A Hybrid Metalogical Framework

Jason Reed

January 31, 2007

Abstract

The methodology by which deductive systems and metatheorems about them are encoded in the logical framework LF is well understood. Many (but not all) of these ideas have already been successfully extended to the case of encoding *stateful* deductive systems in the *linear* logical framework LLF.

The main gap is the heretofore open question of whether one can bring to LLF the full power of *relational metatheory* as used with LF. This technique reduces the statement and proof of a wide class of theorems about deductive systems to the totality of certain typed, recursively defined relations, a property that can often be verified mechanically. The difficulty of making this technique work in the linear case comes from the lack of a sufficiently expressive language to capture invariants on how the *context* of linear hypotheses is manipulated.

I claim that the gap can be closed by use of ideas from *hybrid logic*, an approach to temporal and modal logic that allows explicit reference to modal worlds. These ideas, when appropriately adapted to the setting of logical frameworks using linear logic, provide an elegant and predicative way of effectively quantifying over and manipulating linear contexts. I describe a hybrid logical framework HLF, a conservative extension of LLF in which one can encode precise formal theorems about stateful deductive systems. I propose as a thesis project that this language be used as the foundation of a system capable of mechanically checking proofs of these theorems.

1 Introduction

When designing languages and tools for mechanized mathematical reasoning, a balance must be struck between competing concerns for expressiveness and simplicity. An expressive language can make representation of informal mathematics in it seem easy, but it can be harder to ascertain in a complex language that any particular representation is not itself somehow subtly mistaken. A simpler language is conversely easier to understand, and therefore easier to trust, but may make expressing certain concepts more laborious.

My proposal takes place in a progression of logical frameworks whose designs subscribe to the idea that one should begin at the simple end of this spectrum,

and add new features only with adequate confidence that their usefulness outweighs any difficulty they create in reasoning about the correctness of encodings in the expanded language. Especially it should be the case that any extension is conservative — it should not affect the correctness of any encodings in an earlier framework.

One such step in this progression was from the logical framework LF [HHP93] to the linear logical framework LLF [CP02]. By incorporating features of linear logic [Gir87] LLF among other benefits permits elegant encodings of *stateful* deductive systems, for example programming languages with imperative reference cells. However, the methodology surrounding LF has not been completely extended to LLF. The way that LF is used in practice involves not only encoding deductive systems themselves, but encoding and mechanically verifying meta-theorems *about* such systems. An example of a meta-theorem for a programming language is *type safety*, that executing a well-typed program does not 'go wrong'; a typical meta-theorem for a logic is that it satisfies cut elimination.

There has not yet been any way to similarly check meta-theorems in LLF: it is a logical framework, but not yet an effective *metalogical* framework, as LF is. Although what are believed to be proofs of meta-theorems can be (and indeed have been [CP02]) encoded, the missing piece is being able to *state* the right theorem in a formal way! The principal obstacle is the fact that LLF represents state with a context of linear assumptions, which act as consumable resources. The required behavior of this context must somehow be captured in the statement of a meta-theorem, yet there is no apparent way to do this directly in the language of LLF.

I propose to make a conservative extension of LLF to solve this problem. This extension is in fact most naturally expressed directly as an extension of LF: the linear features of LLF turn out to be particular ways of using the features of the proposed extension.

The extension consists of adding to LF ways of forming new types, and new kinds (the expressions that classify types and families of types). These type and kind constructors are directly inspired by logical connectives in hybrid logic, an approach to modal logic which allows explicit syntactic reference to a notion of 'world' usually found in the semantics of modal logic. There are related hybrid type constructors, which when added to LF, first of all yield a system in which LLF can be faithfully embedded, and secondly make it possible to syntactically refer to the context of stateful deductive systems, allowing precise statements of meta-theorems.

I claim the following:

> **Thesis Statement**: An extension of the type theory of LF with hybrid type and kind constructors provides the foundation for a metalogical framework capable of encoding and verification of metatheorems about stateful deductive systems.

The remainder of this document is divided as follows. Section 2 gives some of the background in (linear) logical framework methodology as used in prior

work. Section 3 explains in more detail the problem I intend to solve. Section 4 describes what work I have already completed toward this goal. Section 5 contains what of the remaining work constitutes the core focus of the proposed thesis project, and section 6 lists other goals I would like to accomplish if time permits. Section 8 gives a projected timeline, and section 7 discusses related work.

2 Background

2.1 What is a logical framework?

A logical framework is a meta-language for encoding deductive systems — other languages, such as logics and programming languages — and for encoding reasoning about them. It consists of a type theory, and an encoding methodology, by which one translates the customary informal descriptions of deductive systems into the type theory. The language being encoded referred to as the *object language*, and the framework in which it is encoded is the *representation language*.

Throughout this proposal I will be concerned with the family of languages descended from the logical framework LF, due to Harper, Honsell and Plotkin [HHP93]. The type theory of LF itself is quite minimal compared to other popular logical frameworks (e.g. Coq, NuPrl, Isabelle). It is essentially just the simply typed λ-calculus with dependent function types $\Pi x{:}A.B$.

The syntax of the terms and types of LF can be given by

$$\text{Terms } M, N ::= \lambda x.M \mid M\ N \mid c \mid x$$

$$\text{Types } A ::= \Pi x{:}A.B \mid a\ M_1 \cdots M_n$$

There is also a language of *kinds*, which classify type families:

$$\text{Kinds } K ::= \Pi x{:}A.K \mid \text{type}$$

The kind $\Pi x_1{:}A_1 \cdots \Pi x_n{:}A_n.\text{type}$ describes type families indexed by n objects, of types A_1 up to A_n. Such a type family can be thought of as a function taking arguments, and returning a type.

We also will write as usual $A \to B$ (resp. $A \to K$) for the degenerate version of $\Pi x{:}A.B$ (resp. $\Pi x{:}A.K$) where x doesn't actually appear in B (resp. K). Terms, just as in the simply-typed λ-calculus, are either function expressions, applications, constants from the signature, or variables. Types are dependent function types, or else instances of type families a from the signature, indexed by a series of terms.

The typing judgment $\Gamma \vdash M : A$ says whether a term M has a type A in a context Γ of hypotheses. Typing rules for the introduction and elimination of functions are as follows:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : \Pi x{:}A.B} \qquad \frac{\Gamma \vdash M : \Pi x{:}A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : \{N/x\}B}$$

From these the behavior of dependent types is apparent from the substitution $\{N/x\}B$ in the function elimination: the *type* of the function application $M\ N$ depends on what the argument N was, for it is substituted for the variable x, which occurs free in B. Any variable in the context can be used to form a well-typed term as well, via the variable rule

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

Since terms appear in types, one must be clear about which pairs of types count as equal. For the time being, we may take the simplistic view that terms (and therefore types that mention terms) are simply considered indistinguishable up to α, β, η convertibility. In Section 4 below we will discuss in more detail a modern treatment of this issue.

2.2 LF methodology

Now the question of whether a given term is well-typed depends on the *signature* of declarations of typed constants and type families. To formally encode a deductive system in LF is to create a signature for which the objects and predicates of the deductive system are in suitable correspondence with well-formed expressions in LF. The remainder of this section provides some examples of typical encoding techniques.

2.2.1 Data Structures

A recursive datatype can be represented by declaring a type, and declaring one constant for each constructor of it. For example, the syntax of the natural numbers can be represented by putting into the signature the declaration of one type, and two constants, for zero and successor:

$$nat : \text{type}$$

$$z : nat. \qquad s : nat \to nat$$

The theories of programming languages and logics often involve variable binding and require that renamings of bound variables are considered equivalent. To encode this explicitly (or to use a different representation, such as deBruijn indices) can be extremely tedious. A common alternative is to use *higher order abstract syntax* [PE89], in which the variables and variable binders in the object language are defined in terms of the variables and function argument binding in the representation language.

The untyped lambda calculus

$$e ::= \lambda x.e \mid e_1\ e_2 \mid x$$

is represented in this style by the type declaration

$$exp : \text{type}$$

and the constant declarations

$$lam : (exp \to exp) \to exp$$

$$app : exp \to exp \to exp$$

The fact that lam requires a function as its input is what makes the encoding higher-order. To encode a lambda term such as $\lambda x.x\ x$, we specifically pass lam a function that uses its argument everywhere the variable bound by that lambda appears. Thus $lam(\lambda x.app\ x\ x)$ is the encoding of $\lambda x.x\ x$.

2.2.2 Judgments

Predicates on data can be represented by taking advantage of dependent function types. The slogan is 'judgments as types': *judgments* (predicates) of objects are encoded as *type families* indexed by those objects.

For example, the usual linear order $\geq$ on natural numbers can be encoded as follows.

$$ge : nat \to nat \to \text{type}$$

$$ge\_z : \Pi M{:}nat.ge\ M\ z$$

$$ge\_s : \Pi M{:}nat.\Pi N{:}nat.ge\ M\ N \to ge\ (s\ M)\ (s\ N)$$

The type family ge is indexed by two natural numbers: because of ge_z and ge_s, it is the case that for any M and N, the type $ge\ M\ N$ is inhabited if and only if M represents a number greater than or equal to the number N represents.

To represent recursive functions, we apply the standard idiom of traditional logic programming languages such as Prolog, viewing functions as relations between their inputs and outputs. For example the function $plus$ on natural numbers is thought of as a relation on triples of nat: the triple $\langle M, N, P\rangle$ belongs to the relation just in case indeed $M + N = P$.

In LF, this relation, just like the binary relation $\geq$ described above, can be encoded as a type family. In this case it is declared as

$$plus : nat \to nat \to nat \to \text{type}$$

$$plus\_z : \Pi N{:}nat.plus\ z\ N\ N$$

$$plus\_s : \Pi M{:}nat.\Pi N{:}nat\Pi P{:}nat.plus\ M\ N\ P$$
$$\to plus\ (s\ M)\ N\ (s\ P)$$

The Twelf system [PS99] has considerable support for reasoning about functions defined in this way. It has a logic programming interpreter, so it can run programs by solving queries. Moreover it has several directives which can be used to verify properties of logic programs:

- *Modes* are descriptions of the intended input-output behavior of a relation. A mode specification for *plus* above is $plus\ +M\ +N\ -P$: its first two arguments M, N are understood as input $(+)$, and its third argument P is output $(-)$. A relation is **well-moded** for a given mode specification if each clause defining it correctly respects the given modes.
- A relation is said to **cover** its inputs, when for any inputs it is given, it will reduce search to another set of goals.
- A relation is **terminating** if this reduction of goals to subgoals always terminates.

These properties are all in principle potentially very difficult (or impossible) to decide exactly, but there are good decidable conservative approximations to them. When a relation R is well-moded, and satisfies coverage and termination properties, then it is a **total** relation: for every set of inputs, there is at least one output such that R relates the given inputs to that output.

Thus for many total relations (for example $plus$ above) it can be mechanically verified in Twelf that they are total.

2.2.3 Metatheorems

A salient advantage of the above style of encoding is the fact that what we might think of as executions of the function *plus* are available as data to be manipulated in the same way as, say, natural numbers. With this it is possible to do meta-logical reasoning in LF [Sch00]: proving metatheorems about entire deductive systems.

An example metatheorem is associativity of *plus*. A constructive interpretation of this claim is that we must exhibit a function that for all numbers n_1, n_2, n_3, yields evidence that $(n_1 + n_2) + n_3 = n_1 + (n_2 + n_3)$. Therefore in LF it suffices to define a total relation that takes as input derivations of

$$plus\ N_2\ N_3\ N_{23}$$

$$plus\ N_1\ N_{23}\ M$$

$$plus\ N_1\ N_2\ N_{12}$$

and outputs a term of type

$$plus\ N_{12}\ N_3\ M$$

Such a relation indeed can be described in LF as follows: (leaving many Πs and their arguments implicit hereafter for brevity)

$$plus\_assoc : plus\ N_1\ N_{23}\ M \to plus\ N_1\ N_2\ N_{12}$$

$$\to plus\ N_2\ N_3\ N_{23} \to plus\ N_{12}\ N_3\ M \to type$$

$$pa/z : plus\_assoc\ plus\_z\ plus\_z\ P\ P.$$

$$pa/s : plus\_assoc\ P_1\ P_2\ P_3\ P_4$$
$$\rightarrow plus\_assoc\ (plus\_s\ P_1)\ (plus\_s\ P_2)\ P_3\ (plus\_s\ P_4)$$

Again, in order for this defined relation to count as a proof of associativity of *plus*, it must be *total* in precisely the same sense as the one mentioned at the end of the previous section, in this case for the mode specification $plus\_assoc +\ P_1\ +\ P_2\ +\ P_3\ -\ P_4$. In this way, the totality-checking facilities of Twelf can be used to verify a wide range of metatheorems.

2.3 Logical frameworks for stateful systems

LF by itself is quite adequate for a wide range of encodings of logical and programming language features, but not as effective for those involving state and imperative update. It is possible to use, for instance, store-passing encodings to represent a programming language with imperative reference cells, but this is inconvenient in much the same way as is 'coding up' references via store-passing when using a pure functional programming language. One also quickly bumps up against the classic so-called *frame problem* when describing large, complex states whose evolution over time is described in terms of small modifications. The generally large 'frame' of state variables that *don't* change in any particular state transition must still be described, thus complicating the encoding.

An elegant solution to these obstacles can be found in the applications of linear logic. Linear logic provides a logical explanation for state, and leads to straightforward encodings of stateful systems. Research on the logical frameworks side [CP02] has proven that it is possible to incorporate these ideas into a conservative extension of LF called LLF (for 'Linear Logical Framework'), yielding a system appropriate for encodings of stateful deductive systems.

In the remainder of this section, we provide a brief survey of the theory and applications of linear logic, and of the framework LLF.

