A Relevant Analysis of Natural Deduction

S.S. Ishtiaq and D.J. Pym
Queen Mary and Westfield College
University of London

Abstract

We study a framework, RLF, for defining natural deduction presentations of linear and other relevant logics. RLF consists in a language together, in a manner similar to that of LF, with a representation mechanism. The language of RLF, the $\lambda\Lambda_\kappa$-calculus, is a system of first-order linear dependent function types which uses a function κ to describe the degree of sharing of variables between functions and their arguments. The representation mechanism is judgements-as-types, developed for linear and other relevant logics. The $\lambda\Lambda_\kappa$-calculus is a conservative extension of the $\lambda\Pi$-calculus and RLF is a conservative extension of LF.

1 Introduction

Linear and other relevant logics have been studied widely in mathematical [4, 16, 25, 36], philosophical [3, 14, 33] and computational [1, 20, 24, 32, 37] logic. We present a study of a logical framework, RLF, for defining natural deduction presentations of such logics.[1] RLF consists in a language together, in a manner similar to that of LF [5, 18, 28], with a representation mechanism. The language of RLF, the $\lambda\Lambda_\kappa$-calculus, is a system of first-order linear dependent function types which uses a function κ to describe the degree of sharing of variables between functions and their arguments. The representation mechanism is judgements-as-types, developed for linear and other relevant logics.

We motivate the $\lambda\Lambda_\kappa$-calculus by considering an abstract form of relevant natural deduction. We specify the $\lambda\Lambda_\kappa$-calculus, a family of first-order dependent type theories with both linear and intuitionistic function spaces, discussing only briefly the possible intermediate systems. The framework RLF is a conservative extension of LF; the notion of conservative extension takes account of the representation mechanism as well as the type theory. The work reported here builds on ideas presented by Pym in [29].

An explanation regarding our use of the word "relevant" is in order. Following Read [33], we use the term relevant for the family of logics which have weaker structural properties than intuitionistic or classical logic, not merely for

[1] RLF, in common with LF, is also able to define Hilbert-type systems, although this is beyond our present scope.

those which have contraction but not weakening. Read's taxonomy would place linear logic (without exponentials) at the lowest point in the "lattice" (we use the word informally and not in any technical sense) of logics. We follow this taxonomy and thus obtain a lattice of logical frameworks, the weakest being RLF, the type theory of which has neither weakening nor contraction.[2] We emphasize that the $\lambda\Lambda_\kappa$-calculus lies properly in the world of relevant logics: the type theory's contexts are a dependently-typed notion of Read's *bunches* [33]. The title of this paper reflects this point of view. Our framework RLF provides a relevant analysis of natural deduction just as LF provides an intuitionistic analysis of natural deduction. In this paper, we do not study the variety of distributivity laws usually considered for relevant contexts [33]. However, such an investigation should fit into our framework, possibly via variations on the $\lambda\Lambda_\kappa$-calculus, quite straightforwardly.

The paper is organized as follows. In § 2, we motivate the $\lambda\Lambda_\kappa$-calculus in the context of a logical framework by considering an abstract form of relevant natural deduction. We formally define it as a type theory and summarize its meta-theory in § 3, concluding the section with a comparison with related work. In § 4, we show that RLF is a conservative extension of LF. In § 5, we illustrate several example encodings in the RLF framework. The object-logics we consider are a fragment of propositional intuitionistic linear logic, the dynamic semantics of ML with references and a relevant λ-calculus. Finally, in § 6, we consider the further work that arises from our study.

2 Motivation

Logical frameworks are formal meta-logics which, *inter alia*, provide languages for describing logics in a manner that is suitable for mechanical implementation. The LF logical framework [5, 18, 28] provides such a metatheory and is suitable for logics which have at least the structural strength of minimal propositional logic. We wish to study a logical framework for describing relevant logics. Now, in order to describe a logical framework one must:

1. Characterize the class of object-logics to be represented;
2. Give a meta-logic or language, together with its meta-logical status *vis-à-vis* the class of object-logics; and
3. Characterize the representation mechanism for object-logics.

The above prescription can conveniently be summarized by the slogan

$$Framework = Language + Representation.$$

We remark that these components are not entirely independent of each other [30]. We will point out some interdependencies later in this section.

[2] In the literature, the terms "sub-structural" and "weak" are sometimes used in this way.

One representation mechanism is that of judgements-as-types, which originates from Martin-Löf's [23] development of Kant's [22] notion of judgement. The two higher-order judgements, the hypothetical $J \vdash J'$ and the general $\bigwedge_{x \in C} . J(x)$, correspond to ordinary and dependent function spaces, respectively. The methodology of judgements-as-types is that judgements are represented as the type of their proofs. A logical system is represented by a signature which assigns kinds and types to a finite set of constants that represent its syntax, its judgements and its rule schemes. An object-logic's rules and proofs are seen as proofs of hypothetico-general judgements $\bigwedge_{x_1 \in C_1} \cdots \bigwedge_{x_m \in C_m} . J \vdash J'$. Representation theorems relate consequence in an object-logic $\vdash_L$ to consequence in an encoded logic $\vdash_{\Sigma_L}$:

$$
\begin{array}{cl}
(X, J_1(\phi_1), \ldots, J_m(\phi_m)) \vdash_L \delta \, : \, J(\phi) & object-consequence \\
\Downarrow & encoding \\
\Gamma_X, x_1{:}J_1(\phi_1), \ldots, x_m{:}J_n(\phi_n) \vdash_{\Sigma_L} M_\delta \, : \, J(\phi) & meta-consequence,
\end{array}
$$

where X is the set of variables that occur in ϕ_i, ϕ; J_i, J are judgements; δ is a proof-object (*e.g.*, a λ-term); Γ_X corresponds to X; each x_i corresponds to a place-holder for the encoding of J_i; and M_δ is a meta-logic term corresponding to the encoding of δ.

In the sequel, we do not consider the complete apparatus of judged object-logics. Our example encodings in § 5 are pathological in the sense that they require only one judgement. For example, the encoding of a fragment of intuitionistic linear logic requires the judgement of $(J_i = J =)$ *proof*. This is in contrast to the general multi-judgement representation techniques [6]. We conjecture that our studies can be applied to the general case, although we defer this development to another occasion.

A certain class of uniform representations is identified by considering surjective encodings between consequences of the object-logic $\vdash_L$ and consequences of the meta-logic $\vdash_{\Sigma_L}$ [19].[3] So, all judgements in the meta-logic have corresponding judgements in the object-logic. The judgement-as-types methodology has the property that encoded systems inherit the structural properties of the meta-logic. It is for this reason that LF — whose language, the $\lambda\Pi$-calculus, admits weakening and contraction — cannot uniformly encode linear and other relevant logics. To illustrate this point, suppose Σ_{ILL} is a uniform encoding of intuitionistic linear logic in LF, and that $\Gamma_X, \Gamma_\Delta \vdash_{\Sigma_{ILL}} M_\delta{:}J(\phi)$ is the image of the object-consequence $(X, \Delta) \vdash_{ILL} \delta{:}J(\phi)$. If $\Gamma_X, \Gamma_\Delta \vdash_{\Sigma_{ILL}} M_\delta{:}J(\phi)$ is provable, then so is $\Gamma_X, \Gamma_\Delta, \Gamma_\Theta \vdash_{\Sigma_{ILL}} M_\delta{:}J(\phi)$. By uniformity, the latter is the image of an object-logic consequence $(X, \Delta, \Theta) \vdash_{ILL} \delta'{:}J(\phi)$, which implies weakening in linear logic, a contradiction.

Thus we seek a language in which weakening and contraction are not forced. We motivate the connectives of the language by considering the natural deduction form of rules for weak logics. We do this in a general way, by considering

[3] The specification in [19] is a stronger one, requiring uniformity over all "presentations" of a given logic. Such concerns are beyond our present scope.

Prawitz's general form of schematic introductions from a more relevant point of view. Prawitz [27] gives these for intuitionistic logic. A schematic introduction rule for an n-ary sentential operator # is represented by an introduction rule of the form below. In the rule, only the bound assumptions for G_j are shown; we elide those for G_k, where $(k \neq j)$, for the sake of readability.

$$\frac{\begin{array}{ccccc} & & [H_{j,1}]\cdots[H_{j,h_j}] & & \\ \vdots & & \vdots & & \vdots \\ G_1 & \cdots & G_j & \cdots & G_p \end{array}}{\#(F_1,\ldots,F_n)}$$

In the above rule, $1 \leq j \leq p$. The Fs, Gs and Hs are formulae constructed in the usual way. An inference infers a formula $\#(F_1,\ldots,F_n)$ from p premises $G_1,\ldots,G_p$ and may bind assumptions of the form $H_{j,1},\ldots,H_{j,h_j}$ that occur above the premise G_j. We let the assumptions be multi-sets, thus keeping the structural rule of exchange. We require that discharge be compulsory. In the case of the natural deduction presentation of intuitionistic linear logic, for instance, we require that $\{F_1,\ldots,F_n\} = \{G_j, H_{j,1}\ldots,H_{j,h_j}\}$. For example, in the rule for $\multimap$-introduction, whose conclusion is $\phi \multimap \psi$, we have $\{F_1,F_2\} = \{\phi,\psi\}$, $G_1 = \psi$ and $H_{1,1} = \phi$.

We annotate the introduction schema below to indicate our method of encoding. The Λ is a linear universal quantifier, o is the type of propositions and $\in$ ranges over both linear $(F{:}o)$ and exponential $(F!o)$ declarations. Each inference — that is, the binding of assumptions $H_{j,1},\ldots,H_{j,h_j}$ above premise G_j and the inference of formula $\#(F_1,\ldots,F_n)$ from premises $G_1,\ldots,G_p$ — is represented by a $\multimap$.

$$\boldsymbol{\Lambda} F_g, G_j, H_{j,k} \in o \qquad \frac{\begin{array}{ccccccc} & & & [H_{j,1}]\Box\cdots\Box[H_{j,h_j}] & & & \\ \vdots & & & \vdots & & & \vdots \\ G_1 & \Box & \cdots & \Box\; G_j\; \Box & \cdots & \Box & G_p \end{array}}{\#(F_1,\ldots,F_n)}$$

The premises $G_1,\ldots,G_p$ are combined either multiplicatively or additively, depending on whether their contexts are disjoint or not. We distinguish between these combinations by the use of two conjunctions; the multiplicative $\otimes$ ("tensor") and the additive & ("with") and so force the structural rules. (In traditional relevance logics, multiplicative is referred to as intentional and additive as extensional.) We use $\Box$ as meta-syntax for both $\otimes$ and &, though mindful of the relationship between the two operators. Full expressivity is recovered by introducing the modality ! ("bang") into our language. The premise $!G$ allows us to depart from relevant inference, and to choose the number of times we use G in the conclusion.

In the meta-logic, then, the schematic introduction rule would be represented by a constant of the following type:

$$\Lambda\, F_g, G_j, H_{j,k} \in o\, . \ldots \Box(\Box_{l \leq h_j}(H_{j,l}) \multimap G_j)\Box \ldots \multimap \#(F_1, \ldots, F_n),$$

where $1 \leq l \leq h_j$ and $\Box_{l \leq h_j}$ represents an iterated $\Box$. From the general encoding formula above, it can be seen that the connectives $\Box$ (*i.e.*, $\otimes$ and $\&$) and $!$ occur only negatively. In the tensor's case, this allows us to curry away the $\otimes$, modulo a permutation of the premises. For example, in the following type, we are able to replace the occurrence of $\otimes$,

$$(\Box_{l \leq h_j}(H_{j,l}) \multimap G_j) \otimes (\Box_{l' \leq h_{j'}}(H_{j',l'}) \multimap G_{j'}) \multimap \#(F_1, \ldots, F_n),$$

by a $\multimap$,

$$(\Box_{l \leq h_j}(H_{j,l}) \multimap G_j) \multimap (\Box_{l' \leq h_{j'}}(H_{j',l'}) \multimap G_{j'}) \multimap \#(F_1, \ldots, F_n).$$

We can also consider a currying away of the $\&$ by a non-dependent version of the additive function space. A language with two kinds of dependent function space is very interesting but is beyond the scope of our current study.

We recapitulate exactly how we have used the three logical constants in the framework: the $\&$ is used to undertake additive conjunction; the Λ is used to quantify and (in its non-dependent form $\multimap$) to represent implication; and the $!$ is used to represent dereliction from relevant inference. We should then be able to formulate a precise idea regarding the completeness of the set $\{\&, \Lambda, !\}$ with respect to all sentential operators that have explicit schematic introduction rules [27, 35].

A similar analysis can be undertaken for the corresponding elimination rule.

Our analysis allows us two degrees of freedom. The first is at the structural level of types. In this section, our main intention has been to motivate a language in which the structural rules of weakening and contraction are not forced, and so to be able to uniformly encode linear logic. But this language is only one of a range of relevant logics [33], which includes, for instance, Anderson and Belnaps's relevance logic [3]. Choosing a different language, with it's particular structural and distributivity properties, would allow us to uniformly encode another class of logics. The family of relevant logics determined by these choices is very interesting from a representational perspective, though we pursue it no further in this paper.

The second, orthogonal, degree of freedom, and one that we do concentrate on in the sequel, concerns the corresponding range of structural choices at the level of terms (as opposed to types). Considering this aspect from the logical point of view, we consider multiple occurrences of the same proof. The degree to which a proof can be shared by propositions is a structural property which determines, via the Curry-Howard-de Bruijn correspondence, a type theory whose functions and arguments share variables to a corresponding degree.

The language that we have motivated in this section, and develop in the sequel, is a type theory in Curry-Howard-de Bruijn correspondence with a $\{\&, \multimap$

$\otimes$, $\multimap$, $\forall$, !}-fragment of intuitionistic linear logic (ILL) extended with a $\forall\!\!\!\forall$ linear universal quantifier. The details of the correspondence are deferred to another occasion.

