A Hybrid Logical Framework

Jason Reed

CMU-CS-09-155

September 4, 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning, Chair
Karl Crary
Robert Harper
Rajeev Goré (Australian National University, Canberra)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This work has been supported by the National Science Foundation under grant CCR-0306313, and by the Fundação para a Ciência e Tecnologia (FCT), Portugal, under a grant from the Information and Communications Technology Institute (ICTI) at CMU.

Keywords: Automated Reasoning, Logical Frameworks, Linear Logic, Hybrid Logic

Abstract

The logical framework LF is a constructive type theory of dependent functions that can elegantly encode many other logical systems. Prior work has studied the benefits of extending it to the linear logical framework LLF, for the incorporation linear logic features into the type theory affords good representations of state change. We describe and argue for the usefulness of an extension of LF by features inspired by hybrid logic, which has several benefits. For one, it shows how linear logic features can be decomposed into primitive operations manipulating abstract resource labels. More importantly, it makes it possible to realize a metalogical framework capable of reasoning about stateful deductive systems encoded in the style familiar from prior work with LLF, taking advantage of familiar methodologies used for metatheoretic reasoning in LF.

Acknowledgments

From the very first computer science course I took at CMU, Frank Pfenning has been an exceptional teacher and mentor. For his patience, breadth of knowledge, and mathematical good taste I am extremely thankful. No less do I owe to the other two major contributors to my programming languages education, Bob Harper and Karl Crary.

Thanks to all the other students, without which grad school wouldn't be a tenth as fun. Anything I've accomplished here is a footnote to time spent over lunches or at the whiteboards with Kevin, Brigitte, Aleks, Kaustuv, Tom, Donna, Spoons, Deepak, William, Chris, Neel, D, Noam, and Dan. Thanks to all of you!

This work is dedicated to my parents — most surely of all I wouldn't be where I am today without their support and encouragement.

Contents

Chapter 1

Introduction

Around the end of the 19th century and the beginning of the 20th, the practice of mathematics saw a dramatic increase in rigor and formalization: existing theories were put on firmer foundations, using axiomatic systems that made the prior informal habits seem strikingly inadequate by comparison. Consider Cauchy and his contemporaries replacing intuitive treatments of infinitesimals in calculus with precise notions of limit. Recall the discovery of paradoxes of naïve set theory, and their resolution by means of careful choice of axioms that prescribe how sets can be built up.

Since then, mathematical work has been produced that unquestionably lives up to the highest standard of rigor, which precisely and completely reduces the results claimed to unambiguous rules of inference, symbol by symbol. But efforts such as the monumental *Principia Mathematica* of Whitehead and Russell remain the exception rather than the rule; most published mathematics to this day remains in paragraphs and formulae meant for human consumption, with gaps, sketches, and hints from which the real proof could be reconstructed by a competent — but extremely patient — reader. But a complaint about solely the human patience required to check fully formal proofs is by now unimportant, in the face of the unbounded patience of software at our disposal.

We find ourselves therefore in the middle of another period of increased formalization, and indeed *mechanization* of reasoning. In addition to precisely formulating the axiomatic theories we mean to reason in, we can also formulate and implement machine representations of those theories, and algorithms to automatically carry out portions of our reasoning, and to validate that given arguments are correct. It is also the problem of *designing* increasingly complex software and hardware artefacts that provides one of the greatest needs for formal reasoning.

What is the way forward, then? Is it to hire an army of programmers to produce the fastest, cleverest, slickest possible automatic theorem prover for statements formulated in, say, Zermelo-Fraenkel set theory once and for all? We think not. By way of analogy, every programmer knows that any one of many models of computation — say, Turing machines, or C++ — is equally *sufficient* to describe the computation she means to express, but can still be an undesirable vehicle of expression of computational ideas, by either containing too *few* means of expression, so that writing any program of realistic size is unnecessarily tedious, or too *many* (or perhaps of any quantity so long as they are poorly organized), so

that their interactions are unpredictable and so the meaning and behavior of the program written is uncertain.

Mathematics and logic are no different. We can judge our foundational formal languages, the languages in which we ultimately formulate ideas — about logic, mathematics, programming languages, specifications, and programs — to be better or worse according to how effectively they allow us to say what we mean. We can judge them according to how natural it is to implement proof-checkers and proof search procedures for them. And finally we may judge them by their simplicity and elegance, so that after we say what we mean, we can be more confident *that* we said what we meant. The way forward is to increase the variety of useful formal languages that we can effectively understand and work with (and not without understanding how they are related to one another, lest we end up with simply a random assortment of incompatible tools) so that when a new problem arises, we have a better chance of not merely expressing it *somehow*, but expressing it *well*.