2.3.1 Introduction to Linear Logic

Linear logic is a substructural logic in which there are *linear hypotheses*, which behave as resources that can (and must) be consumed, in contrast to ordinary logical hypotheses which, having been assumed, remain usable any number of times throughout the entirety of their scope. These latter are therefore termed *unrestricted* hypotheses in contrast to linear ones. Linear logic is 'substructural' because among the usual structural rules that apply to the context of hypotheses — exchange, contraction, and weakening — the latter two are not applicable to linear hypotheses.

Although linear logic was originally presented with a classical (that is to say multiple-conclusion) calculus, we will summarize here a fragment of judgmental intuitionistic linear logic as given by Chang, Chaudhuri, and Pfenning [CCP03], for it is more closely related to the linear logical framework discussed below.

The core notion of linear logic is the *linear hypothetical judgment*, written $\Delta \Vdash A\,\mathrm{true}$, where Δ is a context of hypotheses $A_1\,\mathrm{true}, \ldots, A_n\,\mathrm{true}$. This judgment is read as intuitively meaning 'A can be achieved by using each resource

in Δ exactly once'. For brevity we leave off the repeated instances of true in the sequel and write sequents such as $A_1, \ldots, A_n \Vdash A$.

Which judgements can be derived are determined by inference rules. There is a hypothesis rule

$$\frac{}{A \Vdash A}\, hyp$$

which says that A can be achieved if the set of hypotheses is precisely A. Note that because weakening on the context of linear hypotheses is not permitted, this rule does not allow us to derive $B, A \Vdash A$.

This judgmental notion leads to a new collection of logical connectives. For an idea of how some of them arise, consider the pervasively used example of a vending machine: suppose there is a vending machine that sells sticks of gum and candy bars for a quarter each. If I have one quarter, I can buy a stick of gum, but in doing so, I use up my quarter. This fact can be represented with the *linear implication* $\multimap$ as the proposition $quarter \multimap gum$. Now it is also the case that with my quarter I could have bought a candy bar. Notice specifically that I can achieve both the goals *gum* and *candy* with a quarter, but not both simultaneously. So in some sense *quarter* implies both *gum* and *candy*, but in a different sense from the way *two* quarters implies the attainability of *gum* and *candy*, for with fifty cents I can buy both.

There are accordingly two kinds of conjunction in linear logic, &, which represents the former, 'alternative' notion of conjunction, and $\otimes$, which captures the latter, 'simultaneous' conjunction. The linear logic propositions that represent the facts just discussed are $quarter \multimap gum \mathbin{\&} candy$ (with one quarter I am able to buy both gum and candy, whichever I want) and $quarter \otimes quarter \multimap gum \otimes candy$ (with two quarters I am able to buy both gum and candy at the same time).

$$\frac{\Delta_1 \Vdash A \multimap B \qquad \Delta_2 \Vdash A}{\Delta_1, \Delta_2 \Vdash B} \multimap E \qquad \frac{\Delta, A \Vdash B}{\Delta \Vdash A \multimap B} \multimap I$$

$$\frac{\Delta \Vdash A \mathbin{\&} B}{\Delta \Vdash A} \mathbin{\&} E1 \qquad \frac{\Delta \Vdash A \mathbin{\&} B}{\Delta \Vdash B} \mathbin{\&} E2 \qquad \frac{\Delta \Vdash A \qquad \Delta \Vdash B}{\Delta \Vdash A \mathbin{\&} B} \mathbin{\&} I$$

$$\frac{\Delta_1 \Vdash A \otimes B \qquad \Delta_2, A, B \Vdash C}{\Delta_1, \Delta_2 \Vdash C} \otimes E \qquad \frac{\Delta_1 \Vdash A \qquad \Delta_2 \Vdash B}{\Delta_1, \Delta_2 \Vdash A \otimes B} \otimes I$$

The difference between the two conjunctions is visible from their introduction rules. In $\&I$, in order to establish that $A \mathbin{\&} B$ can be achieved from resources Δ, one must show that A can be achieved from Δ, and that B can be achieved from the *same* set of resources Δ. In $\otimes I$, however, to achieve $A \otimes B$ from some resources, one must exhibit how those resources can be divided into Δ_1 and Δ_2 so that Δ_1 yield A and Δ_2 yield B.

The system described so far can only describe linear hypotheses. So that it can also express everything ordinary intuitionistic logic can, one can add to it the notion of unrestricted hypotheses, and a modal operator ! that mediates between the linear and unrestricted judgments. The proposition $!A$ construed as

a hypothetical resource has the interpretation of an unlimited and unrestricted supply of copies of A — any number of them, zero or more, may be used. We do not give inference rules for ! here, but refer the reader to [CCP03]. The only property of ! that is of interest below is that the usual intuitionistic implication $\Rightarrow$ has a decomposition in linear logic

$$A \Rightarrow B \dashv\vdash (!A) \multimap B$$

2.3.2 Applications of Linear Logic

This notion of consumable hypotheses is useful for representing ephemeral facts of a stateful system — those that may cease to be valid after the state of the system changes.

Consider a finite state machine with states S and a transition relation $R \subseteq S \times \Sigma \times S$, where $\langle s, \sigma, s' \rangle \in R$ means that the system may go from state s to s' if it receives input character σ from some alphabet Σ. This can be represented in linear logic by supposing that there are atomic predicates $state(s)$ and $input(\ell)$ for states $s \in S$ and strings $\ell \in \Sigma^*$, and taking as axioms

$$\Vdash state(s) \otimes input(\sigma\ell) \multimap state(s') \otimes input(\ell)$$

for each $\langle s, \sigma, s' \rangle \in R$. With these assumptions, it is the case that

$$\Vdash state(s) \otimes input(\ell) \multimap state(s') \otimes input(\epsilon)$$

(where ε is the empty string) is derivable just in case the finite machine, started in state s, can in some number of steps read all of the input ℓ, and end up in state s'. Take in particular the system where $S = \{s_1, s_2\}$, $\Sigma = \{a, b\}$, and $R = \{\langle s_1, a, s_2 \rangle, \langle s_2, b, s_2 \rangle, \langle s_2, a, s_1 \rangle\}$. With input ab the system can go from s_1 to s_2, and there is a proof in linear logic of the corresponding judgment, abbreviating $state(s) \otimes input(\ell)$ as $si(s, \ell)$. Here is a sketch of it:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Vdash si(s_2, b) \multimap si(s_2, \varepsilon)} \quad \dfrac{\dfrac{\vdots}{\Vdash si(s_1, ab) \multimap si(s_2, b)} \quad \dfrac{}{si(s_1, ab) \Vdash si(s_1, ab)}\, hyp}{si(s_1, ab) \Vdash si(s_2, b)} \multimap E}{si(s_1, ab) \Vdash si(s_2, \epsilon)} \multimap E}{\Vdash si(s_1, ab) \multimap si(s_2, \epsilon)} \multimap I$$

Note that this claim would not be true if we tried encoding the FSM in ordinary logic by using the axioms

$$\vdash state(s) \wedge input(\sigma\ell) \Rightarrow state(s') \wedge input(\ell)$$

for each $\langle s, \sigma, s' \rangle \in R$. For then in the example FSM we could form a derivation of $state(s_1) \wedge input(a) \Rightarrow state(s_1) \wedge input(\epsilon)$, despite the fact that the above

FSM could not go from state s_1 to itself on input a. The derivation can be constructed in the following way, abbreviating in this case $state(s) \wedge input(\ell)$ as $si(s, \ell)$: First form the derivation

$$\mathcal{D} = \dfrac{\dfrac{\overline{si(s_1,a) \Vdash si(s_2,\varepsilon)}}{si(s_1,a) \Vdash state(s_2)} \wedge E1 \quad \dfrac{\overline{si(s_1,a) \Vdash si(s_1,a)}}{si(s_1,a) \Vdash input(a)} \wedge E2}{si(s_1,a) \Vdash si(s_2,a)} \wedge I \quad hyp$$

Now plug in $\mathcal{D}$ like this:

$$\dfrac{\dfrac{\overline{\Vdash si(s_2,a) \Rightarrow si(s_1,\varepsilon)} \qquad \overset{\mathcal{D}}{si(s_1,a) \Vdash si(s_2,a)}}{si(s_1,a) \Vdash si(s_2,\epsilon)} \multimap E}{\Vdash si(s_1,a) \Rightarrow si(s_1,\epsilon)} \Rightarrow I$$

The usual logical connectives used in this way clearly fail to account for state changes; we were able to cheat and illogically combine some of the information from the previous state of the evolving FSM in which the input string was a, together with the information from a later time when the machine's current state was s_2.

2.3.3 The Linear Logical Framework

LLF is an extension of LF to support a linear hypothetical judgment.

Where the typing judgment of LF is

$$\Gamma \vdash M : A$$

the typing judgment of LLF is

$$\Gamma; \Delta \vdash M : A$$

adding a context Δ of *linear variables* $x \hat{:} A$. It retains from LF the context Γ of *unrestricted variables* — the variables in Δ are resources that must be used exactly once, but the variables in Γ behave as ordinary variables just as in LF, and are allowed to be used without restriction.

Following the slogan of *propositions as types*, LLF adds new type constructors (and new term constructors for them) corresponding to propositional connectives of linear logic. The grammars of terms and types in LF are extended by

$$M ::= \cdots \mid \hat{\lambda} x.M \mid M \mathbin{\hat{}} N \mid \langle M, N \rangle \mid \pi_i M \mid \langle\rangle$$
$$A ::= \cdots \mid A \multimap B \mid A \mathbin{\&} B \mid \top$$

The grammar of the language of kinds remains the same; this will become important in Section 3.1. Inhabiting the linear function type $\multimap$ are linear functions $\hat{\lambda} x.M$, which can be used in linear function application $M \mathbin{\hat{}} N$. The type

$M \,\&\, N$ is a type of pairs, formed by $\langle M, N\rangle$ and decomposed with projections $\pi_1 M, \pi_2 M$. The type $\top$ is a unit for &; it has a single inhabitant $\langle\rangle$. The typing rules for these new constructs are as follows:

$$\frac{\Gamma; \Delta_1 \vdash M : A \multimap B \qquad \Gamma; \Delta_2 \vdash N : A}{\Gamma; \Delta_1, \Delta_2 \vdash M \hat{} N : B} \multimap E \qquad \frac{\Gamma; \Delta, x\hat{:}A \vdash M : B}{\Gamma; \Delta \vdash \hat{\lambda} x.M : A \multimap B} \multimap I$$

$$\frac{\Gamma; \Delta \vdash M : A_1 \,\&\, A_2}{\Gamma; \Delta \vdash \pi_i M : A_i} \,\&E \qquad \frac{\Gamma; \Delta \vdash M : A \qquad \Delta \vdash N : B}{\Gamma; \Delta \vdash \langle M, N\rangle : A \,\&\, B} \,\&I$$

$$\frac{}{\Gamma; \Delta \vdash \langle\rangle : \top} \top I$$

Moreover the existing LF rules must be modified to accommodate the linear context. The LF variable rule splits into two rules, depending on whether the variable used was from the unrestricted or linear context:

$$\frac{x : A \in \Gamma}{\Gamma; \cdot \vdash x : A} \, hyp \qquad \frac{}{\Gamma; x\hat{:}A \vdash x : A} \, lhyp$$

Since every variable in the linear context must be used exactly once, the linear context must be empty in the case of the use of an ordinary unrestricted variable, and must contain exactly the variable used in the case of using a linear variable. The dependent function typing rules become

$$\frac{\Gamma, x : A; \Delta \vdash M : B}{\Gamma; \Delta \vdash \lambda x.M : \Pi x{:}A.B} \Pi I \qquad \frac{\Gamma; \Delta \vdash M : \Pi x{:}A.B \qquad \Gamma; \cdot \vdash N : A}{\Gamma; \Delta \vdash M \; N : \{N/x\}B} \Pi E$$

This means that Πs are still essentially functions of ordinary, non-linear, *unrestricted* arguments: this fact manifests itself in the λ rule as the appearance of the variable x in the unrestricted context, and in the application rule as the fact that the linear context is empty in the typing of N. Since non-dependent unrestricted implication decomposes as $(!A) \multimap B$, and since in a sense the domain of a Π is also unrestricted, one might ask whether a comparable decomposition of Π exists. In other words, is there a 'linear Π,' written as $\Pi x\hat{:}A.B$ such that in some sense $\Pi x{:}A.B \equiv \Pi x\hat{:}(!A).B$? We return to this question briefly in section 3.1 and section 4

2.4 Encodings in the Linear Logical Framework

The two major examples of encodings into LLF given by Cervesato and Pfenning [CP02] are of a programming language, MiniML with references, and of a logic, namely the linear sequent calculus. The first encoding takes advantage of linearity directly to represent state changes resulting from imperative features in the programming language. The second uses the new features introduced in the linear logical framework to easily encode the logic that inspired them. We will focus on this latter encoding as a running example throughout the rest of this proposal.

2.4.1 Linear Sequent Calculus

Just as ordinary higher-order abstract syntax encodes object-language binders as framework-level binders, we can in LLF encode object-language linearity with framework-level linearity.

So that we can later talk about proving cut elimination as a metatheorem, we consider the sequent calculus instead of the natural deduction formulation of the logic. As is typical of sequent calculi, the linear sequent calculus is identical to the natural deduction system in the introduction rules, (except they are instead called 'right rules') but instead of elimination rules it has rules that introduce connectives on the left. For instance, the left rules for $\multimap, \otimes$, and & are:

$$\frac{\Delta_1 \Vdash A \qquad \Delta_2, B \Vdash C}{\Delta_1, \Delta_2, A \multimap B \Vdash C} \multimap L$$

$$\frac{\Delta, A \Vdash C}{\Delta, A \,\&\, B \Vdash C} \,\&L1 \qquad \frac{\Delta, B \Vdash C}{\Delta, A \,\&\, B \Vdash C} \,\&L2$$

$$\frac{\Delta, A, B \Vdash C}{\Delta, A \otimes B \Vdash C} \otimes L$$

The following is an encoding of the linear sequent calculus in LLF. We declare a type for propositions

o : type

and two type families, one for hypotheses, and one for conclusions.