3 The $\lambda\Lambda_\kappa$-calculus

The $\lambda\Lambda_\kappa$-calculus is a first-order dependent type theory for a fragment of linear logic with both intuitionistic and linear function types. The calculus is used for deriving typing judgements. There are three entities in the $\lambda\Lambda_\kappa$-calculus: *objects*, *types* and *families of types*, and *kinds*. Objects (denoted by M, N) are classified by types. Families of types (denoted by A, B) may be thought of as functions which map objects to types. Kinds (denoted by K) classify families. In particular, there is a kind Type which classifies the types. We will use U, V to denote any of the entities.

We assume given three disjoint, countably infinite sets: the meta-variables x, y, z range over the set of variables; c, d range over the set of object-level constants; and a, b range over the set of type-level constants. The abstract syntax of the $\lambda\Lambda_\kappa$-calculus is given by the following grammar:

$$\begin{array}{llll} \textit{Kinds} & K & ::= & \mathsf{Type} \mid \Lambda\, x{:}A\,.K \mid \Lambda\, x!A\,.K \\ \textit{Types} & A & ::= & a \mid \Lambda\, x{:}A\,.B \mid \Lambda\, x!A\,.B \mid \lambda\, x{:}A\,.B \mid \lambda\, x!A\,.B \mid AM \mid A\&B \\ \textit{Objects} & M & ::= & c \mid x \mid \lambda\, x{:}A\,.M \mid \lambda\, x!A\,.M \mid MN \mid \langle M, N\rangle \mid \pi_0 M \mid \pi_1 M\ . \end{array}$$

We write $x{\in}A$ to range over both linear ($x{:}A$) and exponential ($x!A$) variable declarations. The λ and Λ bind the variable x. The object $\lambda\, x{:}A\,.M$ is an inhabitant of the linear dependent function type $\Lambda\, x{:}A\,.B$. The object $\lambda\, x!A\,.M$ is an inhabitant of the type $\Lambda\, x!A\,.B$, which amounts to the Martin-Löf-style $\Pi\, x{:}A\,.B$. The notion of linear free- and bound-variables (LFV, LBV) and substitution may be defined accordingly [10]. When x is not free in B we write $A \multimap B$ and $A \to B$ for $\Lambda\, x{:}A\,.B$ and $\Lambda\, x!A\,.B$, respectively. Our basic study does not include the units, but $\top$ and 1 can be added to the type theory with little difficulty.

We can define the notion of linear occurrence by extending the general idea of occurrence for the λ-calculus [7], though we note that other definitions may be possible.

Definition 3.1 (linear occurrence in U)

1. *x linearly occurs in x;*
2. *If x linearly occurs in U or V (or both), then x linearly occurs in $\lambda\, y{\in}U\,.V$, in $\Lambda\, y{\in}U\,.V$, and in UV, where $x \neq y$;*
3. *If x linearly occurs in both M and N, then x linearly occurs in $\langle M, N\rangle$;*
4. *If x linearly occurs in M, then x linearly occurs in $\pi_i(M)$;*
5. *If x linearly occurs in both A and B, then x linearly occurs in $A\&B$.*

The definition is extended to an inhabited type and kind.

Definition 3.2 (linear occurrence in $U{:}V$) *A variable x linearly occurs in the expression $U{:}V$ if it linearly occurs in U, in V, or in both.*

In the sequel we will often refer informally to the concept of a linearity constraint. Essentially this means that all linear variables declared in the context are used: a production-consumption contract. But we depart from the usual resource-conscious logics idea that formulae are produced in the antecedent and consumed in the succedent. Given this, the judgement $x{:}A, y{:}cx \vdash_\Sigma y{:}cx$ in which the linear x is consumed by the (type of) y declared after it and the y itself is consumed in the succedent, is a valid one.

In the $\lambda\Lambda_\kappa$-calculus signatures are used to keep track of the types and kinds assigned to constants. Contexts are used to keep track of the types, both linear and exponential, assigned to variables. The abstract syntax for signatures and contexts is given by the following grammar:

$$\mathit{Signatures} \quad \Sigma ::= \langle\rangle \mid \Sigma, a!K \mid \Sigma, c!A \qquad\qquad \mathit{Contexts} \quad \Gamma ::= \langle\rangle \mid \Gamma, x{:}A \mid \Gamma, x!A \ .$$

So signatures and contexts consists of finite sequences of declarations. The dependency aspect of the type theory requires that "[bases] have to become linearly ordered" [8, page 198]. We assume the usual extraction functions $(dom(\Gamma), ran(\Gamma))$ related to such lists. We also define the following two functions which extract out just the *lin*ear and *exp*onential parts of a context:

$$\begin{array}{rcl@{\qquad}rcl} lin(\langle\rangle) & = & \langle\rangle & exp(\langle\rangle) & = & \langle\rangle \\ lin(\Gamma, x{:}A) & = & lin(\Gamma), x{:}A & exp(\Gamma, x{:}A) & = & exp(\Gamma) \\ lin(\Gamma, x!A) & = & lin(\Gamma) & exp(\Gamma, x!A) & = & exp(\Gamma), x!A \end{array}$$

The $\lambda\Lambda_\kappa$-calculus is a formal system for deriving the following judgements:

$$\begin{array}{ll} \vdash \Sigma\ \mathsf{sig} & (\Sigma \text{ is a valid signature}) \\ \vdash_\Sigma \Gamma\ \mathsf{context} & (\Gamma \text{ is a valid context in } \Sigma) \\ \Gamma \vdash_\Sigma K\ \mathsf{Kind} & (K \text{ is a valid kind in } \Sigma \text{ and } \Gamma) \\ \Gamma \vdash_\Sigma A{:}K & (A \text{ has a kind } K \text{ in } \Sigma \text{ and } \Gamma) \\ \Gamma \vdash_\Sigma M{:}A & (M \text{ has a type } A \text{ in } \Sigma \text{ and } \Gamma) \end{array}$$

We write $\Gamma \vdash_\Sigma U{:}V$ for either of $\Gamma \vdash_\Sigma A{:}K$ or $\Gamma \vdash_\Sigma M{:}A$, and $\Gamma \vdash_\Sigma X$ for $\Gamma \vdash_\Sigma K\ \mathsf{Kind}$ or $\Gamma \vdash_\Sigma U{:}V$. We abuse notation and also write $\Gamma \vdash_\Sigma X$ to indicate the derivability of X in the $\lambda\Lambda_\kappa$-calculus, in which case the K or U is said to be *valid* in the signature Σ and context Γ.

The definition of the type theory depends crucially on the following three notions:

1. The joining together of two contexts to form a third must be undertaken so that the order of declarations and type of variables (linear versus intuitionistic) is respected;
2. The idea of linear variable occurrences allows us to form contexts of the form $x{:}A, x{:}A$, for some type constant A in the signature. That is, contexts in which repeated but distinct declarations of the same variable are possible;

3. Following a joining of contexts, certain occurrences of linear variables – those that are shared by a function and its argument – are identified with one another. This sharing is implemented by the κ function.

These notions will be explicated at appropriate points in the sequel.

We now present the rules for deriving judgements in Tables 1 and 2 below. To save space, we place any side-conditions along with the premises. The rules are conveniently separated into a linear and an exponential set, the latter relating directly to the intuitionistic $\lambda\Pi$-calculus.

Valid Signatures

$$\frac{}{\vdash \langle\rangle\ \mathsf{sig}}\,(\Sigma)$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \quad \vdash_\Sigma K\ \mathsf{Kind} \quad a \notin \Sigma}{\vdash \Sigma, a!K\ \mathsf{sig}}\,(\Sigma K!) \qquad \frac{\vdash \Sigma\ \mathsf{sig} \quad \vdash_\Sigma A{:}\mathsf{Type} \quad c \notin \Sigma}{\vdash \Sigma, c!A\ \mathsf{sig}}\,(\Sigma A!)$$

Valid Contexts

$$\frac{\vdash \Sigma\ \mathsf{sig}}{\vdash_\Sigma \langle\rangle\ \mathsf{context}}\,(\Gamma)$$

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{context} \quad \Delta \vdash_\Sigma A{:}\mathsf{Type} \quad [\Xi;\Gamma;\Delta] \quad (x \notin dom(\Xi) \text{ or } x{:}A \in \Xi)}{\vdash_\Sigma \Xi, x{:}A\ \mathsf{context}}\,(\Gamma A)$$

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{context} \quad \Delta \vdash_\Sigma A{:}\mathsf{Type} \quad [\Xi;\Gamma;\Delta] \quad (x \notin dom(\Xi) \text{ or } x{:}A \in \Xi)}{\vdash_\Sigma \Xi, x!A\ \mathsf{context}}\,(\Gamma A!)$$

Valid Kinds

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{context}}{\Gamma \vdash_\Sigma\ \mathsf{Type}\ \mathsf{Kind}}\,(KAx) \qquad \frac{\Gamma, x{:}A \vdash_\Sigma K\ \mathsf{Kind}}{\Gamma \vdash_\Sigma \Lambda\, x{:}A\,.K\ \mathsf{Kind}}\,(K\Lambda\mathcal{I}1)$$

$$\frac{\Gamma \vdash_\Sigma A{:}\mathsf{Type} \quad \Delta \vdash_\Sigma K{:}\mathsf{Kind} \quad [\Xi';\Gamma;\Delta] \quad \Xi = \Xi' \backslash (lin(\Gamma) \cap lin(\Delta))}{\Xi \vdash_\Sigma A \multimap K{:}\mathsf{Kind}}\,(K\Lambda\mathcal{I}2)$$

$$\frac{\Gamma, x!A \vdash_\Sigma K\ \mathsf{Kind}}{\Gamma \vdash_\Sigma \Lambda\, x!A\,.K\ \mathsf{Kind}}\,(K\Lambda!\mathcal{I})$$

Table 1: $\lambda\Lambda_\kappa$-calculus

The signature formation rules enforce intuitionistic behaviour by allowing only a constant of exponential type to extend the signature. The context formation rules allow only types to be assigned to variables. We distinguish between extending the context by linear $(\Xi, x{:}A)$ and exponential $(\Xi, x!A)$ variables. The context formation rules introduce two particular characteristics of the type theory. The first one is that of joining the premise contexts for the multiplicative rules. The join must respect the ordering of the premise contexts and the concept of linear versus exponential variables. A method to join Γ and Δ into Ξ –

Valid Families of Types

$$\frac{\vdash_\Sigma !\Gamma \ \mathsf{context} \quad a!K \in \Sigma}{!\Gamma \vdash_\Sigma a{:}K}\,(Ac)$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma B{:}\mathsf{Type}}{\Gamma \vdash_\Sigma \Lambda x{:}A\,.B \ :\ \mathsf{Type}}\,(A\Lambda\mathcal{I}1) \quad \frac{\Gamma \vdash_\Sigma A{:}\mathsf{Type} \quad \Delta \vdash_\Sigma B{:}\mathsf{Type} \quad [\Xi';\Gamma;\Delta] \quad \Xi = \Xi' \backslash (lin(\Gamma) \cap lin(\Delta))}{\Xi \vdash_\Sigma A \multimap B{:}\mathsf{Type}}\,(A\Lambda\mathcal{I}2)$$

$$\frac{\Gamma, x!A \vdash_\Sigma B{:}\mathsf{Type}}{\Gamma \vdash_\Sigma \Lambda x!A\,.B \ :\ \mathsf{Type}}\,(A\Lambda!\mathcal{I})$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma B{:}K}{\Gamma \vdash_\Sigma \lambda x{:}A\,.B \ :\ \Lambda x{:}A\,.K}\,(A\lambda\Lambda\mathcal{I}) \quad \frac{\Gamma \vdash_\Sigma B \ :\ \Lambda x{:}A\,.K \quad \Delta \vdash_\Sigma N{:}A \quad [\Xi';\Gamma;\Delta] \quad \Xi = \Xi' \backslash \kappa(\Gamma,\Delta)}{\Xi \vdash_\Sigma BN \ :\ K[N/x]}\,(A\Lambda\mathcal{E})$$

$$\frac{\Gamma, x!A \vdash_\Sigma B{:}K}{\Gamma \vdash_\Sigma \lambda x!A\,.B \ :\ \Lambda x!A\,.K}\,(A\lambda\Lambda!\mathcal{I}) \quad \frac{\Gamma \vdash_\Sigma B \ :\ \Lambda x!A\,.K \quad !\Delta \vdash_\Sigma N{:}A \quad [\Xi;\Gamma;!\Delta]}{\Xi \vdash_\Sigma BN \ :\ K[N/x]}\,(A\Lambda!\mathcal{E})$$

$$\frac{\Gamma \vdash_\Sigma A{:}\mathsf{Type} \quad \Gamma \vdash_\Sigma B{:}\mathsf{Type}}{\Gamma \vdash_\Sigma A\&B{:}\mathsf{Type}}\,(A\&\mathcal{I})$$

$$\frac{\Gamma \vdash_\Sigma A{:}K \quad \Delta \vdash_\Sigma K' \ \mathsf{Kind} \quad K \equiv K' \quad [\Xi;\Gamma;\Delta]}{\Xi \vdash_\Sigma A{:}K'}\,(A\equiv)$$