This thesis presents and argues for the benefits of one particular such language. It is not meant as a representation language that is everything to everyone, but rather it is a step forward for a certain class of applications that will be described, and solves a definite problem in treating that class. We claim:

> **Thesis Statement**
> A logical framework with hybrid type and kind constructors provides a good foundation for a metalogical framework capable of encoding and verification of reasoning about stateful deductive systems.

Let us now transition to a narrower view of the problem domain, and at least begin to make preliminary sense of the so far undefined terms in this claim. The comments immediately below merely provide a very high-level view of the ideas contained in the thesis. A more complete account of the background and motivation for the present work, with examples, is given in Chapter 2.

Logical Frameworks

By the phrase 'logical framework' we mean a language for encoding deductive systems and for encoding reasoning about them. A deductive system, in turn, is a formal language, such as logic or programming language, in which the primary objects of attention are *deductions*, also referred to as *derivations*, which are made of, for example, provability of propositions in a logic, facts about typing, program evaluation, and program equivalence. A logical framework in this sense 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*. The family of logical frameworks this work will focus on descends from LF [HHP93], a typed lambda calculus with dependent types, discussed further in Section 2.1.

We use the term logical framework in a sense that is at least intuitively more specific than merely the idea of a formal language as a locus of expression of formal ideas, as discussed up to this point. To whatever extent one can agree as to what counts as purely

logic, (as opposed to more general mathematical means) a logical framework is meant to consist of purely *logical* tools, leaving the user free to implement whichever concrete axioms she wishes. In this sense, we might say that the logical framework in which the axioms of Zermelo-Fraenkel set theory are formulated is simply first-order classical logic. The design space of theories that can be axiomatized in a logical framework is very large, but exploring the space of logical frameworks themselves is more usefully constrained: there is a reasonable consensus on a collection of diagnostic tests — in the form of theorems we expect to hold — to distinguish a logical framework that is well-designed from one that is not.

Metalogical Frameworks

While a logical framework is a setting where deductions exist, and we may speak of basic facts about them — such as a deduction being well-formed, or arising from another derivation in a simple fashion, perhaps by replacing part of it by yet a third derivation — often we wish to describe complex relationships and mappings between derivations, to prove theorems about classes of derivations, and so on. A setting in which this is possible is a *metalogical framework*. It is so named because often a deductive system by itself, especially if it is naturally viewed as a logic, provides a its own notion of truth and provability, and so to prove facts *about* the deductive system as a whole (as opposed to within it) constitutes a viewpoint one level up from merely working in a logical framework. These facts are therefore called metatheorems. Examples of typical metatheorems are type safety of a programming language, consistency of a logic, and bijectivity of a translation between one deductive system and another.

Stateful Deductive Systems

By *stateful* deductive systems we mean to suggest a constellation of representational challenges that are common when treating, for example, the operational semantics of programming languages that have stateful features, e.g. reference cells that are updated imperatively, discarding old values. Most ordinary logical reasoning is *monotonic* in the sense that assumptions, once made, remain valid throughout their scope. But reasoning about state change is naturally *nonmonotonic*: for example, the assumption that a memory cell currently holds a certain value must be forgotten when it is overwritten by a new value. 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 collection of state variables that *don't* change (termed the 'frame') in any particular state transition must still be described, thus complicating the encoding.

The problem that motivated this work was considering an existing *logical* framework designed for representing stateful deductive systems (the linear logical framework LLF [CP02] discussed in 2.3.3) and attempting to generalize it to a *meta*logical framework fully capable of reasoning about stateful deductive systems so represented. A more complete discussion of the this problem is found in Section 2.6. The fundamental issue is that LLF, although capable of representing a significant portion of the reasoning necessary for proofs

of metatheorems about stateful deductive systems of interest, lacks the type-theoretic means to concisely and accurately express their *statement*.

Hybrid Type Constructors

Our solution is to begin with a different logical framework in the first place, and increase the expressive power of LLF by extending it with new type constructors, so that one can state and prove a variety of metatheorems in the extended system. These type constructors have a logical basis, arising from a variant of what is called hybrid logic.

Hybrid logic originally arose from the need to make more expressive extensions of temporal and modal logics without going so far as to pass directly to encoding the Kripke semantics of a modal logic in first-order logic and reasoning there. It is therefore a compromise 'hybrid' between first-order and modal logic. What hybrid logic adds in a modal logic setting is a selection of features to explicitly label and mention modal worlds.