$hyp : o \to$ type
$conc : o \to$ type

The propositional connectives $\multimap, \&, \otimes$ are encoded as constructors of the type of propositions

$lol : o \to o \to o$
$amp : o \to o \to o$
$tensor : o \to o \to o$

Thereafter we can encode the left and right inference rules $\multimap L$ and $\multimap R$ for $\multimap$ as two constants

$lolr : (hyp\ A \multimap conc\ B) \multimap conc\ (lol\ A\ B)$
$loll : conc\ A \multimap (hyp\ B \multimap conc\ C) \multimap (hyp\ (lol\ A\ B) \multimap conc\ C)$

and similarly for the rules $\&R$, $\&L_1$, and $\&L_2$:

$ampr : conc\ A\ \&\ conc\ B \multimap conc\ (amp\ A\ B)$
$ampl1 : (hyp\ A \multimap conc\ C) \multimap (hyp\ (amp\ A\ B) \multimap conc\ C)$
$ampl2 : (hyp\ B \multimap conc\ C) \multimap (hyp\ (amp\ A\ B) \multimap conc\ C)$

and for the rules $\otimes R$ and $\otimes L$:

$tensorr : conc\ A \multimap conc\ B \multimap conc\ (tensor\ A\ B)$
$tensorl : (hyp\ A \multimap hyp\ B \multimap conc\ C) \multimap (hyp\ (tensor\ A\ B) \multimap conc\ C)$

Finally, the init rule

$$\frac{}{A \Vdash A}\ init$$

is represented by the declaration

$init : hyp\ A\ \multimap conc\ A$

The encoding uses higher-order function types to represent the structure of the context, and uses linearity in the framework to represent linearity of hypotheses in the object language.

The representation of a derivation such as

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \Vdash A}\ init \quad \dfrac{}{B \Vdash B}\ init}{A \multimap B, A \Vdash B}\ \multimap L}{A \multimap B \Vdash A \multimap B}\ \multimap R}{\Vdash (A \multimap B) \multimap (A \multimap B)}\ \multimap R$$

can be built up as follows. The end goal is a derivation of $\Vdash (A \multimap B) \multimap (A \multimap B)$, which will be represented as an LLF term M of type

$$conc\ (lol\ (lol\ A\ B)\ (lol\ A\ B))$$

The last proof rule used was $\multimap R$, so M will be $lolr\ \hat{}\ (\hat{\lambda} x.M_1)$ for some M_1 such that $\cdot; x \hat{:} hyp\ (lol\ A\ B) \vdash M_1 : conc\ (lol\ A\ B)$. That the constructor $lolr$ requires a linear function corresponds exactly to the fact that the inference rule $\multimap R$ requires a derivation with a linear hypothesis. Working up through the proof, we use $\multimap R$ again, and so we choose M_1 to b be $lolr\ \hat{}\ (\hat{\lambda} y.M_2)$ for some M_2 such that

$$\cdot; x \hat{:} hyp\ (lol\ A\ B), y : hyp\ A \vdash M_2 : conc\ B$$

And then M_2 should be a use of $loll$, to match the use of $\multimap L$; subsequently at the leaves of the proof tree, we must use $init$. The final representation of the proof is

$$M = lolr\ \hat{}\ (\hat{\lambda} x.lolr\ \hat{}\ (\hat{\lambda} y.loll\ \hat{}\ (init\ \hat{}\ y)\ \hat{}\ (\hat{\lambda} z.init\ \hat{}\ z)\ \hat{}\ x))$$

3 Problem: Mechanized Metatheory for LLF

What is missing from the above methodology is a complete analogue to the way metatheorems are encoded LF. It is still possible to write down the essential computational content of the proofs themselves in LLF (and in fact has been done [CP02]) but it is not known how to capture the *statement* of theorems about stateful deductive systems in terms of LLF relations. The proofs that have been carried out are still encoded as relations, but the type of the relation is insufficiently precise to capture the intended theorem.

Casting the problem in terms of thinking of proofs as programs, we may say that it is still possible to write the programs we want in LLF, but not to give these programs precise enough types to ensure that they *are* the programs we meant to write. The goal of the proposed thesis project is, essentially, to establish a language of types to solve this problem.

3.1 Cut Admissibility in Intuitionistic and Linear Logic

We can examine how this problem arises in the case of trying to mechanically verify a proof of cut admissibility for the sequent calculus encoded above.

First we note some facts about how encoding a structural proof [Pfe95, Pfe00] of cut admissibility works for ordinary intuitionistic logic. The theorem to be shown is that the cut rule, which allows us to eliminate a 'proof detour' A, is admissible:

Theorem 3.1 (Intuitionistic Cut Admissibility) *If* $\Gamma \vdash A$ *and* $\Gamma, A \vdash C$ *then* $\Gamma \vdash C$.

This is represented as a relation mapping derivations of $\mathit{conc}\ A$ and $\mathit{hyp}\ A \to \mathit{conc}\ C$ to a derivation of $\mathit{conc}\ C$.

$$\mathit{ca} : \mathit{conc}\ A \to (\mathit{hyp}\ A \to \mathit{conc}\ C) \to \mathit{conc}\ C \to \text{type} \tag{1}$$

The context $\Gamma = A_1, \ldots, A_n$ of hypotheses in the statement of the theorem corresponds in the encoding to an LF context of variables $x_1 : \mathit{hyp}\ A_1, \ldots, x_n : \mathit{hyp}\ A_n$ that might occur in the two input, and one output derivation. This strategy — representing the object language context with the representation context — is effective because the context Γ is shared across the two premises and conclusion of the theorem. When forming a term in the type family *ca* (that is, a derivation of of some instance of the theorem) all variables in the LF context are naturally available for unrestricted use in both premises and the conclusion.

Now the statement of cut admissibility, on the other hand, is

Theorem 3.2 (Linear Cut Admissibility) *If* $\Delta_1 \Vdash A$ *and* $\Delta_2, A \Vdash C$*, then* $\Delta_1, \Delta_2 \Vdash C$.

In the linear sequent calculus the nontrivial relationships of the various contexts to one another is essential for the meaning of the cut admissibility theorem. Yet

there is no evident way of writing the theorem as an LLF relation that captures these invariants.

The simplest attempt to adapt (∗) is to simply replace the type-level → with a ⊸, yielding

$$ca : conc\ A \to (hyp\ A \multimap conc\ C) \to conc\ C \to \text{type} \tag{2}$$

However, the use of unrestricted function space → discards any information that might have been known about the context used to make terms of type *conc A*, *hyp A* ⊸ *conc C*, and *conc C*: recall that the unrestricted arrow requires its argument to be well-typed in an empty linear context.

What seems desirable is to use linear connectives somehow to express the fact that Δ_1 and Δ_2 are disjoint contexts, and that the context Δ_1, Δ_2 in the output derivation is the combination of them. Suggestively, one might try to write

$$ca : ((conc\ A \otimes (hyp\ A \ \multimap conc\ C)) \,\&\, conc\ C) \multimap \text{type} \tag{3}$$

to capture the fact that two input derivations have disjoint contexts, and that the output derivation of *conc C* has the same context as the two input derivations taken together. The multiplicative conjunction ⊗ expresses that two goals are to be achieved with disjoint parts of the current context, and the additive conjunction that two goals are to be achieved with the same context.

The main problem here is that this isn't even a valid declaration of an LLF type family — LLF does not have '⊸ type' in its language of kinds. Another apparent problem is that LLF doesn't have ⊗, but this can be avoided by a standard currying transformation of the encoding where the type $A_1 \otimes A_2$ is simulated by a declared constant type t_{A_1,A_2} : type and a constant $create_{A_1,A_2}$: $A_1 \multimap A_2 \multimap t_{A_1,A_2}$ (or else by working in CLF, which does have ⊗). To have a linear function space whose codomain is the kind 'type' would mean that there would be a notion of type family whose indices were themselves somehow linear. This was the approach suggested by early work [Pfe94] on encoding linear cut elimination. Although work by Ishtiaq, Pym, et al. [IP98, Ish99] has studied this sort of dependent-substructural function space in other logics, to the best of our knowledge it has not been carried out adequately with index objects that are actually linear in the sense of Girard's linear logic, as opposed to obeying substructural disciplines as found in relevant logic and bunched logic.

Using → type instead of ⊸ type to stay within LLF yields the type family

$$ca : ((conc\ A \otimes (hyp\ A \ \multimap conc\ C)) \,\&\, conc\ C) \to \text{type} \tag{4}$$

but this use of the unrestricted arrow has the same problem as in (2).

It is nonetheless possible to encode a correct proof of the linear cut admissibility theorem, but only by representing object-language linear by unrestricted LF variables — it is this feature of the representation that loses essential information about the use of linear resources. Consequently it is possible to write *incorrect* cases of the proof of this theorem, and they will nonetheless typecheck.

We first explain the encoding of a case from the correct theorem, and go on to show how it can be modified to yield an unsound 'proof'. Suppose the

derivations of $\Delta_1 \Vdash A$ and $\Delta_2, A \Vdash C$ are named $\mathcal{E}$ and $\mathcal{F}$, respectively. If they happen to both introduce the top-level propositional connective of the cut formula A, then the situation is called a *principal cut*. In the principal cut case for $\multimap$, the cut formula A is of the form $A_1 \multimap A_2$ and $\mathcal{E}$ and $\mathcal{F}$ are built out of derivations as follows:

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}_1\\ \Delta_1, A_1 \Vdash A_2\end{matrix}}{\Delta_1 \Vdash A_1 \multimap A_2}\multimap R \quad \dfrac{\begin{matrix}\mathcal{D}_2 & \mathcal{D}_3\\ \Delta_{21} \Vdash A_1 & \Delta_{22}, A_2 \Vdash C\end{matrix}}{\Delta_{21}, \Delta_{22}, A_1 \multimap A_2 \Vdash C}\multimap L}{\Delta_1, \Delta_{21}, \Delta_{22} \Vdash C}\, cut$$

From this we can construct a derivation that only uses the cut rule (or equivalently applies the induction hypothesis of the admissibility theorem) at smaller cut formulae:

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}_2 & \mathcal{D}_1\\ \Delta_{21} \Vdash A_1 & \Delta_1, A_1 \Vdash A_2\end{matrix}}{\Delta_1, \Delta_{21} \Vdash A_2}\, cut \quad \begin{matrix}\mathcal{D}_3\\ \Delta_{22}, A_2 \Vdash C\end{matrix}}{\Delta_1, \Delta_{21}, \Delta_{22} \Vdash C}\, cut$$

This reasoning is encoded as

$$\begin{array}{l}
prems : o \to \text{type}\\
p : conc\ A \multimap (hyp\ A \multimap conc\ C) \multimap prems\ C\\
ca : (prems\ C\ \&\ conc\ C) \to \text{type}\\
ca/lol/principal :\\
\quad \Pi\mathcal{D}_1{:}(hyp\ A_1 \multimap conc\ A_2).\ \Pi\mathcal{D}_2{:}(conc\ A_1).\\
\quad \Pi\mathcal{D}_3{:}(hyp\ A_2 \multimap conc\ C).\ \Pi\mathcal{D}_4{:}(conc\ A_2).\ \Pi\mathcal{D}_5{:}(conc\ C).\\
\qquad ca\ \langle p\,\hat{}\,(lolr\,\hat{}\,(\hat{\lambda}x.\mathcal{D}_1\,\hat{}\,x))\,\hat{}\,(\hat{\lambda}z.loll\,\hat{}\,\mathcal{D}_2\,\hat{}\,(\hat{\lambda}x.\mathcal{D}_3\,\hat{}\,x)\,\hat{}\,z), \mathcal{D}_5\rangle\\
\qquad\quad \leftarrow ca\langle p\,\hat{}\,\mathcal{D}_2\,\hat{}\,(\hat{\lambda}z.\mathcal{D}_1\,\hat{}\,x), \mathcal{D}_4\rangle\\
\qquad\quad \leftarrow ca\langle p\,\hat{}\,\mathcal{D}_4\,\hat{}\,(\hat{\lambda}z.\mathcal{D}_3\,\hat{}\,x), \mathcal{D}_5\rangle
\end{array}$$

Here *prems* is an instance of the currying technique mentioned above in order to simulate $\otimes$. The long prefix of Πs at the beginning names all derivations used in the case. The first subsequent line establishes that this clause treats the case where the first derivation is constructed from $\multimap R$ (i.e. using *lolr* in LLF) and the second from $\multimap L$ (i.e. using *loll* in LLF). The two subgoals contain the instructions to cut $\mathcal{D}_2$ against $\mathcal{D}_1$ to yield a derivation $\mathcal{D}_4$ of *conc* A_2, and to then cut $\mathcal{D}_4$ against $\mathcal{D}_3$ to yield $\mathcal{D}_5$, a derivation of *conc* C.