Valid Objects

$$\frac{\vdash_\Sigma !\Gamma \ \mathsf{context} \quad c!A \in \Sigma}{!\Gamma \vdash_\Sigma c{:}A}\,(Mc)$$

$$\frac{\Gamma \vdash_\Sigma A{:}\mathsf{Type}}{\Gamma, x{:}A \vdash_\Sigma x{:}A}\,(MVar) \quad \frac{\Gamma \vdash_\Sigma A{:}\mathsf{Type}}{\Gamma, x!A \vdash_\Sigma x{:}A}\,(MVar!)$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma M{:}B}{\Gamma \vdash_\Sigma \lambda x{:}A\,.M \ :\ \Lambda x{:}A\,.B}\,(M\lambda\Lambda\mathcal{I}) \quad \frac{\Gamma \vdash_\Sigma M \ :\ \Lambda x{:}A\,.B \quad \Delta \vdash_\Sigma N{:}A \quad [\Xi';\Gamma;\Delta] \quad \Xi = \Xi' \backslash \kappa(\Gamma,\Delta)}{\Xi \vdash_\Sigma MN \ :\ B[N/x]}\,(M\Lambda\mathcal{E})$$

$$\frac{\Gamma, x!A \vdash_\Sigma M{:}B}{\Gamma \vdash_\Sigma \lambda x!A\,.M \ :\ \Lambda x!A\,.B}\,(M\lambda\Lambda!\mathcal{I}) \quad \frac{\Gamma \vdash_\Sigma M \ :\ \Lambda x!A\,.B \quad !\Delta \ \vdash_\Sigma \ N{:}A \quad [\Xi;\Gamma;!\Delta]}{\Xi \vdash_\Sigma MN \ :\ B[N/x]}\,(M\Lambda!\mathcal{E})$$

$$\frac{\Gamma \vdash_\Sigma M{:}A \quad \Gamma \vdash_\Sigma N{:}B}{\Gamma \vdash_\Sigma \langle M, N\rangle \ :\ A\&B}\,(M\&\mathcal{I}) \quad \frac{\Gamma \vdash_\Sigma M \ :\ A_0\&A_1}{\Gamma \vdash_\Sigma \pi_i M \ :\ A_i}\,(M\&\mathcal{E}_i)\ (i \in \{0,1\})$$

$$\frac{\Gamma \vdash_\Sigma M{:}A \quad \Delta \vdash_\Sigma A'{:}\mathsf{Type} \quad A \equiv A' \quad [\Xi;\Gamma;\Delta]}{\Xi \vdash_\Sigma M{:}A'}\,(M\equiv)$$

Table 2: $\lambda\Lambda_\kappa$-calculus (continued)

denoted by $[\Xi;\Gamma;\Delta]$ – is defined in Section 3.1 below.

In order to motivate the second characteristic of the type theory, consider the following simple, apparently innocuous, derivation. We assume that $A!\mathsf{Type}$ and $c!A \multimap \mathsf{Type}$ are declared in the signature Σ. We note that the argument type, cx, is a dependent one; the linear x is free in it.

$$\dfrac{\dfrac{\dfrac{\vdots}{x{:}A \vdash_\Sigma cx{:}\mathsf{Type}}}{x{:}A, z{:}cx \vdash_\Sigma z{:}cx}}{x{:}A \vdash_\Sigma \lambda\, z{:}cx\,.z \;:\; \Lambda\, z{:}cx\,.cx} \quad \dfrac{\dfrac{\vdots}{x{:}A \vdash_\Sigma cx{:}\mathsf{Type}}}{x{:}A, y{:}cx \vdash_\Sigma y{:}cx}$$
$$\overline{\quad x{:}A, x{:}A, y{:}cx \vdash_\Sigma (\lambda\, z{:}cx\,.z)y \;:\; cx \quad}$$

The problem is that an excess of linear xs now appear in the combined context after the application step. (In this step, the types match literally. However this problem arises where they are equal too.) Our solution is to recognize the two xs as two *distinct* occurrences of the *same* variable, the one occuring in the argument type cx, and to allow a degree of freedom in sharing these occurrences. It is now necessary to formally define a binding strategy for multiple occurrences; this we do in § 3.2 below. The sharing aspect is implemented via the κ function, defined in § 3.3 below. One implication of this solution is that repeated declarations of the same variable are allowed in contexts. For this reason, the usual side-condition of $x \notin dom(\Xi)$ is absent from the rules for valid contexts, though of course we don't allow the same variable to inhabit two distinct types.

The $(K\Lambda\mathcal{I})$ and $(A\Lambda\mathcal{I})$ pair of rules form linear function spaces. The first of each pair, in which $x \in FV(B)$, constructs linear dependent function spaces. The second rule of each pair constructs the ordinary linear function spaces. There are two side conditions for the latter rules: the first joins the premise contexts and the second then does a necessary book-keeping for those occurrences of linear variables which are identified with each other under the current binding strategy. The side-conditions in the $(A\Lambda\mathcal{E})$ and $(M\Lambda\mathcal{E})$ rules are of a similar nature. The κ function selects those such "critical" linear occurrences. These occurrences are removed to give the conclusion context. It can be seen that these side-conditions are type-theoretically and, via the propositions-as-types correspondence, logically natural.

The essential difference between linear and intuitionistic function spaces can be observed by considering the $(M\Lambda\mathcal{E})$ and $(M\Lambda!\mathcal{E})$ rules. For the latter, the context for the argument $N{:}A$ is an entirely intuitionistic one $(!\Delta)$, which allows the function to use N as many times as it likes.

Example 3.1 *We end this sub-section with an example of a derivation which does not involve sharing. Let* $A!\mathsf{Type}$, $d!A \multimap \mathsf{Type}$, $e!\Lambda\, y{:}A\,.dy \in \Sigma$. *Then we construct*

$$\dfrac{\dfrac{\dfrac{\vdash_\Sigma \langle\rangle\ \mathsf{context}}{\vdash_\Sigma e : \Lambda\, y{:}A\,.dy} \quad \dfrac{\vdash_\Sigma A{:}\mathsf{Type}}{x{:}A \vdash_\Sigma x{:}A}}{\dfrac{x{:}A \vdash_\Sigma ex : dx}{\vdash_\Sigma \lambda\, x{:}A\,.ex : \Lambda\, x{:}A\,.dx}} \quad \dfrac{\vdash_\Sigma A{:}\mathsf{Type}}{z{:}A \vdash_\Sigma z{:}A}}{z{:}A \vdash_\Sigma (\lambda\, x{:}A\,.ex)z : (dx)[z/x]}$$

Now, $(\lambda\, x{:}A\,.ex)z \to_\beta ez$ *and* $ez{:}dz$*, which maintains the linear occurrence of the variable* z*.*

3.1 Context joining

The method of joining two contexts is a ternary relation $[\Xi;\Gamma;\Delta]$, to be read as "the contexts Γ and Δ are joined to form the context Ξ". Or, for proof-search: "the context Ξ is split into the contexts Γ and Δ".

The first rule for defining $[\Xi;\Gamma;\Delta]$ states that an empty context can be formed by joining together two empty contexts. The second and third rules comply with the linearity constraint, and imply that the linear variables in Ξ are exactly those of Γ and Δ. The last rule takes account of the intuitionistic behaviour of exponential variables. In search, the intuitionistic variable $x!A$ would be sent both ways when the context is split.

$$\dfrac{}{[\langle\rangle;\langle\rangle;\langle\rangle]}\ (\text{JOIN})$$

$$\dfrac{[\Xi;\Gamma;\Delta]}{[\Xi, x{:}A;\Gamma, x{:}A;\Delta]}\ (\text{JOIN-L}) \qquad \dfrac{[\Xi;\Gamma;\Delta]}{[\Xi, x{:}A;\Gamma;\Delta, x{:}A]}\ (\text{JOIN-R})$$

$$\dfrac{[\Xi;\Gamma;\Delta]}{[\Xi, x!A;\Gamma, x!A;\Delta, x!A]}\ (\text{JOIN-!})$$

Table 3: Context joining

Further, the context joining relation must respect the ordering of the contexts and the linearity constraint (as defined by the binding strategy in the next section). That is, if $\vdash_\Sigma \Gamma$ context, $\vdash_\Sigma \Delta$ context and $[\Xi;\Gamma;\Delta]$, then $\vdash_\Sigma \Xi$ context (and *vice versa* for when Ξ is split into Γ and Δ). We remark that if we were also studying the distribution laws for relevant contexts, then the context joining relation would need to take regard of these context equalities.

We make a brief remark about the $[\Xi;\Gamma;\Delta]$ relation with regard to logic programming. As we noted above, in proof-search (the basis of logic programming) the relation $[\Xi;\Gamma;\Delta]$ is read as "split Ξ into Γ and Δ". An implementation of the $\lambda\Lambda_\kappa$-calculus as a logic programming language would have to calculate such splittings, perhaps using techniques similar to those for Lolli and Lygon [17, 20], although it would be interesting to consider approaches in which $[\Xi;\Gamma;\Delta]$ remained unevaluated for as long as possible during search. Such an approach would resemble matrix methods [39].

3.2 Multiple occurrences

Consider the multiple occurrences idea from a proposition-as-types reading. Then $x{:}A, x{:}A$ can be understood as two uses of the same proof of the same proposition, as opposed to $x{:}A, y{:}A$, which can be seen as distinct proofs of the same proposition. Though this idea can be seen, in the presence of the binding strategy that we are about to define, as an internalization of α-conversion, it allows us a degree of freedom, that at the structural level of terms (as opposed to types), which is useful in dealing with variable sharing (§ 3.3).

In this section, we define the "left-most free occurrence of x" in U and a corresponding binding strategy for it. We use this in the sequel, later noting that it can be generalized.

Definition 3.3 *The left-most linear occurrence of x in U is defined as follows, provided that $x \in LFV(U)$. We use @ to denote atoms (constants and variables) and say "x,@ distinct" if @ is a, c or y.*
(Constant, Variable) The constant and variable cases are trivial:

$$\begin{array}{lcll} lm_x(@) & = & \{\} & x,@ \text{ distinct} \\ lm_x(x) & = & \{x\} & \end{array}$$

(Abstraction) We adopt the usual technique of capture-avoiding substitution for the case where another occurrence of x has already been bound. By induction, the λ (Λ) binds a given occurrence — the left-most one — of x in U. So we can α-convert this to $\lambda\, z{\in}A\,.V[z/x]$ ($\Lambda\, z{\in}A\,.V[z/x]$) and continue. We give the cases for the λ binder; the ones for Λ are exactly similar.

$$\begin{array}{lcll} lm_x(\lambda\, y{\in}A\,.V) & = & \left\{ \begin{array}{ll} lm_x(A) & x \in LFV(A) \\ lm_x(V) & \text{otherwise} \end{array} \right\} & x,y \text{ distinct} \\ lm_x(\lambda\, x{\in}A\,.V) & = & lm_x(\lambda\, z{\in}A\,.V[z/x]) & z \text{ new} \end{array}$$

(Application) The left-most occurrence of x in VM is in V or, failing that, in M. The case where V is a constant or variable is straight-forward:

$$\begin{array}{lcll} lm_x(@M) & = & lm_x(M) & x,@ \text{ distinct} \\ lm_x(xM) & = & \{x\} & \end{array}$$

Otherwise, we need to check whether x is free in V or not:

$$lm_x(VM) = \left\{ \begin{array}{ll} lm_x(V) & x \in LFV(V) \\ lm_x(M) & \text{otherwise} \end{array} \right.$$

(Pairing) We deal with the additive cases by a disjoint union of the left-most occurrence of x in both components of the pair:

$$\begin{array}{lcll} lm_x(\langle M,N\rangle) & = & lm_x(M) \uplus lm_x(N) & \\ lm_x(\pi_i(M)) & = & lm_x(M) & i \in \{0,1\} \\ lm_x(A\&B) & = & lm_x(A) \uplus lm_x(B) & \end{array}$$

We define the left-most occurrence of $x{:}A$ in a context Γ as the first declaration of $x{:}A$ in Γ. Similarly, the right-most occurrence of $x{:}A$ in Γ is the last such declaration.

The binding strategy now formalizes the concept of linearity constraint:

Definition 3.4 (left-most binding) *Assume $\Gamma, x{:}A, \Delta \vdash_\Sigma U{:}V$ and that $x{:}A$ is the right-most occurrence of x in the context. Then x binds:*

1. *The first left-most occurrence of x in $ran(\Delta)$, if there is such a declaration;*
2. *The unbound left-most linear occurrences of x in $U{:}V$.*

There is no linearity constraint for intuitionistic variables: the right-most occurrence of $x!A$ in the context binds all the unbound xs used in the type of a declaration in Δ and all the occurrences of x in $U{:}V$.

The rules for deriving judgements are now read according to the strategy in place. For example, in the $(M\lambda\Lambda\mathcal{I})$ rule, the $\lambda(\Lambda)$ binds the left-most occurrence of x in $M(B)$. Similarly, in the (admissible) cut rule, the term $N{:}A$ cuts with the left-most occurrence of $x{:}A$ in the context $\Delta, x{:}A, \Delta'$. In the corresponding intuitionistic rules, the $\lambda!(\Lambda!)$ binds all occurrences of x in $M(B)$ and $N{:}A$ cuts all occurrences of $x!A$ in the context $\Delta, x!A, \Delta'$.

In the sequel we use the left-most binding and cutting strategy as discussed above. We remark that there is a general ij strategy, that of binding the i^{th} variable from the left and cutting the j^{th} variable from the left.

3.3 Variable sharing

Variable sharing is a central notion which allows linear dependency to be set up. In fact, this notion is already implicit in our definition (3.1) of linear occurrence. The $\lambda\Lambda_\kappa$-calculus uses a function κ which implements the degree of sharing of variables between functions and their arguments.

We define κ by considering the situation when either of the two contexts Γ or Δ are of the form $\ldots, x{:}A$ or $\ldots, x{:}A, y{:}Bx$. The only case when the two declarations of $x{:}A$ are not identified with each other is when both Γ and Δ are of the form $\ldots, x{:}A, y{:}Bx$.