We propose a hybrid logical framework HLF, which likewise adds a notion of explicit label, but which is suited to reasoning about the resource discipline present in LLF instead of modal worlds. This extra feature makes it possible to state and prove metatheorems of interest about stateful deductive systems, making HLF a viable metalogical framework.

1.1 Contributions

Below is the structure of the remainder of the thesis, organized by the technical contributions made.

HLF

Chapter 3 presents the type theory of HLF. It is a descendant of LF containing hybrid type operators that provide enough expressive power to represent stateful deductive systems, and metatheorems about them. The standard set of theorems are proved that establish that the type theory is suitably coherent, principally the substitution and identity theorems.

Embeddings into HLF

It is important to understand the formal relationship between HLF and existing logical frameworks and logical systems. Barring technicalities related to the additive unit $\top$, HLF is a generalization of the linear logical framework LLF. One can also take those technicalities into account, and describe a variant of HLF that embeds LLF exactly, or a variant of LLF that embeds into HLF exactly. This is shown in Section 4.1.

In addition, a fragment of bunched logic [OP99] possesses an embedding into HLF, as shown in Section 4.2, a perhaps surprising fact given the significant differences between bunched logic and linear logic, the latter which is ostensibly at the foundation of HLF. This fact is taken as evidence that for suitably simple fragments, the meaning of linear and bunched logics' resource disciplines coincide, and it is only interactions among the connectives of those logics taken as a whole that gives them their different behavior.

Unification

In Section 5.1 we begin discussion an important algorithm for logical frameworks in general, HLF being no exception. Unification is the problem of solving equations where the unknowns are syntactic expressions. It plays a role in many other useful analyses of terms and signatures in logical frameworks. We discuss a novel higher-order unification algorithm for LF that works by constraint simplification, and how it can be extended to work with the new features introduced in HLF.

Coverage Checking

Logical frameworks can represent complex inductive proofs by their constructive content, as programs in a logic programming style. To check that a proof is correct requires checking that the corresponding program executes correctly no matter what input it is given, and so checking *coverage* of case analyses plays a central role. In Section 5.2 we discuss the extensions to the coverage checking algorithm necessary to make to accommodate HLF.

Applications of HLF

In Chapter 6 we discuss some applications of HLF, which demonstrate on ways that we can reason in HLF about stateful deductive systems. The first example is of a proof of cut admissibility for linear logic, and the second is a proof of type soundness for the simple programming language MiniML.

Conclusion

Finally, Chapter 7 describes possible future work, and summarizes the dissertation.

Chapter 2

Background

2.1 The Logical Framework LF

Throughout this work our concern is with the family of languages descended from the logical framework LF, due to Harper, Honsell and Plotkin [HHP93]. We first provide a rough overview of LF, and go into more technical detail regarding properties of the type theory in discussion of specific extensions of it.

The structure LF itself is quite minimal compared to other popular logical frameworks (e.g., Coq, NuPrl, Isabelle). It is a typed λ-calculus with the only type constructor being the dependent function space $\Pi x{:}A.B$.

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

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

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

where c denotes the use of a declared constant term, a the use of a constant type family, and x the use of a variable. The theory is parametrized over a signature Σ to which the declared constant terms and type families belong.

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

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

The kind $\Pi x_1{:}A_1 \cdots \Pi x_n{:}A_n.\,\mathsf{type}$ 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 base types, 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 is, for it is substituted for the variable x, which may occur 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 simplified point of view that terms (and therefore types that mention terms) are simply considered indistinguishable up to α, β, η convertibility, temporarily ignoring the algorithmic problems this identification entails. In Section 3.1.3 below we will discuss in more detail a modern treatment of this issue, using the technique of *hereditary substitution* to maintain canonical forms of terms.

2.2 LF methodology

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 intended 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 is 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 : type .
$z : nat.$
$s : nat \to nat.$

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.

For example, the propositions of a logic that has a first-order universal and existential quantifier, and equality over individuals has propositions

$$\begin{array}{rl} \text{Propositions} & o ::= o \supset o \mid o \wedge o \mid \forall x : \iota.o \mid \exists x : \iota.o \mid \iota = \iota \\ \text{Individuals} & \iota ::= x \end{array}$$

and is represented in this style by the type declarations

o : type .
ι : type .

and the constant declarations

$and : o \to o \to o.$
$imp : o \to o \to o.$
$all : (\iota \to o) \to o.$
$exists : (\iota \to o) \to o.$
$eq : \iota \to \iota \to o.$

The fact that *app* and *exists* require a function as their argument is what makes the encoding higher-order. To encode a proposition such as $\exists x.x = x$, we specifically pass *exists* an LF function that uses its argument everywhere the variable bound by that existential quantifier appears. Thus $exists\ (\lambda x.eq\ x\ x)$ is the encoding of $\exists x.x = x$.