To see why the type system is not helping enough to determine that this case is correctly reasoned, imagine that we incorrectly wrote the $\multimap$ right rule as

$$\dfrac{\Delta, A, A \Vdash B}{\Delta \Vdash A \multimap B}\multimap R^{bad}$$

which in LLF would be encoded as

$$lolrbad : (hyp\ A \multimap hyp\ A \multimap conc\ B) \multimap conc\ (lol\ A\ B)$$

It is easy to check that this rule destroys cut admissibility: there is a proof using cut

$$\dfrac{\dfrac{\dfrac{\vdots}{A \otimes A \multimap B, A, A \Vdash B}}{A \otimes A \multimap B \Vdash A \multimap B}\multimap R^{bad} \qquad \begin{array}{c}\vdots\\ A, A \multimap B \Vdash B\end{array}}{A \otimes A \multimap B, A \Vdash B}\multimap L$$

but no cut-free proof of $A \otimes A \multimap B, A \Vdash B$ — and the reason for this is the extra linear occurrence of A. Yet there is still a well-typed (but erroneous!) LLF proof case

$$\begin{array}{l} ca/lolbad/principal : \\ \quad \Pi\mathcal{D}_1{:}(hyp\ A_1 \multimap hyp\ A_1 \multimap conc\ A_2). \\ \quad \Pi\mathcal{D}'_1{:}(hyp\ A_1 \multimap conc\ A_2).\Pi\mathcal{D}_2{:}(conc\ A_1). \\ \quad \Pi\mathcal{D}_3{:}(hyp\ A_1 \multimap conc\ C).\Pi\mathcal{D}_4{:}(conc\ A_2). \\ \quad \Pi\mathcal{D}_5{:}(conc\ C). \\ \qquad ca\ \langle p\hat{}(lolrbad\hat{}(\hat{\lambda}x.\hat{\lambda}y.\mathcal{D}_1\hat{}x\hat{}y))\hat{}(\lambda z.loll\hat{}\mathcal{D}_2\hat{}(\hat{\lambda}x.\mathcal{D}_3\hat{}x)\hat{}z), \mathcal{D}_5\rangle \\ \qquad\quad \leftarrow (\Pi y{:}hyp\ A_1.ca\ \langle p\hat{}\mathcal{D}_2\hat{}(\hat{\lambda}x.\mathcal{D}_1\hat{}x\hat{}y), \mathcal{D}_1\hat{}y\rangle) \\ \qquad\quad \leftarrow ca\langle p\hat{}\mathcal{D}_2\hat{}(\hat{\lambda}x.\mathcal{D}'_1\hat{}x), \mathcal{D}_4\rangle) \\ \qquad\quad \leftarrow ca\langle p\hat{}\mathcal{D}_4\hat{}(\hat{\lambda}x.\mathcal{D}_3\hat{}x), \mathcal{D}_5\rangle \end{array}$$

which corresponds to cutting $\mathcal{D}_2$ *twice* into $\mathcal{D}_1$! In a paper proof this would result in transforming

$$\dfrac{\dfrac{\mathcal{D}_1}{\Delta_1, A_1, A_1 \Vdash A_2}}{\Delta_1 \Vdash A_1 \multimap A_2}\multimap R^{bad} \quad \dfrac{\mathcal{D}_2 \quad \mathcal{D}_3}{\Delta_{21} \Vdash A_1 \quad \Delta_{22}, A_2 \Vdash C}\multimap L}{\Delta_1, \Delta_{21}, \Delta_{22} \Vdash C}\,cut$$

into

$$\dfrac{\dfrac{\mathcal{D}_2}{\Delta_{21} \Vdash A_1} \quad \dfrac{\mathcal{D}_2 \quad \mathcal{D}_1}{\Delta_{21} \Vdash A_1 \quad \Delta_1, A_1, A_1 \Vdash A_2}\,cut}{\Delta_1, \Delta_{21}, A_1 \Vdash A_2}\,cut \qquad \mathcal{D}_3}{\Delta_1, \Delta_{21}, \Delta_{21} \Vdash A_2 \qquad \Delta_{22}, A_2 \Vdash C}\,cut$$

$$\Delta_1, \Delta_{21}, \Delta_{21}, \Delta_{22} \Vdash C$$

The fact that two copies of Δ_{21} arrive in the context is a clear mistake, yet the type system LLF and encoding methodology above do not provide any clear mechanism for preventing it: all derivations are quantified by the unrestricted dependent type constructor Π, and carry no information about the linear context they are supposed to be valid in.

4 Preliminary Results

My initial attempt to solve this problem was searching for a type system with a 'linear Π' that supported kinds of the form $A \multimap$ type. The system I arrived at, although technically sound, seemed inconvenient and unattractive in several ways, and moreover did not seem a good fit for typical example metatheorems such as the one given above. Since it was not a very fruitful line of investigation, I will not go into any further detail about it in this document.

Instead, what now seems most promising is an extension of LF with features similar to those found in *hybrid logic*. Hybrid logic [ABM01, Bla00, BdP06, CMS06] was originally developed as an approach to temporal and modal logics with Kripke-like semantics. One posits new language features at a purely syntactic level, which effectively reify the Kripke possible worlds in the semantics. The name 'hybrid' comes from the fact that the resulting logic is somewhere between traditional modal logic and the opposite extreme of simply embedding (the Kripke semantics of) modal logic in first-order logic and reasoning there. Instead, a limited but expressive supply of logical tools for accessing Kripke worlds are provided, leading to a logic that is often (depending on which version is considered) still decidable, and more suited to particular domain applications than general first-order logic.

I claim that similar benefits can be obtained with (at least the LLF fragment of) linear logic. One can allow explicit mention of certain 'resource labels' inspired not by Kripke semantics, but the resource semantics given by Galmiche and Méry [GM03]. Because they nonetheless behave somewhat like Kripke worlds, I use 'world' and 'label' more or less interchangeably in the sequel.

The language of LF extended with these labels (and appropriate type constructors that manipulate them) can express enough concepts of linear logic to generalize LLF, and also has enough expressive power to accurately encode the above theorems as relations.

In the following subsections I will attempt to motivate the design and key ideas of this new system in terms of basic considerations about representing resource consumption, after which I describe the system in more formal detail. Finally, solutions are sketched to the theorem encoding problems for the running example.

4.1 Toward a Hybrid Reconstruction of LLF

We have seen already that a linear hypothesis must be used exactly once, and that this behavior can be enforced by restricting the use of substructural rules in the context.

Consider the following alternate strategy for controlling uses of hypotheses: instead of $\Gamma \vdash M : A$, take as the basic typing judgment $\Gamma \vdash M : A\ [U]$, where U is a mapping of variables in Γ to numbers, indicating how often they are used. Thus we might have

$$x : A, y : B, z : C \vdash c\ x\ x\ y : D\ [x \Rightarrow 2, y \Rightarrow 1, z \Rightarrow 0]$$

Linearity in this system could be imposed after the fact by saying that the only valid typing derivations are those that end with every linear variable being mapped to 1 in U. The typing rule for $\multimap$ could enforce linearity of functions as follows:

$$\frac{\Gamma, x : A \vdash M : B \ [U, x \Rightarrow 1]}{\Gamma \vdash \hat{\lambda} x.M : A \multimap B \ [U]}$$

However, this does not make it any easier to quantify over what resources are used. If we were to come up with a notion of well-formedness of usage specification Us, it would depend on the shape of the current context, and we would likely be back in the same spot of having to quantify over contexts.

The solution is an alternate, more abstract representation of the same information found in usage specifications U. Let there be a separate syntactic class of *worlds*

$$p, q ::= \alpha \mid p * q \mid \epsilon$$

which may be world variables α from the context, combinations $p * q$ of two worlds, or the empty world ϵ. The binary operator $*$ is assumed to form a commutative monoid with ϵ: that is, $*$ is commutative and associative, and $p * \epsilon$, $\epsilon * p$, and p are considered equal.

Worlds can be used to encode information about how often variables are used in a very simple way. If each linear variable $x_1, \ldots, x_n$ in the context is associated with a unique world variable from $\alpha_1, \ldots, \alpha_n$, and an world expression

$$\overbrace{\alpha_1 * \cdots * \alpha_1}^{k_1 \text{ times}} * \cdots * \overbrace{\alpha_n * \cdots * \alpha_n}^{k_n \text{ times}}$$

represents the situation where, for all i, the variable x_i is used k_i times.

The central judgment of the type theory is now

$$\Gamma \vdash M : A[p]$$

read as, "M has type A in context Γ, and resources p are consumed to produce M." Contexts may now include also world variables:

$$\Gamma ::= \cdots \mid \Gamma, \alpha : \mathsf{world}$$

and the rule for typing variables now specifies that to simply use a hypothesis directly requires using no resources:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A[\epsilon]}$$

similarly constants from the signature Σ exist without using any resources:

$$\frac{c : A \in \Sigma}{\Gamma \vdash c : A[\epsilon]}$$

In order to explicate the previously vague notion of how worlds are 'associated' with variables, let there be a new type constructor @ that takes a type and a world, which internalizes the notion of the new judgmental feature $[p]$:

$$A ::= \cdots \mid A@p$$

Its typing rules are

$$\frac{\Gamma \vdash M : A[p]}{\Gamma \vdash M : (A@p)[q]}$$

$$\frac{\Gamma \vdash M : (A@p)[q]}{\Gamma \vdash M : A[p]}$$

Note that neither of these rules introduce any new term constructs: the type $A@p$ is merely a *refinement* of the type A, meaning that $A@p$'s inhabitants are a subset of A's. Now we can give typing rules for $\multimap$, which define linear functions in terms of world labels. They are:

$$\frac{\Gamma, \alpha : \mathsf{world}, x : A@\alpha \vdash M : B[p * \alpha]}{\Gamma \vdash \hat{\lambda} x.M : A \multimap B[p]}$$

$$\frac{\Gamma \vdash M : A \multimap B[p] \qquad \Gamma \vdash N : A[q]}{\Gamma \vdash M \hat{} N : B[p * q]}$$

In the introduction rule, a fresh world variable α is created, and the function argument is hypothesized not at type A, but at type $A@\alpha$ — the information at type A itself it is only available by applying the @ elimination rule and thereby *consuming resource* α. Furthermore, the body of the function *must* consume the resource α exactly once, along with whatever resources p the containing expression was already required to consume.

In the elimination rule, if the function consumes resources p, and its argument consumes resources q, then the application consumes the combined resources $p * q$, just as the usual rule for $\multimap$ elimination features a combination Δ_1, Δ_2 of contexts. Note, however, that here the context Γ is shared across both premises, because resource use is completely accounted for by the world annotations.

4.2 Hybrid Logical Connectives

The main reason to tease out the information about resource use into a separate object is to be able to manipulate it with other logical connectives, and corresponding type operators. The most common connectives found in the literature on hybrid logic are the universal quantifier $\forall \alpha.A$ over worlds, and a somewhat more peculiar quantifier, the 'local binding' $\downarrow \alpha.A$ operator, which binds a variable α to the *current* world at the time it is analyzed.

Both of these type constructors again introduce no term constructors. The typing rules for $\downarrow$ allows access to the 'current world' via its typing rules as follows:

$$\frac{\Gamma \vdash M : (\{p/\alpha\}A)[p]}{\Gamma \vdash M : (\downarrow\alpha.A)[p]}$$

$$\frac{\Gamma \vdash M : (\downarrow\alpha.A)[p]}{\Gamma \vdash M : (\{p/\alpha\}A)[p]}$$

And the typing rules for $\forall$ are:

$$\frac{\Gamma, \alpha : \mathsf{world} \vdash M : A[p]}{\Gamma \vdash M : (\forall\alpha.A)[p]}$$

$$\frac{\Gamma \vdash M : (\forall\alpha.A)[p] \qquad \Gamma \vdash q : \mathsf{world}}{\Gamma \vdash M : (\{q/\alpha\}A)[p]}$$

Moreover, we can add universal quantification to the language of kinds:

$$K ::= \cdots \mid \forall\alpha.K$$

Since this only involves quantification over objects of a particular syntactic sort (that is, worlds) which themselves are not assumed linearly (that is, once we hypothesize a world to exist, there is no linear occurrence discipline that says we must use it in some label) this extension to the language of kinds involves far less complication than would be required by adding $\multimap$. The ability to write $\forall$ in kinds will be central to the technique I propose below for encoding statements of metatheorems in HLF.

Before getting to that, however, it is expedient to explain how these extra connectives are already useful for providing (together with ordinary function types) a decomposition of $\multimap$. It turns out that the terms of type $A \multimap B$ are in bijective correspondence with those of type

$$\forall\alpha.\downarrow\beta.(A@\alpha) \to (B@(\beta * \alpha))$$

(the scope of $\forall$ and $\downarrow$ are generally both meant to extend as far to the right as possible) In fact, we can take this type as a definition of $\multimap$ as mere syntactic sugar. One effect this has is to collapse λ and $\hat{\lambda}$ (and $M\ N$ with $M \hat{} N$) as term constructors, which I argue in Section 4.3.4 to be benign.

To see why this definition of $\multimap$ works, notice that for every derivation

$$\frac{\Gamma, \alpha : \mathsf{world}, x : A@\alpha \vdash M : B[p * \alpha]}{\Gamma \vdash \hat{\lambda}x.M : A \multimap B[p]}$$

that would take place in the system with a 'first-class' $\multimap$, there is a derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \alpha : \mathsf{world}, x : A@\alpha \vdash M : B[p * \alpha]}{\Gamma, \alpha : \mathsf{world}, x : A@\alpha \vdash M : B@(p * \alpha)[p]}}{\Gamma, \alpha : \mathsf{world} \vdash \lambda x.M : A@\alpha \to B@(p * \alpha)[p]}}{\Gamma, \alpha : \mathsf{world} \vdash \lambda x.M : \downarrow\beta.A@\alpha \to B@(\beta * \alpha)[p]}}{\Gamma \vdash \lambda x.M : \forall\alpha.\downarrow\beta.A@\alpha \to B@(\beta * \alpha)[p]}$$

in the system with a defined $\multimap$, and a similar correspondence holds for the elimination rule.