Definition 3.5 *The function κ is defined for the binary, multiplicative $(A\Lambda\mathcal{E})$, $(M\Lambda\mathcal{E})$ and (Cut) rules*

$$\frac{\Gamma \vdash_\Sigma U : \Lambda z{:}C.V \quad \Delta \vdash_\Sigma N{:}C \quad [\Xi'; \Gamma; \Delta] \quad \Xi = \Xi' \backslash \kappa(\Gamma, \Delta)}{\Xi \vdash_\Sigma UN : V[N/x]} \,(A\Lambda\mathcal{E}), (M\Lambda\mathcal{E})$$

$$\frac{\begin{array}{c}\Pi(\Delta[N/x]) \\ \vdots \\ \Delta, z{:}C, \Delta' \vdash_\Sigma U{:}V\end{array} \quad \Gamma \vdash_\Sigma N{:}C \quad [\Xi'; \Gamma; \Delta, \Delta'[N/x]] \quad \Xi = \Xi' \backslash \kappa(\Gamma, \Delta)}{\Xi \vdash_\Sigma (U{:}V)[N/z]} \,(\mathrm{Cut}).$$

For each $x{:}A$ occurring in both Γ and Δ, construct from right to left as follows:[4]

$$\kappa(\Gamma,\Delta) \;=\; \{\} \qquad\qquad \textit{if } lin(\Gamma)\cap lin(\Delta)=\emptyset$$

$$\kappa(\Gamma,\Delta) \;=\; \left\{ \begin{array}{ll} \{x{:}A \in lin(\Gamma)\cap lin(\Delta) \mid & \textit{either (i) there is no } y{:}B(x) \textit{ to} \\ & \quad \textit{the right of } x{:}A \textit{ in } \Gamma \\ & \textit{or (ii) there is no } y{:}B(x) \textit{ to} \\ & \quad \textit{the right of } x{:}A \textit{ in } \Delta \\ & \textit{or both (i) and (ii)}\} \end{array} \right\} \quad \textit{otherwise}$$

The second clause of the definition can also be stated as follows: in at least one of Γ and Δ there is no $y{:}B(x)$ to the right of the occurrence of $x{:}A$. This clause is needed to form a consistent type theory which allows the formation of sufficiently complex dependent types. By this, we mean types such as $\Lambda x_1{:}A_1 \ldots \Lambda x_n{:}A_n(x_1,\ldots,x_{n-1})\,.\,A$ in which the abstracting types depend upon previously abstracted variables. In binary rules, it can be that some variables must occur, in order to establish the well-formedness of types in each premise, in the contexts of both premises, and must occur only once in order to establish the well-formedness of types in the conclusion. However, it is possible for other variables occurring in both premises to play a role in the logical structure of the proof; these variables must be duplicated in the conclusion. These requirements are regulated by κ.

In the absence of sharing of variables, when the first clause only applies, we still obtain a useful linear dependent type theory, with a linear dependent function space but without the dependency of the abstracting A_is on the previously abstracted variables. For example, we use such a type theory to encode the dynamic semantics of ML with references in § 5 later.

With the definition of κ given above, we can consider the following example.

Example 3.2 *Suppose $A!\mathsf{Type}, c!A \multimap \mathsf{Type} \in \Sigma$. The we construct the following:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash_\Sigma c{:}A \multimap \mathsf{Type} \quad \dfrac{\vdash_\Sigma A{:}\mathsf{Type}}{x{:}A \vdash_\Sigma x{:}A} \quad (*)}{x{:}A \vdash_\Sigma cx{:}\mathsf{Type}}}{x{:}A, z{:}cx \vdash_\Sigma z{:}cx}}{x{:}A \vdash_\Sigma \lambda\, z{:}cx\,.z \;:\; \Lambda\, z{:}cx\,.cx} \quad \dfrac{\dfrac{\vdash_\Sigma c{:}A \multimap \mathsf{Type} \quad \dfrac{\vdash_\Sigma A{:}\mathsf{Type}}{x{:}A \vdash_\Sigma x{:}A} \quad (*)}{x{:}A \vdash_\Sigma cx{:}\mathsf{Type}}}{x{:}A, y{:}cx \vdash_\Sigma y{:}cx} \quad (**)}{x{:}A, y{:}cx \vdash_\Sigma (\lambda\, z{:}cx\,.z)y \;:\; cx}$$

The $(*)$ denotes the context join to get $x{:}A$. The $(**)$ side-condition is more interesting. First, the premise contexts are joined together to get $x{:}A, x{:}A, y{:}cx$. Then, κ removes the extra occurrence of $x{:}A$ and so restores the linearity constraint. A similar situation arises when the y is cut in for the z:

$$\dfrac{x{:}A, z{:}cx \vdash_\Sigma z{:}cx \quad x{:}A, y{:}cx \vdash_\Sigma y{:}cx \quad (**')}{x{:}A, y{:}cx \vdash_\Sigma (z{:}cx)[y/z]}$$

[4] Formally, $\kappa(\Gamma,\Delta)$ is defined recursively on the structure of Γ and Δ, read from right to left. We adopt the following informal notation for ease of expression.

The function κ is not required, *i.e.*, its use is vacuous, when certain restrictions of the $\lambda\Lambda_\kappa$-calculus type theory are considered. For instance, if we restrict type-formation to be entirely intuitionistic so that type judgements are of the form $!\Gamma \vdash_\Sigma A{:}\mathsf{Type}$, then we recover the $\{\Pi, \multimap, \&\}$-fragment of Cervesato and Pfenning's $\lambda^{\Pi\multimap\&\top}$ type theory [12]. Our fragment does not include $\top$, the unit of &; we will remark on this while stating the subject reduction property in § 3.5 later. Like the simple dependency case above, this restricted type theory is useful too; we use it to encode a fragment of intuitionistic linear logic in § 5 later.

3.4 Definitional equality

The definitional equality relation that we consider here is the β-conversion of terms at all three levels. The definitional equality relation, $\equiv$, between terms at each respective level is defined to be the symmetric and transitive closure of the parallel nested reduction relation, $\rightarrow$, defined in Table 4 below. We note that, in the β-rules, substitution is performed only for the bound occurrences of x. The transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$.

$$\frac{}{U \rightarrow U}\,(\rightarrow refl) \qquad \frac{A \rightarrow A' \quad M \rightarrow M'}{\lambda\, x{\in}A\,.M \rightarrow \lambda\, x{\in}A'\,.M'}\,(\rightarrow M\lambda)$$

$$\frac{A \rightarrow A' \quad K \rightarrow K'}{\Lambda\, x{\in}A\,.K \rightarrow \Lambda\, x{\in}A\,.K'}\,(\rightarrow K\Lambda) \qquad \frac{M \rightarrow M' \quad N \rightarrow N'}{MN \rightarrow M'N'}\,(\rightarrow M app)$$

$$\frac{A \rightarrow A' \quad B \rightarrow B'}{\Lambda\, x{\in}A\,.B \rightarrow \Lambda\, x{\in}A\,.B'}\,(\rightarrow A\Lambda) \qquad \frac{M \rightarrow M' \quad N \rightarrow N'}{(\lambda\, x{\in}A\,.M)N \rightarrow M'[N'/x]}\,(\rightarrow M\beta)$$

$$\frac{A \rightarrow A' \quad B \rightarrow B'}{\lambda\, x{\in}A\,.B \rightarrow \lambda\, x{\in}A'\,.B'}\,(\rightarrow A\lambda) \qquad \frac{M \rightarrow M' \quad N \rightarrow N'}{\langle M, N\rangle \rightarrow \langle M', N'\rangle}\,(\rightarrow M\&)$$

$$\frac{A \rightarrow A' \quad M \rightarrow M'}{AM \rightarrow A'M'}\,(\rightarrow A app) \qquad \frac{M \rightarrow M'}{\pi_i M \rightarrow \pi_i M'}\,(\rightarrow M\pi)$$

$$\frac{B \rightarrow B' \quad N \rightarrow N'}{(\lambda\, x{\in}A\,.B)N \rightarrow B'[N'/x]}\,(\rightarrow A\beta) \qquad \frac{M \rightarrow M'}{\pi_0\langle M, N\rangle \rightarrow M'}\,(\rightarrow M\pi_0)$$

$$\frac{A \rightarrow A' \quad B \rightarrow B'}{A\&B \rightarrow A'\&B'}\,(\rightarrow A\&) \qquad \frac{N \rightarrow N'}{\pi_1\langle M, N\rangle \rightarrow N'}\,(\rightarrow M\pi_1)$$

Table 4: Parallel nested reduction

We remark that while β-conversion is sufficient for our current purposes, we forsee little difficulty (other than that for the $\lambda\Pi$-calculus [13, 34]) in strengthening the definitional equality relationship by the η-rule.

3.5 Basic properties of the $\lambda\Lambda_\kappa$-calculus

In this section, we summarize the basic properties of the $\lambda\Lambda_\kappa$-calculus, the proofs of which can be obtained by adapting the techniques of [18]. Note that we are concerned here with just the basic, $\{\&, \Lambda, !\}$-fragment, of the type theory.

The choice of the reduction relation $\rightarrow$ allows us to prove confluency:

Lemma 3.1 (CR Property) *If $U \rightarrow^* U'_m$ and $U \rightarrow^* U''_n$, then there exists a $V_{n,m}$ such that $U'_m \rightarrow^* V_{n,m}$ and $U''_n \rightarrow^* V_{n,m}$.*

Proof The lemma is proven in several steps. First, we show that substitution is conserved under reduction. Then we prove, by induction on the sum of the lengths of the proofs $U \rightarrow U'$ and $U \rightarrow U''$, the diamond property: that if $U \rightarrow U'$ and $U \rightarrow U''$, then there exists a V such that $U' \rightarrow V$ and $U'' \rightarrow V$. CR then follows by an induction on the number of β-steps. $\square$

The following lemma analyses how a type assignment for an abstraction can be obtained. It is a specific (part 4) and specialized (for linearity) case of Barendregt's Generation Lemma for Pure Type Systems (PTS) [8].

Lemma 3.2 (Inversion) *If $\Gamma \vdash_\Sigma \lambda\, x{\in}A\,.U : \Lambda\, x{\in}A\,.V$, then $\Gamma, x{\in}A \vdash_\Sigma U{:}V$.*

Proof Consider a derivation of $\lambda\, x{\in}A\,.U : \Lambda\, x{\in}A\,.V$. The conversion rules do not change the term $\lambda\, x{\in}A\,.U$. We follow the branch of the derivation until the term $\lambda\, x{\in}A\,.U$ is introduced for the first time. This can be done by an abstraction rule. The conclusion of the abstraction rule is

$$\Gamma \vdash_\Sigma \lambda\, x{\in}A\,.U : \Lambda\, x{\in}B\,.W$$

with $\Lambda\, x{\in}A\,.V \equiv \Lambda\, x{\in}B\,.W$. The statement of the lemma follows by inspection of the abstraction. $\square$

We remark that, given the next theorem, the above lemma could allow weakening or contraction in the intuitionistic parts of Γ.

We recall our earlier comments, in § 2, regarding how the consideration of a particular language allows us to admit certain structural rules. The next theorem details this for the $\lambda\Lambda_\kappa$-calculus. We comment on the form of the admissible structural rules. The exchange and contraction rules are inherited from dependent and linear type theory, respectively. The rule for weakening requires that the context for proving the well-typedness of A is entirely intuitionistic. The rule for dereliction requires the derelicting of the free variables in the linear type too. Cut comes in two forms, one for cutting a linear variable and one for cutting an exponential one. The rules are read according to the current binding strategy. The extra side-conditions for exchange enforce the context well-formedness in accordance with the left-most binding strategy.

Theorem 3.1 (Structural Admissibilities) *The following structural rules are admissible:*

1. Exchange*: If* $\Gamma, x{\in}A, y{\in}B, \Delta \vdash_\Sigma U{:}V$*, then* $\Gamma, y{\in}B, x{\in}A, \Delta \vdash_\Sigma U{:}V$*, provided* $x \notin FV(B)$*,* $y \notin FV(A)$ *and* $\Gamma \vdash_\Sigma B{:}\mathsf{Type}$*;*
2. Weakening*: If* $\Gamma \vdash_\Sigma U{:}V$ *and* $!\Delta \vdash_\Sigma A{:}\mathsf{Type}$*, then* $\Xi, x!A \vdash_\Sigma U{:}V$*, where* $[\Xi; \Gamma; !\Delta]$*;*
3. Dereliction*: If* $\Gamma, x{:}A \vdash_\Sigma U{:}V$*, then* $\Gamma', x!A \vdash_\Sigma U{:}V$*, where* Γ' *is* Γ *in which the free variables of* A *have been derelicted too;*
4. Contraction*: If* $\Gamma, x!A, y!A \vdash_\Sigma U{:}V$*, then* $\Gamma, x!A \vdash_\Sigma (U{:}V)[x/y]$*;*
5. Cut*: If* $\Pi(\Delta[N/x])$ *is a sub-proof of* $\Delta, x{:}A, \Delta' \vdash_\Sigma U{:}V$ *and* $\Gamma \vdash_\Sigma N{:}A$*, then* $\Xi \vdash_\Sigma (U{:}V)[N/x]$*, where* $[\Xi'; \Gamma; \Delta, \Delta'[N/x]]$ *and* $\Xi = \Xi' \backslash \kappa(\Gamma, \Delta)$*;*
6. Cut!*: If* $\Delta, x!A, \Delta' \vdash_\Sigma U{:}V$ *and* $!\Gamma \vdash_\Sigma N{:}A$*, then* $\Xi \vdash_\Sigma (U{:}V)[N/x]$*, where* $[\Xi; \Gamma; \Delta, \Delta'[N/x]]$*.*

Proof By induction on the structure of the proof of the premises. We illustrate a few representative cases.
Admissibility of Dereliction:

1. $(MVar)$. $\Gamma, x{:}A \vdash_\Sigma x{:}A$ because $\Gamma \vdash_\Sigma A{:}\mathsf{Type}$. In the case where there are no free variables in A, we just use $(MVar!)$ to get $\Gamma, x!A \vdash_\Sigma x{:}A$.