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.

$ge : nat \to nat \to$ 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

$plus : nat \to nat \to nat \to \mathsf{type}$.
$plus\_z : \Pi N{:}nat.plus\ z\ N\ N.$
$plus\_s : \Pi M{:}nat.\Pi N{:}nat.\Pi P{:}nat.plus\ M\ N\ P$
$\quad \to plus\ (s\ M)\ N\ (s\ P).$

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; that is, if run as a logic program, every clause will compute definite outputs if given definite inputs.
- A relation is said to **cover** its inputs if it has enough clauses so that for any inputs it is given, some clause will match them it will reduce search to another set of goals.
- A relation is **terminating** if this process of 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 set of outputs such that R relates the given inputs to those outputs.

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

2.2.3 Metatheorems

The above style of encoding has the advantage that we have essentially defined a datatype of *executions of the function plus*. Inhabitants of this type, that is, of the type $plus\ M\ N\ P$, are available as a datatype to be manipulated in the same way as, say, natural numbers. With this it is possible to do meta-logical reasoning in LF [Sch00a]: that is, to prove metatheorems *about* a deductive system, because constructive proofs of metatheorems are essentially just functions on these data structures.

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, as supported by the Twelf implementation)

$$\begin{aligned}
&plus_assoc : plus\ N_1\ N_{23}\ M \rightarrow plus\ N_1\ N_2\ N_{12} \\
&\qquad \rightarrow plus\ N_2\ N_3\ N_{23} \rightarrow plus\ N_{12}\ N_3\ M \rightarrow type. \\
&pa/z : plus_assoc\ plus_z\ plus_z\ P\ P. \\
&pa/s : plus_assoc\ P_1\ P_2\ P_3\ P_4 \\
&\qquad \rightarrow plus_assoc\ (plus_s\ P_1)\ (plus_s\ P_2)\ P_3\ (plus_s\ P_4).
\end{aligned}$$

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 sufficient 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.

An elegant solution to these obstacles can be found in the applications of linear logic [Gir87]. 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 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 judgments 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 $\mathit{quarter} \multimap \mathit{gum}$. 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 $\mathit{quarter} \multimap \mathit{gum}\ \&\ \mathit{candy}$ (with one quarter I am able to buy both gum and candy, whichever I want) and $\mathit{quarter} \otimes \mathit{quarter} \multimap \mathit{gum} \otimes \mathit{candy}$ (with two quarters I am able to buy both gum and candy at the same time).

The natural deduction introduction and elimination rules for the connectives described are as follows:

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

$$\frac{\Delta \vdash A \,\&\, B}{\Delta \vdash A} \,\&E1 \qquad \frac{\Delta \vdash A \,\&\, B}{\Delta \vdash B} \,\&E2 \qquad \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \,\&\, B} \,\&I$$

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

The difference between the two conjunctions is visible from their introduction rules. In $\&I$, in order to establish that $A \,\&\, B$ 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. With an added notion of unrestricted hypotheses and a modal operator ! that mediates between the linear and unrestricted judgments, full linear logic is strictly more expressive than ordinary intuitionistic logic. 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 property of ! that is of interest below is that the usual intuitionistic implication $\supset$ has a decomposition in linear logic

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

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

$$\vdash state(s) \otimes input(\ell) \multimap state(s') \otimes input(\varepsilon)$$

(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:

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

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

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

for each $\langle s, \sigma, s' \rangle \in R$. For then in the example FSM we could form a derivation of $state(s_1) \wedge input(a) \supset state(s_1) \wedge input(\varepsilon)$, 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

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

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

$$\dfrac{\dfrac{}{\vdash si(s_2,a) \supset si(s_1,\varepsilon)} \qquad \begin{matrix} \mathcal{D} \\ si(s_1,a) \vdash si(s_2,a) \end{matrix}}{\dfrac{si(s_1,a) \vdash si(s_1,\varepsilon)}{\vdash si(s_1,a) \supset si(s_1,\varepsilon)} \supset I} \multimap E$$

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

In this section we describe LLF [CP02], an extension of the type theory of LF by type operators sensitive to the resource discipline of linear logic as described above. We begin by focusing on how LLF most obviously differs from LF. Where the basic typing judgment of LF is

$$\Gamma \vdash M : A$$

that of LLF is instead

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

which has an additional context Δ of *linear variables*, written $x \hat{:} A$. 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 certain propositional connectives of linear logic. The grammars of terms and types in LF are extended by

$$M ::= \cdots \mid \hat{\lambda} x.M \mid M \hat{} N \mid \langle M, N \rangle \mid \pi_i M \mid \langle\rangle$$

$$A ::= \cdots \mid A \multimap B \mid A \,\&\, B \mid \top$$

The grammar of the language of kinds remains the same; this will become important in Section 2.6. 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 $M \mathbin{\&} N$ 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 canonical inhabitant $\langle\rangle$. The typing rules for these new constructs are as follows:

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

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

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

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, we might wonder whether there is 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 2.6.