4.3 HLF Formal Description

In order to give a formal account of HLF, I take advantage of a pair of fairly modern pieces of logical frameworks machinery.

One is the idea of *hereditary substitution*, pioneered by the formulation of CLF, which can be seen as a Curry-Howard analogue to the logical technique of structural cut elimination [Pfe95, Pfe00]. It allows the following approach to equality on terms: Instead of first considering all possible terms, and thereafter identifying them up to β and η conversion, consider *only* terms already in canonical (β-normal, η-long) form. One constrains the syntax of the system to only allow these terms to be written.

Ordinarily, substitution of a term for a variable — even if both terms involved were originally themselves canonical — may create a beta-redex, e.g. $\{\lambda x.\lambda z.M/y\}(y\ c_1\ c_2) = (\lambda x.\lambda z.M)\ c_1\ c_2$. However, the syntax of the language of terms is supposed to rule out such non-β-normal terms. The definition of substitution must therefore be modified to carry out β-reductions, which themselves involve substitutions, and so on hereditarily, until the resulting term is normal.

The key technical result to be shown, then, is that on well-typed terms, this process always terminates and yields an answer of the correct type. One benefit of using hereditary substitution and considering only canonical forms of terms is that equality of terms and types is defined by α-equivalence alone, vastly simplifying previous accounts of decidability of typechecking, which typically involved rather complicated logical relations arguments.

The second device I employ is a presentation in *spine form*, [CP97b] which is the λ-calculus analogue of the logical notion of *focussing* developed by Andreoli [And92], and later extended by Girard [Gir01]. Focussing enjoins us to carry out eliminations of certain logical connectives, (the 'negative' connectives, in Girard's terminology) all at once in a sequence, until we reach a positive proposition, or else perhaps an atomic proposition. Conspicuously, all of the type operators in HLF are in the negative fragment.

All arguments to a function, and all projections from a pair are therefore constrained to appear in *spines*, and all terms involving these eliminations take

the form of a head, that is a variable or constant, applied to a spine that reduces it to base type. For instance, instead of $\lambda x.\lambda y.(\pi_1(c\ x))\ y$, we write $\lambda x.\lambda y.c \cdot (x; \pi_1; y)$. The spine is a list of instructions: first apply to x, then take the first component, then apply to y.

4.3.1 Language

The complete syntax of the language is as follows:

$$
\begin{array}{rrl}
\text{Worlds} & p, q, r & ::= \alpha \mid p * q \mid \epsilon \\
\text{Kinds} & K & ::= \Pi x{:}A.K \mid \forall \alpha.K \mid \text{type} \\
\text{Types} & A & ::= \Pi x{:}A.B \mid a \cdot S \mid \forall \alpha.B \mid \downarrow \alpha.B \mid A@p \mid A \ \& \ B \mid \top \\
\text{Terms} & M & ::= \lambda x.M \mid c \cdot S \mid x \cdot S \mid \langle M_1, M_2 \rangle \mid \langle\rangle \\
\text{Spines} & S & ::= () \mid (M; S) \mid (\pi_i; S) \\
\text{Contexts} & \Gamma & ::= \cdot \mid \Gamma, x : A \mid \Gamma, \alpha : \mathsf{world}
\end{array}
$$

4.3.2 Judgments

There are four typing judgments:

$$
\begin{array}{l}
\Gamma \vdash p : \mathsf{world} \\
\Gamma \vdash M : A[p] \\
\Gamma \vdash A : \text{type} \\
\Gamma \vdash S : A[p] > C[r]
\end{array}
$$

The first simply checks that a world expression is well-formed. The second is type-checking parameterized by a world as discussed above, and the third checks well-formedness of types, notably *not* parameterized by any world. The last is the typing judgment is on spines: $\Gamma \vdash S : A[p] > C[r]$ says that S is a spine that, if a head (variable or constant) of type $A@p$ is applied to it, it will yield a term of type C consuming resources r.

4.3.3 Type Checking

We write R to stand for either $x \cdot S$ or $c \cdot S$. The relation $\equiv_{ACU}$ is equivalence of worlds up to **A**ssociativity and **C**ommutativity of $*$ and **U**nit laws for $*$ and ϵ.

$$
\frac{\Gamma \vdash R : a \cdot S[p] \qquad S =_\alpha S' \qquad p \equiv_{ACU} q}{\Gamma \vdash R : a \cdot S'[q]}
$$

$$
\frac{\Gamma, x : A \vdash M : B[p]}{\Gamma \vdash \lambda x.M : \Pi x{:}A.B[p]} \qquad \frac{\Gamma, \alpha : \mathsf{world} \vdash M : B[p]}{\Gamma \vdash M : \forall \alpha.B[p]}
$$

$$
\frac{\Gamma \vdash M : (\{p/\alpha\}B)[p]}{\Gamma \vdash M : \downarrow \alpha.B[p]} \qquad \frac{\Gamma \vdash M : A[q]}{\Gamma \vdash M : A@q[p]}
$$

$$\frac{\Gamma \vdash M_1 : A_1[p] \qquad \Gamma \vdash M_2 : A_2[p]}{\Gamma \vdash \langle M_1, M_2 \rangle : A_1 \;\&\; A_2[p]} \qquad \frac{}{\Gamma \vdash \langle\rangle : \top[p]}$$

$$\frac{c : A \in \Sigma \qquad \Gamma \vdash S : A[\epsilon] > C[r]}{\Gamma \vdash c \cdot S : C[r]}$$

$$\frac{x : A \in \Gamma \qquad \Gamma \vdash S : A[\epsilon] > C[r]}{\Gamma \vdash x \cdot S : C[r]}$$

4.3.4 Spine Typing

$$\frac{}{\Gamma \vdash () : a \cdot S[p] > a \cdot S[p]}$$

$$\frac{\Gamma \vdash M : A[\epsilon] \qquad \Gamma \vdash S : \{M/x\}B[p] > C[r]}{\Gamma \vdash (M; S) : \Pi x{:}A.B[p] > C[r]}$$

$$\frac{\Gamma \vdash q : \mathsf{world} \qquad \Gamma \vdash S : \{q/\alpha\}B[p] > C[r]}{\Gamma \vdash S : \forall \alpha.B[p] > C[r]}$$

$$\frac{\Gamma \vdash S : \{p/\alpha\}B[p] > C[r]}{\Gamma \vdash S : \downarrow \alpha.B[p] > C[r]}$$

$$\frac{\Gamma \vdash S : A[q] > C[r]}{\Gamma \vdash S : A@q[p] > C[r]}$$

$$\frac{\Gamma \vdash S : A_i[p] > C[r]}{\Gamma \vdash (\pi_i; S) : A_1 \;\&\; A_2[p] > C[r]}$$

It is worth noting that types do not appear in terms, for this means that worlds (which do appear in types, via $A@p$) in turn do not appear in terms. Because of this, equality of terms remains simply the identity up to α-conversion.

The term language, we claim, is essentially identical to that of LLF, if it were specified in spine form, and with syntactically enforced canonical forms. Linear and ordinary lambdas (and respectively, their applications) are identified, but this identification is consistent with the absence of type annotations that already comes with the canonical-forms only paradigm. Arguably the difference between unrestricted functions and linear functions is just a difference of type annotation on the argument, and so need not be recorded in the term.

4.3.5 Hereditary Substitution

Substitution $\{N/x\}^C$ of N for x at type C is a partial function from types to types, from terms to terms, and from spines to spines. It is annotated with a type to ensure that calls to the substitution function always terminates with an answer, or (this possibility being sensible because it is a partial function) being undefined. Other, less algorithmic presentations are possible, but we are interested in showing that there is a terminating type-checking algorithm, so this is the most appropriate.

Substitution $\{p/\alpha\}^{\mathsf{world}}$ of the world p for the world variable α is also a function from types to types, and from worlds to worlds.

The definition of substitution in almost all cases is straightforward homomorphism. Let σ abbreviate $\{N/x\}^C$ and σ' abbreviate $\{p/\alpha\}^{\mathsf{world}}$.

$$\sigma(\lambda y.M) = \lambda y.(\sigma M)$$
$$\sigma\langle M_1, M_2\rangle = \langle \sigma M_1, \sigma M_2\rangle \qquad \sigma\langle\rangle = \langle\rangle$$
$$\sigma(y \cdot S) = y \cdot (\sigma S) \quad (\text{if } x \neq y) \qquad \sigma(c \cdot S) = c \cdot (\sigma S)$$
$$\sigma() = ()$$
$$\sigma(M; S) = (\sigma M; \sigma S)$$
$$\sigma(\pi_i; S) = (\pi_i; \sigma S)$$

$$\sigma\epsilon = \epsilon \qquad \sigma(p * q) = \sigma p * \sigma q$$
$$\{p/\alpha\}^{\mathsf{world}}\alpha = p$$
$$\sigma\beta = \beta \qquad (\text{if } \alpha \neq \beta)$$

In the case of substitution into a λ, the usual convention is taken that we first rename bound variables if necessary to be distinct from the variable being substituted for, in order to avoid capture.

The remaining case is when a variable being substituted for is actually used, which invokes another partial function, $[N \mid S]^C$, pronounced 'N reduces against S at type C'.

$$\{N/x\}^C(x \cdot S) = [N \mid \{N/x\}^C S]^C$$

The 'reduces' function carries out β-reductions, and is mutually recursive with substitution. It is defined as follows:

$$[\lambda x.M \mid (N; S)]^{\Pi x:A.B} = [\{N/x\}^A M \mid S]^B$$
$$[R \mid ()]^{a \cdot S} = R$$
$$[M \mid S]^{\forall \alpha.A} = [M \mid S]^A$$
$$[M \mid S]^{\downarrow \alpha.A} = [M \mid S]^A$$
$$[M \mid S]^{A@p} = [M \mid S]^A$$
$$[\langle M_1, M_2\rangle \mid (\pi_i; S)]^{A_1 \& A_2} = [M_i \mid S]^{A_i}$$

($[N \mid S]^C$ is undefined otherwise, if none of the above clauses apply.)

4.4 Metatheory of HLF

As mentioned, the metatheory of logical frameworks in the LF family traditionally involved spending considerable effort [HP01, HP05] showing that an algorithm for typechecking coincided with a (not obviously decidable) definition of the typing judgment that involved a *type conversion rule*

$$\frac{\Gamma \vdash M : A \qquad A \equiv_{\beta\eta} B}{\Gamma \vdash M : B}$$

The more recent approach, considering canonical forms only, does not have this rule, and pushes the work of establishing $\beta\eta$-equivalence into the definition of substitution. Therefore the requisite theorems are those that establish that definition of substitution is correct. Since substitution carries out β reductions hereditarily, it is not apparent that it is well-founded as a function definition. However, since every substitution and reduction operation above is indexed by the type of the object being substituted, the type index can be seen to always decrease, and this ensures termination. The recursive definition of $\{N/x\}^C M$ and $[N \mid S]^C$ is well-founded according to a lexicographic order that puts priority on the size of C decreasing, and then is ordered by the subject M, or in the case of reduction, simultaneously the size of N and S. This is essentially the same termination order as in the structural proof of cut admissibility [Pfe95, Pfe00].

Given that substitution is a well-defined partial function, we want to be certain that it always yields an answer on well-typed terms, and that that answer is itself appropriately typed.

Theorem 4.1 (Substitution Property) *Assume Γ, A, p are well-formed. Suppose $\Gamma \vdash M : A[\epsilon]$, and $\Gamma \vdash r : \mathsf{world}$. Let σ abbreviate $\{M/x\}^A$, and σ' abbreviate $\{r/\alpha\}^{\mathsf{world}}$.*

- *If $\Gamma, x : A, \Gamma' \vdash N : B[p]$ then $\Gamma, \sigma\Gamma' \vdash \sigma N : \sigma B[p]$.*
- *If $\Gamma, x : A, \Gamma' \vdash S : A[p] > C[q]$, then $\Gamma, \sigma\Gamma' \vdash \sigma S : \sigma A[p] > \sigma C[q]$.*
- *If $\Gamma, x : A, \Gamma' \vdash B : \text{type}$ then $\Gamma, \sigma\Gamma' \vdash \sigma B : \text{type}$.*
- *If $\Gamma, \alpha : \mathsf{world}, \Gamma' \vdash N : B[p]$ then $\Gamma, \sigma'\Gamma' \vdash N : \sigma' B[\sigma' p]$.*
- *If $\Gamma, \alpha : \mathsf{world}, \Gamma' \vdash S : A[p] > C[q]$, then $\Gamma, \sigma'\Gamma' \vdash S : \sigma' A[\sigma' p] > \sigma' C[\sigma' q]$.*
- *If $\Gamma, \alpha : \mathsf{world}, \Gamma' \vdash B : \text{type}$ then $\Gamma, \sigma'\Gamma' \vdash \sigma' B : \text{type}$.*
- *If $\Gamma \vdash N : A[p]$ and $\Gamma \vdash S : A[p] > C[q]$, then $\Gamma \vdash [N \mid S]^A : C[q]$.*

Proof By lexicographic induction on the type of the object being substituted, and subsequently the typing derivation of the object being substituted into. Some standard lemmas are required to show that consecutive substitutions appropriately commute with each other. ■

The substitution property justifies replacing the possibility of carrying out β-reductions in types with the requirement that all types and terms mentioned are already *beta*-normal. There is a parallel theorem that must be shown to justify the requirement that all types and terms mentioned are already η-long. It states that for any variable, there is an η-expansion of it, which behaves as an identity for the operation of substitution.

Theorem 4.2 (Identity Property) *For every well-formed type A, there is a term $\eta^*_A(x)$ such that*

- $x : A \vdash \eta^*_A(x) : A$.
- *If* $\Gamma, x : A \vdash M : B$, *then* $[\eta^*_A(y)/y]^A M = M$.
- *If* $\Gamma \vdash M : A$, *then* $[M/y]^A(\eta^*_A(y)) = M$.

