation. We will rely on this deductive system in the following discussion.

When deriving a judgment of the form $\Psi \vdash_\Sigma U \Uparrow\!\!\Downarrow V$, the *canonical system* in Figure 7 assumes that the validity of both the signature Σ and the context Ψ has been checked "beforehand". Consequently, the rules dealing with constants, variables, and other atomic terms do not need their premiss anymore; they are the leaves of the derivations. When applying the rules for λ-abstraction or when checking that a dependent type is valid (rules **fc_dep** and **kc_dep**), we need to check that the type of the assumption added to the context is valid. This ensures the validity of the context in the premiss of the rules. In rule **oa_lapp**, which relies on context splitting, the contexts occurring in the premisses differ from the context in the conclusion only by the removal of some linear assumptions. These contexts are however valid since in our language, linear variables cannot occur free in types, and therefore no assumption can depend on

Signatures

$$\frac{}{\vdash \cdot \Uparrow Sig}\ \textbf{sc\_dot} \qquad \frac{\vdash \Sigma \Uparrow Sig \quad \cdot \vdash_\Sigma A \Uparrow \mathrm{TYPE}}{\vdash \Sigma, c{:}A \Uparrow Sig}\ \textbf{sc\_obj} \qquad \frac{\vdash \Sigma \Uparrow Sig \quad \cdot \vdash_\Sigma K \Uparrow Kind}{\vdash \Sigma, a{:}K \Uparrow Sig}\ \textbf{sc\_fam}$$

Contexts

$$\frac{}{\vdash_\Sigma \cdot \Uparrow Ctx}\ \textbf{cc\_dot} \qquad \frac{\vdash_\Sigma \Psi \Uparrow Ctx \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}}{\vdash_\Sigma \Psi, x{:}A \Uparrow Ctx}\ \textbf{cc\_int} \qquad \frac{\vdash_\Sigma \Psi \Uparrow Ctx \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}}{\vdash_\Sigma \Psi, u \hat{:} A \Uparrow Ctx}\ \textbf{cc\_lin}$$

Kinds

$$\frac{}{\overline{\Psi} \vdash_\Sigma \mathrm{TYPE} \Uparrow Kind}\ \textbf{kc\_type} \qquad \frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE} \quad \overline{\Psi}, x{:}A \vdash_\Sigma K \Uparrow Kind}{\overline{\Psi} \vdash_\Sigma \Pi x{:}A.\, K \Uparrow Kind}\ \textbf{kc\_dep}$$

Types/type families

$$\frac{\overline{\Psi} \vdash_\Sigma P \downarrow \mathrm{TYPE}}{\overline{\Psi} \vdash_\Sigma P \Uparrow \mathrm{TYPE}}\ \textbf{fc\_a} \qquad \frac{}{\overline{\Psi} \vdash_\Sigma \top \Uparrow \mathrm{TYPE}}\ \textbf{fc\_top} \qquad \frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE} \quad \overline{\Psi} \vdash_\Sigma B \Uparrow \mathrm{TYPE}}{\overline{\Psi} \vdash_\Sigma A \,\&\, B \Uparrow \mathrm{TYPE}}\ \textbf{fc\_with}$$

$$\frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE} \quad \overline{\Psi} \vdash_\Sigma B \Uparrow \mathrm{TYPE}}{\overline{\Psi} \vdash_\Sigma A \multimap B \Uparrow \mathrm{TYPE}}\ \textbf{fc\_limp} \qquad \frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE} \quad \overline{\Psi}, x{:}A \vdash_\Sigma B \Uparrow \mathrm{TYPE}}{\overline{\Psi} \vdash_\Sigma \Pi x{:}A.\, B \Uparrow \mathrm{TYPE}}\ \textbf{fc\_dep}$$

$$\frac{}{\overline{\Psi} \vdash_{\Sigma, a:K, \Sigma'} a \downarrow K}\ \textbf{fa\_con} \qquad \frac{\overline{\Psi} \vdash_\Sigma P \downarrow \Pi x{:}A.\, K \quad \overline{\Psi} \vdash_\Sigma N \Uparrow A}{\overline{\Psi} \vdash_\Sigma P\, N \downarrow \mathrm{NF}([N/x]K)}\ \textbf{fa\_iapp}$$

Objects

$$\frac{\Psi \vdash_\Sigma M \downarrow P}{\Psi \vdash_\Sigma M \Uparrow P}\ \textbf{oc\_a} \qquad \frac{}{\Psi \vdash_\Sigma \langle\rangle \Uparrow \top}\ \textbf{oc\_unit} \qquad \frac{\Psi \vdash_\Sigma M \Uparrow A \quad \Psi \vdash_\Sigma N \Uparrow B}{\Psi \vdash_\Sigma \langle M, N\rangle \Uparrow A \,\&\, B}\ \textbf{oc\_pair}$$

$$\frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE} \quad \Psi, u \hat{:} A \vdash_\Sigma M \Uparrow B}{\Psi \vdash_\Sigma \hat{\lambda} u{:}A.\, M \Uparrow A \multimap B}\ \textbf{oc\_llam} \qquad \frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE} \quad \Psi, x{:}A \vdash_\Sigma M \Uparrow B}{\Psi \vdash_\Sigma \lambda x{:}A.\, M \Uparrow \Pi x{:}A.\, B}\ \textbf{oc\_ilam}$$

$$\frac{}{\overline{\Psi} \vdash_{\Sigma, c:A, \Sigma'} c \downarrow A}\ \textbf{oa\_con} \qquad \frac{}{\overline{\Psi}, u \hat{:} A, \overline{\Psi'} \vdash_\Sigma u \downarrow A}\ \textbf{oa\_lvar} \qquad \frac{}{\overline{\Psi}, x{:}A, \overline{\Psi'} \vdash_\Sigma x \downarrow A}\ \textbf{oa\_ivar}$$

(No rule for $\top$)

$$\frac{\Psi \vdash_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash_\Sigma \mathrm{FST}\, M \downarrow A}\ \textbf{oa\_fst} \qquad \frac{\Psi \vdash_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash_\Sigma \mathrm{SND}\, M \downarrow B}\ \textbf{oa\_snd}$$

$$\frac{\Psi' \vdash_\Sigma M \downarrow A \multimap B \quad \Psi'' \vdash_\Sigma N \Uparrow A \quad \Psi = \Psi' \bowtie \Psi''}{\Psi \vdash_\Sigma M\,\hat{}\,N \downarrow B}\ \textbf{oa\_lapp} \qquad \frac{\Psi \vdash_\Sigma M \downarrow \Pi x{:}A.\, B \quad \overline{\Psi} \vdash_\Sigma N \Uparrow A}{\Psi \vdash_\Sigma M\, N \downarrow \mathrm{NF}([N/x]B)}\ \textbf{oa\_iapp}$$

Figure 7: Canonical Deduction System for *LLF*

them (see Lemma 2.4).

Another novelty of the canonical system is the absence of a rule corresponding to **oaa_c**. We already know that this rule is admissible, but we already saw that this rule and only this rule is responsible for the possibility of deriving judgments containing β-redices. Without it, the canonical system can only derive η-long terms that are β-normal, i.e., canonical, as the name of this system implies. As we motivated earlier, this limitation is welcome in many respects.

An important consequence of the elimination of **oaa_c** is that the resulting system is deterministic: every well-formed judgment matches the conclusion of at most one inference rule.

The equivalence between the algorithmic and the canonical system in Figure 7 is expressed by means of the following soundness and completeness theorems.

Theorem 2.25 (*Soundness of the canonical system*)

i. If $\vdash \Sigma \Uparrow Sig$, $\vdash_\Sigma \Psi \Uparrow Ctx$ *and* $\mathcal{C} :: \Psi \vdash_\Sigma U \Updownarrow V$, *then* $\Psi \vdash^a_\Sigma U \Updownarrow V$;

ii. If $\vdash \Sigma \Uparrow Sig$ *and* $\mathcal{C} :: \vdash_\Sigma \Psi \Uparrow Ctx$, *then* $\vdash^a_\Sigma \Psi \Uparrow Ctx$;

iii. If $\mathcal{C} :: \vdash \Sigma \Uparrow Sig$, *then* $\vdash^a \Sigma \Uparrow Sig$.

Moreover, Σ, Ψ, U *and* V *are in normal form.*

Proof.

By induction on the structure of $\mathcal{C}$. ☑

The converse is not true in general. It holds, however, whenever all the entities appearing in a derivable algorithmic judgment are in normal form. We know by the normal form corollary 2.19 that every derivable judgment in that system is equivalent to a judgment that mentions only normal terms. Therefore, the correspondence between the algorithmic and the canonical system is not perfect, since it preserves derivability but not derivations, in general. As we said earlier, when performing search and when encoding a deductive system, we are only interested in terms that are η-long and β-normal, i.e., canonical.

Theorem 2.26 (*Completeness of the canonical system*)

i. If $\Psi \vdash^a_\Sigma M \Updownarrow A$, *then*
- *if* A *is a base type, then* $\mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(M) \Updownarrow A$,
- *if* $\mathrm{NF}(M) = \langle\rangle, \langle M', M''\rangle, \hat{\lambda} u : A'. M'$ *or* $\lambda x : A'. M'$, *then* $\mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(M) \Uparrow A$
- *if* $\mathrm{NF}(M) = x, c, \mathrm{FST}\, M', \mathrm{SND}\, M', M' \hat{}\, M''$ *or* $M'\, M''$, *then* $\mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(M) \downarrow A$

ii. If $\overline{\Psi} \vdash^a_\Sigma A \Updownarrow K$, *then* $\mathrm{NF}(\overline{\Psi}) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(A) \Updownarrow K$;

iii. If $\overline{\Psi} \vdash^a_\Sigma K \Uparrow Kind$, *then* $\mathrm{NF}(\overline{\Psi}) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(K) \Uparrow Kind$;

iv. If $\vdash^a_\Sigma \Psi \Uparrow Ctx$, *then* $\vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(\Psi) \Uparrow Ctx$;

v. If $\vdash^a \Sigma \Uparrow Sig$, *then* $\vdash \mathrm{NF}(\Sigma) \Uparrow Sig$.

Proof.

We proceed by induction on the given judgments after applying the Normal Forms Corollary 2.19 and taking into consideration the admissibility of rule **oaa_c**, which derives from the analogous property of rule `opa_c` in the pre-canonical system. ☑

We will adopt the system presented in this section to compare the features of *LLF* to *LF*. We will rely on a canonical system adapted from [Pfe92] for our comparison. Details can be found in [Cer96]. We will distinguish the λ^{Π} equivalents of the judgments presented earlier by annotating them with the superscript $^{\mathrm{LF}}$.

As already pointed out in Section 2.1, *LLF* enriches the language of *LF* with a number of linear type constructors and the corresponding object level operators. However, all syntactic entities of λ^{Π} are available in our language. This embedding is maintained at the semantic level. The canonical system for $\lambda^{\Pi\multimap\&\top}$ in Figure 7 differs from the λ^{Π} system with the same name only by the addition of rules that deal with the linear entities of our language. Therefore, every judgment derivable in λ^{Π} has an isomorphic derivation in $\lambda^{\Pi\multimap\&\top}$, i.e., our language extends *LF* also from the semantic point of view.

Theorem 2.27 (*Extension over LF*)

i. If $\mathcal{LF} :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma} U \Uparrow\!\!\Downarrow V$, *then* $\Gamma \vdash_{\Sigma} U \Uparrow\!\!\Downarrow V$;

ii. If $\mathcal{LF} :: \vdash^{\mathrm{LF}}_{\Sigma} \Gamma \Uparrow \mathit{Ctx}$, *then* $\vdash_{\Sigma} \Gamma \Uparrow \mathit{Ctx}$;

iii. If $\mathcal{LF} :: \vdash^{\mathrm{LF}} \Sigma \Uparrow \mathit{Sig}$, *then* $\vdash \Sigma \Uparrow \mathit{Sig}$.

Proof.

We proceed by induction on the structure of $\mathcal{LF}$. All cases are immediate as soon as we notice that, for a λ^{Π} context Γ, $\overline{\Gamma} = \Gamma$. ☑

LLF has also the converse property of being *conservative* over *LF*, i.e., every derivable $\lambda^{\Pi\multimap\&\top}$ judgment that mentions only entities in the λ^{Π} fragment of the syntax has a corresponding derivation in λ^{Π}. This also entails that every judgment that is not derivable in *LF* remains such in *LLF*.

Theorem 2.28 (*Conservativity over LF*)

Let Σ, Γ, U *and* V *be an LF signature, an LF context and two LF terms, respectively, then*

i. If $\mathcal{C} :: \Gamma \vdash_{\Sigma} U \Uparrow\!\!\Downarrow V$, *then* $\Gamma \vdash^{\mathrm{LF}}_{\Sigma} U \Uparrow\!\!\Downarrow V$;

ii. If $\mathcal{C} :: \vdash_{\Sigma} \Gamma \Uparrow \mathit{Ctx}$, *then* $\vdash^{\mathrm{LF}}_{\Sigma} \Gamma \Uparrow \mathit{Ctx}$;

iii. If $\mathcal{C} :: \vdash \Sigma \Uparrow \mathit{Sig}$, *then* $\vdash^{\mathrm{LF}} \Sigma \Uparrow \mathit{Sig}$.

Proof.

We proceed by induction on the structure of $\mathcal{C}$. We need to remember that for an *LF* context Γ, $\overline{\Gamma} = \Gamma$. ☑

These properties have important consequences. Not only every judgment derivable in *LF* is derivable also in our language, but, more importantly, all the representation techniques, adequacy theorems, and examples developed for *LF* remain valid for *LLF*. This inheritance endows our formalism with a rich library of concrete applications.

2.9 A Concrete Syntax for *LLF*

In this section, we extend the concrete syntax of *Elf* [Pfe94a] to express the linear operators of *LLF*. In doing so, we want to fulfill two constraints: first of all, existing *Elf* programs should not undergo any syntactic alteration (unless they define some of the reserved identifiers that we will introduce) if we were to execute them in an implementation of *LLF* relying on the new syntax. In other words, the extension we propose should be conservative with respect to the syntax of *Elf*. Second, we want at avoid a proliferation of operators: keeping their number as small as possible will make future extensions easier to accommodate if their inclusion appears beneficial.