   Now suppose $y \in FV(A)$. If y is of an exponential type then we are done. Otherwise, we consider the step where the y is introduced and replace the application of $(MVar)$ with $(MVar!)$;
2. $(M\lambda\Lambda\mathcal{I})$. $\Gamma, x{:}A \vdash_\Sigma \lambda\, y{:}B\,.M \;:\; \Lambda\, y{:}B\,.C$ because $\Gamma, x{:}A, y{:}B \vdash_\Sigma M{:}C$. There are two sub-cases, depending on how the linear variables $x{:}A$ is consumed.

   (a) $x \notin FV(B)$. That is, x has a linear occurrence in $M{:}C$. An Exchange puts the judgment in the form where we can apply the induction hypothesis. So we get $\Gamma', y{:}B, x!A \vdash_\Sigma M{:}C$, where Γ' is Γ with the $z \in LFV(A)$ are derelicted too. Then we apply Exchange and $(M\lambda\Lambda\mathcal{I})$ to get $\Gamma', x!A \vdash_\Sigma \lambda\, y{:}B\,.M \;:\; \Lambda\, y{:}B\,.C$.

   (b) $x \in FV(B)$. This case is argued similarly to the one before.

Admissibility of Cut:

1. $(MVar)$. $\Delta, x{:}A \vdash_\Sigma x{:}A$ because $\Delta \vdash_\Sigma A$:Type. We have to show that $\Xi \vdash_\Sigma (x{:}A)[M/x]$. This follows from the assumption $\Gamma \vdash_\Sigma M{:}A$ with the $\Xi = \Xi' \backslash \kappa(\Gamma, \Delta)$ side-condition needed to remove the excess occurrences in Γ which type A ;
2. $(M\Lambda\mathcal{E})$. $\Delta, x{:}A, \Delta' \vdash_\Sigma MN \;:\; C[N/y]$ because $\Phi \vdash_\Sigma M \;:\; \Lambda\, y{:}B\,.C$ and $\Psi \vdash_\Sigma N{:}B$, with $[\Delta, x{:}A, \Delta'; \Phi; \Psi]$. There are two sub-cases to consider, depending on whether or not x is a shared variable, as regulared by κ.

   (a) For the non-sharing case, the proof proceeds according to which context the $x{:}A$ is sent to. So suppose $x{:}A \in \Phi$ (the case for $x{:}A \in \Psi$ is similar). By induction hypothesis we get $\Upsilon \vdash_\Sigma (M \;:\; \Lambda\, y{:}B\,.C)[M/x]$, where $[\Upsilon; \Phi_0, \Phi_n[M/x]; \Gamma]$ and $\Phi = \Phi_0, x{:}A, \Phi_n$. Then we use $(M\Lambda\mathcal{E})$ to construct $\Xi \vdash_\Sigma (MN : (C[N/z]))[M/x]$, with $[\Xi; \Upsilon; \Psi]$. We have elided the details of substitution.

(b) For the sharing case, $x{:}A$ will be sent to both branches. That is, $x{:}A \in \Phi$ and $x{:}A \in \Psi$. The argument then proceeds as above, using the induction hypothesis on each branch. $\Box$

The following unicity properties are desirable from both a type-theoretic and pragmatic perspective:

Lemma 3.3 (Unicity of Types and Kinds, UT) *If $\Gamma \vdash_\Sigma U{:}V$ and $\Gamma \vdash_\Sigma U{:}V'$, then $V \equiv V'$.*

Proof By induction on the structure of the proof of the premises. We omit the details. $\Box$

Lemma 3.4 (Extended Unicity of Domains, EUD) *If $\lambda\, x{\in}A\,.U$ inhabits $\Lambda\, x{\in}B\,.V$, then $A \equiv B$.*

Proof CR determines, up to definitional equality, the term $\lambda\, x{\in}A\,.U$. UT does the same for the type $\Lambda\, x{\in}B\,.V$. This is sufficient to allow us to infer the result. $\Box$

Our definition (3.1) of linear occurrence is motivated by the desire for the type theory to have the subject reduction property. However, it is important that no linear variables are lost during reduction. We stop to consider this problem before proceeding to show the property. Consider the following instance of application:

$$\frac{\begin{array}{c}\vdots\\ \Gamma \vdash_\Sigma \lambda\, x{:}A\,.y \,:\, A \multimap B\end{array} \qquad \begin{array}{c}\vdots\\ z{:}A \vdash_\Sigma z{:}A\end{array}}{\Gamma, z{:}A \vdash_\Sigma (\lambda\, x{:}A\,.y)z \,:\, B} \tag{1}$$

We suppose that the type of the function is $A \multimap B$. After a β-reduction, we have $\Gamma, z{:}A \vdash_\Sigma y{:}B$, which leaves the $z{:}A$ hanging. Now, we supposed that $\Gamma \vdash_\Sigma \lambda\, x{:}A\,.y \,:\, A \multimap B$ be provable. By inversion, we must then have $\Gamma, x{:}A \vdash_\Sigma y{:}B$ provable. By our definition of linear occurrence, this can be so for one (or both) of the following reasons:

1. $x \in FV(y)$, which is not true in this case (but, in general, in simple types we may have a sufficiently complex M for all to be well);
2. $x \in FV(B)$, so the x is consumed by the B. That is, the type of the function $\lambda\, x{:}A\,.y$ is not $A \multimap B$ but rather $\Lambda\, x{:}A\,.B(x)$. So, in (1), the conclusion of the application is of the form $\Gamma, z{:}A \vdash_\Sigma (\lambda\, x{:}A\,.y)z \,:\, B[z/x]$ and hence we still have a linear occurrence of z.

So it follows that a situation as simple as (1), with the loss of an occurrence of a variable from the succedent, cannot arise in the type theory.

The subject reduction property is proved for $\rightarrow_1$, the one-step reduction relation in the basic type theory. ($\rightarrow^*$ and $\rightarrow_1^*$ define the same relation.)

Lemma 3.5 (Subject Reduction) *If* $\Gamma \vdash_\Sigma U{:}V$ *and* $U \rightarrow_1 U'$, *then* $\Gamma \vdash_\Sigma U'{:}V$.

Proof By simultaneous induction on the structure of the proof of the premises. The two main cases are when the last step of the typing derivation is either rule $(M\Lambda\mathcal{E})$ or $(M\Lambda!\mathcal{E})$; and the last reduction step is rule $(\rightarrow M\beta)$. We consider the first of these cases. So, suppose

$$\Xi \vdash_\Sigma (\lambda\, x{:}A\,.M)N \;:\; B \quad \text{and} \quad (\lambda\, x{:}A\,.M)N \rightarrow_1 M[N/x]$$

and the first of these arises because

$$\Gamma \vdash_\Sigma \lambda\, x{:}A\,.M \;:\; \Lambda\, x{:}C\,.D \quad \text{and} \quad \Delta \vdash_\Sigma N{:}C$$

with $[\Xi';\Gamma;\Delta]$, $\Xi = \Xi' \backslash \kappa(\Gamma,\Delta)$ and $B = D[N/x]$.

By Lemma 3.2, we have that $\Gamma, x{:}A \vdash_\Sigma M{:}D$. By Lemma 3.4 we have that $A \equiv C$ and by the $(M \equiv)$ rule we have that $\Delta \vdash_\Sigma N{:}A$. Then we use the Cut rule to construct

$$\frac{\Gamma, x{:}A \vdash_\Sigma M{:}D \quad \Delta \vdash_\Sigma N{:}A \quad [\Xi';\Gamma;\Delta] \quad \Xi = \Xi'\backslash\kappa(\Gamma,\Delta)}{\Xi \vdash_\Sigma (M{:}D)[N/x]}$$

The conclusion is $M[N/x]{:}D[N/x]$. $\square$

The type theory extended with $1{:}\top$, the unit of &, does not have the stated subject reduction property. The reason is illustrated by the following derivation, in which we assume that $A!\mathsf{Type} \in \Sigma$:

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash_\Sigma \Gamma, x{:}A \text{ context}}}{\dfrac{\Gamma, x{:}A \vdash_\Sigma 1{:}\top}{\Gamma \vdash_\Sigma \lambda\, x{:}A\,.1 \;:\; A \multimap \top}} \qquad \dfrac{\vdots}{z{:}A \vdash_\Sigma z{:}A}}{\Gamma, z{:}A \vdash_\Sigma (\lambda\, x{:}A\,.1)z \;:\; \top}$$

After a β-reduction we have $\Gamma, z{:}A \vdash_\Sigma 1{:}\top$, and the z is left hanging. However, we conjecture that such an extended type theory will have a weaker form of subject reduction, in which $\Gamma' \subseteq \Gamma$. The conjecture arises from a consideration of cut-elimination in linear type theory in the presence of $1{:}\top$. The point is that β-reduction in an example such as the one above effects not only terms but also proofs and so should therefore properly be considered an inference rule of the type theory.

All reduction sequences in the type theory terminate:

Theorem 3.2 (Strong Normalization) *All valid terms are strongly normalizing:*

1. *If* $\Gamma \vdash_\Sigma K$ Kind*, then* K *is strongly normalizing;*
2. *If* $\Gamma \vdash_\Sigma U{:}V$*, then* U *is strongly normalizing.*

The proof idea, again, a variation on an argument in [18], is to define a faithful "dependency- and linearity-less" translation τ of kinds and type families to S, the set of simple types constructed by $\times$ and $\rightarrow$ over a given base type ω, and $|.|$, of type families and objects to $\Lambda(K)$, the set of untyped λ-terms of over a set of constants $K = \{\pi_\sigma | \sigma \in S\}$. Let $\vdash^\lambda$ denote type assignment following Curry (with products) together with the infinite set of rules for K

$$\vdash^\lambda \pi_\sigma{:}\omega \rightarrow (\sigma \rightarrow \omega) \rightarrow \omega$$

for each $\sigma \in \Sigma$.

The translation embeds $\lambda\Lambda_\kappa$ into such Curry-typable terms of the untyped λ-calculus in a structure preserving way. The dependency aspect is lost by, for instance, forgetting about the variable x in the term $\Lambda\, x{:}A\,.B$. The linear aspect is lost by translating linear and exponential variables in exactly the same manner.

Definition 3.6 (Translation to simple types)

$$
\begin{array}{lcll}
\tau : K \rightarrow S & & & \\
\tau(\mathsf{Type}) & = & \omega & \\
\tau(\Lambda\, x{\in}A\,.K) & = & \tau(A) \rightarrow \tau(K) & \\
 & & & \\
\tau : A \rightarrow S & & & \\
\tau(a) & = & a & \\
\tau(\Lambda\, x{\in}A\,.B) & = & \tau(A) \rightarrow \tau(B) & \\
\tau(\lambda\, x{\in}A\,.B) & = & \tau(B) & \\
\tau(AM) & = & \tau(A) & \\
\tau(A\&B) & = & \tau(A) \times \tau(B) & \\
 & & & \\
|.| : A \rightarrow \Lambda(K) & & & \\
|a| & = & a & \\
|\Lambda\, x{\in}A\,.B| & = & \pi_{\tau(A)}|A|(\lambda x.|B|) & \\
|\lambda\, x{\in}A\,.B| & = & (\lambda y.\lambda x.|B|)|A| & y \notin FV(B) \\
|AM| & = & |A||M| & \\
|A\&B| & = & |A| \times |B| & \\
 & & & \\
|.| : M \rightarrow \Lambda(K) & & & \\
|c| & = & c & \\
|x| & = & x & \\
|\lambda\, x{\in}A\,.M| & = & (\lambda y.\lambda x.|M|)|A| & y \notin FV(M) \\
|MN| & = & |M||N| & \\
|\langle M, N\rangle| & = & \langle |M|, |N| \rangle & \\
|\pi_i M| & = & \pi_i |M| & i \in \{0, 1\}
\end{array}
$$

We note some minor technicalities to do with the translation. The translation of $\Gamma \vdash_\Sigma U{:}V$ is given by $\tau(\Sigma), \tau(\Gamma) \vdash^\lambda |U|{:}\tau(V)$; the signature and context are dealt with in the obvious way. That is, $\tau(\langle\rangle) = \langle\rangle$; $\tau(\Gamma, x{\in}A) = \tau(\Gamma), x{:}\tau(A)$ and $\tau(\Sigma, c!A) = \tau(\Sigma), c{:}\tau(A)$, *etc.* The binding strategy is utilized to give occurrences unique names.

The next two lemmata show that the translation is sufficiently faithful. We will abuse notation somewhat and take the symbols such $\equiv$, $\rightarrow$, $[M/x]$, *etc.* to mean similar relations in the simply-typed λ-calculus.

Lemma 3.6

1. *If $A \equiv A'$, then $\tau(A) = \tau(A')$.*
2. *If $K \equiv K'$, then $\tau(K) = \tau(K')$.*

Proof By induction on the structure of the proofs that if $A \rightarrow A'$, then $\tau(A) = \tau(A')$ and that if $K \rightarrow K'$ then, $\tau(K) = \tau(K')$. The lemma follows from the fact that $\equiv$ is defined to be the symmetric and transitive closure of $\rightarrow$. We omit the details. $\square$

Lemma 3.7

1. $|M[N/x]| = |M|[|N|/x]$.
2. $|B[N/x]| = |B|[|N|/x]$.

Proof By induction on the structure of M and B respectively. We omit the details. $\square$

The next lemma shows the consistency of the translation.