Proof After making a slight generalization to η-expansion of a partially applied head, the proof proceeds by lexicographic induction first on the type A, and subsequently on the structure of the term M, if any. ■

Decidability of typechecking is in our case still not quite immediate, for the $\forall$ left rule

$$\frac{\Gamma \vdash q : \mathsf{world} \qquad \Gamma \vdash S : \{q/\alpha\}B[p] > C[r]}{\Gamma \vdash S : \forall\alpha.B[p] > C[r]}$$

leaves the identity of the instantiating world q up to nondeterministic guessing. However, we can proceed with typechecking leaving a free variable in place of q, and eagerly solve constraints $\equiv_{ACU}$ that arise from the rule

$$\frac{\Gamma \vdash R : a \cdot S[p] \qquad S =_\alpha S' \qquad p \equiv_{ACU} q}{\Gamma \vdash R : a \cdot S'[q]}$$

as they arise. In this way typechecking reduces to unification over a term language with one constant and one binary operator that together satisfy the axioms of a commutative monoid. Fortunately, this problem is known to be decidable [Sti81].

Lastly we can show that the system is a conservative generalization of an appropriate presentation of LLF: assume as mentioned above that it is given in spine form, with canonical forms syntactically enforced, and with the evident definition of hereditary substitution. Let it have only one λ (and one notion of application) for both linear and unrestricted functions. As pointed out in Section 4.3.4, this syntactic collapse is entirely appropriate in canonical-forms-only style, since it is analogous to the removal of type annotations on function arguments, which are already absent.

Make the following definitions, which relate LLF contexts to HLF contexts, formalizing the intuition of Section 4.1:

Definition Suppose Δ is an LLF context, of the form $x_1 : A_1, \ldots x_n : A_n$.
Let $\Delta^{@}$ be the HLF context

$$(\alpha_1 : \mathsf{world}, x_1 : A_1@\alpha_1, \ldots, \alpha_n : \mathsf{world}, x_n : A_n@\alpha_n)$$

Let α_Δ be the world $\alpha_1 * \cdots * \alpha_n$.
Let $\Delta\lfloor_p$ be the LLF context $x_{i_1} : A_{i_1}, \ldots, x_{i_n} : A_{i_n}$, if $p = \alpha_{i_1} * \cdots * \alpha_{i_m}$.

Now the following results hold:

Theorem 4.3 (Soundness) *Suppose Δ is a valid LLF context, and A is a valid LLF type.*

- *If $\Gamma, \Delta^{@} \vdash M : A[p]$ then $\Gamma; \Delta\lfloor_p \vdash_{LF} M : A$.*
- *If $\Gamma, \Delta^{@} \vdash S : A[p] > C[r]$ and $r =_{ACU} p * q$, then $\Gamma; \Delta\lfloor_q \vdash_{LF} S : A > C$.*

Proof By induction on the typing derivations. ■

Theorem 4.4 (Completeness) *Suppose Γ is well-formed, and suppose A is a valid type in Γ.*

- *If $\Gamma; \Delta \vdash_{LF} M : A$, then $\Gamma, \Delta^{@} \vdash M : A[\alpha_\Delta]$*
- *If $\Gamma; \Delta \vdash_{LF} S : A > C$, then $\Gamma, \Delta' \vdash S : A[p] > C[q]$, for any Δ' that extends $\Delta^{@}$ and any p such that $\Delta' \vdash p : \mathsf{world}$, for some $q \equiv_{ACU} \alpha_\Delta * p$.*

Proof By induction on the typing derivations. ■

4.5 Application: Cut Admissibility in Linear Logic

We continue with the running example, and show how the problems encoding linear cut admissibility as a metatheorem in LLF can be solved with HLF. The desired cut admissibility theorem can now be encoded as

$$\begin{array}{l} ca : \forall\alpha.\forall\beta. \\ \quad (conc\ A)@\alpha \\ \quad \rightarrow (hyp\ A \multimap conc\ C)@\beta \\ \quad \rightarrow (conc\ C)@(\alpha * \beta) \\ \quad \rightarrow \text{type} \end{array}$$

The reason this encoding is correct follows directly from the form that terms can take at types $(conc\ A)@p$, $(hyp\ A \multimap conc\ C)@q$, and $(conc\ C)@(p * q)$ for particular worlds p and q. Using the definitions in the previous section, assume a context Γ is of the form $\Delta^{@}$ for an LLF context Δ. Any instance

$$ca\ M_1\ M_2\ M_3$$

of the type family ca has the property that there exist worlds p, q (instantiating the variables α, β) such that

$$\Delta^{@} \vdash M_1 : \mathit{conc}\ A[p]$$
$$\Delta^{@} \vdash M_2 : \mathit{hyp}\ A \multimap \mathit{conc}\ C[q]$$
$$\Delta^{@} \vdash M_3 : \mathit{conc}\ C[p * q]$$

The second derivation, for example, by inversion on the rules defining $\multimap$, indicates that M_2 is of the form $\lambda x.M_2'$, and

$$\Delta^{@}, \alpha : \mathsf{world}, x : \mathit{hyp}\ A@\alpha \vdash M_2' : \mathit{conc}\ C[\alpha_\Delta]$$

or in other words

$$(\Delta, x \hat{:} \mathit{hyp}\ A)^{@} \vdash M_2' : \mathit{conc}\ C[\alpha_{\Delta,x}]$$

and M_2' must be a term that linearly uses every hypothesis in $\Delta, x \hat{:} \mathit{hyp}\ A$. We can see therefore that the kind of this type family is sufficiently precise to allow as inputs pairs of derivations that consume different sets of resources p and q respectively from the amalgamated context Γ, and enforces that the output derivation must use both p and q together.

There is a concern worth discussing about whether it is necessary to enforce disjointness of p and q. HLF as presented cannot (at least it is not obvious that it can naturally) capture a requirement of disjointness, but this is no obstacle to the correctness of the present metatheorem. Its interpretation is that for *every* set of resources p and q, the two input derivations using p and q respectively can be transformed into a derivation using $p * q$. For linear cut elimination we only need this result for when p and q happen to be disjoint, and so it follows as a special case of the metatheorem.

A typical clause of the HLF proof of linear cut admissibility looks much like the existing proof LLF, except that the types of Π-bound variables are modified to account for resource use. The principal cut case of $\multimap$ is now encoded as

$$\begin{array}{l}
\mathit{ca/lol/principal} : \forall\alpha.\forall\beta.\forall\gamma. \\
\quad \Pi\mathcal{D}_1{:}(\mathit{hyp}\ A_1 \multimap \mathit{conc}\ A_2)@\alpha.\Pi\mathcal{D}_2{:}(\mathit{conc}\ A_1)@\beta. \\
\quad \Pi\mathcal{D}_3{:}(\mathit{hyp}\ A_2 \multimap \mathit{conc}\ C)@\gamma.\Pi\mathcal{D}_4{:}(\mathit{conc}\ A_2)@(\alpha * \beta). \\
\quad \Pi\mathcal{D}_5{:}(\mathit{conc}\ C)@(\alpha * (\beta * \gamma)). \\
\qquad \mathit{ca}\ (\mathit{lolr}\ \lambda x.\mathcal{D}_1\ x)\ (\mathit{loll}\ \mathcal{D}_2\ (\lambda x.\mathcal{D}_3\ x))\ \mathcal{D}_5 \\
\qquad\quad \leftarrow \mathit{ca}\ \mathcal{D}_2\ (\lambda x.\mathcal{D}_1\ x)\ \mathcal{D}_4 \\
\qquad\quad \leftarrow \mathit{ca}\ \mathcal{D}_4\ (\lambda x.\mathcal{D}_3\ x)\ \mathcal{D}_5
\end{array}$$

which represents the same reasoning as in 3.1, only here the behavior of the linear contexts in the object language is correctly captured. Note in particular how the variables α, β, γ correspond exactly to $\Delta_1, \Delta_{21}, \Delta_{22}$ in the informal reasoning. Moreover, trying to translate the example incorrect proof case given above leads directly to a type error, because the types $\mathit{conc}\ C@(\alpha * \beta * \gamma)$ and $\mathit{conc}\ C@(\alpha * \beta * \beta * \gamma)$ (the types of the required conclusion, and the result of the reasoning contained in the bad proof, respectively) are not equal.

I claim the similarity of this proof in HLF to the existing one encoded in LLF is an important benefit of the system I propose: that in order to write

proofs, very little if any new thinking is required outside the proof techniques already developed for LLF. It is likely that type reconstruction can discover the correct types (and therefore world labels) for implicitly Π-bound variables in most cases, and so even the relatively minor extra work involved in writing them might be avoided.

5 Proposal Core

5.1 HLF Type Theory

I propose first of all to settle on a final type theory for HLF, ideally not very different (perhaps not at all) from the present one. It will need to satisfy the metatheoretic properties expected of such an extension of LF. In the canonical-forms-only style, since the complexity of determining equality of terms is pushed into the definition of substitution, a central result is showing that the substitution property (Theorems 4.1) and identity property (4.2) hold.

Furthermore for the system to be usable in practice we must show that typechecking is decidable, and that canonical forms are structured as required for known encodings to be adequate.

5.2 Logic Programming

The main task after the type theory is established is to understand how logic programming works over the type theory. This is so that the later work of devising algorithms for checking metatheoretic properties such as coverage is done with a clear idea of their intended meaning in terms of logic program execution.

Since logic programming over LF [Pfe91] is well understood, it remains to understand what incremental effect adding labels has on the known operational semantics.

This can be divided into two parts. First, to understand how it works globally assuming the particular constraint solving problem for worlds (and the commutative monoid equational theory already described) has been adequately solved. In other words, we assume the constraint solving problem has an 'oracle' which either outputs a most general unifier, or signals that it cannot find one, and outputs remaining constraints. Similar to the eager constraint-solving of higher-order unification constraints, it may be possible to proceed even when there are not immediately most general unifiers, and allow later instantiations to put unification problems back into a more tractable problem fragment.

To develop the remainder of the algorithm even assuming world constraint-solving is understood may still be somewhat nontrivial, because of subtleties in the behavior of higher-order unification in the presence of linearity [CP97a].

An example of this concern arises in trying to solve the unification problem

$$x : B \vdash X \mathbin{\hat{}} x \doteq c \mathbin{\hat{}} (Y_1\ x) \mathbin{\hat{}} (Y_2\ x) \qquad (*)$$

in a signature where a is a base type and $c : a \multimap a \multimap a$, for the free variables $X : B \multimap a, Y_1, Y_2 : B \to a$. If we simply run ordinary unification on this, ignoring linearity at first, we get the solution

$$X := \lambda x.c \mathbin{\hat{}} (Y_1\ x) \mathbin{\hat{}} (Y_2\ x)$$

which, on its face, is not linearly valid: x is potentially used twice, possibly not at all, depending on the instantiations of Y_1, Y_2. The answer is that there are two maximally general solutions, which are incomparable in the instantiation order,

$$(X := \lambda x.c\ (Z_1 \mathbin{\hat{}} x)\ W_2, Y_1 := \lambda x.Z_1 \mathbin{\hat{}} x, Y_2 := W_2)$$

$$X := \lambda x.c\ W_1\ (Z_2 \mathbin{\hat{}} x), Y_1 := W_1, Y_2 := \lambda x.Z_2 \mathbin{\hat{}} x$$

where $Z_1, Z_2 : B \multimap A$ and $W_1, W_2 : A$.

The message to be taken away from this example that is seems *linear* higher-order unification problems cannot be solved by doing *ordinary* higher-order unification and filtering out only those instances of the solution that are well-typed as linear terms.

However, consider the same unification problem in the language of HLF. In it the variable X gets the type $\forall\alpha.B@\alpha \to a@\alpha$, and c has the type (expanding the HLF definition of $\multimap$)

$$\forall\beta_1, \beta_2.(a@\beta_1) \to (a@\beta_2) \to a$$

For the expression

$$X := \lambda x.c\ (Y_1\ x)\ (Y_2\ x)$$

to be well-typed, it must be that

$$Y_i\ x : o_i[p_i]$$

such that $p_1 * p_2 = \alpha$.

The solutions to the linear unification problem are exactly the HLF solutions satisfying this constraint. There is still not a most general unifier, but for this example, in contrast to LLF, the 'two-phase' strategy of doing ordinary higher-order unification and subsequently imposing linearity constraints does yield a complete set of unifiers, because of the expressivity of HLF's language of worlds.

Moreover, the formalism of HLF suggests an explanation for why the above example fails to have a most general unifier, by way of an analogy to the use of patterns in higher-order unification [DHKP96]. Briefly, a pattern is a term whose head is a free variable X applied to a sequence of distinct bound variables (and importantly not constants or other terms) $x_1, \ldots, x_n$. Unification equations one side of which are a pattern can be immediately simplified, and if the other side has no free variables at all, immediately solved.

There is a sense in which the left hand side of $(*)$ is *not* a pattern, in that $X \mathbin{\hat{}} x$ hides an application to ϵ, which is not a variable, but a constant. The type of X, of X is $\forall\alpha.B@\alpha \to a@\alpha$, this ϵ the instantiation of α. I conjecture that there is a generalization of the notion of higher-order pattern that includes

these 'applications' to world expressions, and predict that unification problems involving these generalized patterns satisfy the same simplification properties already enjoyed by patterns.