The set of special characters of *Elf* consists of % : .) ( ] [} {. We extend these with two symbols: , and ^. $\lambda^{\Pi\multimap\&\top}$ object and type family constants are consequently represented as identifiers consisting of any non-empty string that does not contain spaces or the characters % : .) ( ] [} { , ^. As in *Elf*, identifiers must be separated from each other by whitespace (i.e., blanks, tabs and new lines) or special characters. We augment the set of reserved identifiers of *Elf* (type, -> and <-) with <T>, &, -o, o-, <fst> and <snd>. Although not properly an identifier, the symbol () is also reserved; this string is forbidden in *Elf*.

The following table associates every $\lambda^{\Pi\multimap\&\top}$ operator to its concrete representation. Terms in the λ^{Π} sublanguage of *LLF* are mapped to the syntax of *Elf*. This language offers the convenience of writing -> as <- with the arguments reversed in order to give a more operational reading to a program, when desired: under this perspective, we read the expression A <- B as "A *if* B". We extend this possibility to linear implication, -o. Clearly, when we use o-, the arguments should be swapped.

	Abstract syntax	*Concrete syntax*		
Kinds	TYPE	type		
	$\Pi x : A.\, K$	{x:A}K	A -> K	K <- A
Types	$P\, M$	P M		
	$\top$	<T>		
	$A \,\&\, B$	A & B		
	$A \multimap B$	A -o B	B o- A	
	$\Pi x : A.\, B$	{x:A}B	A -> B	B <- A
Objects	$\langle\rangle$	()		
	$\langle M, N\rangle$	M,N		
	FST M	<fst> M		
	SND M	<snd> M		
	$\hat{\lambda} x : A.\, M$	[x^A]M		
	$M \,\hat{}\, N$	M ^ N		
	$\lambda x : A.\, M$	[x:A]M		
	$M\, N$	M N		

The next table gives the relative precedence and associativity of these operators. As in *Elf*, parentheses are available to override these behaviors.

Precedence	*Operator*			*Position*
highest	_ _	_ ^ _		left associative
	_ & _			right associative
	_ -o _	_ -> _		right associative
	_ o- _	_ <- _		left associative
	<fst> _	<snd> _		left prefix
	_,_			right associative
lowest	{_:_}_	[_:_]_	[_^_]_	left prefix

As in *Elf*, a signature declaration $c{:}A$ is represented by the program clause

$$c \ : \ A.$$

Type family constants are declared similarly. For practical purposes, it is convenient to provide a means of declaring linear assumptions. Indeed, whenever the object formalism we want to represent requires numerous linear hypotheses, it is simpler to write them as program clauses than to rely on some initialization routine that assumes them in the context during its execution.

To this end, we permit declarations of the form

$$c \mathbin{\hat{}} A.$$

with the intent that this declaration should be inserted in the context as a linear assumption.

We retain from *Elf* the use of `%` for comments and interpreter directives. We adopt the conventions available in that language in order to enhance the readability of *LLF* programs [Pfe91]. In particular, we permit keeping the type of bound variables implicit whenever they can be effectively reconstructed. We write `{`x`}`B, `[`x`]`B and `[`x`^]`B when maintaining implicit the type A of the variable x in `{`x:A`}`B, `[`x:A`]`B and `[`x`^`A`]`B, respectively. Similar conventions apply to dependent kinds. As in *Elf*, the binders for variables quantified at the head of a clause can be omitted altogether if we write these variables with identifiers starting with a capital letter. Moreover, the arguments instantiating them can be kept implicit when using these declarations.

Finally, we relax the requirement of writing *LLF* declarations only in η-long form. With sufficient typing information it is always possible to transform a signature to that format.

The encoding we present in the next section is written according to this syntactic extension of *Elf* and adopts the conventions just described. Since no implementation of *LLF* is currently available, it has not been possible to test these programs. Therefore their correctness is not guaranteed. However, we partially type-checked them by promoting the new constructs to the syntax of *Elf* and loading the result into the current interpreter for this language. They are therefore syntactically correct up to the linearity constraints which have not been checked mechanically.

3 The Methodology of Linear Meta-Representation

LF and *Elf* constitute a remarkable tool for studying existing logics and programming languages, and an ideal playground for experimenting with alternative constructs in the design phase of new languages. The range of practical applicability of these formalisms is limited by their foundation on intuitionistic type theory. All the formal systems that have been successfully encoded in *LF* (functional and logic programming languages [MP91, Pfe92], λ-calculi [Pfe99] and a number of logics [HHP93, Pfe95]) share a fundamental characteristic: whenever a judgment mentions a context, a bottom-up reading of the inference rules for it may add items, but they never remove assumptions. We call contexts with this property *permanent*, in contrast with *volatile* contexts, free from this restriction. Object formalisms admitting arbitrary operations on their context cannot be effectively encoded in *LF*: the standard technique, representing object context items as *LF* assumptions, is not sound in this case since *LF* assumptions are permanent. The alternative is to represent the object context as a term in *LF* and implement explicitly the operations required to access and manipulate it. This is undesirable since it makes the adequacy results difficult to prove and often complicates the encoding of meta-theoretic properties to the point of making it hardly manageable in practice.

This situation is quite unfortunate since most formalisms of practical significance rely on a volatile context in an essential manner. The languages used for programming commercial applications are imperative: they have a store and assignment instructions to change the value of variables. Most real-world problems carry a state that changes with time. Many new logics and type theories are inherently bound to destructive context manipulations. Permanent contexts are insufficient even for more traditional formalisms, for example when studying efficient proof-search procedures for intuitionistic logic [Dyc92]. These applications are encoded awkwardly not only in *LF*, but in general in all intuitionistic frameworks based on traditional languages.

The linear type theory $\lambda^{\top\&\multimap\Pi}$ presented in the previous section retains all the desirable properties of *LF* and also augments this formalism with linear assumptions, admitting volatile manipulations, and with a suitable set of operators to manage them. These new features overcome the above deficiency of *LF*: if we represent the volatile context of an object language as linear assumptions in $\lambda^{\top\&\multimap\Pi}$, destructive context operations in the object formalism can be modeled by an appropriate combination of linear operators.

The linear logical framework *LLF* is founded on the type theory $\lambda^{\top\&\multimap\Pi}$ and combines as its meta-representation methodology the *judgments-as-types* technique of *LF* with the above observation. The resulting formalism broadens the range of applicability of *LF* to formal systems revolving around a volatile context, such as imperative programming languages, systems based on state transformations and presentations of logics relying on destructive context operations. In this respect, *LLF* brings the potential benefits of logical frameworks closer to real-world applications.

The present section illustrates the added expressiveness of *LLF* as a logical framework by describing the meta-representation methodology it adopts first abstractly and then on a concrete case study. The formalism we want to represent is an imperative extension of *Mini-ML* [HM90, MP91, Pfe92], a purely functional restriction of the programming language *ML* [MTHM96]. More precisely, we augment that language with a store and imperative instructions to access and modify the values it contains, we formalize the typing and evaluation semantics of these constructs and show that this extended language enjoys the type preservation property. We call this language *MLR*, for *Mini-ML with References*. The linear assumptions of *LLF* can be used to encode individual memory cells and the linear operators of our type theory offer effective tools to model manipulations on them.

We review the judgments-as-types representation methodology and extend it to handle volatile assumptions in Section 3.1. Then, we give a precise but informal presentation of the syntax, semantics and of the type preservation property for *MLR* in Section 3.2. Finally, we show how to encode these different aspects in *LLF* in Section 3.3. Appendix A contains the complete *LLF* signature for this example. In the following, we will concentrate mainly on the novel constructions available in *MLR*, referring the reader to the literature [CDDK86, HM90, MP91, Pfe92] for aspects already present in *Mini-ML*.

3.1 Judgments-as-Types Revisited

A logical framework consists of a representation language and a methodology that prescribes how to best take advantage of the constructs it offers when encoding an object formalism. *LF* [HHP93] relies on λ^{Π} as its meta-language and the *judgments-as-types* methodology for achieving effective meta-representations.

We will describe this technique by analyzing the following simplified rule of inference from the case study in this section.

$$\frac{\Gamma, x{:}\tau \vdash^{e} e : \tau}{\Gamma \vdash^{e} \mathbf{fix}\, x.\, e : \tau}\ \mathbf{tpe\_fix}$$

Ignoring for the moment the context Γ, it specifies that the fix-point expression $\mathbf{fix}\, x.\, e$ has type τ if e has type τ assuming that the variable x has also type τ. We will emphasize the fact that x can occur in e by writing $e(x)$. Given a closed expression $\mathbf{fix}\ x.\, e(x)$, the judgment in the conclusion of **tpe_fix** postulates that $\mathbf{fix}\, x.\, e(x)$ has type τ (we need to provide a derivation to ascertain that this is indeed the case). We call a judgment of this form *simple*. The judgments-as-type representation methodology encodes simple judgments as λ^{Π} base types. In Section 3.3, we will use the type family constants EXP and TP, both of kind TYPE to classify the expressions and the types of the object language, respectively. The general form of the typing judgment above relates an expression and a type, and therefore we encode it as a type family TPE, of kind $\mathrm{EXP} \rightarrow \mathrm{TP} \rightarrow \mathrm{TYPE}$. Given representations $(\mathrm{FIX}\,\lambda x{:}\mathrm{EXP}.\, \ulcorner e \urcorner\, x)$ and $\ulcorner \tau \urcorner$ (to be explained below) for the closed expression $\mathbf{fix}\ x.\, e(x)$ and for the object language type τ, the simple judgment $\Gamma \vdash^{e} \mathbf{fix}\, x.\, e(x) : \tau$ is represented as

$$\mathrm{TPE}\ (\mathrm{FIX}\ (\lambda x{:}\mathrm{EXP}.\, \ulcorner e \urcorner\, x))\ \ulcorner \tau \urcorner$$

The judgment in the premiss of rule **tpe_fix** is different in nature. Indeed, it specifies that the expression $e(x)$ has type τ *if* we assume that the variable x has also type τ. A judgment of this form is called *hypothetical* since it intrinsically relies on the hypothesis that x has type τ. Notice also that x is a bound variable in $\mathbf{fix}\ x.\, e(x)$, but it is free in $e(x)$. Therefore, that premiss expresses the fact that $e(x)$ has type τ for a generic expression x of type τ. The judgment $\Gamma, x : \tau \vdash^{e} e(x) : \tau$ is therefore said to be *parametric* in x. The judgments-as-types representation methodology encodes hypothetical and parametric judgments by means of simple and dependent function types respectively. The premiss of the rule above, which is parametric in x and hypothetical in $x{:}\tau$, is represented as follows

$$\Pi x{:}\mathrm{EXP}.\, \mathrm{TPE}\ x\ \ulcorner \tau \urcorner \rightarrow \mathrm{TPE}\ (\ulcorner e \urcorner\, x)\ \ulcorner \tau \urcorner$$

Notice that instantiating the parameter x with some term e' yields a hypothetical judgment postulating that $e(e')$ has type τ assuming that e' has type τ. This reduces to a simple judgment as soon as we provide a derivation for this hypothesis.

An attempt at finding a canonical *LF* derivation with the above type reduces to searching for a derivation for the base type $\mathrm{TPE}\ (\ulcorner e \urcorner\, x)\ \ulcorner \tau \urcorner$ after having added the assumptions $x : \mathrm{EXP}$

and $t_x : \text{TPE}\ x\ \ulcorner\tau\urcorner$ to the context of LF. Viewing this as an alternate encoding for the premiss of rule **tpe_fix** illustrates the manner an object context is encoded according to the judgments-as-types methodology: each item in the context of the object formalism is represented as one or more assumptions in the context of LF. This technique offers the further advantage that we can rely on the primitive operations of LF to simulate the look up of object level assumptions. Less sophisticated representations, for example those that encode the object context as a term, must provide explicit access operations.

Observe that rule **tpe_fix** can be read as a judgment that is parametric in the (functional) expression e and the type τ, and hypothetical in the derivability of its premiss. Indeed, it is encoded as the following declaration

$$\begin{aligned}\text{TPE\_FIX}:\ & \Pi e : \text{EXP} \to \text{EXP}.\, \Pi\tau : \text{TP}.\\ & \quad (\Pi x : \text{EXP}.\, \text{TPE}\ x\ \tau \to \text{TPE}\ (e\ x)\ \tau)\\ & \to \text{TPE}\ (\text{FIX}\ (\lambda x : \text{EXP}.\, e\ x))\ \tau\end{aligned}$$

or, taking advantage of the concrete syntax of *Elf* (see Section 2.9),

```
tpe_fix :   ({x:exp} tpe x T -> tpe (E x) T)
         -> tpe (fix ([x:exp] E x)) T.
```

In summary, the judgments-as-types representation methodology for LF encodes simple judgments as base types, hypothetical and parametric judgments as simple and dependent types respectively, and element of the object context as items in the context of LF. Moreover, derivations for a simple judgments are naturally represented as terms of the corresponding base type.

The judgments-as-types methodology interacts particularly well with *higher-order abstract syntax*, a technique for the representation of the syntactic level of an object formalism that encodes object variables as meta-variables and relies on the λ-abstraction of λ^Π to emulate generic object-level binding constructs. Above, we encoded the fix-point expression $\mathbf{fix}\ x.\, e(x)$, that bind the variable x in $e(x)$ as $(\text{FIX}\ (\lambda x : \text{EXP}.\ulcorner e\urcorner\ x))$. We used the λ-abstraction of LF to express binding, and consequently encoded the operator **fix** by means of the LF constant `fix` that accepts a functional operator (`fix : (exp -> exp) -> exp`).

The faithfulness of the representation of an object formalism is captured by means of *adequacy theorems* that relate the entities being represented to their encoding. An important advantage of the judgments-as-types technique with respect to less sophisticated approaches is that it produces encodings very close to the notations being formalized. This makes the adequacy theorems easy to prove.