Lemma 3.8

1. *If $\Gamma \vdash_\Sigma A{:}K$, then $\tau(\Sigma), \tau(\Gamma) \vdash^\lambda |A|{:}\tau(K)$.*
2. *If $\Gamma \vdash_\Sigma M{:}A$ then, $\tau(\Sigma), \tau(\Gamma) \vdash^\lambda |M|{:}\tau(A)$.*

Proof By induction on the structure of the proof of the premises. We illustrate the argument with a few representative cases.

1. (Mc). $!\Gamma \vdash_\Sigma c{:}A$ because $\vdash_\Sigma !\Gamma$ $\mathsf{context}$, with $c!A \in \Sigma$. Trivial, as $\tau(A)$ is always a well-formed type.
2. $(M\lambda\Lambda\mathcal{I})$. $\Gamma \vdash_\Sigma \lambda\, x{:}A\,.M \,:\, \Lambda\, x{:}A\,.B$ because $\Gamma, x{:}A \vdash_\Sigma M{:}B$. By induction hypothesis we have that

$$\tau(\Sigma), \tau(\Gamma), x{:}\tau(A) \vdash^\lambda |M| \,:\, \tau(B)$$

Therefore

$$\tau(\Sigma), \tau(\Gamma) \vdash^\lambda \lambda x.|M| \,:\, \tau(A) \rightarrow \tau(B)$$

and

$$\tau(\Sigma), \tau(\Gamma) \vdash^\lambda (\lambda y.\lambda x.|M|)|A| \,:\, \tau(A) \rightarrow \tau(B)$$

which is $\tau(\Sigma), \tau(\Gamma) \vdash^\lambda |\lambda\, x{:}A\,.M| \,:\, \tau(\Lambda\, x{:}A\,.B)$.

3. $(M\Lambda\mathcal{E})$. $\Xi \vdash_\Sigma MN : B[N/x]$ because $\Gamma \vdash_\Sigma M : \Lambda\, x{:}A\,.K$ and $\Delta \vdash_\Sigma N{:}A$, with $[\Xi';\Gamma;\Delta]$ and $\Xi = \Xi' \backslash \kappa(\Gamma,\Delta)$. By IH twice we have

$$\tau(\Sigma), \tau(\Gamma) \vdash^\lambda |M| : \tau(A) \to \tau(B)$$

and

$$\tau(\Sigma), \tau(\Delta) \vdash^\lambda |N| : \tau(A)$$

Then we construct

$$\dfrac{\dfrac{\dfrac{\tau(\Sigma), \tau(\Gamma) \vdash^\lambda |M| : \tau(A) \to \tau(B)}{\tau(\Sigma), \tau(\Gamma), \tau(\Delta) \vdash^\lambda |M| : \tau(A) \to \tau(B)}\,W}{\tau(\Sigma), \tau(\Xi) \vdash^\lambda |M| : \tau(A) \to \tau(B)}\,X,C \quad \dfrac{\dfrac{\tau(\Sigma), \tau(\Delta) \vdash^\lambda |N| : \tau(A)}{\tau(\Sigma), \tau(\Delta), \tau(\Gamma) \vdash^\lambda |N| : \tau(A)}\,W}{\tau(\Sigma), \tau(\Xi) \vdash^\lambda |N| : \tau(A)}\,X,C,}{\tau(\Sigma), \tau(\Xi) \vdash^\lambda (|M||N|) : \tau(B)}\,\mathrm{APP}$$

where the double line indicates a series of applications of the indicated rule. The weakenings (W) introduce $\tau(\Delta)$ and $\tau(\Gamma)$ into the left and right premises. The exchanges (X) and contractions (C) are used to eliminate duplicate (exponential, in the original type theory) variables. These are necessary so as to get the premises of the $\to$-elimination rule into additive form. The conclusion of the proof tree is $\tau(\Sigma), \tau(\Xi) \vdash^\lambda |MN| : \tau(B[N/x])$, as required. And $\tau(B) = \tau(B[N/x])$ as there is no type dependency in the simply-typed λ-calculus.

4. $(M\&\mathcal{I})$. $\Gamma \vdash_\Sigma \langle M, N\rangle : A\&B$ because $\Gamma \vdash_\Sigma M{:}A$ and $\Gamma \vdash_\Sigma N{:}B$. By induction hypothesis twice we have

$$\tau(\Sigma), \tau(\Gamma) \vdash^\lambda |M| : \tau(A)$$

and

$$\tau(\Sigma), \tau(\Gamma) \vdash^\lambda |N| : \tau(B)$$

The rule for $\times$-introduction then gives us that $\tau(\Sigma), \Gamma \vdash^\lambda \langle |M|, |N|\rangle : \tau(A) \times \tau(B)$, which is $\tau(\Sigma), \tau(\Sigma), \tau(\Gamma) \vdash^\lambda |\langle M, N\rangle| : \tau(A\&B)$. □

□

The extra combinatorial complexity of $\lambda\Lambda_\kappa$-calculus terms owing to the possibility of reductions within type labels is not lost by the translation.

Lemma 3.9

1. *If $A \to_1 A'$, then $|A| \to_1^+ |A'|$.*
2. *If $M \to_1 M'$, then $|M| \to_1^+ |M'|$,*

where $\to_1^+$ is the transitive closure of $\to_1$ for the untyped λ-calculus.

Proof By induction on the proof of $A \to_1 A'$ and $M \to_1 M'$. The only non-trivial cases arise when the last rule applied is one of the β-rules, or one of the Λ-rules. In the first case we have, for example,

$$|(\lambda\, x{:}A\,.M)N| \to_1^+ (\lambda\, x\,.|M|)|N| \to_1^+ |M|[|N|/x]$$

which is $\tau(M[N/x])$, by Lemma 3.7. In the second case, Lemma 3.6 suffices for the result. $\square$

We can now give the proof of strong normalization. Suppose there was an infinite reduction in the $\lambda\Lambda_\kappa$-calculus. Then this would be translated into a reduction in the simply-typed λ-calculus. As the translation is faithful, the reduction in the simply-typed λ-calculus would be infinite too. But this cannot be so, as the simply-typed λ-calculus with pairing is known to be strongly normalizing [15]. So there cannot be an infinite reduction in the $\lambda\Lambda_\kappa$-calculus.

Predicativity arises as a corollary of Theorem 3.2. Finally, we have:

Theorem 3.3 (Decidability) *All assertions of the $\lambda\Lambda_\kappa$-calculus are decidable.*

Proof The argument is the same as for $\lambda\Pi$. We observe that, firstly, the complexity of the proof of a judgement is determined by proofs of strictly smaller measure; and, secondly, the form of a judgement completely determines its proof. The main method underlying this argument involves replacing the conversion rules with a (better behaved) normal-order reduction strategy. $\square$

3.6 Related systems

In this section, we briefly compare the $\lambda\Lambda_\kappa$-calculus to the appropriate fragments of other linear type theories. Abramsky's [1] and Benton's [9] linear type theories are in propositions-as-types correspondence with a propositional ILL. Our concern is with a predicate ILL. Consider a linear version of the Barendregt cube, displayed in so-called standard orientation. Then Abramsky's and Benton's type theories correspond to the $\lambda \to$ and $\lambda 2$ nodes; our type theory corresponds to the λP node.

Another difference between Abramsky's and Benton's studies and ours is one of motivation; we study the $\lambda\Lambda_\kappa$-calculus as the language of a logical framework. A comparison with Cervesato and Pfenning's work [12] is, perhaps, more appropriate in this case. Their work claims to be inspired by our study's origins [29]. We remark that the description of the LLF framework lacks an account of a notion of representation and that the $\lambda^{\Pi \multimap \& \top}$ type theory is a fragment of the $\lambda\Lambda_\kappa$-calculus lacking, *inter alia*, linear dependent function types. To be precise, the $\{\Pi, \multimap, \&\}$-fragment of $\lambda^{\Pi \multimap \& \top}$ can be recovered by restricting type-formation to be intuitionistic, with the consequence that the use of κ is vacuous. We have noted this restricted type theory in § 3.3.

The key point to make in these comparisons is as follows. It is the construction of the linear dependent function space that necessitates an investigation into various structural properties. These are then explicated by the technical device of multiple occurrences. If our concern were non-dependent $(\multimap, \rightarrow)$ or intuitionistic dependent (Π) function spaces, then we could do without such analyses.

4 Conservativity

In this section we show that RLF is a conservative extension of LF. We will need the following translation between the $\lambda\Pi$- and $\lambda\Lambda_\kappa$-calculi. This is reminiscent of the translation of IL into ILL which maps $\phi \rightarrow \psi$ to $!\phi \multimap \psi$ [16].

Definition 4.1 ($\ulcorner - \urcorner : \lambda\Pi \rightarrow \lambda\Lambda_\kappa$) *We first define a translation for signatures and contexts. The clauses capture the intuitionistic–linear distinction between the two languages; the image is always of an exponential type.*

$$\ulcorner \langle\rangle \urcorner \;=\; \langle\rangle \qquad \begin{array}{rcl} \ulcorner \Sigma, c{:}U \urcorner & = & \ulcorner \Sigma \urcorner, c!U \\ \ulcorner \Gamma, x{:}A \urcorner & = & \ulcorner \Gamma \urcorner, x!A \end{array}$$

For the succedent of the typing judgement, $\ulcorner - \urcorner$ is defined by induction on the structure of the conclusion. We give only the cases for typed objects, $M{:}A$; the other cases are similar.

$$\begin{array}{rcl} \ulcorner c{:}A \urcorner & = & c{:}A \\ \ulcorner x{:}A \urcorner & = & x{:}A \end{array} \qquad \begin{array}{rcl} \ulcorner \lambda\, x{:}A\,.M \,:\, \Pi\, x{:}A\,.B \urcorner & = & \lambda\, x!A\,.\ulcorner M \urcorner \,:\, \Lambda\, x!A\,.\ulcorner B \urcorner \\ \ulcorner MN \,:\, B \urcorner & = & \ulcorner M \urcorner \ulcorner N \urcorner \,:\, \ulcorner B \urcorner \end{array}$$

The abstraction clause deals with the fact that the binding $x{:}A$ is a negative occurrence of a variable.

Now, our argument must capture the property of conservative extension not only at the level of the type theory but also at the level of a framework.[5] That is, we need to consider, for an arbitrary object-logic L, a translation from its definition in LF, via an encoding $\mathcal{E}$ and signature Σ_L, to its definition in RLF, via an encoding $\mathcal{E}'$ and signature Σ'_L, where both $\mathcal{E}$ and $\mathcal{E}'$ are standard judgements-as-types encodings.

Lemma 4.1 (Conservativity) *Let L be an object-logic. Let $\mathcal{E}$ be a uniform encoding of L in LF. For every provable L-consequence $(X;\Delta) \vdash_L \phi$, if*

$$\mathcal{E}(X), \mathcal{E}(\Delta) \vdash^{\lambda\Pi}_{\Sigma_L} M{:}\mathcal{E}(\phi) \tag{2}$$

then there is a uniform encoding $\mathcal{E}'$

$$\mathcal{E}'(X), \mathcal{E}'(\Delta) \vdash^{\lambda\Lambda_\kappa}_{\Sigma'_L} M'{:}\mathcal{E}'(\phi).$$

[5] Conservativity at the level of the type theory is an immediate consequence of Definition 4.1.

Proof We define $\mathcal{E}'$ as follows:

$$\mathcal{E}'(X) \;=\; \ulcorner\mathcal{E}(X)\urcorner \qquad \mathcal{E}'(\Gamma) \;=\; \ulcorner\mathcal{E}(\Gamma)\urcorner \qquad \mathcal{E}'(M') \;=\; \ulcorner\mathcal{E}(M)\urcorner\ ,$$

where M and M' are proof-realizers for the proposition ϕ in assumption Δ.

We now have to show that $\mathcal{E}'$ is a uniform encoding. The proof is by induction on the structure of (2). An interesting case is weakening. So suppose, $\Gamma, \Delta \vdash^{\lambda\Pi}_{\Sigma_L} M{:}A$ because $\Gamma \vdash^{\lambda\Pi}_{\Sigma_L} M{:}A$. Translating the latter consequence into the $\lambda\Lambda_\kappa$-calculus gives us $!\Gamma \vdash^{\lambda\Lambda_\kappa}_{\Sigma'_L} M'{:}A$. This can be weakened to get $!(\Gamma, \Delta) \vdash^{\lambda\Lambda_\kappa}_{\Sigma'_L} M'{:}A$. From the definition of $\ulcorner - \urcorner$, this is the image of $\Gamma, \Delta \vdash^{\lambda\Pi}_{\Sigma_L} M{:}A$. Now, by assumption, $\mathcal{E}$ is a uniform encoding, so $\Gamma, \Delta \vdash^{\lambda\Pi}_{\Sigma_L} M{:}A$ is an image of some object-consequence. □

5 Example encodings

In this section, we illustrate several encodings in RLF. The intention is to bring out the essential characteristics of the $\lambda\Lambda_\kappa$-calculus language — the weak structural properties, linear dependent function space and variable sharing — which allow these encodings to be undertaken uniformly via the judgements-as-types mechanism. The object-logic syntax and inference rules are not considered to be consumable resources and are encoded as (intuitionistic) constants in the signature. We adopt the notational convention that the meta-logic be expressed in **bold** print, so as not to confuse it with the object-logic it encodes.

We state representation theorems for each of the three encodings we undertake. In order to do this, we need a notion of *canonical* (essentially, long $\beta\eta$-normal) form. The definitions and lemmata needed for the characterization of canonical forms in the $\lambda\Lambda_\kappa$-calculus are similar to that for the $\lambda\Pi$-calculus; we omit them from this presentation. In the following, we will often say that a function is a "compositional bijection"; this simply means that it is a bijection and commutes with substitution.