Once the general notion of unification is figured out, I can take apart the black box and actually address the constraint solving problem itself, looking to prior work on ACU unification. It is known that the unifiability problem for ACU is decidable, but it still must be determined under which circumstances most general unifiers are guaranteed. I conjecture that the image of the translation of LLF is, informally speaking, 'well-behaved', but already it does not possess MGUs in every case, as witnessed by simple equations like $\exists\beta_1{:}\mathsf{world}.\exists\beta_2{:}\mathsf{world}.\beta_1 * \beta_2 \doteq \alpha$, which can indeed arise from type-checking, e.g.

$$\alpha : \mathsf{world}, x : a@\alpha, f : \top \multimap \top \multimap o \vdash f\ \top\ \top : o$$

This term typechecks if and only if there is some way to use up the resource α given two resource-sinks $\top$, which arises as precisely the equation above; it has two maximally general solutions, $\beta_1 := \epsilon, \beta_2 := \alpha$ and $\beta_1 := \alpha, \beta_2 := \epsilon$, neither of which is an instance of the other.

It is the case that full ACU unification[1] is required for typechecking general HLF terms. The reduction from an arbitrary ACU unification problem to a typechecking problem is rather simple:

Given a set of equations

$$E = \{p_1 \doteq q_1, \ldots, p_n \doteq q_n\}$$

over world variables $\alpha_1, \ldots, \alpha_{n'}, \beta_1, \ldots \beta_m$, then there exists a set of $r_1, \ldots, r_m$ such that every equation in $\{r_1/\beta_1\}^{\mathsf{world}} \cdots \{r_m/\beta_m\}^{\mathsf{world}} E$ is true, if and only if the typing

$$\alpha_1 : \mathsf{world}, \ldots, \alpha_{n'} : \mathsf{world} \vdash M : A$$

holds in a signature with one base type $o : \mathrm{type}$, where

$$\begin{aligned} M &= \lambda f.f \cdot (\lambda x_1.x_1; \cdots; \lambda x_n.x_n) \\ A &= (\forall\beta_1 \ldots \forall\beta_m.Id_1 \to \cdots \to Id_n \to o) \to o \\ Id_i &= o@p_i \to o@q_i \end{aligned}$$

5.3 Metatheory of Logic Programming

The three main metatheoretic properties of logic programs we care about are modes, coverage, and termination.

Mode checking is the simplest of the three, though it is not clear how worlds and modes interact. For the time being, all examples I have treat worlds as input arguments, but conceivably it would be useful to be able to express nontrivial computations that yield worlds as output. In section 6.3 I suggest how this might be accomplished.

[1] Although not, as is sometimes found in the literature, in the sense of allowing arbitrary extra term constructors apart from the binary function symbol and nullary unit.

Coverage is probably the hardest task. In previous work [SP03], coverage is approached through the notions of immediate coverage, and splitting. Whether a clause immediately covers some subgoal depends on higher-order pattern unification [DHKP96], in possibly a different variant than is required for mere type-checking. The notion of pattern is already something which may require attention, as alluded to in Section 5.2. However, it seems that with the way linearity is implemented in HLF in terms of a separation of worlds and terms, it may come out to something more simple, so that the pattern language only needs to be extended to handle pairs, which is relatively straightforward and already adequately studied [Dug98].

Finally, I have to consider termination. Since worlds are always finite concatenations of world parameters, there is a simple and natural termination ordering, one that is slightly more general than the subterm ordering would be for ordinary terms, namely the length of such sequences. It would be desirable therefore to devise an example where it is useful to 'swap' world variables around in a less uniform way while nonetheless reducing overall length.

5.4 Applications

As part of the thesis I intend to establish that the framework is in fact capable of expressing interesting properties about stateful deductive systems. The current set of examples I believe to work includes extending the cut admissibility theorem for the linear sequent calculus, and the type preservation theorem for MiniML with references due Cervesato and Pfenning, to precise statements and effective extensions of their proofs. Also I can achieve a simple proof of correctness the logic program that nondeterministically permutes a list using the linear context.

6 Additional Potential Thesis Work

Having committed to the core of the proposal, there remain a number of other ideas that I would like to pursue as part of the thesis project, if there is enough time.

6.1 Ordered Logic

Ordered logic, a substructural logic that eliminates exchange as well as contraction and weakening, was studied by Lambek [Lam58] in the context of mathematical linguistics, and later studied by Polakow [Pol01], who built the logical framework OLF based on it. Since the proposed type system here (much like other proposals of labelled deduction) cleanly separates the algebra of world-labels from the rest of the logic, I expect it to be able to encode OLF and reasoning about systems encoded in it by modifying the algebra on worlds.

For instance, the ordered linear arrows $\twoheadrightarrow$ and $\rightarrowtail$ that append a hypothesis to the right (respectively to the left) side of the ordered context can be simulated

by adding a new associative, but not commutative, operator $\bullet$ to worlds:

$$p ::= \cdots \mid p \bullet q$$

Given this, the encoding of $\multimap$ can be modified to yield

$$A \twoheadrightarrow B =_{def} \forall\alpha.{\downarrow}\beta.A@\alpha \rightarrow B@(\beta \bullet \alpha)$$
$$A \rightarrowtail B =_{def} \forall\alpha.{\downarrow}\beta.A@\alpha \rightarrow B@(\alpha \bullet \beta)$$

Here are several simple examples that seem promising as applications for the substructural metatheory I have worked out so far:

6.1.1 List Reverse

Polakow's thesis [Pol01] discusses how the ordered context can be used as a queue to reverse a list. Assuming lists are defined by

elt : type
$list$: type
$nil : list$
$cons : elt \rightarrow list \rightarrow$ type

the program is

$queued : elt \rightarrow$ type
$rev : list \rightarrow list \rightarrow$ type
$rev/nil : rev\ L\ L$
$rev/incons : rev\ (cons\ H\ TL)\ K \twoheadleftarrow (queued\ H \twoheadleftarrow rev\ TL\ K)$
$rev/outcons : rev\ nil\ (cons\ H\ K) \twoheadleftarrow queued\ H \twoheadleftarrow rev\ nil\ K$

Here one might like to prove that this implementation of reverse is correct with respect to a definitional implementation in pure LF:

$drev : list \rightarrow list \rightarrow$ type
$revacc : list \rightarrow list \rightarrow list \rightarrow$ type
$drev/rule : drev\ L\ K \leftarrow revacc\ L\ nil\ K$
$revacc/cons : revacc\ (cons\ H\ TL)\ ACC\ K \leftarrow revacc\ TL\ (cons\ H\ ACC)\ K$
$revacc/nil : revacc\ nil\ ACC\ ACC$

This can be stated at the top level as

$$correct : rev\ L\ K \rightarrow drev\ L\ K \rightarrow \text{type}$$

which then requires an obvious lemma

$$lemma1 : rev\ L\ K \rightarrow revacc\ L\ nil\ K \rightarrow \text{type}$$

which in turn requires a less apparent generalization. We must account for the intermediate stages of execution of *rev*, during which the ordered context is populated. The solution is to write a predicate which mediates between the state of the ordered context, and an ordinary list that it represents:

$$
\begin{array}{l}
med : list \to \text{type} \\
med/nil : med\ nil \\
med/cons : \forall p.\forall q.(med\ (cons\ H\ TL))@(p \bullet q) \\
\qquad \leftarrow (med\ K)@p \\
\qquad \leftarrow (queued\ H)@q
\end{array}
$$

The contract of *med* is that the type $(med\ L)@p$ is inhabited iff p is an ordered list of world variables associated with hypotheses $queued H_1, \ldots queued H_n$, and L is the list $[H_1, \ldots, H_n]$. Given this, the required generalization is

$$lemma2 : \forall p.(rev\ L\ K)@p \to (med\ ACC)@p \to revacc\ L\ ACC\ K \to \text{type}$$

6.1.2 Ordered Cut Admissibility

Ordered logic has, just as linear logic, a sequent calculus, which satisfies a cut-admissibility property:

Proposition 6.1 *If* $\Omega \vdash A$, *and* $\Omega_L, A, \Omega_R \vdash C$, *then* $\Omega_L, \Omega, \Omega_R \vdash C$.

This can be encoded as

$$
\begin{array}{l}
ca : \forall \alpha.\forall \beta_L.\forall \beta_R. \\
\qquad (conc\ A)@\alpha \\
\qquad \to (\forall \beta.(hyp\ A)@\beta \to (conc\ C)@(\beta_L \bullet \beta \bullet \beta_R)) \\
\qquad \to (conc\ C)@(\beta_L \bullet \alpha \bullet \beta_R) \\
\qquad \to \text{type}
\end{array}
$$

Note the use of quantifiers to express the notion of the 'hole' in the middle of the context into which the cut takes place. With ordered logic, this complication means that there are not, as with linear logic, even any suggestive encodings into analogous 'type family with ordered linear arguments' comparable to the attempt in Section 3.1 The use of labels and quantifiers on the other hand addresses with the problem with hardly any difficulty at all.

6.2 Language Simplification

There remain opportunities for further reduction, compilation, and explanation of the proposed type theory in terms of other languages with fewer primitives.

I speculate that it is possible to 'compile away' all use of $\downarrow$, and any use of @ at types that are not base types. Observe for instance that the following pairs of types are equal in terms of the objects that inhabit them:

$$(\downarrow \alpha.A)@p \equiv (\{p/\alpha\}A)@p$$

$$(\Pi x{:}A.B)@p \equiv \Pi x{:}(A@\epsilon).(B@p)$$
$$(\forall \alpha.A)@p \equiv \forall \alpha.(A@p)$$
$$(A@q)@p \equiv A@q$$

These equations let us 'push down' instances of @ toward base types, eliminating $\downarrow$ along the way. This compilation is a sort of elaboration, in that it is not particularly desirable to work in its target language: it is convenient to be able to use high-level connectives like $\multimap$ defined locally in terms of $\downarrow$. However, it may turn out to be simpler to describe the metatheoretic properties in terms of the lower-level language.

Moreover the theory of worlds up to certain equalities and the behavior of $\forall$ seem like they could be explained away by a suitable variant of proof irrelevant LF. However this reduction does not appear to simplify, but rather complicates, direct coverage analysis.

6.3 Kind Language Modifications

In the current proposed type system of HLF, the kind language of LF is extended with a $\forall$. An alternative is to allow something of the form more akin to a genuine Π for worlds,

$$\text{Kinds } K ::= \cdots \mid \Pi\alpha{:}\mathsf{world}.K$$

In this case the use of a type family so Π-indexed by a world would actually mention a world in its list of arguments. If we declared

$$o : \text{type} \qquad k : o \qquad a : \Pi\alpha{:}\mathsf{world}.o@\alpha \to \text{type}$$

then $a\ \epsilon\ k$ would be a type. Contrast this to $\forall$, which does not require any explicit application to a world. If instead we declared of a that

$$a : \forall\alpha.o@\alpha\text{type}$$

then $a\ k$ by itself, with no application to the world ϵ, would be a type.

The $\Pi\alpha{:}\mathsf{world}$ above, which would allow worlds to appear in the argument list of atomic types, could potentially complicate type-checking, but on the other hand it would make it possible to name world-arguments to type families explicitly, so that, for example, they could be described as input or output arguments in mode specifications.

6.4 Implementation

An implementation of the system would be desirable, to ensure the correctness of all the examples. At a bare minimum, to faithfully implement all the theory I have proposed to work out, it would need to typecheck terms in canonical form, and do mode, coverage and termination checking.

Another important feature for practicality of use — one that is already quite standard in using LF — is term and type reconstruction at the front-end, so that the user can omit information such as implicit Π quantification

and corresponding arguments, when they can be inferred. This would involve significant use of unification algorithms, perhaps of a different variant from the use of unification in other parts of the implementation. In particular care is required when most general unifiers (possibly) do not exist, and unification over ACU introduces further opportunities for this difficulty to arise, apart from the fact that it is already possible in higher-order unification outside the pattern fragment.

6.5 The Positive Fragment

HLF, like LLF, only includes *negative* connectives from linear logic, those that have invertible introduction rules. It would be nice to support positive connectives such as $1, \otimes, \oplus, !$, which have invertible *elimination* rules, but it is not apparent how to integrate them with the labelled framework. Watkins et al. solved this problem insofar as adding positive connectives to LLF, yielding the concurrent logical framework CLF. The threat to conservativity of canonical forms that the elimination forms of these new connectives pose was resolved by introducing a *monad* to isolate the concurrent effects.

Particularly worrying for the chances of successfully wedding HLF with positive connectives is the apparent ability [Ree06] to represent a logical connective related to the additive arrow in the logic of bunched implications, as

$$\downarrow\alpha.A@\alpha \rightarrow B@\alpha$$

There are some claims [O'H03] based on category-theoretic reasoning that bunched and linear logic are fundamentally incompatible. In linear logic, & does not distribute over $\oplus$: the sequent $A \mathbin{\&} (B \oplus C) \vdash (A \mathbin{\&} B) \oplus (A \mathbin{\&} C)$ is not derivable. In bunched logic, the additive conjunction (there usually written $\wedge$ instead of &) does distribute over disjunction (in bunched logic usually written $\vee$ instead of $\oplus$). The claim is that this is necessarily so, given the fact that $\wedge$ is a left adjoint (its right adjoint is the additive arrow) and categorically left adjoints must preserve (i.e. distribute over) colimits, an example of which is disjunction.

If in HLF there is a sufficiently strong sense in which the function type above is right adjoint to &, then we would expect some sort of distributivity between & and $\oplus$, in violation of it being equivalent to full linear logic.

6.6 Encoding Modal Logics

Another collection of logical ideas that would be useful to incorporate would obviously be those from modal logic, from which hybrid logic originally arose. It is not unthinkable to add in machinery to track an accessibility relation $\leq$ between world labels, but this would complicate the constraint solving problem considerably.