We just saw that the judgments-as-types representation methodology encodes the context of the object language in the context of LF, and relies on the operations of this meta-language to emulate object-level manipulations. Here, and in the remainder of this paper, we view and describe operations on the context as they arise when we construct derivations "bottom-up", that is, from the judgment in question towards the axioms. This view is the most natural one to elucidate the examples and anticipates the logic programming interpretation of LLF. For example, instead of saying that we *discharge* a hypothesis in rule **opc_ilam** in Figure 3 we say that we *introduce* a hypothesis. From this point of view, λ^{Π} offers two operations on its context: insertion and look-up. In particular, the context can only grow during the bottom-up construction of a derivation. Therefore, the judgments-as-types methodology in λ^{Π} cannot capture object languages that perform deletion on their context. Consider as an example the following inference rule, taken from the case study in the next section:

$$\frac{S, c = v, S' \rhd K \;\vdash\; \textbf{return}\ \langle\rangle \hookrightarrow a}{S, c = v', S' \rhd K \;\vdash\; c :=_2^* v \hookrightarrow a}\ \textbf{ev\_assign}_2^*$$

This rule describes the semantics of assignment in an imperative programming language. It specifies that, in order to assign the value v to the cell c, we must update the binding $c = v'$ in the store with $c = v$. An elegant encoding of this system in *LF* would represent each cell-value pair in the store as a meta-level assumption. However, λ^{Π} does not provide means to simulate the deletion of the old binding, $c = v'$.

In contrast, we can easily achieve this effect in *LLF* since its meta-language is linear. Indeed, looking up a *linear* assumption in $\lambda^{\Pi\multimap\&\top}$ removes it from the context. This suggests encoding each cell-value pair $c = v$ present at any instant in the store of the object language as an *LLF* linear assumption `Cn ^ contains c` $\ulcorner v \urcorner$.

The linear type constructors of $\lambda^{\Pi\multimap\&\top}$ provide the necessary means to manipulate such assumptions. We rely on $\multimap$ to enter them in the context of *LLF* and take advantage of the context splitting semantics of this operator to isolate them in order to access them. The additive product type constructor, &, offers means to duplicate or share linear assumptions among its two conjuncts. This operator can also be used to express selection between exclusive alternatives, although we will not take advantage of this feature here. Finally, the unit type, $\top$, permits discarding unused linear hypotheses. Furthermore, the dependent type constructor Π can be used to express constraints on the contents of the linear portion of the context. It also offers the possibility of introducing intuitionistic assumptions in the context of $\lambda^{\Pi\multimap\&\top}$.

These different features will be illustrated in detail in Section 3.3. We just show the encoding of the rule above:

```
ev_assign*2 : contains C V'
               -o (contains C V
                     -o ev K (return unit) A)
               -o ev K (assign*2 (rf C) V) A.
```

The presence of the linear operators of $\lambda^{\Pi\multimap\&\top}$ broadens the range of applicability of the judgments-as-types methodology to formal systems that require occasionally deleting assumptions from context-like structures that appear in the judgments that define them. Furthermore, linearity in our logical framework can be integrated into higher-order abstract syntax as a convenient manner of encoding languages relying on linear binders [Cer96]. Notice that these possibility extend conservatively the representation policy of *LF*: whenever an object formalism does not rely on context deletion or on linear binders, it can be encoded in *LLF* exactly in the same manner as it would be represented in *LF*.

3.2 *Mini-ML* with References

Critical choices in the implementation of programming languages depend on the validity of meta-theoretic properties. Type preservation in *Standard ML* [MTHM96], for example, guarantees that no typing error can arise during evaluation; therefore execution can be sped up significantly by disregarding type information at run-time. Meta-theoretic properties in the presence of non-functional features, included in most concrete languages, are difficult to prove and therefore the formal analysis of imperative extensions of purely functional programming languages has received great attention in the literature. The addition of references and their interaction with polymorphism has been analyzed with different tools, ranging from the complex domain-theoretic approach of Damas [Dam85] to the syntactic formulation of Harper [Har94]. The latter idea was adapted from Wright and Felleisen, who additionally consider continuations and exceptions [WF94].

The proofs of these properties are long and error-prone. Therefore, recent work has investigated the possibility of partially automating their generation or at least their verification. Chirimar gives *Forum* specifications for a language with references, exceptions and continuations and uses the meta-theory of *Forum* [Mil94] to study program equivalence [Chi95]. VanInwegen [Van96] formally proved properties such as value soundness for most of *Standard ML* with the help of the *HOL* theorem prover [GM93].

In this section, we define *MLR* as an extension of *Mini-ML* with references and imperative instructions, and study aspects of its meta-theory. Although our principal objective is to demonstrate the expressive power of *LLF*, our presentation differs in some aspects from the formulations and proofs in the literature and therefore might be interesting in itself. We will point out differences and similarities with other approaches as they arise.

Expressions and Store

Since its introduction in [CDDK86], the language *Mini-ML* and variants of it have been used for case studies in the presentation of logical frameworks [HM90, MP91, Pfe92]. *Mini-ML* is a purely functional restriction of the programming language *ML* [MTHM96]. More specifically, it is a small functional programming language based on a simply typed λ-calculus. It includes numerals, conditional expressions, pairs, polymorphic definitions, recursion and functional expressions.

We consider an extension of *Mini-ML* with a store and imperative instructions in the style of *ML* to access and modify the values it contains. We call this language *Mini-ML with References*, or *MLR* for short. The *store* of an *MLR* program is defined as a collection of *cells* each containing a value. We will sometimes use *location* or *address* as synonyms of cell. *MLR* retains all the constructs of *Mini-ML* but enriches the syntax of its expressions with the necessary operations to manipulate individual cells. The resulting language is specified by the following grammar, where we have separated out the constructs not present in standard presentations of *Mini-ML* with a double bar ($\|$). Cells c and stores S are not directly accessible to the programmer, but it is customary and convenient to enrich the syntax in order to represent intermediate stages during computation.

$$
\begin{array}{llll}
\textit{Expressions:} & e ::= & x & (\textit{Variables}) \\
 & & \mid \mathbf{z} \mid \mathbf{s}\, e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\, x \Rightarrow e_2 & (\textit{Natural numbers}) \\
 & & \mid \langle\rangle & (\textit{Unit element}) \\
 & & \mid \langle e_1, e_2 \rangle \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e & (\textit{Pairs}) \\
 & & \mid \mathbf{lam}\, x.\, e \mid e_1\, e_2 & (\textit{Functions}) \\
 & & \mid \mathbf{letval}\, x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{letname}\, x = e_1\ \mathbf{in}\ e_2 & (\textit{Definitions}) \\
 & & \mid \mathbf{fix}\, x.\, e & (\textit{Recursion}) \\
 & & \parallel c \mid \mathbf{ref}\, e \mid \mathbf{!}e & (\textit{References}) \\
 & & \mid e_1 := e_2 \mid e_1 ; e_2 & (\textit{Commands}) \\
\textit{Stores:} & S ::= & \cdot \mid S, c = v &
\end{array}
$$

In these productions, c ranges over the lexical category of memory locations, while we use the letter x for variables. The meta-variable v denotes values, that we will define shortly. The polymorphism in *MLR* is restricted to values, which is generally accepted as superior to the imperative type variables present in previous versions of *SML* [LW91]. We achieve this by distinguishing two forms of **let**. The expression **ref** e dynamically allocates a cell and initializes it with the value of e. The contents of a cell can be inspected by dereferencing it with **!** and modified with an assignment (:=). Differently from [WF94], but consistently with the main stream in the literature (including the definition of *Standard ML* [MTHM96]), we choose this operation not to return the assigned object, but the unit element $\langle\rangle$. The sequencing operator (;) is typically used as a means of chaining a series of assignments with some interesting final value; it is syntactic sugar for the expression ($\mathbf{letval}\, x = e_1\ \mathbf{in}\ e_2$) when x does not occur in e_2. As is normally the case in functional languages, *MLR* does not offer explicit means to deallocate memory cells.

All these constructs are available in *Standard ML* [MTHM96] with the exception of addresses themselves (c), which cannot be manipulated directly in that language. We require *MLR* programs not to mention locations directly so that cells are always guaranteed to be initialized. Thus cells are created dynamically with **ref** and can be named by binding them to variables with one of the two **let** constructs of *MLR*.

As in *ML*, the reference cells of *MLR* encompass two distinct features of imperative programming languages such as *C* or *Pascal*. First of all, they play the role of the imperative variables of these languages and can be used as such (except for the necessity of dereferencing them explicitly in order to access their value). Second, we can use them as pointers in data structures, although their usefulness is rather limited in this respect due to the absence of recursive data types in *MLR*. Such data structures could be easily added to the language.

Typing

The language of types of *MLR* augments the typing constructs typically present in *Mini-ML*, namely natural numbers, unit, pairing and functional types, with one new constructor: for each type τ, the type τ **ref** for references to objects of type τ. The syntax of types is summarized in the following grammar:

$$\begin{array}{rrl} \textit{Types:} & \tau ::= & \mathbf{nat} \mid \mathbf{1} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \\ & \| & \tau\ \mathbf{ref} \end{array}$$

On the basis of this definition, the static semantics of *MLR* naturally extends the traditional typing rules of *Mini-ML*. The possibility of expressions to mention cells requires introducing a *store context* as a means to declare the type of free locations. More precisely, the item $c{:}\tau$ in a store context declares τ as the type of the values that c can contain; c itself has consequently type τ **ref**. Contexts, as usual, assign types to free variables. They are constructed according to the following grammar:

$$\begin{array}{rl} \textit{Contexts:} & \Gamma ::= \cdot \mid \Gamma, x{:}\tau \\ \textit{Store contexts:} & \Omega ::= \cdot \mid \Omega, c{:}\tau \end{array}$$

We rely on the usual convention that the names of the variables and the cells declared in stores and context stores, respectively, are distinct. Moreover, we treat both forms of contexts as multisets.

The presence of a store context in the typing rules for *MLR* is necessary even if we forbid the users to write addresses directly in their programs. It accounts for cells dynamically allocated during evaluation, which may appear in intermediate results and in the final answer.

We express the fact that the *MLR* expression e has type τ with respect to a store context Ω and a context Γ with the judgment

$$\Omega; \Gamma \vdash^{e} e : \tau$$

The inference rules for this judgment are displayed in Figure 8. The upper part of this figure shows the rules for the functional core of *MLR*. The changes with respect to the usual rules for *Mini-ML* are limited to the systematic inclusion of a store context in the judgments.

The central part of Figure 8 shows the rules for the novel features of *MLR*. As for the functional case, they express the conditions under which an expression can be statically accepted as meaningful. For example, rule **tpe_deref** enforces that only references be dereferenced.

We present in the lower part of Figure 8 the rules for typing a store. The judgments we consider have the form

$$\Omega' \vdash^{S} S : \Omega$$

that we interpret as requiring that the type of each value v stored in S coincides with the type of the corresponding cell as specified in Ω. The store context Ω' gives the type of the cells v may mention. We will always be interested in top-level judgments of the form $\Omega \vdash^{S} S : \Omega$ since a store will in general refer circularly to its own cells. Rule **tpS_cell** prevents expressions containing free variables from being inserted in the store.

Expressions

$$\frac{}{\Omega;\Gamma,x{:}\tau \vdash^e x:\tau}\,\textbf{tpe\_x} \qquad \frac{}{\Omega;\Gamma \vdash^e \langle\rangle:\mathbf{1}}\,\textbf{tpe\_unit}$$

$$\frac{}{\Omega;\Gamma \vdash^e \mathbf{z}:\mathbf{nat}}\,\textbf{tpe\_z} \qquad \frac{\Omega;\Gamma \vdash^e e:\mathbf{nat}}{\Omega;\Gamma \vdash^e \mathbf{s}\,e:\mathbf{nat}}\,\textbf{tpe\_s}$$

$$\frac{\Omega;\Gamma \vdash^e e:\mathbf{nat} \quad \Omega;\Gamma \vdash^e e_1:\tau \quad \Omega;\Gamma,x{:}\mathbf{nat} \vdash^e e_2:\tau}{\Omega;\Gamma \vdash^e \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z}\ \Rightarrow\ e_1 \mid \mathbf{s}\,x\ \Rightarrow\ e_2:\tau}\,\textbf{tpe\_case}$$

$$\frac{\Omega;\Gamma \vdash^e e_1:\tau_1 \quad \Omega;\Gamma \vdash^e e_2:\tau_2}{\Omega;\Gamma \vdash^e \langle e_1,e_2\rangle:\tau_1\times\tau_2}\,\textbf{tpe\_pair} \qquad \frac{\Omega;\Gamma \vdash^e e:\tau_1\times\tau_2}{\Omega;\Gamma \vdash^e \mathbf{fst}\,e:\tau_1}\,\textbf{tpe\_fst} \qquad \frac{\Omega;\Gamma \vdash^e e:\tau_1\times\tau_2}{\Omega;\Gamma \vdash^e \mathbf{snd}\,e:\tau_2}\,\textbf{tpe\_snd}$$

$$\frac{\Omega;\Gamma,x{:}\tau_1 \vdash^e e:\tau_2}{\Omega;\Gamma \vdash^e \mathbf{lam}\,x.\,e:\tau_1\to\tau_2}\,\textbf{tpe\_lam} \qquad \frac{\Omega;\Gamma \vdash^e e_1:\tau_2\to\tau_1 \quad \Omega;\Gamma \vdash^e e_2:\tau_2}{\Omega;\Gamma \vdash^e e_1\ e_2:\tau_1}\,\textbf{tpe\_app}$$

$$\frac{\Omega;\Gamma \vdash^e e_1:\tau_1 \quad \Omega;\Gamma,x{:}\tau_1 \vdash^e e_2:\tau_2}{\Omega;\Gamma \vdash^e \mathbf{letval}\,x\ =\ e_1\ \mathbf{in}\,e_2:\tau_2}\,\textbf{tpe\_letval} \qquad \frac{\Omega;\Gamma \vdash^e [e_1/x]e_2:\tau}{\Omega;\Gamma \vdash^e \mathbf{letname}\,x\ =\ e_1\ \mathbf{in}\,e_2:\tau}\,\textbf{tpe\_letname}$$

$$\frac{\Omega;\Gamma,x{:}\tau \vdash^e e:\tau}{\Omega;\Gamma \vdash^e \mathbf{fix}\,x.\,e:\tau}\,\textbf{tpe\_fix}$$

$$\frac{}{\Omega,c{:}\tau;\Gamma \vdash^e c:\tau\ \mathbf{ref}}\,\textbf{tpe\_cell} \qquad \frac{\Omega;\Gamma \vdash^e e:\tau}{\Omega;\Gamma \vdash^e \mathbf{ref}\,e:\tau\ \mathbf{ref}}\,\textbf{tpe\_ref} \qquad \frac{\Omega;\Gamma \vdash^e e:\tau\ \mathbf{ref}}{\Omega;\Gamma \vdash^e \mathbf{!}e:\tau}\,\textbf{tpe\_deref}$$

$$\frac{\Omega;\Gamma \vdash^e e_1:\tau_1 \quad \Omega;\Gamma \vdash^e e_2:\tau_2}{\Omega;\Gamma \vdash^e e_1;e_2:\tau_2}\,\textbf{tpe\_seq} \qquad \frac{\Omega;\Gamma \vdash^e e_1:\tau\ \mathbf{ref} \quad \Omega;\Gamma \vdash^e e_2:\tau}{\Omega;\Gamma \vdash^e e_1 := e_2:\mathbf{1}}\,\textbf{tpe\_assign}$$