5.1 ILL

Our first encoding is that of the $\{\otimes, \&\}$-fragment of propositional intuitionistic linear logic (ILL). We will work through the ILL object-logic in slightly more detail than the others. In this encoding, we work with a restricted type theory in which type formation is entirely intuitionistic; we have discussed such a type theory in § 3.3 previously. Such a restriction picks out the system of Cervesato and Pfenning [12] from amongst the others.

The natural deduction style rules for this logic are given in Table 5 below and are taken from [38]. The lower-case Greek letters ϕ, ψ, χ range over propositions of the ILL object-logic. For the rest of this sub-section, $i \in \{0, 1\}$.

$$\dfrac{\begin{array}{cc}\Gamma & \Delta\\ \vdots & \vdots\\ \phi & \psi\end{array}}{\phi\otimes\psi}\,(\text{TENSOR-I})\qquad \dfrac{\begin{array}{cc}\Gamma & \Delta,[\phi,\psi]\\ \vdots & \vdots\\ \phi\otimes\psi & \chi\end{array}}{\chi}\,(\text{TENSOR-E})\qquad \dfrac{\begin{array}{cc}\Gamma & \Gamma\\ \vdots & \vdots\\ \phi & \psi\end{array}}{\phi\&\psi}\,(\text{WITH-I})\qquad \dfrac{\begin{array}{c}\Gamma\\ \vdots\\ \phi_0\&\phi_1\end{array}}{\phi_i}\,(\text{WITH-E}_i)$$

Table 5: A fragment of ILL

The signature Σ_{ILL} begins with the declarations $\iota!\mathsf{Type}$ and $o!\mathsf{Type}$ to represent the syntactic categories of individuals and propositions of ILL. Next, each of the two formula-constructors are declared as constants in the signature Σ_{ILL}:

$$\otimes\,!\,o \multimap o \multimap o \qquad\qquad \&\,!\,o \multimap o \multimap o\ .$$

Terms (formulae) are encoded by a function $\mathcal{E}_X$ which maps terms (formulae) with free variables in X to terms of type ι (o) in Σ_{ILL}, Γ_X:

$$\mathcal{E}_X(\phi\otimes\psi) \;=\; \otimes\ \mathcal{E}_X(\phi)\ \mathcal{E}_X(\psi) \qquad\qquad \mathcal{E}_X(\phi\&\psi) \;=\; \&\ \mathcal{E}_X(\phi)\ \mathcal{E}_X(\psi)\ .$$

There is one basic judgement, the judgement that the formula ϕ has a proof, $\vdash_{ILL} \phi\ proof$. This is represented by declaring the constant $proof!o \multimap \mathsf{Type}$ in the signature. A proof of a formula ϕ is represented by a term of type $proof(\mathcal{E}(\phi))$.

The multiplicative operator $\multimap$ is used to represent the inference in the object-logic. It is also used – as a curried version of a meta-logical multiplicative conjunction – to combine the representation of the premises of the $\otimes$ rules, which are represented by the following declarations in the signature Σ_{ILL}:

$$\begin{array}{rcl}\text{TENSOR-I} & ! & \mathbf{\Lambda}\,\phi,\psi!o\,.\,proof(\phi)\multimap proof(\psi)\multimap proof(\otimes(\phi,\psi))\\ \text{TENSOR-E} & ! & \mathbf{\Lambda}\,\phi,\psi,\chi!o\,.\,proof(\otimes(\phi,\psi))\multimap\\ & & \quad (proof(\phi)\multimap proof(\psi)\multimap proof(\chi))\multimap proof(\chi)\ .\end{array}$$

We need the distinguished additive operator & to represent the additive rules. An alternative might be to use an additive function space (as described in [38], for instance), although it would appear that such a connective forces contexts to be more complex structures. Recall, $i\in\{0,1\}$ in this sub-section.

$$\begin{array}{rcl}\text{WITH-I} & ! & \mathbf{\Lambda}\,\phi,\psi!o\,.\,proof(\phi)\ \&\ proof(\psi)\multimap proof(\&(\phi,\psi))\\ \text{WITH-E}_i & ! & \mathbf{\Lambda}\,\phi_0,\phi_1!o\,.\,proof(\&(\phi_0,\phi_1))\multimap proof(\phi_i)\end{array}$$

Valid proofs of ILL are labelled trees with the constraint that assumption packets contain exactly one proposition and all such packets are uniquely labelled [10]. A valid proof Π of ϕ, with respect to a proof context $(X;\Delta)$,[6] is

[6] As usual, linearity is at the level of *propositions*-as-types; the set of variables X is implicitly !-ed.

denoted by the assertion $(X;\Delta) \vdash \Pi{:}\phi$, where X is a finite set of variables of first-order logic, Δ is the list of uniquely labelled assumptions $\{\xi_1{:}\phi_1, \ldots, \xi_n{:}\phi_n\}$, and $FV(dom(\Delta)) \subseteq X$. We remark that the treatment of the ILL quantifiers in RLF is essentially the same as that in LF. The rules for proving assertions of the form $(X;\Delta) \vdash \Pi{:}\phi$ are given in Table 6 below.

$$\frac{}{(X;\xi{:}\phi) \vdash \mathrm{HYP}_\phi(\xi){:}\phi}$$

$$\frac{(X;\Delta) \vdash \Pi{:}\phi \quad (X;\Delta') \vdash \Pi'{:}\psi}{(X;\Delta,\Delta') \vdash \text{TENSOR-I}_{\phi,\psi}(\Pi,\Pi'){:}\phi \otimes \psi} \qquad \frac{(X;\Delta) \vdash \Pi{:}\phi \otimes \psi \quad (X;\Delta',\xi{:}\phi,\xi'{:}\psi) \vdash \Pi'{:}\chi}{(X;\Delta,\Delta') \vdash \text{TENSOR-E}_{\phi,\psi,\chi}(\Pi,\xi,\xi'{:}\Pi'){:}\chi}$$

$$\frac{(X;\Delta) \vdash \Pi{:}\phi \quad (X;\Delta) \vdash \Pi'{:}\psi}{(X;\Delta) \vdash \text{WITH-I}_{\phi,\psi}(\Pi,\Pi'){:}\phi\&\psi} \qquad \frac{(X;\Delta) \vdash \Pi{:}\phi_0\&\phi_1}{(X;\Delta) \vdash \text{WITH-E}_{i\ \phi_0,\phi_1}(\Pi){:}\phi_i}$$

Table 6: Some valid proof expressions of ILL

The encoding function $\mathcal{E}_{(X;\Delta)}$ can be defined to encode proofs of ILL. The two $\otimes$ cases, for instance, are as follows:

$$\begin{array}{rcl}
\mathcal{E}_{(X;\Delta,\Delta')}(\text{TENSOR-I}_{\phi,\psi}(\Pi,\Pi')) & = & \text{TENSOR-I}\ \mathcal{E}_X(\phi)\ \mathcal{E}_X(\psi)\ \mathcal{E}_{(X;\Delta)}(\Pi) \\
 & & \mathcal{E}_{(X;\Delta')}(\Pi') \\
\mathcal{E}_{(X;\Delta,\Delta')}(\text{TENSOR-E}_{\phi,\psi,\chi}(\Pi,\xi,\xi'{:}\Pi')) & = & \text{TENSOR-E}\ \mathcal{E}_X(\phi)\ \mathcal{E}_X(\psi)\ \mathcal{E}_X(\chi) \\
 & & \mathcal{E}_{(X;\Delta)}(\Pi) \\
 & & \lambda\xi{:}proof(\mathcal{E}_X(\phi)). \\
 & & \lambda\xi'{:}proof(\mathcal{E}_X(\psi)).\mathcal{E}_{(X;\Delta')}(\Pi')
\end{array}$$

A proof context $(X;\Delta)$, with $X = \{x_1, \ldots, x_m\}$ and $\Delta = \{\xi_1{:}\phi_1, \ldots, \xi_n{:}\phi_n\}$, is mapped to $\Gamma_{X,\Delta} = x_1!\iota, \ldots, x_m!\iota, \xi_1{:}proof(\mathcal{E}(\phi_1)), \ldots, \xi_n{:}proof(\mathcal{E}(\phi_n))$.

The encoding basically illustrates the propositions-as-types correspondence for a $\{\otimes, \&\}$-fragment of ILL. So we can expect a strong representation theorem.

Theorem 5.1 (Representation for ILL) *The encoding functions are compositional bijections. That is, for every ILL-formula ϕ:*

1. $X \vdash_{ILL} \phi$ *if and only if* $\Gamma_X \vdash_{\Sigma_{ILL}} \mathcal{E}_X(\phi){:}\iota$;
2. $(X;\Delta) \vdash_{ILL} \Pi{:}\phi$ *if and only if* $\Gamma_{X,\Delta} \vdash_{\Sigma_{ILL}} M_\Pi{:}proof(\mathcal{E}_X(\phi))$,

where Π is an ILL proof-object and M_Π is a canonical object of the $\lambda\Lambda_\kappa$-calculus.

Proof The encoding functions $\mathcal{E}_X$ and $\mathcal{E}_{(X;\Delta)}$ are clearly injective. Surjectivity is established by defining decoding functions $\mathcal{D}_X$ and $\mathcal{D}_{(X;\Delta)}$ which are leftinverse to $\mathcal{E}_X$ and $\mathcal{E}_{(X;\Delta)}$:

$$\begin{array}{rcl} \mathcal{D}_X(\otimes\ M_1\ \ M_2) & = & \mathcal{D}_X(M_1)\ \otimes\ \mathcal{D}_X(M_2) \\ \mathcal{D}_X(\&\ M_1\ \ M_2) & = & \mathcal{D}_X(M_1)\ \&\ \mathcal{D}_X(M_2) \end{array}$$

$$\begin{array}{rcl} \mathcal{D}_{(X;\Delta,\Delta')}\left(\begin{array}{l} \text{TENSOR-I}\ M_1\ M_2 \\ P_1\ P_2 \end{array}\right) & = & \text{TENSOR-I}_{\mathcal{D}_{(X;\Delta)}(M_1),\mathcal{D}_{(X;\Delta')}(M_2)}(\Pi_1,\Pi_2) \\ \mathcal{D}_{(X;\Delta,\Delta')}\left(\begin{array}{l} \text{TENSOR-E}\ \ M_1\ \ M_2 \\ M_3\ \ P_1 \\ \lambda\,\xi{:}proof(M_1)\,. \\ \lambda\,\xi'{:}proof(M_2)\,.P_2 \end{array}\right) & = & \begin{array}{l} \text{TENSOR-E}_{\mathcal{D}_{(X;\Delta)}(M_1),\mathcal{D}_{(X;\Delta)}(M_2),\mathcal{D}_{(X;\Delta')}(M_3)} \\ \quad(\Pi_1,\xi,\xi'{:}\Pi_2) \end{array} \\ & & \begin{array}{l} where\ \ \Pi_1 = \mathcal{D}_{(X;\Delta)}(P_1) \\ and\ \ \Pi_2 = \mathcal{D}_{(X;\Delta')}(P_2) \end{array} \end{array}$$

That the decoding functions are total and well-defined follows from the definition of canonical forms and the signature. By induction on formulae and proof expressions, respectively, we get $\mathcal{D}_X(\mathcal{E}_X(\phi)) = \phi$ and $\mathcal{D}_{(X;\Delta)}(\mathcal{E}_{(X;\Delta)}(\Pi)) = \Pi$. Again, by a similar induction, we get that the encoding commutes with substitution. $\square$

The encoding can be extended to deal with a $\{\top,\otimes,\oplus,\multimap,1\}$-fragment of propositional ILL. The representation of the ILL units forces the design of the type theory. A meta-logical $\top$ is required to directly represent the object-logic $\top$; linearity constraints in the type theory mean that an encoding of $\Gamma \vdash_{ILL} \top$ would not be a valid $\lambda\Lambda_\kappa$-calculus judgement. The case for $\perp$ would be similar.

5.2 ML with references

Our second encoding is that of the programming language ML extended with references (MLR), a reworking of an example in Cervesato and Pfenning [11, 12]. In our reworking, we exploit the use of the Λ which is not available to Cervesato and Pfenning. Consequently, we are in the full $\lambda\Lambda_\kappa$-calculus type theory, in which κ's action is non-trivial.

The basic MLR logic judgement is of the form $S \triangleright K \vdash_{MLR} i \longrightarrow a$ which means: the program i is evaluated with the store S and continuation K and leaves an answer (a store–expression pair) a. The signature Σ_{MLR} begins with the declarations $store!\mathsf{Type}$, $cont!\mathsf{Type}$, $instr!\mathsf{Type}$ and $ans!\mathsf{Type}$ to represent the syntactic categories of store, continuations, expressions and answers. Evaluation is represented by the following declaration:

$$ev\,!\,cont \multimap instr \multimap answer \multimap \mathsf{Type}\ .$$

We are really only interested in the rule for evaluating re-assignment. This can be stated as follows:

$$\frac{S, c = v', S' \ \triangleright\ K \vdash_{MLR} \bullet \longrightarrow A}{S, c = v, S' \ \triangleright\ K \vdash_{MLR} \mathtt{ref}\, c := v' \longrightarrow A}\,,$$

where $\bullet$ is the MLR unit expression.