It seems *prima facie* like it has a minimal impact on the inference rules themselves. In one place in the typing rules, world equality would become world inequality. However, if $\forall$ at the kind level is replaced (or augmented

with the presence) of a true Π over worlds, (which may be motivated anyhow by concerns with the metatheory) it would be necessary compare the argument lists of type families for inequality. This would likely result in imposing a sort of positivity on occurrences of worlds as arguments to type families, which seems unsettlingly arbitrary.

7 Related Work

7.1 Metalogical Approaches

McCreight and Schürmann [MS03] aim to solve the same general problem as I do, but their approach differs in that they devise a meta-logic that explicitly refers to, and quantifies over entire contexts. This leads to some complications with operations on contexts: well-formedness of contexts depends on well-formedness of (dependent!) types within them, and so their definitions are therefore more complicated.

The world labels in HLF effectively *stand for* object language contexts in most examples, but they themselves are not representation contexts, and are therefore are of a much more predicative flavor. What worlds *are*, and when they are well-formed can be explained solely in terms of simple algebraic properties, i.e. the theory of a commutative monoid. The fact that they can then effectively stand for contexts is simply a consequence of the expressiveness of the type system as a representation language.

7.2 Constraint Domains

It is reasonable, especially in light of the fact that we already consider variants of the language of world-labels for ordered logic, to suppose that the appropriate level of generality leaves open the algebraic theory of worlds to a sufficiently language of syntactic sorts, inhabitants of those sorts, and rules determining when two objects are considered equal.

Research on constraint domains and Constraint Handling Rules [Frü98] may afford a general strategy for accounting for equational theories, of which the ACU theory we require for encoding LLF is just a special case. This would involve considerably more open-ended work than I am willing to commit to in this thesis, but it is certainly an interesting avenue of future work, for if the rewrite rules determining equality can be themselves expressed as LF clauses of a declared equality relation, then by staying within a suitably reflective version of LF, one could subsume the expressivity a wide range of logical frameworks affording open-ended constraint domains. Already Roberto Virga has studied constraint domains [Vir99] such as the integers, rationals, and strings for LF, but for our applications we require constraint domains that are open-ended (that new worlds can be hypothesized by ∀) and that the equational theory on hypothetical worlds can be defined equationally.

7.3 Hybrid Logic

Hybrid logic arose out of the desire for more expressive formulation of modal logics. While it has traditionally been concerned with classical logics, there has been some recent forays into constructive hybrid logics, by Chadha, Macedonia and Sassone [CMS06], Murphy, Crary, Harper and Pfenning [VCHP04], and dePaiva and Braüner [BdP03, BdP06].

The implication — or, equivalently, function type — in dePaiva and Braüner's work is particularly interesting, since it differs from the trivialization of the Π I have proposed: its argument and body are both supposed to be at the same world that the function is typed. Its introduction and elimination rules in our notation would be

$$\frac{\Gamma, x : A@p \vdash M : B[p]}{\Gamma \vdash \lambda x.M : A \rightharpoonup B[p]} \qquad \frac{\Gamma \vdash M : A[p] \qquad \Gamma \vdash S : B[p] > C[q]}{\Gamma \vdash (M; S) : A \rightharpoonup B[p] > C[q]}$$

However, much like the reconstruction of $\multimap$, this arrow can be simulated with $\downarrow$ operator. It looks like

$$A \rightharpoonup B = \downarrow\alpha.\Pi x{:}A@\alpha.B$$

Work by Hardt and Smolka [HS06] also encountered a situation where the operators $\downarrow$ and @ can be 'compiled away' in favor of types that simply have indices of some kind. However, their work involves Kripke worlds with accessibility, rather than resources labels with a monoid operation.

7.4 Bunched Semantics

The semantics of BI, the logic of bunched implications [OP99, Pym99, Pym02] have been studied by Pym, O'Hearn, and Yang [POY04], and Galmiche and Méry [GM03].

Our interest is primarily in the algebraic forcing semantics, in which the clauses for bunched additive conjunction $\wedge$ and 'magic wand' $-\!\!*$ are similar in structure to the HLF rules for & and $\multimap$.

The general algebraic setup is that there is a monoid M with operation $*$ (featuring an additional preorder structure that captures the intuitionistic properties of BI), whose elements m act as Kripke worlds in a relation $\vDash$. The pertinent clauses are

$$m \vDash A \wedge B \text{ iff } m \vDash A \text{ and } m \vDash B$$

$$m \vDash A -\!\!* B \text{ iff } (\forall n \in M)(n \vDash A \text{ implies } m * n \vDash B)$$

The similarity between the first and the introduction rule for & is evident. For the second, observe that the $\forall n$ in the semantics is represented by the hypothetical world variable α, and the English-language implication is replaced by the assumption that there is a variable $x : A@\alpha$.

This relationship between the semantics of BI and the syntactic system of HLF supports the idea that we should perhaps be able to prove that a fragment of BI embeds faithfully in HLF, realizing a syntactic interpretation of the semantic machinery of Kripke forcing.

7.5 Substructural Logic Programming

Dale Miller has written a survey [Mil04] of the field of linear logic programming.

Extant linear logic programming languages include Lolli [Hod92, HM91, HM94, Hod94], a typeless first-order linear logic programming language, and Forum [Mil96], an extension of Lolli to a multiple-conclusion sequent, meant to capture some notion of concurrency among the conclusions. The type system of CLF [KWW03a, KWW03b] also supports a logic programming language, LolliMon [LPPW05]. Andreoli's early work with focussing also inspired a language LO he developed with Pareschi [AP90], which, like its logical counterpart in the original focussing system, was classical.

A particularly relevant piece of work to ours is the linear constraint management system [Hod92, CHP00] used in it. This system was developed to efficiently track how knowledge of linear uses of variables is managed during logic programming proof search. We expect it to be fruitful to compare their system to simply eagerly performing constraint-solving on labels — ideally they might turn out to be isomorphic, providing a alternative logical explanation for why their algorithm works.

Polakow, who developed the ordered logical framework OLF [Pol01], also considered ordered logic programming [Pol00]. Examples of ordered logic programs appear above in section 6.1.

8 Conclusion and Plan

I have proposed a novel type system as the foundation of a metalogical framework for mechanical verification of metatheorems about deductive systems in substructural logics. The type system is rooted in established logical ideas from hybrid logic, and can be applied to achieve precise statement of metatheorems whose formalization has already been studied.

The proposed thesis project consists of adapting known and well-studied algorithms to the expanded language. Because of the size of, and experience with, this body of work, I claim that it is feasible to complete the project in a reasonable amount of time.

Specifically, my intended date of completion of the thesis project is May of 2008. Beginning work in December 2006, this gives 18 months. Here is a rough allocation of that time, based on my current understanding of the difficulties of various parts of the project. These are not necessarily consecutive blocks of time; I specifically expect that work on the foundational issues might be spread out over the duration of the project if requirements 'downstream' in particular algorithms require changes to the type theory.

- **Basic type theory**: *2 months*
  - Definition of language of terms, types, kinds.
  - Type system, appropriate definition of hereditary substitution.
  - Proof of substitution property
  - Proof of decidability of typechecking
  - Proof of correctness of embedding of LLF
- **Logic Programming**: *4 months*
  - Operational semantics apart from world-specific constraint-solving: *2 months*. This chiefly involves figuring out how unification works with respect to $\forall$ and @.
  - Operational semantics of constraint-solving: *2 months*. This will involve substantial digging into the literature of ACU unification.
- **Coverage Checking**: *5 months*.
  - Splitting: *4 months* Here I need to figure out more about unification in HLF, and decide what sort of case splitting specifically on world labels can be supported. The counterexamples known from the study of linear unification [CP97a] must be dealt with, although it is quite possible it goes away due to the identification of λ and $\hat{\lambda}$.
  - General Algorithm: *1 month*
- **Termination Checking**: *2 months*.

  Here I need to investigate what termination measures extend to the labelled terms.
- **Implementation**: *3 months*
  - Typechecking: *0 months* This is already effectively done.
  - Metatheorem Checking: *3 months* Extend algorithms from Twelf to match up with the theory developed above.
- **Examples**: *2 months*

  Develop existing examples more, and once the implementation is finished, put them in a form that can be mechanically checked.

References

[ABM01] C. Areces, P. Blackburn, and M. Marx. Hybrid logics: Characterization, interpolation and complexity. *Journal of Symbolic Logic*, 66(3):977–1010, 2001.

[And92] J. M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[AP90] Jean-Marc Andreoli and Remo Pareschi. Lo and behold! concurrent structured processes. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 44–56, New York, NY, USA, 1990. ACM Press.

[BdP03] T. Braüner and V. de Paiva. Towards constructive hybrid logic. *Elec. Proc. of Methods for Modalities*, 3, 2003.

[BdP06] Torben Braüner and Valeria de Paiva. Intuitionistic hybrid logic. To appear., 2006.

[Bla00] P. Blackburn. Representation, reasoning, and relational structures: a hybrid logic manifesto. *Logic Journal of IGPL*, 8(3):339–365, 2000.

[CCP03] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131, Carnegie Mellon University, 2003.

[CHP00] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000. Special issue on Proof Search in Type-Theoretic Languages, D. Galmiche and D. Pym, editors.

[CMS06] R. Chadha, D. Macedonio, and V. Sassone. A Hybrid Intuitionistic Logic: Semantics and Decidability. *Journal of Logic and Computation*, 16(1):27, 2006.

[CP97a] Iliano Cervesato and Frank Pfenning. Linear higher-order pre-unification. In G. Winskel, editor, *Twelfth Annual Symposium on Logic in Computer Science — LICS'97*, pages 422–433, Warsaw, Poland, 29 – 2 1997. IEEE Computer Society Press.

[CP97b] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, Department of Computer Science, Carnegie Mellon University, April 1997.

[CP02] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Inf. Comput.*, 179(1):19–75, 2002.

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.

[Dug98] Dominic Duggan. Unification with extended patterns. *Theoretical Computer Science*, 206(1):1–50, 1998.

[Frü98] T. Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37(1):95–138, 1998.

[Gir87] J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[Gir01] J.Y. Girard. Locus Solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(03):301–506, 2001.

[GM03] Didier Galmiche and Daniel Méry. Semantic Labelled Tableaux for Propositional BI. *Journal of Logic and Computation*, 13(5):707–753, 2003.

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

[HM91] Joshua S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on*, pages 32–42, 1991.

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

[Hod92] J.S. Hodas. Lolli: an extension of λProlog with linear logic context management. In *Proceedings of the 1992 workshop on the λProlog programming language, Philadelphia*, 1992.

[Hod94] J. Hodas. *Logic Programming in Intuitionistic Linear Logic*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[HP01] Robert Harper and Frank Pfenning. On the equivalence and canonical forms in the LF type theory. Technical report, Carnegie Mellon University, 2001.

[HP05] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *Transactions on Computational Logic*, 6:61–101, January 2005.

[HS06] Moritz Hardt and Gert Smolka. Higher-order syntax and saturation algorithms for hybrid logic. In *Proceedings of HyLo 2006*, Electronic Notes in Theoretical Computer Science, 2006. To Appear.

[IP98] Samin S. Ishtiaq and David J. Pym. A relevant analysis of natural deduction. *Journal of Logic and Computation*, 8(6):809–838, 1998.

[Ish99] Samin S. Ishtiaq. *A Relevant Analysis of Natural Deduction*. PhD thesis, Queen Mary and Westfield College, University of London, 1999.

[KWW03a] Frank Pfenning Kevin Watkins, Iliano Cervesato and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, 2003.

[KWW03b] Frank Pfenning Kevin Watkins, Iliano Cervesato and David Walker. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, 2003.

[Lam58] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65(3):154–170, 1958.

[LPPW05] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 35–46, New York, NY, USA, 2005. ACM Press.

[Mil96] D. Miller. A multiple-conclusion meta-logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[Mil04] D. Miller. An overview of linear logic programming, 2004. To appear in a book on linear logic, edited by Thomas Ehrhard, Jean-Yves Girard, Paul Ruet, and Phil Scott. Cambridge University Press.

[MS03] Andrew McCreight and Carsten Schürmann. A meta linear logical framework. Draft manuscript., 2003.

[O'H03] P. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4), July 2003.

[OP99] P.W. O'Hearn and D.J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[PE89] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1989.

[Pfe91] F. Pfenning. Logic programming in the LF logical framework. *Logical Frameworks*, pages 149–181, 1991.

[Pfe94] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CS-94-222, Carnegie Mellon University, 1994.

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

[Pfe00] Frank Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.

[Pol00] Jeff Polakow. Linear logic programming with an ordered context. In *PPDP '00: Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 68–79, New York, NY, USA, 2000. ACM Press.

[Pol01] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University School of Computer Science, 2001.

[POY04] D.J. Pym, P.W. O'Hearn, and H. Yang. Possible worlds and resources: The semantics of BI. *Theoretical Computer Science*, 315(1):257–305, 2004.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Pym99] David J. Pym. On bunched predicate logic. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 183–192, Trento, Italy, 1999. IEEE Computer Society Press.

[Pym02] D.J. Pym. *The Semantics and Proof Theory of the Logic of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[Ree06] Jason Reed. Hybridizing a logical framework. In *Proceedings of the International Workshop on Hybrid Logic 2006*. Elsevier, 2006. To be published.

[Sch00] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

[SP03] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the Theorem Proving in Higher Order Logics 16th International Conference*, volume 2758 of *LNCS*, pages 120–135, Rome, Italy, September 2003. Springer.

[Sti81] Mark E. Stickel. A unification algorithm for associative-commutative functions. *Journal of the ACM*, 28(3):423–434, 1981.

[VCHP04] Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In H. Ganzinger, editor, *Proceedings of the 19th Annual Symposium on Logic in Computer Science (LICS'04)*, pages 286–295, Turku, Finland, July 2004. IEEE Computer Society Press. Extended version available as Technical Report CMU-CS-04-105.

[Vir99] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999.