Store

$$\frac{}{\Omega \vdash^S \cdot:\cdot}\,\textbf{tpS\_empty} \qquad \frac{\Omega \vdash^S S:\Omega' \quad \Omega;\cdot \vdash^e v:\tau}{\Omega \vdash^S (S,c=v):(\Omega',c{:}\tau)}\,\textbf{tpS\_cell}$$

Figure 8: Typing Rules in *MLR*, Expressions and Store

Evaluation

An *MLR* expression e will in general mention reference cells whose values are contained in the store. The evaluation of e will typically not only retrieve these values, but also change them or create new cells. Therefore, as e is evaluated, the store will undergo transformations, and by the time a value for e is eventually produced, it might appear very different from the store we started with. This observation suggests an evaluation judgment of the form

$$S;e \hookrightarrow S';v$$

where S is the store prior to evaluating e, and S' results from the evaluation of e to v: cells in e refer to S while cells in v refer to S'. This formulation extends the traditional evaluation judgment for *Mini-ML* [HM90, MP91, Pfe92].

The dynamic semantics of functional languages enriched with imperative features, such as *MLR*'s references, is normally expressed in the literature in this manner. We will instead adopt a different strategy and present the reductions occurring during the execution of an *MLR* program as continuation-based evaluation rules. This choice has been dictated by our intention to encode the semantics of *MLR* in *LLF*. A direct representation of the judgment above, although possible, would have resulted in a less elegant encoding.

Differently from more declarative formulations, a continuation-based execution strategy imposes a strict order of evaluation on the different subexpressions of any given construct in the language. This order respects the expected flow of data and is therefore natural. For example, when computing the value of an expression of the form ($\mathbf{letval}\ x = e_1\ \mathbf{in}\ e_2$) we will first evaluate e_1, obtain a value v', substitute it for x in e_2 and only then evaluate the resulting expression.

An effective implementation of this strategy requires sequentializing the evaluation of the subexpressions of constructs with more than one argument. One of them is evaluated immediately while the evaluation of the others is postponed until a value has been produced for it. Clearly, if a subexpression depends on the value of another, we process it last. We realize this idea by maintaining a stack of expressions to be evaluated, called a *continuation*.

Postponing the evaluation of an expression e_2 in favor of another expression e_1 is achieved by pushing the former into the continuation. Since, as when evaluating ($\mathbf{letval}\ x = e_1\ \mathbf{in}\ e_2$) for example, the value of e_1 might need to be substituted for some free variable x in e_2, we wrap a binder for x around e_2 and thus insert an object of the form $\lambda x.\, e_2$ into the continuation. For uniformity, it is convenient to take this measure every time we insert an item into the stack. As soon as e_1 has been fully evaluated to a value v, $\lambda x.\, e_2$ is extracted from the continuation, v is substituted for the variable x in e_2, and $[v/x]e_2$ is evaluated in turn.

The necessity of distinguishing expressions still to be evaluated from values being returned requires the introduction of the new syntactic layer of *instructions*. Specifically, we write $\mathbf{eval}\ e$ for the request to evaluate an expression e and denote the intention to return a value v as $\mathbf{return}\ v$. Instructions are needed also for the purpose of handling partially evaluated expression.

While evaluating a *Mini-ML* expression simply yielded a value, *MLR* expressions will in general produce objects mentioning cells. Therefore the result of the evaluation of an instruction i must include not only a final value v but also the final store S' it draws its references from; moreover, as a measure of hygiene, we mark the cells c that have been introduced during the evaluation process by binding them in front of the pair (S', v) by means of the $\mathbf{new}\ c.\, \_$ operator. The resulting object is called an *answer* and is indicated with the letter a.

The structure of instructions, continuations and answers is given by the following grammar, where we have indicated with the double bar the instructions introduced in correspondence to the imperative constructs of *MLR*.

Instructions

$$\frac{\Omega;\Gamma \vdash^e e : \tau}{\Omega;\Gamma \vdash^i \mathbf{eval}\, e : \tau}\,\textbf{tpi\_eval} \qquad \frac{\Omega;\Gamma \vdash^e v : \tau}{\Omega;\Gamma \vdash^i \mathbf{return}\, v : \tau}\,\textbf{tpi\_return}$$

$$\frac{\Omega;\Gamma \vdash^e v : \mathbf{nat} \quad \Omega;\Gamma \vdash^e e_1 : \tau \quad \Omega;\Gamma, x{:}\mathbf{nat} \vdash^e e_2 : \tau}{\Omega;\Gamma \vdash^i \mathbf{case}^*\ v\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\,x \Rightarrow e_2 : \tau}\,\textbf{tpi\_case}^*$$

$$\frac{\Omega;\Gamma \vdash^e v : \tau_1 \quad \Omega;\Gamma \vdash^e e : \tau_2}{\Omega;\Gamma \vdash^i \langle v, e\rangle^* : \tau_1 \times \tau_2}\,\textbf{tpi\_pair}^* \qquad \frac{\Omega;\Gamma \vdash^e v : \tau_1 \times \tau_2}{\Omega;\Gamma \vdash^i \mathbf{fst}^*\ v : \tau_1}\,\textbf{tpi\_fst}^* \qquad \frac{\Omega;\Gamma \vdash^e v : \tau_1 \times \tau_2}{\Omega;\Gamma \vdash^i \mathbf{snd}^*\ v : \tau_2}\,\textbf{tpi\_snd}^*$$

$$\frac{\Omega;\Gamma \vdash^e v : \tau_2 \to \tau_1 \quad \Omega;\Gamma \vdash^e e : \tau_2}{\Omega;\Gamma \vdash^i \mathbf{app}^*\ v\ e : \tau_1}\,\textbf{tpi\_app}^*$$

$$\frac{\Omega;\Gamma \vdash^e v : \tau}{\Omega;\Gamma \vdash^i \mathbf{ref}^*\ v : \tau\ \mathbf{ref}}\,\textbf{tpi\_ref}^* \qquad \frac{\Omega;\Gamma \vdash^e v : \tau\ \mathbf{ref}}{\Omega;\Gamma \vdash^i \mathbf{deref}^*\, v : \tau}\,\textbf{tpi\_deref}^*$$

$$\frac{\Omega;\Gamma \vdash^e v : \tau\ \mathbf{ref} \quad \Omega;\Gamma \vdash^e e : \tau}{\Omega;\Gamma \vdash^i v :=_1^* e : \mathbf{1}}\,\textbf{tpi\_assign1}^* \qquad \frac{\Omega;\Gamma \vdash^e v_1 : \tau\ \mathbf{ref} \quad \Omega;\Gamma \vdash^e v_2 : \tau}{\Omega;\Gamma \vdash^i v_1 :=_2^* v_2 : \mathbf{1}}\,\textbf{tpi\_assign2}^*$$

Continuations

$$\frac{}{\Omega \vdash^K \mathbf{init} : \tau \Rightarrow \tau}\,\textbf{tpK\_init} \qquad \frac{\Omega; x{:}\tau_1 \vdash^i i : \tau \quad \Omega \vdash^K K : \tau \Rightarrow \tau_2}{\Omega \vdash^K K, \lambda x.\, i : \tau_1 \Rightarrow \tau_2}\,\textbf{tpK\_lam}$$

Answers

$$\frac{\Omega \vdash^S S : \Omega \quad \Omega; \cdot \vdash^e v : \tau}{\Omega \vdash^a (S, v) : \tau}\,\textbf{tpa\_val} \qquad \frac{\Omega, c{:}\tau' \vdash^a a : \tau}{\Omega \vdash^a \mathbf{new}\ c.\, a : \tau}\,\textbf{tpa\_new}$$

Figure 9: Typing Rules in *MLR*, Instructions, Continuations and Answers

$$\begin{array}{rrcl}
\textit{Instructions:} & i & ::= & \mathbf{eval}\, e \mid \mathbf{return}\, v \\
 & & \mid & \mathbf{case}^*\ v\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\,x \Rightarrow e_2 \\
 & & \mid & \langle v, e\rangle^* \mid \mathbf{fst}^*\ v \mid \mathbf{snd}^*\ v \\
 & & \mid & \mathbf{app}^*\ v\ e \\
 & & \| & \mathbf{ref}^*\ v \mid \mathbf{deref}^*\, v \mid v :=_1^* e \mid v_1 :=_2^* v_2 \\
\textit{Continuations:} & K & ::= & \mathbf{init} \mid K, \lambda x.\, i \\
\textit{Answers:} & a & ::= & (S, v) \mid \mathbf{new}\ c.\, a
\end{array}$$

The typing rules for objects in these three categories are displayed in Figure 9. Notice that the type of an answer coincides with the type of the embedded value. Rule **tpa_val** requires that the store it is paired with be well-typed, while rule **tpa_new** constrains every occurrence of the cells bound in an answer to be consistently typed.

Values constitute the subclass of expressions that evaluate to themselves. They are specified by the following grammar.

$$\begin{array}{lll} \textit{Values:} & v ::= & x \mid \mathbf{z} \mid \mathbf{s}\, v \mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid \mathbf{lam}\, x.\, e \\ & \quad \parallel & c \end{array}$$

On the basis of this definition, we can justify the uses of the term "value" in the above presentation. Not only does **return** operates only on values, but computation places a value at the heart of answers and the contents of every cell in the store is a value. See [Cer96] for a formal statement of these properties.

We model the continuation-based semantics of the imperative constructs of *MLR* by means of a judgment of the form

$$S \triangleright K \vdash i \hookrightarrow a$$

where i is the instruction to be executed, K is the current continuation, S is the store with respect to which i is to be evaluated and a is the final answer produced as the result of the evaluation.

The inference rules for evaluation are given in Figures 10–11. The evaluation of instructions in the functional core of *MLR* never accesses the store. The only exception is rule **ev_init** which must package the current store together with the produced value in order to construct the final answer. The inference rules concerned with non-functional expressions of *MLR* and the corresponding instructions are separated out by a dotted line in Figures 10 and 11, respectively.

Cells (rule **ev_cell**) simply evaluate to themselves, as any value. The sequencing instruction $e_1; e_2$ has a simple semantics too: it evaluates e_1, disregards the returned value, and then proceed with the evaluation of e_2 (rule **ev_seq**).

The evaluation of $\mathbf{ref}\, e$ computes the value v of e (rule **ev_ref**), allocates a new cell c in the store, initializes it with v and finally returns c itself. Notice that rule **ev_ref*** has also the effect of binding c in the final answer by means of $\mathbf{new}\, c.\, \_$. $\mathbf{!}e$ evaluates its argument to a reference cell and returns the value associated to it. Similarly, the evaluation of $e_1 := e_2$ first evaluates e_1 to a store location c, computes the value v of e_2 and replaces the former contents of c with v. The returned value is $\langle\rangle$.