The ML memory is modelled by a set of (cell,expression)-pairs. Each such pair is represented by a linear hypothesis of type *contains* which holds a lvalue (the cell) and its rvalue (the expression).

$$cell\,!\,\mathsf{Type} \qquad exp\,!\,\mathsf{Type} \qquad contains\,!\,cell \multimap exp \multimap \mathsf{Type}$$

The rule for re-assignment evaluation is encoded as follows:

$$\begin{array}{lll} \text{EV-REASS} & ! & \mathbf{\Lambda}\, c!cell\,.\,\mathbf{\Lambda}\, v, v'!exp\,. \\ & & \quad ((contains\ c\ v') \multimap (ev\ K\ \bullet\ A)) \multimap \\ & & \quad (\mathbf{\Lambda}\, v{:}exp\,.\,(contains\ c\ v)) \multimap (ev\ K\ (c := v')\ A) \end{array}$$

where the assignment instruction $c := v$ is shown in the usual (infix) form for reasons of readability. The rule can also be encoded in such a fashion that the linear property of the memory is formalized via the Λ quantifier. We will illustrate this idea soon. For now, based on our re-working of the MLR example, we can state the following by referring to [12].

Theorem 5.2 (Representation for MLR) *The encoding functions are compositional bijections. That is, for all stores S of shape $\langle c_1, v_1\rangle, \ldots, \langle c_n, v_n\rangle$, continuations K, instructions i and answers A (which are closed except for possible occurrences of free cells),*

$$\begin{array}{l} S \triangleright K \vdash_{MLR} \Pi{:}i \longrightarrow a \quad \textit{if and only if} \\ \left[\begin{array}{c} c_1!cell, \ldots, c_n!cell, p_1{:}(contains\ c_1\ \mathcal{E}(v_1)), \\ \ldots, p_m{:}(contains\ c_m\ \mathcal{E}(v_m)) \end{array}\right] \vdash_{\Sigma_{MLR}} M_\Pi{:}(ev\ \mathcal{E}(K)\ \mathcal{E}(i)\ \mathcal{E}(a)), \end{array}$$

where Π is a proof object of MLR and M_Π is a canonical object of the $\lambda\Lambda_\kappa$-calculus.

One property that it is desirable to show for the MLR logic is type preservation; in the context of a store Ω, if $S \triangleright K \vdash_{MLR} i \longrightarrow a$, i is a valid instruction of type τ, K is a valid continuation of type $\tau \to \tau'$ and S is a valid store, then a is a valid answer of type τ. The main difference in our reworking of this example is how the proof of type preservation for the EV-REASS rule, prEV-REASS, is encoded.

$$\begin{array}{lll} \text{prEV-REASS} & ! & \mathbf{\Lambda}\, c!cell\,.\,\mathbf{\Lambda}\, v, v'!exp\,.\,\mathbf{\Lambda}\, p{:}(contains\ c\ v)\,. \\ & & \quad (\mathbf{\Lambda}\, p'{:}(contains\ c\ v')\,.\,(prCell\ p'\ c\ v') \multimap (ev\ K\ \bullet\ A)) \multimap \\ & & \quad (prCell\ p\ c\ v) \multimap (prEv\ K\ (x := v')\ A) \end{array}$$

In the above type, *prCell* and *prEv* are the proofs of type preservation over cells and for evaluations, respectively. We note that the types of p and p' have no linear free variables in them. That is, the type theory we have employed in the encoding does not involve the notion of sharing.

Now, the cells could have been quantified intuitionistically (as they are in [12]) instead of linearly. In that case, a sub-proof of $\Gamma \vdash_\Sigma \text{prEV-REASS}{:}U$, where U is the above type of prEV-REASS, would consist of an instance of Π-introduction. But this would allow us to admit garbage: (cell,expression)-pairs which are occupying memory space but not being used. The linear quantification gives us a better representation of memory management. The above encoding realizes the intuition that we are making general statements about linear variables, so the Λ and not the Π quantifier should be used.

The encoded version of MLR type preservation can be stated and shown as in [12]. We omit the details.

5.3 A λ_I-calculus

Our last example is that of the equational theory of a type theory similar to Church's λI-calculus, in which abstraction is only allowed if the abstracted variable is free in the body of the function. We use the full expressiveness of the $\lambda\Lambda_\kappa$-calculus type theory, with the crucial notion of variable sharing. This allows the Λ quantifier to capture the (traditional) notion of relevance. By contrast, in the encoding of the λI-calculus in Avron *et al.* [5], the relevance constraint is enforced by introducing extraneous language to axiomatize relevance in domain theory.

The signature Σ_{λ_I} begins with the declaration $o!\mathsf{Type}$ to represent the syntactic category of terms. The next three constants represent the object-logic abstraction and application operations, and the equality judgement:

$$\lambda_I\,!\,(o \multimap o) \multimap o \qquad app\,!\,o \multimap o \multimap o \qquad =\,!\,o \multimap o \multimap \mathsf{Type}\ .$$

The axioms and rules of the equational theory of the relevant λ-calculus are encoded as follows:

$$\begin{array}{rcl}
E_0 & ! & \mathbf{\Lambda}\, x{:}o\,.\,x = x \\
E_1 & ! & \mathbf{\Lambda}\, x{:}o\,.\,\mathbf{\Lambda}\, y{:}o\,.\,x = y \multimap y = x \\
E_2 & ! & \mathbf{\Lambda}\, x{:}o\,.\,\mathbf{\Lambda}\, y{:}o\,.\,\mathbf{\Lambda}\, z{:}o\,.\,x = y \multimap y = z \multimap x = z \\
E_3 & ! & \mathbf{\Lambda}\, x{:}o\,.\,\mathbf{\Lambda}\, x'{:}o\,.\,\mathbf{\Lambda}\, y{:}o\,.\,\mathbf{\Lambda}\, y'{:}o\,.\,x = x' \multimap y = y' \multimap \\
 & & \quad app(x,y) = app(x',y') \\
\beta & ! & \mathbf{\Lambda}\, x{:}o \multimap o\,.\,\mathbf{\Lambda}\, y{:}o\,.\,app(\lambda_I(x),y) = xy
\end{array}$$

The first three constant declarations, E_0 to E_2, encode the reflexivity, symmetry and transitivity properties of the object-logic judgement, $=$. The constant declaration E_3 encodes the object-logic rule of congruence with respect to application. Finally, the constant declaration β encodes application.

Now, the definition of κ means that the $\Lambda x{:}o$ quantifies over all occurrences of x in its body. Like the ILL example before, the encoding is illustrating a propositions-as-types correspondence. This allows us to state a stronger representation theorem than that given in Avron *et al.* [5].

Theorem 5.3 (Representation of λ_I) *The encoding functions $\mathcal{E}$ are compositional bijections:*

1. $X \vdash_{\lambda_I} M$ *if and only if* $x_1{:}o, \ldots, x_n{:}o \vdash_{\Sigma_{\lambda_I}} \mathcal{E}_X(M){:}o$, *for* $x_i \in FV(M)$; *and*

2. $(M_1 = N_1), \ldots, (M_n = N_n) \vdash_{\lambda_I} \Pi{:}(M = N)$ *if and only if* $x_1{:}(\mathcal{E}(M_1) = \mathcal{E}(N_1))$, $\ldots$, $x_n{:}(\mathcal{E}(M_n) = \mathcal{E}(N_n)) \vdash_{\Sigma_{\lambda_I}} M_\Pi{:}(\mathcal{E}(M) = \mathcal{E}(N))$, *where* Π *is a proof object of* λ_I *and* M_Π *is a canonical object of the* $\lambda\Lambda_\kappa$*-calculus.*

6 Further work

In this paper we have studied a framework, RLF, for uniformly encoding natural deduction presentations of weak logics. Further work based on this paper falls into two groups. Firstly, we can continue the proof-theoretic work by exploring the hyper-cube of intuitionistic and linear λ-cubes, with "diagonal" edges determined by a translation of the form considered in Definition 4.1. We have already mentioned an extension of our current study to include the distributivity laws relating to contexts. We are currently studying the proposition-as-types correspondence and a Gentzenization of the $\lambda\Lambda_\kappa$-calculus. Secondly, a study of the semantics of the $\lambda\Lambda_\kappa$-calculus would bring together and generalize the Kripke models of linear logic and typed λ-calculus [2, 26]. We note that many characteristics of functorial Kripke models of the $\lambda\Pi$-calculus [31] — Π-formation as right adjoint to weakening, for instance — are not immediately applicable in our case. We comment that, besides proof- and model-theoretic semantics, it is also important to study the theory of meaning of the type theory [27]; our discussion in § 2 barely touched on this.

Acknowledgements

We are grateful to Iliano Cervesato, Dale Miller, Peter O'Hearn, Frank Pfenning, Gordon Plotkin and an anonymous referee for their comments on this work. The partial support of the UK EPSRC is gratefully acknowledged.

References

[1] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] G. Allwein and J. Michael Dunn. Kripke Models for Linear Logic. *Journal of Symbolic Logic*, 58(2):514–545, 1993.

[3] A.R. Anderson and J.D. Belnap. *Entailment: The Logic of Relevance and Necessity.* Princeton University Press, 1975.

[4] A. Avron. The semantics and proof theory of linear logic. *Theoretical Computer Science*, 57:161–184, 1988.

[5] A. Avron, F. Honsell, I.A. Mason and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–354, 1992.

[6] A. Avron, F. Honsell, M. Miculan and C. Paravano. Encoding modal logics in logical frameworks, 1997. Manuscript.

[7] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[8] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 118–310. Oxford Science Publications, 1992.

[9] P.N. Benton. A mixed linear and non-linear logic: Proofs, terms and models (preliminary report). Technical Report 352, Computer Laboratory, University of Cambridge, 1994.

[10] G.M. Bierman. On intuitionistic linear logic. Technical Report 346, University of Cambridge Computer Laboratory, August 1994.

[11] I. Cervesato. *A Linear Logical Framework*. PhD thesis, Università di Torino, 1996.

[12] I. Cervesato and F. Pfenning. A Linear Logical Framework. In E Clarke, editor, *11th LICS, New Brunswick, NJ*, pages 264–275. IEEE Computer Society Press, 1996.

[13] T. Coquand. An algorithm for testing conversion in type theory. In Huet and Plotkin [21], pages 255–279.

[14] J.M. Dunn. Relevance logic and entailment. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume III, chapter 3, pages 117–224. D Reidel Publishing Company, 1986.

[15] R.O. Gandy. Proofs of strong normalization. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–478. Academic Press, 1980.

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

[17] J.A. Harland, D.J. Pym and J. Winikoff. Programming in Lygon: An overview. In M Wirsing and M Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 391–405. Springer-Verlag, 1996.

[18] R. Harper, F. Honsell and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[19] R. Harper, D. Sannella and A. Tarlecki. Structured theory representations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.

[20] J.S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[21] G. Huet and G. Plotkin, editors. *Logical Frameworks.* Cambridge University Press, 1991.

[22] I. Kant. *Immanuel Kants Logik (Edited by G.B. Jäsche).* Friedrich Nicolovius, Königsberg, 1800. In translation: R.S. Hartman and W. Schwarz, Dover Publications, Inc., 1988.

[23] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996. (Also: Technical Report 2, Scuola di Specializiazzione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1982.).

[24] M. Masseron, C. Tollu and J. Vauzeilles. Generating plans in linear logic. *Theoretical Computer Science*, 113:349–370 and 371–375, 1993.

[25] R.K. Meyer. Relevant arithmetic. *Polish Academy of Sciences, Institute of Philosophy and Bulletin of the Section of logic*, 5:133–137, 1976.

[26] J.C. Mitchell and E. Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51:99–124, 1991.

[27] D. Prawitz. Proofs and the meaning and completeness of the logical constants. In J. Hintikka, J. Nuniluoto and E. Saarinen, editors, *Essays on Mathematical and Philosophical Logic*, pages 25–40. D Reidel Publishing Company, 1978.

[28] D.J. Pym. *Proofs, Search and Computation in General Logic.* PhD thesis, University of Edinburgh, 1990. Available as Edinburgh University Computer Science Department Technical Report ECS-LFCS-90-125.

[29] D.J. Pym. A relevant analysis of natural deduction. Lecture at Workshop, EU Espirit Basic Research Action 3245, Logical Frameworks: Design, Implementation and Experiment, Båstad, Sweden, May 1992. (Joint work with D. Miller and G. Plotkin.)

[30] D.J. Pym. A note on representation and semantics in logical frameworks. In D. Galmiche, editor, *Proc. of CADE-13 Workshop on Proof-search in Type-theoretic Languages, Rutgers University, New Brunswick, NJ*, 1996. (Also: Technical Report 725, Department of Computer Science, Queen Mary & Westfield College, University of London.).

[31] D.J. Pym. Functorial Kripke models of the $\lambda\Pi$-calculus, 1997. Lecture at Isaac Newton Institute for Mathematical Sciences, Semantics Programme, Workshop on Categories and Logic Programming, Cambridge, 1995. Paper(s) in preparation.

[32] D.J. Pym and J.A. Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, 1994.

[33] S. Read. *Relevant Logic: A Philosophical Examination of Inference.* Basil Blackwell, 1988.

[34] A. Salvesen. A proof of the Church-Rosser property for the Edinburgh LF with η-conversion. Lecture given at the First Workshop on Logical Frameworks, Sophia-Antipolis, France, May 1990.

[35] P. Schroeder-Heister. Generalized rules for quantifiers and the completeness of the intuitionistic operators $\&, \vee, \supset, \curlywedge, \forall, \exists$. In M.M. Richter *et al.*, editor, *Computation and Proof Theory, Logic Colloquim Aachen*, volume 1104 of *LNM*, pages 399–426. Springer-Verlag, 1983.

[36] P. Schroeder-Heister. Structural frameworks, substructural logics, and the role of elimination inferences. In Huet and Plotkin [21], pages 385–403.

[37] N. Tennant. *Autologic*. Edinburgh University Press, 1992.

[38] A.S. Troelstra. *Lectures on Linear Logic*. CSLI, 1992.

[39] L.A. Wallen. *Automated Deduction in Non-Classical Logics*. MIT Press, 1990.