We conclude our discussion about evaluation with a few words about the interaction of references and polymorphism. The question is subtle and has received great attention in the literature [Tof90, Har94, LW91]. Consider for example the following *MLR* expression.

$$\begin{array}{l} \mathbf{letname}\, f \;=\; \mathbf{ref}\, (\mathbf{lam}\, x.\, x) \\ \;\mathbf{in}\; f := \mathbf{lam}\, x.\, \mathbf{s}\; x; \\ \qquad \mathbf{!}f\; \langle\rangle \end{array}$$

At first sight, this expression allocates a cell and initializes it with the identity function, which has polymorphic type $\tau \to \tau$ for any τ. In the body of **letname**, we first update it to the successor function, of type $\mathbf{nat} \to \mathbf{nat}$, and then apply it to $\langle\rangle$, of type $\mathbf{1}$. Clearly, something is wrong, but the typing rules of *MLR* accept the program above as a correct expression of type

Expressions

(No **ev_x**)

$$\frac{S \triangleright K \vdash \textbf{return } \mathbf{z} \hookrightarrow a}{S \triangleright K \vdash \textbf{eval } \mathbf{z} \hookrightarrow a}\,\mathbf{ev\_z} \qquad \frac{S \triangleright K, \lambda x.\, \textbf{return } \mathbf{s}\ x \vdash \textbf{eval } e \hookrightarrow a}{S \triangleright K \vdash \textbf{eval } \mathbf{s}\ e \hookrightarrow a}\,\mathbf{ev\_s}$$

$$\frac{S \triangleright K, \lambda y.\, \textbf{case}^*\ y \textbf{ of } \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \vdash \textbf{eval } e \hookrightarrow a}{S \triangleright K \vdash \textbf{eval case } e \textbf{ of } \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \hookrightarrow a}\,\mathbf{ev\_case}$$

$$\frac{S \triangleright K \vdash \textbf{return } \langle\rangle \hookrightarrow a}{S \triangleright K \vdash \textbf{eval } \langle\rangle \hookrightarrow a}\,\mathbf{ev\_unit} \qquad \frac{S \triangleright K, \lambda x.\, \langle x, e_2\rangle^* \vdash \textbf{eval } e_1 \hookrightarrow a}{S \triangleright K \vdash \textbf{eval } \langle e_1, e_2\rangle \hookrightarrow a}\,\mathbf{ev\_pair}$$

$$\frac{S \triangleright K, \lambda x.\, \textbf{fst}^*\ x \vdash \textbf{eval } e \hookrightarrow a}{S \triangleright K \vdash \textbf{eval fst } e \hookrightarrow a}\,\mathbf{ev\_fst} \qquad \frac{S \triangleright K, \lambda x.\, \textbf{snd}^*\ x \vdash \textbf{eval } e \hookrightarrow a}{S \triangleright K \vdash \textbf{eval snd } e \hookrightarrow a}\,\mathbf{ev\_snd}$$

$$\frac{S \triangleright K \vdash \textbf{return lam } x.\, e \hookrightarrow a}{S \triangleright K \vdash \textbf{eval lam } x.\, e \hookrightarrow a}\,\mathbf{ev\_lam} \qquad \frac{S \triangleright K, \lambda x.\, \textbf{app}^*\ x\ e_2 \vdash \textbf{eval } e_1 \hookrightarrow a}{S \triangleright K \vdash \textbf{eval } e_1\ e_2 \hookrightarrow a}\,\mathbf{ev\_app}$$

$$\frac{S \triangleright K, \lambda x.\, \textbf{eval } e_2 \vdash \textbf{eval } e_1 \hookrightarrow a}{S \triangleright K \vdash \textbf{eval letval } x = e_1 \textbf{ in } e_2 \hookrightarrow a}\,\mathbf{ev\_letval} \qquad \frac{S \triangleright K \vdash \textbf{eval } [e_1/x]e_2 \hookrightarrow a}{S \triangleright K \vdash \textbf{eval letname } x = e_1 \textbf{ in } e_2 \hookrightarrow a}\,\mathbf{ev\_letname}$$

$$\frac{S \triangleright K \vdash \textbf{eval } [\textbf{fix } x.\, e/x]e \hookrightarrow a}{S \triangleright K \vdash \textbf{eval fix } x.\, e \hookrightarrow a}\,\mathbf{ev\_fix}$$

....

$$\frac{S \triangleright K \vdash \textbf{return } c \hookrightarrow a}{S \triangleright K \vdash \textbf{eval } c \hookrightarrow a}\,\mathbf{ev\_cell}$$

$$\frac{S \triangleright K, \lambda x.\, \textbf{ref}^*\ x \vdash \textbf{eval } e \hookrightarrow a}{S \triangleright K \vdash \textbf{eval ref } e \hookrightarrow a}\,\mathbf{ev\_ref} \qquad \frac{S \triangleright K, \lambda x.\, \textbf{deref}^*\ x \vdash \textbf{eval } e \hookrightarrow a}{S \triangleright K \vdash \textbf{eval } !e \hookrightarrow a}\,\mathbf{ev\_deref}$$

$$\frac{S \triangleright K, \lambda x.\, \textbf{eval } e_2 \vdash \textbf{eval } e_1 \hookrightarrow a}{S \triangleright K \vdash \textbf{eval } e_1; e_2 \hookrightarrow a}\,\mathbf{ev\_seq} \qquad \frac{S \triangleright K, \lambda x.\, x :=_1^* e_2 \vdash \textbf{eval } e_1 \hookrightarrow a}{S \triangleright K \vdash \textbf{eval } e_1 := e_2 \hookrightarrow a}\,\mathbf{ev\_assign}$$

Figure 10: Evaluation in *MLR*, Expressions

1. Is there a flaw in the definition of the static semantics of our language? Fortunately, no. A closer analysis reveals that, since the evaluation of **letname** substitutes **ref** (**lam** $x.\, x$) for every occurrences of f in its body, the expression above reduces to

$$\textbf{ref } (\textbf{lam } x.\, x) := \textbf{lam } x.\, \mathbf{s}\ x;$$
$$(!\textbf{ref } (\textbf{lam } x.\, x))\ \langle\rangle$$

Each occurrence of **ref** (**lam** $x.\, x$) evaluates to a different cell that is typed according to its use. The expression above would not be typable if we had used **letval** in place of **letname**.

Languages with explicit type variables solve the same problem by distinguishing between *ap-*

Values

$$\frac{}{S \rhd \mathbf{init} \vdash \mathbf{return}\ v \hookrightarrow (S, v)}\ \mathbf{ev\_init} \qquad \frac{S \rhd K \vdash [v/x]i \hookrightarrow a}{S \rhd K, \lambda x.\, i \vdash \mathbf{return}\ v \hookrightarrow a}\ \mathbf{ev\_cont}$$

Auxiliary instructions

$$\frac{S \rhd K \vdash \mathbf{eval}\ e_1 \hookrightarrow a}{S \rhd K \vdash \mathbf{case}^*\ \mathbf{z}\ \mathbf{of}\ \mathbf{z}\ \Rightarrow\ e_1 \mid \mathbf{s}\ x\ \Rightarrow\ e_2 \hookrightarrow a}\ \mathbf{ev\_case_1^*}$$

$$\frac{S \rhd K \vdash \mathbf{eval}\ [v/x]e_2 \hookrightarrow a}{S \rhd K \vdash \mathbf{case}^*\ \mathbf{s}\ v\ \mathbf{of}\ \mathbf{z}\ \Rightarrow\ e_1 \mid \mathbf{s}\ x\ \Rightarrow\ e_2 \hookrightarrow a}\ \mathbf{ev\_case_2^*}$$

$$\frac{S \rhd K, \lambda x.\, \mathbf{return}\ \langle v, x\rangle \vdash \mathbf{eval}\ e \hookrightarrow a}{S \rhd K \vdash \langle v, e\rangle^* \hookrightarrow a}\ \mathbf{ev\_pair^*}$$

$$\frac{S \rhd K \vdash \mathbf{return}\ v_1 \hookrightarrow a}{S \rhd K \vdash \mathbf{fst}^*\ \langle v_1, v_2\rangle \hookrightarrow a}\ \mathbf{ev\_fst^*} \qquad \frac{S \rhd K \vdash \mathbf{return}\ v_2 \hookrightarrow a}{S \rhd K \vdash \mathbf{snd}^*\ \langle v_1, v_2\rangle \hookrightarrow a}\ \mathbf{ev\_snd^*}$$

$$\frac{S \rhd K, \lambda x.\, \mathbf{eval}\ e_1 \vdash \mathbf{eval}\ e_2 \hookrightarrow a}{S \rhd K \vdash \mathbf{app}^*\ (\mathbf{lam}\ x.\, e_1)\ e_2 \hookrightarrow a}\ \mathbf{ev\_app^*}$$

$$\frac{(S, c = v) \rhd K \vdash \mathbf{return}\ c \hookrightarrow a}{S \rhd K \vdash \mathbf{ref}^*\ v \hookrightarrow \mathbf{new}\ c.\, a}\ \mathbf{ev\_ref^*} \qquad \frac{(S, c = v, S') \rhd K \vdash \mathbf{return}\ v \hookrightarrow a}{(S, c = v, S') \rhd K \vdash \mathbf{deref}^*\ c \hookrightarrow a}\ \mathbf{ev\_deref^*}$$

$$\frac{S \rhd K, \lambda x.\, c :=_2^* x \vdash \mathbf{eval}\ e \hookrightarrow a}{S \rhd K \vdash c :=_1^* e \hookrightarrow a}\ \mathbf{ev\_assign_1^*} \qquad \frac{(S, c = v, S') \rhd K \vdash \mathbf{return}\ \langle\rangle \hookrightarrow a}{(S, c = v', S') \rhd K \vdash c :=_2^* v \hookrightarrow a}\ \mathbf{ev\_assign_2^*}$$

Figure 11: Evaluation in *MLR*, Values and Auxiliary Instructions

plicative and *imperative* type variables in order to avoid problems such as the above [MTHM96, Har94]. Restricting polymorphism to values has also been proposed as a solution to this problem [Tof90] and has been adopted in the current definition of *Standard ML* [MTHM96]. This language offers only one form of **let**, but it takes different courses of action depending on whether it defines a value or an arbitrary expression. Our treatment is slightly more general since it makes the call-by-name semantics of **letname** directly available: for example, the above expression does not type-check in *SML*.

Type Preservation

We conclude this section with the statement of the type preservation theorem for *MLR* and of the lemmas it depends on. For reasons of space, we will not formalize the proof of these results in *LLF*. The interested reader can find an encoding of this proof in our linear logical framework in [Cer96].

The type preservation theorem states that the type of an expression does not change as the result of evaluation. The proof of the type preservation theorem relies on a number of auxiliary lemmas. The first is *weakening*: whenever an expression is well-typed in a given context and store, it remains well-typed by considering further assumptions and additional cells.

Lemma 3.1 (*Weakening*)

i. *If* $\mathcal{T} :: \Omega;\Gamma \vdash^{e} e : \tau$, *then* $\Omega,\Omega';\Gamma,\Gamma' \vdash^{e} e : \tau$;

ii. *If* $\mathcal{T} :: \Omega;\Gamma \vdash^{i} i : \tau$, *then* $\Omega,\Omega';\Gamma,\Gamma' \vdash^{i} i : \tau$;

iii. *If* $\mathcal{T} :: \Omega \vdash^{K} K : \tau_1 \Rightarrow \tau_2$, *then* $\Omega,\Omega' \vdash^{K} K : \tau_1 \Rightarrow \tau_2$;

iv. *If* $\mathcal{T} :: \Omega \vdash^{S} S : \overline{\Omega}$, *then* $\Omega,\Omega' \vdash^{S} S : \overline{\Omega}$;

v. *If* $\mathcal{T} :: \Omega \vdash^{a} a : \tau$, *then* $\Omega,\Omega' \vdash^{a} a : \tau$.

Proof.

We proceed by induction on the structure of $\mathcal{T}$. The parts of this lemma should be proved in the order they are presented. ☑

The second auxiliary property we need is the *substitution lemma*: it states that free variables in a well-typed expression can be substituted for expressions of the same type and the result will be well-typed.

Lemma 3.2 (*Substitution*)

i. *If* $\mathcal{T} :: \Omega;\Gamma,x{:}\tau' \vdash^{e} e : \tau$ *and* $\Omega;\Gamma \vdash^{e} e' : \tau'$, *then* $\Omega;\Gamma \vdash^{e} [e'/x]e : \tau$;

ii. *If* $\mathcal{T} :: \Omega;\Gamma,x{:}\tau' \vdash^{i} i : \tau$ *and* $\Omega;\Gamma \vdash^{e} e' : \tau'$, *then* $\Omega;\Gamma \vdash^{i} [e'/x]i : \tau$.

Proof.

We proceed by induction on the structure of $\mathcal{T}$. ☑

The next result is mainly technical as it generalizes the typing rules for the store to items possibly different from the rightmost.

Lemma 3.3 (*Inversion on the store typing*)

i. *If* $\Omega \vdash^{S} (S', c = v, S'') : \overline{\Omega}$, *then* $\overline{\Omega} = \Omega', c{:}\tau, \Omega''$ *with* $\Omega \vdash^{S} S' : \Omega'$, $\Omega;\cdot \vdash^{e} v : \tau$ *and* $\Omega \vdash^{S} S'' : \Omega''$;

ii. *If* $\Omega \vdash^{S} S' : \Omega'$, $\Omega;\cdot \vdash^{e} v : \tau$ *and* $\Omega \vdash^{S} S'' : \Omega''$, *then* $\Omega \vdash^{S} (S', c = v, S'') : (\Omega', c{:}\tau, \Omega'')$. □

As in the functional case, type preservation ensures that the type of an expression is identical to the type of its value. Intermediate evaluation steps require us to take into account arbitrary continuations and stores. We have the following generalization.

Theorem 3.4 (*Type preservation*)

If $S \triangleright K \vdash i \hookrightarrow a$ *with* $\Omega; \cdot \vdash^i i : \tau$, $\Omega \vdash^K K : \tau \Rightarrow \tau'$ *and* $\Omega \vdash^S S : \Omega$, *then* $\Omega \vdash^a a : \tau'$.

Proof.

We proceed by induction on the structure of a derivation of the evaluation judgment and inversion on the derivations of the typing judgments. ☑

The type preservation result is formalized as follows at the top level of evaluation.

Corollary 3.5 (*Type preservation*)

If $\cdot \triangleright \textbf{init} \vdash \textbf{eval}\, e \hookrightarrow a$ *with* $\cdot; \cdot \vdash^e e : \tau$, *then* $\cdot \vdash^a a : \tau$. □

3.3 Representation in *LLF*

In this subsection, we give an *LLF* representation of the syntax of *MLR*, of its static and dynamic semantics and show how to exploit the resulting encoding of computations. The representation we propose is a natural extension of the *LF* code for *Mini-ML* found in the literature [MP91]. In particular, it retains its structure, its elegance, and the ease of proving its adequacy with respect to the informal presentation we just concluded. We describe the main issues in the representation by displaying fragments of the code and a limited number of adequacy statements. A complete treatment can be found in Appendix A. It is interesting to compare the result of our encoding with similar endeavors in the literature.

VanInwegen used the *HOL* theorem prover [GM93] to verify properties about a substantial portion of *Standard ML* [Van96]. She adopted a brute-force approach to the meta-representation problem, encoding, for example, contexts as terms. This choice resulted in a complex representation, in the absence of adequacy results, and only in the partial achievement of the main goal of this endeavor: a formal proof of type preservation for that language. Although on a much simpler fragment, our use of higher-order abstract syntax, of parametric and hypothetical judgments, and of the linear features of *LLF* avoids these difficulties completely.

Chirimar used *Forum* [Mil94] to represent a language similar to *MLR* with the addition of exceptions and continuations [Chi95], but without any emphasis on typing. He took advantage of the higher-order nature of *Forum* and of its linear constructs. The resulting program is as elegant as our code and is proved adequate with respect to the informal specification of the object language. The absence of proof-terms in *Forum* prevents the direct manipulation of object-level derivations and no attempt is made to use that meta-language to investigate meta-theoretic properties such as type preservation.

Syntax

The representation of the syntactic level of *MLR* is based on higher-order abstract syntax and does not require the expressive power of the linear constructs of *LLF*. It lies therefore in the *LF* fragment of this language.

As normally done in *LF*, every syntactic category of the object language is mapped to a distinguished base type. The type families necessary to encode the syntactic categories of *MLR* are given by the following declarations:

```
exp   : type.        cell   : type.
tp    : type.        store  : type.
instr : type.        cv     : type.
cont  : type.        answer : type.
```

The four declarations on the right encode expressions, instructions, types and continuations. The four on the left are needed to represent the imperative features of *MLR* programs. `cell` corresponds to the lexical category of memory cells. `cv` and `store` will be used to represent the store. Finally, `answer` deals with final answers.

We encode the abstract syntax of *MLR* expressions, as described in the grammar of Section 3.3, by means of the representation function $\ulcorner \_ \urcorner$. This function maps every production to an *LLF* object constant that, when applied to the representation of the subexpressions that it relates, yields an object that has type `exp`. The function $\ulcorner \_ \urcorner$ is inductively defined on the left-hand side of the table below (we have separated out the treatment of the imperative constructs); its right-hand side gives the type of the constants used in the encoding.

$\ulcorner x \urcorner$	$=$	x		
$\ulcorner \mathbf{z} \urcorner$	$=$	`z`	`z`	`: exp.`
$\ulcorner \mathbf{s}\ e \urcorner$	$=$	`s` $\ulcorner e \urcorner$	`s`	`: exp -> exp.`
$\ulcorner$ **case** e $\urcorner$	$=$	`case` $\ulcorner e \urcorner$	`case`	`:   exp`
of z $\Rightarrow e_1$		$\ulcorner e_1 \urcorner$		`-> exp`
$\mid$ **s** $x \Rightarrow e_2$		`([`x`:exp]`$\ulcorner e_2 \urcorner$`)`		`-> (exp -> exp) -> exp.`
$\ulcorner \langle\rangle \urcorner$	$=$	`unit`	`unit`	`: exp.`
$\ulcorner \langle e_1, e_2 \rangle \urcorner$	$=$	`pair` $\ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner$	`pair`	`: exp -> exp -> exp.`
$\ulcorner \mathbf{fst}\ e \urcorner$	$=$	`fst` $\ulcorner e \urcorner$	`fst`	`: exp -> exp.`
$\ulcorner \mathbf{snd}\ e \urcorner$	$=$	`snd` $\ulcorner e \urcorner$	`snd`	`: exp -> exp.`
$\ulcorner \mathbf{lam}\ x.\ e \urcorner$	$=$	`lam ([`x`:exp]`$\ulcorner e \urcorner$`)`	`lam`	`: (exp -> exp) -> exp.`
$\ulcorner e_1\ e_2 \urcorner$	$=$	`app` $\ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner$	`app`	`: exp -> exp -> exp.`
$\ulcorner \mathbf{letval}\ x = e_1$ $\urcorner$	$=$	`letval` $\ulcorner e_1 \urcorner$	`letval`	`:   exp`
in e_2		`([`x`:exp]`$\ulcorner e_2 \urcorner$`)`		`-> (exp -> exp) -> exp.`
$\ulcorner \mathbf{letname}\ x = e_1$ $\urcorner$	$=$	`letname` $\ulcorner e_1 \urcorner$	`letname`	`:   exp`
in e_2		`([`x`:exp]`$\ulcorner e_2 \urcorner$`)`		`-> (exp -> exp) -> exp.`
$\ulcorner \mathbf{fix}\ x.\ e \urcorner$	$=$	`fix ([`x`:exp]`$\ulcorner e \urcorner$`)`	`fix`	`: (exp -> exp) -> exp.`
$\ulcorner c \urcorner$	$=$	`rf` c	`rf`	`: cell -> exp.`
$\ulcorner \mathbf{ref}\ e \urcorner$	$=$	`ref` $\ulcorner e \urcorner$	`ref`	`: exp -> exp.`
$\ulcorner \mathbf{!} e \urcorner$	$=$	`!`$\ulcorner e \urcorner$	`!`	`: exp -> exp.`
$\ulcorner e_1 := e_2 \urcorner$	$=$	`assign` $\ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner$	`assign`	`: exp -> exp -> exp.`
$\ulcorner e_1 ; e_2 \urcorner$	$=$	`seq` $\ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner$	`seq`	`: exp -> exp -> exp.`

The representation of most expressions reflects directly the abstract syntax of *MLR*. We take advantage of higher-order abstract syntax in the representation of cells, variables and binding constructs of *MLR*. Variables are encoded as *LLF* variables (of type `exp`). The fact that an object-level construct binds a variables x in a sub-expression e is then modeled by using the λ-abstraction of *LLF* in order to bind x in $\ulcorner e \urcorner$. Cells appear as hypotheses $c : \texttt{cell}$ in the context of *LLF*, similarly to free variables. Their representation as expressions is mediated by the constant `rf`, that maps entities of type `cell` to objects of type `exp`.

As an example, consider again the following *MLR* expression from the previous section:

$$\begin{array}{l}\textbf{ref}\,(\textbf{lam}\,x.\,x) := \textbf{lam}\,x.\,\textbf{s}\,x;\\ (\textbf{!ref}\,(\textbf{lam}\,x.\,x))\,\langle\rangle\end{array}$$

It is represented by the following *LLF* term of type `exp`:

```
letname
  (ref (lam ([x] x)))
  ([f] (seq
        (assign f (lam ([x] (s x))))
        (app (! f) unit))))
```

The faithfulness of this representation with respect to the object level syntax of expressions consists of a number of properties that we summarize in the following adequacy theorem:

Theorem 3.6 (*Adequacy of the representation of MLR expressions*)

There is a compositional bijection $\ulcorner\_\urcorner$ *between MLR expression with free variables among* $x_1, \ldots, x_n$ *and cells* $c_1, \ldots, c_m$*, and canonical LLF objects* M *such that the judgment*

$$x_1 : \texttt{exp}, \ldots, x_n : \texttt{exp}, c_1 : \texttt{cell}, \ldots, c_m : \texttt{cell} \vdash_\Sigma M \Uparrow \texttt{exp}$$

is derivable. □

Compositionality in this statement means that the representation function commutes with substitution, i.e., that for every *MLR* expression e and e', $\ulcorner [e'/x]e \urcorner = [\ulcorner e' \urcorner / x]\ulcorner e \urcorner$. It confirms the correct application of higher-order abstract syntax in our encoding. Note that compositionality is not needed for cells since they are never subject to substitution.

Due to the complexity of our object language, we do not display the simple but long and somewhat tedious inductive proof of this statement. The interested reader is referred to [Cer96] for a full treatment. The techniques used in order to prove adequacy theorems for *LLF* encodings naturally extends the methods successfully applied for years in the more restricted setting of *LF*. In particular, they retain their simplicity in our richer applicative arena. This contrasts with other proposals, e.g., the treatment of linearity in *LF* itself [Pfe94b], where adequacy theorems have complex proofs even for simple object languages.

Types, instructions and continuations are represented in a similar way. The *LLF* declarations for the constants needed in their encoding can be found in Appendix A. We omit displaying the statements of the respective adequacy theorems since they do not introduce new concepts. They can be found in [Cer96].

The formalization of *MLR* we propose relies on two distinct representations of the store. In most circumstances, the store is efficiently encoded in a distributed fashion in the context of *LLF*. We refer to this encoding as the *internal* representation. When an answer is eventually produced, the store is given an *external* representation as an *LLF* term, and packaged with the final value. This latter encoding is defined as follows:

$$\begin{array}{rcll} \ulcorner . \urcorner & = & \texttt{estore} & \texttt{estore : store} \\ \ulcorner S, c = v \urcorner & = & \texttt{with } \ulcorner S \urcorner \texttt{ (holds } c \ulcorner v \urcorner \texttt{)} & \texttt{with \ \ : store -> cv -> store.} \\ & & & \texttt{holds \ : cell -> exp -> cv.} \end{array}$$

The representation of answers directly expresses the grammatical rules:

$$\begin{array}{rcll} \ulcorner (S, v) \urcorner & = & \texttt{close } \ulcorner S \urcorner \ulcorner v \urcorner & \texttt{close : store -> exp -> answer.} \\ \ulcorner \mathbf{new}\, c.\, a \urcorner & = & \texttt{new ([}c\texttt{:cell]}\ulcorner a \urcorner\texttt{)} & \texttt{new \ \ : (cell -> answer) -> answer.} \end{array}$$

The declarations for these constants are repeated in Appendix A. The adequacy theorems that link them to the syntax of *MLR* are reported in [Cer96].

Static Semantics

On the basis of the above encoding of the syntax of *MLR*, we will now describe the meta-representation of the static semantics of this language.

As for syntax, the representation of the static semantics of *MLR* does not rely on the linear features of *LLF*. The resulting code lies therefore in the *Elf* fragment of our logical framework. We have the following declarations for the type families that model the various typing judgments presented in Section 3.2:

```
tpe : exp -> tp -> type.
tpi : instr -> tp -> type.
tpK : cont -> tp -> tp -> type.

tpc : cell -> tp -> type.
tpS : store -> type.
tpa : answer -> tp -> type.
```

Again we have separated out the declarations that suffice for the functional core of *MLR* from the type families that are required to handle the imperative aspects of this language. The first three represent the typing judgments for expressions, instructions and continuations, while `tpS` and `tpa` encode respectively the store and answer typing judgments, and `tpc` records the type of individual cells in the store.

We illustrate the representation of the static semantics of *MLR* by displaying how to encode a typing derivation for expressions. The remaining typing judgments are treated similarly and the resulting *LLF* declarations are presented in Appendix A.

In Section 3.2, we denoted the fact that an expression e has type τ assuming the types given in Γ for its free variables and the types given in Ω for its reference cells as the hypothetical judgment $\Omega;\Gamma \vdash^e e : \tau$. We represent the schematic form of this judgment by means of the *LLF* type family `tpe`. This family accepts two parameters: the representation of an expression and the representation of a type. Therefore, the instance above will be encoded as the *LLF* base type `tpe` $\ulcorner e \urcorner$ $\ulcorner \tau \urcorner$. The context Γ is taken into consideration only when checking that this term is indeed derivable in *LLF*. Then, we will encode each pair $x_i\!:\!\tau_i$ in Γ by means of an *LLF* hypothesis

$$t_i : \texttt{tpe}\ x_i\ \ulcorner \tau_i \urcorner.$$

where the free variable x_i is declared as an expression ($x_i\!:\!\texttt{exp}$). Similarly, we encode every item $c_j\!:\!\tau'_j$ in this context as the (intuitionistic) assumption

$$t'_j : \texttt{tpc}\ c_j\ \ulcorner \tau'_j \urcorner$$

in the context of *LLF*, where c_j is declared as a cell ($c_j\!:\!\texttt{cell}$). Note that `tpc` only serves the purpose of making typing assumptions for cells. We write $\ulcorner\Gamma\urcorner$ and $\ulcorner\Omega\urcorner$ for the encoding we just outlined for the context Γ and the store context Ω, respectively.

The inference rules defining the derivability of the typing judgment for *MLR* are encoded according to the technique presented in Section 3.1. We consider two rules as additional examples, the remaining declarations can be found in Appendix A. Rule **tpe_z** associates the type **nat** to the numeral **z**. We represent it by means of the *LF* constant `tpe_z` that relate `z` to `nat`:

$$\ulcorner \frac{}{\Omega;\Gamma \vdash^e \mathbf{z} : \mathbf{nat}}\,\textbf{tpe\_z} \urcorner \quad = \quad \texttt{tpe\_z : tpe z nat.}$$

Rule **tpe_case** specifies how to type-check a conditional expression. We repeat it from Figure 8.

$$\frac{\Omega;\Gamma \vdash^e e : \mathbf{nat} \quad \Omega;\Gamma \vdash^e e_1 : \tau \quad \Omega;\Gamma, x\!:\!\mathbf{nat} \vdash^e e_2 : \tau}{\Omega;\Gamma \vdash^e \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 : \tau}\,\textbf{tpe\_case}$$

This rule has multiple premisses. It is hypothetical because the rightmost premiss inserts the assumption $x\!:\!\mathbf{nat}$ into the context. It is also parametric since the variable x is bound in the case construct, but it appears as a new symbol both in the added hypothesis and in the expression e_2 type-checked by the rightmost premiss. We represent this rule by means of the *LLF* constant `tpe_case` and encode its structure in the associated type. We have the declaration:

```
tpe_case :   tpe E nat
          -> tpe E1 T
          -> ({x:exp} tpe x nat -> tpe (E2 x) T)
          -> tpe (case E E1 ([x:exp] E2 x)) T.
```

Notice the quantification over `x` and the embedded implication with antecedent `tpe x nat` in the encoding of the third premiss. In this declaration, the *LLF* variables `E`, `E1`, `E2` and `T` correspond to the schematic variables e, e_1, e_2 and τ respectively. They are implicitly quantified at the front of the declaration.

It is worth noticing that there is no declaration in correspondence of rule **tpe_x**:

$$\frac{}{\Omega; \Gamma, x{:}\tau \vdash^e x : \tau}\ \textbf{tpe\_x}$$

Since assumptions are represented directly in the context of *LLF*, a judgment of the form `tpe x` T, where T is the representation of some concrete type τ, will be validated by accessing the context of *LLF* rather than the signature, and succeed precisely when t_x : `tpe x` T appears in it as an assumption. Similar considerations hold for reference cells.

We have now the means for representing derivations of expression typing judgments. The adequacy theorem below ensures that whenever $\mathcal{T}$ is a (valid) derivation for the *MLR* judgment $\Omega; \Gamma \vdash^e e : \tau$, then $\ulcorner \mathcal{T} \urcorner$ is a canonical inhabitant of the *LLF* type `tpe` $\ulcorner e \urcorner$ $\ulcorner \tau \urcorner$ with respect to the proper encoding of Γ and Ω, and vice versa.

Theorem 3.7 (*Adequacy of the representation of MLR expression typing*)

Given an MLR expression e and a type τ, there is a compositional bijection $\ulcorner \_ \urcorner$ between derivations of

$$\Omega; \Gamma \vdash^e e : \tau$$

and LLF objects M such that

$$\ulcorner \Omega \urcorner, \ulcorner \Gamma \urcorner \vdash_\Sigma M \Uparrow \texttt{tpe}\ \ulcorner e \urcorner\ \ulcorner \tau \urcorner$$

is derivable. □

Dynamic Semantics

Differently from the case of the syntax and of the static semantic, the representation of evaluation relies heavily on the linear features of *LLF*. It is based on the following four type families:

```
ev       : cont -> instr -> answer -> type.

contains : cell -> exp -> type.
collect  : store -> type.
read     : cell -> exp -> type.
```

which we will describe in turn.

Assuming the appropriate representation functions $\ulcorner \_ \urcorner$ for continuations, instructions and answers, we model the continuation-based judgment $S \rhd K \vdash i \hookrightarrow a$ as the *LLF* base type `ev` $\ulcorner K \urcorner$ $\ulcorner i \urcorner$ $\ulcorner a \urcorner$. The store S mentioned by this judgment is represented in a distributed fashion in the context of *LLF*. Each item $c = v$ in S is modeled by two assumptions: first of all, we need to declare c as a cell and we do so by means of the assumption c : `cell`, second, we represent the fact that the current contents of c is v by a *linear* hypothesis of the form $h \mathbin{\hat{:}}$ `contains` c $\ulcorner v \urcorner$. The first assumption should clearly be intuitionistic since c may be mentioned many times in K, i, a and S. In contrast, the second must be linear since assignment updates the value associated with a cell destructively. If h were an intuitionistic hypothesis, we would have no means of prohibiting the old value from being accessed.

An anticipated presentation of the adequacy theorem for the representation of evaluation might be helpful in gaining familiarity with the internal representation of the store.

Theorem 3.8 (*Adequacy of the representation of MLR evaluation*)

Given an MLR continuation K, an instruction i, a store $S = (c_1 = v_1, \ldots, c_n = v_n)$ and an answer a, where K, i, S and a are closed except for the possible presence of free cells, there is a compositional bijection $\ulcorner \_ \urcorner$ between derivations of

$$S \rhd K \vdash i \hookrightarrow a$$

and LLF objects M such that

$$\begin{bmatrix} c_1 : \texttt{cell}, & \ldots, & c_n : \texttt{cell}, \\ h_1 \mathbin{\hat{:}} \texttt{contains}\ c_1\ \ulcorner v_1 \urcorner, & \ldots, & h_n \mathbin{\hat{:}} \texttt{contains}\ c_n\ \ulcorner v_n \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \texttt{ev}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner a \urcorner$$

is derivable. □

Four rules in the deductive system for continuation-based evaluation presented in Figures 10–11 access the store directly. We will illustrate the use of the linear features of *LLF* on the representation of these rules. The remaining rules are encoded according to a similar technique, but the linear aspects of our meta-language are not required. The complete code is displayed in Appendix A.

Rule **ev_ref***, repeated below on the left, creates a new location c in the store and initializes it with the argument v of **ref***. Its representation on the right models these actions on the context of *LLF*: the new cell is intuitionistically assumed when processing the dependent type `{c:cell}`, while the resolution of the embedded linear implication has the effect of asserting `contains c V` in the linear part of the context. The linear arrow between the antecedent and the consequent of this clause allows the access of the linear hypotheses in the recursive call. Notice how the newly created cell `c` is bound in the final answer.

$$\ulcorner \dfrac{S, c = v \rhd K \vdash \textbf{return}\ c \hookrightarrow a}{S \rhd K \vdash \textbf{ref}^*\ v \hookrightarrow \textbf{new}\ c.\, a}\ \textsf{ev\_ref}^* \urcorner =$$

```
ev_ref* : ({c:cell} contains c V
                 -o ev K (return (rf c)) (A c))
             -o ev K (ref* V) (new ([c:cell] A c)).
```

Of the three rules that realize assignment in *MLR*, only $\textbf{ev\_assign}_2^*$ accesses the store. The declaration `ev_assign*2` below mimics the destructive update of the contents of the cell c (written `C` in the clause) in the store in two steps. First the old value is retrieved by `contains C V'`.

Since it appears as a linear assumption, accessing it causes its removal from the linear context of *LLF*. Since the other antecedent of this clause is reached through a multiplicative connective, the remaining linear hypotheses will be passed to it. This term inserts the new value v (i.e. `V`) of c in the representation of the store by means of the antecedent `contains C V` of the embedded linear implication.

$$\ulcorner \frac{S, c = v, S' \rhd K \vdash \textbf{return}\ \langle\rangle \hookrightarrow a}{S, c = v', S' \rhd K \vdash c :=_2^* v \hookrightarrow a}\textbf{ev\_assign}_2^* \urcorner = \left[\begin{array}{l}\texttt{ev\_assign*2 : contains C V'} \\ \qquad\qquad\texttt{-o (contains C V} \\ \qquad\qquad\qquad\texttt{-o ev K (return unit) A)} \\ \qquad\qquad\texttt{-o ev K (assign*2 (rf C) V) A.}\end{array}\right.$$

Dereferencing a cell c (rule **ev_deref***) can be modeled in *LLF* in two manners. The rule below illustrates a general technique that make uses of additives. Using additive conjunction has the effect of duplicating the linear context and giving a copy to each of its conjuncts. In this manner, the evaluation, modeled by the second conjunct, can proceed with the original contents of the store.

$$\ulcorner \frac{S, c = v, S' \rhd K \vdash \textbf{return}\ v \hookrightarrow a}{S, c = v, S' \rhd K \vdash \textbf{deref}^*\ c \hookrightarrow a}\textbf{ev\_deref}^* \urcorner = \left[\begin{array}{l}\texttt{ev\_deref* :\quad (read C V} \\ \qquad\qquad\quad\texttt{\& ev K (return V) A)} \\ \qquad\qquad\texttt{-o ev K (deref* (rf C)) A.}\end{array}\right.$$

The conjunct `read C V` looks up its image of the linear context in search of the assumption `contains C V` and relies on the additive unit of *LLF*, written `<T>` to discard the rest. The definition of `read` consists of the following single clause:

```
read_val :   contains C V
          -o <T>
          -o read C V.
```

Rule **ev_deref*** could have been alternatively modeled similarly to assignment by first accessing the linear assumption `contains C V` directly. In order to balance its consequent removal from the linear context of *LLF*, this same assumption should be re-entered in the context before returning the value `V`. We would have the following declaration:

```
ev_deref*' :   contains C V
            -o (contains C V -o ev K (return V) A)
            -o ev K (deref* (rf C)) A.
```

Finally, rule **ev_init** pairs up the store and the final value in order to produce the answer. We model this behavior by means of the auxiliary procedure `collect` which translates the internal representation of the store, as linear *LLF* assumptions, to its external representation, as an object of type `store`.

$$\ulcorner \frac{}{S \rhd \textbf{init} \vdash \textbf{return}\ v \hookrightarrow (S, v)}\textbf{ev\_init} \urcorner = \left[\begin{array}{l}\texttt{ev\_init :\quad collect S} \\ \qquad\qquad\texttt{-o ev init (return V) (close S V).}\end{array}\right.$$

The code for `collect` is displayed below.

```
col_empty : collect estore.
col_cv    :   contains C V
            -o collect S
            -o collect (with S (holds C V)).
```

Since the use of multiplicatives removes the assumptions `contains C V` as they are retrieved, each recursive access to `collect` adds a different item to the external representation of the store. Clause `col_empty` is provable only when the linear part of the context of *LLF* is empty, and therefore only when the complete store of *MLR* has been externalized.

The effectiveness of the representation we just illustrated relied heavily on the ability to remove objects from the context of *LLF*. The role of the linear features of our formalism is crucial for this purpose. Using *LF* on this problem would have produced awkward encodings with prohibitive consequence for the development of the meta-theory of *MLR* [Cer96]: a first alternative would have relied entirely on the external representation of the store, implementing all the operations required to access and modify it explicitly. A second alternative would be to proceed as we did, with the tedious addition of declarations aimed at checking the linearity of the resulting derivations a posteriori.

A derivation $\mathcal{E}$ for an evaluation judgment $S \triangleright K \vdash i \hookrightarrow a$ is a trace of the computation that an continuation-based *MLR* interpreter performs when evaluating the instruction i and the continuation K to the final answer K with respect to the store contents S. According to the above adequacy theorem, such derivations are faithfully represented by the terms M inhabiting the *LLF* type encoding this judgment. We conclude this section by illustrating how to take advantage of this internal representation of *MLR* computations. We only give a small example here—more interesting examples such as the proof of type preservation, or a cut elimination procedure for classical linear logic can be found in [Cer96]. Specifically, we will give *LLF* declarations that permit counting the number of reference cells dynamically allocated during the evaluation.

In order to achieve this purpose, we first give the following declarations for natural numbers:

```
num : type.
zero : num.
succ : num -> num.
```

The counting judgment relates a *MLR* computation to the number of cells it allocates. It is represented by the following type family

```
count : ev K I A -> num -> type.
```

We implement the counting procedure in *LLF* by unfolding the representation of an *MLR* computation. We ignore the steps that do not allocate memory cells, but increment by one the counter every time rule **ev_ref*** is applied. We show three declarations, corresponding to the initialization step performed by rule **ev_init**, the allocation of a new cell by rule **ev_ref***, and one of the numerous cases where nothing happens (rule **ev_z**):

```
cnt_init :   count (ev_init C) zero.
cnt_ref* :   ({c:cell}{d:contains c V} count (C c d) N)
          -> count (ev_ref* ([c:cell] [d:contains c V] C c d)) (succ N).
cnt_z    :   count C N
          -> count (ev_z C) N.
```

We conclude this section by displaying the *LLF* base type that ascertains that the *MLR* expression

$$\begin{array}{l}\textbf{letname}\ f\ =\ \textbf{ref}\ (\textbf{lam}\ x.\ x)\\ \ \textbf{in}\ f := \textbf{lam}\ x.\ \textbf{s}\ x;\\ \quad \textbf{!}f\ \langle\rangle\end{array}$$

allocates two cells. The concrete syntax of this term is as follows, where the second argument is the representation of the evaluation of the above expression:

```
count
 (ev_letname ^
  (ev_seq ^
   (ev_assign ^
    (ev_ref ^
     (ev_lam ^
      (ev_cont ^
       (ev_ref* ^
        ([c1:cell] [Cn1^contains c1 (lam [x:exp] x)] ev_cont ^
         (ev_assign*1 ^
          (ev_lam ^
           (ev_cont ^
            (ev_assign*2 ^ Cn1 ^
             ([Cn1'^contains c1 (lam [x:exp] s x)] ev_cont ^
              (ev_app ^
               (ev_deref ^
                (ev_ref ^
                 (ev_lam ^
                  (ev_cont ^
                   (ev_ref* ^
                    ([c2:cell] [Cn2^contains c2 (lam [x:exp] x)] ev_cont ^
                     (ev_deref* ^
                      (read_val ^ () ^ Cn2 ,
                       ev_cont ^
                        (ev_app* ^
                         (ev_unit ^
                          (ev_cont ^
                           (ev_unit ^
                            (ev_init ^ (col_cv ^ Cn2 ^ (col_cv ^ Cn1' ^ col_empty)))
 )))))))))))))))))))))))))) (succ (succ zero))
```

4 Conclusion and Future Work

In this paper, we have presented the linear logical framework *LLF* as an extension of *LF* with internal support for the representation of state-based problems. We have demonstrated its expressive power by providing a usable representation of the syntax and the semantics of an imperative variant of the functional programming language *Mini-ML*; space reasons prevented us from extending this encoding to aspects of the meta-theory of this language, such as a proof of its type preservation property [Cer96]. Additional substantial case studies we have completed include the formalization of a proof of cut elimination for classical linear logic, translations between minimal linear natural deduction and sequent calculus, as well as a number of puzzles and solitaires. The interested reader may access them on the World-Wide Web at [CP] or in [Cer96].

The representation language of *LLF*, $\lambda^{\Pi\multimap\&\top}$, conservatively extends *LF*'s λ^{Π} with constructs from linear logic. We can think of it as the type theory freely generated from the type constructors $\top$, $\&$, $\multimap$, and Π. This choice of constructors is complete in the sense that they suffice to represent full intuitionistic or classical linear logic. Further, adding any other linear connective as a free type constructor destroys the property that usable canonical forms exist. This property is crucial in the proofs of adequacy theorems for encodings and also for the interpretation of *LLF* as an abstract logic programming language.

The meta-representation methodology of *LLF* extends the judgments-as-types technique adopted in *LF* with a direct way to map state-related constructs and behaviors onto the linear operators of $\lambda^{\Pi\multimap\&\top}$. The resulting representations retain the elegance and immediacy that characterize *LF* encodings, and the ease of proving their adequacy.

LLF maintains the computational nature of *LF* and is amenable to an efficient implementation as a logic programming language in the style of *Elf* [Pfe91, Pfe94a]. The experience gained with linearity in the language *Lolli* [HM94, CHP96] applies directly to our formalism. New aspects, in particular the definition of effective algorithms for performing linear higher-order unification, are under investigation [CP96].

LLF generalizes other formalisms based on linear logic such as *Forum* [Mil94] by making linear objects available for representations, by permitting proof terms and by providing linear types. It is closely related to the system *RLF* of Ishtiaq and Pym [IP96]. Our approach is orthogonal to general logics in the style of *LU* [Gir93].

We plan to release in the near future an interpreter for *LLF* in the style of the current implementation of *Elf*. We are also considering implementing a *WAM*-like compiler for this language, although numerous theoretical issues still need to be settled. We also intend to investigate a generalization of the type constructors $\&$ and $\multimap$ of $\lambda^{\Pi\multimap\&\top}$ to linear Σ and Π types, respectively, although it currently appears that this would greatly complicate the type theory while it is not clear how much would be gained. We would also like to explore the possibility of automatically verifying that a signature implements a meta-proof of a meta-theorem analogously to *schema-checking* for *LF* [Roh97].

References

[Bar80] H. P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics.* North-Holland, 1980.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Laboratory for Foundations of Computer Sciences, University if Edinburgh, 1996.

[CDCV81] Mario Coppo, Maria Dezani-Ciancaglini, and B. Venneri. Functional character of solvable terms. *Zeitschrift für mathematische Logic und Grundlagen der Mathematik*, 27:45–58, 1981.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 13–27. ACM Press, 1986.

[Cer96] Iliano Cervesato. *A Linear Logical Framework.* PhD thesis, Dipartimento di Informatica, Università di Torino, February 1996.

[Chi95] Jawahar Lal Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD thesis, University of Pennsylvania, May 1995.

[CHP96] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, pages 67–81, Leipzig, Germany, March 1996. Springer-Verlag LNAI 1050.

[Chu40] Alonzo Church. A formulation of a simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Coq91] 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.

[CP] Iliano Cervesato and Frank Pfenning. The linear logical framework *LLF*. Accessible on the World-Wide Web as `http://www.cs.cmu.edu/~iliano/LLF/`.

[CP96] Iliano Cervesato and Frank Pfenning. Linear higher-order pre-unification. In D. Galmiche, editor, *Informal Proceedings of the Workshop on Proof Search in Type-Theoretic Language*, New Brunswick, New Jersey, July 1996.

[Dam85] Luis M.M. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1985.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter Rewrite Systems, pages 243–320. MIT Press, 1990.

[Dyc92] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3):795–807, September 1992.

[Fel91] Amy Felty. Encoding dependent types in an intuitionistic logic. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*, pages 214–251. Cambridge University Press, 1991.

[Gan80] R.O. Gandy. Proofs of strong normalization. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 457–477. Academic Press, 1980.

[Geu93] Herman Geuvers. *Logics and Type Systems.* PhD thesis, Katholieke Universiteit Nijmegen, 1993.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir93] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a Theorem Proving Environment for Higher-order Logic.* Cambridge University Press, 1993.

[Har94] Robert Harper. A simplified account of polymorphic references. *Information Processing Letter*, 51:201–206, 1994.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HM90] John Hannan and Dale Miller. From operational semantics to abstract machines: Preliminary results. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 323–332, Nice, France, 1990.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

[Hod94] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[IP96] Samin Ishtiaq and David Pym. A relevant analysis of natural deduction, December 1996. Unpublished.

[LW91] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 291–302, Orlando, Florida, January 21-23 1991. ACM press.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994.

[MP91] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[MPP92] Dale Miller, Gordon Plotkin, and David Pym. A relevant analysis of natural deduction. Talk given at the Workshop on Logical Frameworks, Båstad, Sweden, May 1992.

[MTHM96] Robin Milner, Mads Tofte, Robert Harper, and Dave McQueen. *The Definition of Standard ML, Revised 1996.* MIT Press, 1996. Forthcoming.

[Pfe99] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*, 199? To appear. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186, September 1992.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe92] Frank Pfenning. Computation and deduction. Unpublished lecture notes, revised May 1994, May 1992.

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.

[Pfe94a] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Pfe94b] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[Pfe94c] 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.

[Pfe95] Frank Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.

[Pfe96] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.

[Roh97] Ekkehard Rohwedder. *Verifying the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 1997. Forthcoming.

[Sal90] Anne Salvesen. The Church-Rosser theorem for LF with $\beta\eta$-reduction. Unpublished notes to a talk given at the First Workshop on Logical Frameworks in Antibes, France, May 1990.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information & Computation*, 89:1–34, November 1990.

[Tro86] Anne S. Troelstra. Strong normalization for typed terms with surjective pairing. *Notre Dame Journal of Formal Logic*, 27(4):547–550, October 1986.

[Van96] Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1996.

[Wad90] Philip Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, April 1990. North-Holland.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information & Computation*, 115(1):38–94, November 1994.

A Formalization of *MLR*

A.1 Syntax

```
exp    : type.
tp     : type.
instr  : type.
cont   : type.
cell   : type.
cv     : type.
store  : type.
answer : type.


% Expressions

z       : exp.
s       : exp -> exp.
case    : exp -> exp -> (exp -> exp) -> exp.
unit    : exp.
pair    : exp -> exp -> exp.
fst     : exp -> exp.
snd     : exp -> exp.
lam     : (exp -> exp) -> exp.
app     : exp -> exp -> exp.
letval  : exp -> (exp -> exp) -> exp.
letname : exp -> (exp -> exp) -> exp.
fix     : (exp -> exp) -> exp.
rf      : cell -> exp.
ref     : exp -> exp.
!       : exp -> exp.
seq     : exp -> exp -> exp.
assign  : exp -> exp -> exp.


% Types

nat   : tp.
one   : tp.
cross : tp -> tp -> tp.
arrow : tp -> tp -> tp.
tref  : tp -> tp.


% Instructions

eval     : exp -> instr.
```

```
return   : exp -> instr.
case*    : exp -> exp -> (exp -> exp) -> instr.
pair*    : exp -> exp -> instr.
fst*     : exp -> instr.
snd*     : exp -> instr.
app*     : exp -> exp -> instr.
ref*     : exp -> instr.
deref*   : exp -> instr.
assign*1 : exp -> exp -> instr.
assign*2 : exp -> exp -> instr.


% Continuations

init : cont.
klam : cont -> (exp -> instr) -> cont.


% Store

estore : store.
with   : store -> cv -> store.
holds  : cell -> exp -> cv.


% Answers

close : store -> exp -> answer.
new   : (cell -> answer) -> answer.
```

A.2 Typing

```
tpc : cell -> tp -> type.
tpe : exp -> tp -> type.
tpi : instr -> tp -> type.
tpK : cont -> tp -> tp -> type.
tpS : store -> type.
tpa : answer -> tp -> type.


% Expressions

tpe_z       :    tpe z nat.
tpe_s       :    tpe E nat
              -> tpe (s E) nat.
tpe_case    :    tpe E nat
              -> tpe E1 T
```

```
              -> ({x:exp} tpe x nat
                     -> tpe (E2 x) T)
              -> tpe (case E E1 ([x:exp] E2 x)) T.
tpe_unit    :    tpe unit one.
tpe_pair    :    tpe E1 T1
              -> tpe E2 T2
              -> tpe (pair E1 E2) (cross T1 T2).
tpe_fst     :    tpe E (cross T1 T2)
              -> tpe (fst E) T1.
tpe_snd     :    tpe E (cross T1 T2)
              -> tpe (snd E) T2.
tpe_lam     :    ({x:exp} tpe x T1
                     -> tpe (E x) T2)
              -> tpe (lam ([x:exp] E x)) (arrow T1 T2).
tpe_app     :    tpe E1 (arrow T2 T1)
              -> tpe E2 T2
              -> tpe (app E1 E2) T1.
tpe_letval  :    tpe E1 T1
              -> ({x:exp} tpe x T1
                     -> tpe (E2 x) T2)
              -> tpe (letval E1 ([x:exp] E2 x)) T2.
tpe_letname :    tpe (E2 E1) T
              -> tpe (letname E1 ([x:exp] E2 x)) T.
tpe_fix     :    ({x:exp} tpe x T
                     -> tpe (E x) T)
              -> tpe (fix ([x:exp] E x)) T.
tpe_cell    :    tpc C T
              -> tpe (rf C) (tref T).
tpe_ref     :    tpe E T
              -> tpe (ref E) (tref T).
tpe_deref   :    tpe E (tref T)
              -> tpe (! E) T.
tpe_seq     :    tpe E1 T1
              -> tpe E2 T2
              -> tpe (seq E1 E2) T2.
tpe_assign  :    tpe E1 (tref T)
              -> tpe E2 T
              -> tpe (assign E1 E2) one.


% Instructions

tpi_eval     :    tpe E T
               -> tpi (eval E) T.
tpi_return   :    tpe V T
               -> tpi (return V) T.
tpi_case*    :    tpe V nat
```

```
               -> tpe E1 T
               -> ({x:exp} tpe x nat
                      -> tpe (E2 x) T)
               -> tpi (case* V E1 ([x:exp] E2 x)) T.
tpi_pair*    :    tpe V T1
               -> tpe E T2
               -> tpi (pair* V E) (cross T1 T2).
tpi_fst*     :    tpe V (cross T1 T2)
               -> tpi (fst* V) T1.
tpi_snd*     :    tpe V (cross T1 T2)
               -> tpi (snd* V) T2.
tpi_app*     :    tpe V (arrow T2 T1)
               -> tpe E T2
               -> tpi (app* V E) T1.
tpi_ref*     :    tpe V T
               -> tpi (ref* V) (tref T).
tpi_deref*   :    tpe V (tref T)
               -> tpi (deref* V) T.
tpi_assign*1 :    tpe V (tref T)
               -> tpe E T
               -> tpi (assign*1 V E) one.
tpi_assign*2 :    tpe V1 (tref T)
               -> tpe V2 T
               -> tpi (assign*2 V1 V2) one.


% Continuations

tpK_init :    tpK init T T.
tpK_lam  :    ({x:exp} tpe x T1
                  -> tpi (I x) T)
           -> tpK K T T2
           -> tpK (klam K ([x:exp] I x)) T1 T2.


% Store

tpS_empty :    tpS estore.
tpS_with  :    tpS S
            -> tpc C T
            -> tpe V T
            -> tpS (with S (holds C V)).


% Answers

tpa_close :    tpS S
```

```
           -> tpe V T
           -> tpa (close S V) T.
tpa_new   :   ({c:cell} tpc c T'
                  -> tpa (A c) T)
           -> tpa (new ([c:cell] A c)) T.
```

A.3 Evaluation

```
%%% Contents of a cell

contains: cell -> exp -> type.     % Only run-time assumptions


%%% Collection of all the cells in the store

collect: store -> type.

col_empty :    collect estore.
col_cv    :    contains C V
           -o collect S
           -o collect (with S (holds C V)).


%%% Reading the value of a cell from the store

read: cell -> exp -> type.

read_val  :    contains C V
           -o <T>
           -o read C V.


%%% Evaluation

ev : cont -> instr -> answer -> type.


% Expressions

ev_z       :    ev K (return z) A
            -o ev K (eval z) A.
ev_s       :    ev (klam K ([x:exp] return (s x))) (eval E) A
            -o ev K (eval (s E)) A.
ev_case    :    ev (klam K ([x:exp] case* x E1 E2)) (eval E) A
            -o ev K (eval (case E E1 E2)) A.
ev_unit    :    ev K (return unit) A
            -o ev K (eval unit) A.
```

```
ev_pair    :    ev (klam K ([x:exp] pair* x E2)) (eval E1) A
             -o ev K (eval (pair E1 E2)) A.
ev_fst     :    ev (klam K ([x:exp] fst* x)) (eval E) A
             -o ev K (eval (fst E)) A.
ev_snd     :    ev (klam K ([x:exp] snd* x)) (eval E) A
             -o ev K (eval (snd E)) A.
ev_lam     :    ev K (return (lam E)) A
             -o ev K (eval (lam E)) A.
ev_app     :    ev (klam K ([x:exp] app* x E2)) (eval E1) A
             -o ev K (eval (app E1 E2)) A.
ev_letval  :    ev (klam K ([x:exp] eval (E2 x))) (eval E1) A
             -o ev K (eval (letval E1 E2)) A.
ev_letname :    ev K (eval (E2 E1)) A
             -o ev K (eval (letname E1 E2)) A.
ev_fix     :    ev K (eval (E (fix E))) A
             -o ev K (eval (fix E)) A.
ev_cell    :    ev K (return (rf C)) A
             -o ev K (eval (rf C)) A.
ev_ref     :    ev (klam K ([x:exp] ref* x)) (eval E) A
             -o ev K (eval (ref E)) A.
ev_deref   :    ev (klam K ([x:exp] deref* x)) (eval E) A
             -o ev K (eval (! E)) A.
ev_seq     :    ev (klam K ([x:exp] eval E2)) (eval E1) A
             -o ev K (eval (seq E1 E2)) A.
ev_assign  :    ev (klam K ([x:exp] assign*1 x E2)) (eval E1) A
             -o ev K (eval (assign E1 E2)) A.


% Values

ev_init     :    collect S
              -o ev init (return V) (close S V).
ev_cont     :    ev K (I V) A
              -o ev (klam K I) (return V) A.


% Auxiliary instructions

ev_case*1   :    ev K (eval E1) A
              -o ev K (case* z E1 E2) A.
ev_case*2   :    ev K (eval (E2 V)) A
              -o ev K (case* (s V) E1 E2) A.
ev_pair*    :    ev (klam K ([x:exp] return (pair V x))) (eval E) A
              -o ev K (pair* V E) A.
ev_fst*     :    ev K (return V1) A
              -o ev K (fst* (pair V1 V2)) A.
ev_snd*     :    ev K (return V2) A
```

```
              -o ev K (snd* (pair V1 V2)) A.
ev_app*     :    ev (klam K ([x:exp] eval (E1 x))) (eval E2) A
              -o ev K (app* (lam E1) E2) A.
ev_ref*     :    ({c:cell} contains c V
                     -o ev K (return (rf c)) (A c))
              -o ev K (ref* V) (new A).
ev_deref*   :    (read C V
               & ev K (return V) A)
              -o ev K (deref* (rf C)) A.
ev_assign*1 :    ev (klam K ([x:exp] assign*2 (rf C) x)) (eval E) A
              -o ev K (assign*1 (rf C) E) A.
ev_assign*2 :    contains C V'
              -o (contains C V
                   -o ev K (return unit) A)
              -o ev K (assign*2 (rf C) V) A.
```