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 remainder of this work, to illustrate various concepts as they arise.

2.4.1 Encoding 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 admissibility 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

$$o : \mathsf{type}$$

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

$$hyp : o \to \mathsf{type}$$
$$conc : o \to \mathsf{type}$$

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

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

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

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

2.5 Problem: Mechanized Metatheory for LLF

What is absent in the above methodology is a satisfying analogue to the conventional way metatheorems are encoded LF. We will see that it is still possible sometimes to write down the essential computational content of the proofs themselves in LLF but it is not known how to accurately 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 clauses of a type family viewed as a relation, but in a certain sense it's the *wrong* relation — the kind 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. A central contribution of this work is to give a language of types to solve this problem.

2.6 Cut Admissibility

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 detour in a proof through a lemma A — is admissible:

Theorem 2.6.1 (Intuitionistic Cut Admissibility) *If* $\Gamma \vdash A$ *and* $\Gamma, A \vdash C$ *then* $\Gamma \vdash C$.

The situation before applying cut admissibility is that we can prove the lemma A, and, separately from an assumption of A, we can prove C. Cut admissibility says that whenever there is a proof that takes such a detour, there is also a direct proof.

This is represented in LF as a total relation that relates derivations of *conc* A and *hyp* $A \to$ *conc* C to derivations of *conc* C.

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

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 $x_1 : hyp\ A_1, \ldots, x_n : hyp\ A_n$ that might occur in the two input, and one output derivation. It is worth pointing out that this strategy — representing the object language context with the representation context — is effective because the pertinent context Γ is uniformly shared across the two premises and conclusion

of the theorem, and so we need not say anything particular about it. 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.

For the linear case, however, this pleasant state of affairs as concerns the context is no longer the case. The statement of cut admissibility is

Theorem 2.6.2 (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 — that the context of the conclusion is the combination of the contexts of the two premises of the theorem — is essential for the meaning of the theorem.

However, there is no evident way of writing the theorem as an LLF relation that captures these invariants. To illustrate, we discuss a few attempts, and why they fail. One simple attempt to adapt $(*)$ is to replace the one type-level $\rightarrow$ with a $\multimap$, yielding

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

However, the use of unrestricted function space $\rightarrow$ discards any information that might have been known about the context used to make terms of type $conc\ A$, $hyp\ A \multimap conc\ C$, 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 themselves to somehow 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

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

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 $\otimes$ 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.

One apparent problem with this putative encoding is that LLF lacks the linear connective $\otimes$, but this can be remedied by a standard currying transformation [Pfe94] of the encoding where the type $A_1 \otimes A_2$ is simulated by a declared constant type $t_{A_1,A_2} : \mathsf{type}$ and a constant $create_{A_1,A_2} : A_1 \multimap A_2 \multimap t_{A_1,A_2}$ (or else perhaps by working in CLF [WCPW03a, WCPW03b], which does have $\otimes$).

A deeper problem is that this isn't even a valid declaration of an LLF type family — LLF does not support '$\multimap \mathsf{type}$' with its language of kinds language of kinds, and indeed it is not clear how to add such a constructor, for then the interaction between linearity and dependent types becomes very unclear.

To have a linear function space whose codomain is a kind, such as type, would mean that there would be a notion of type family whose indices were themselves somehow linear. Imagine, supposing we extended the kind language of LLF to include

$$\text{Kinds } K ::= \cdots \mid A \multimap K$$

that we formed the signature containing $o : \mathsf{type}$ and $a : o \multimap \mathsf{type}$, the idea behind a being that it is a type family indexed by o, but which somehow linearly consumes its argument even in the very act of type formation. In this case, it is not obvious what to think about a simple term such as $\hat{\lambda}x.y\,\hat{}\,x$ in a linear context containing $y \mathbin{\hat{:}} o \multimap a\,\hat{}\,x$. One might expect it to have type $o \multimap a\,\hat{}\,x$, since it appears to be the η-expansion of the variable y, yet the attempted typing derivation

$$\dfrac{\dfrac{\overline{x \mathbin{\hat{:}} o \vdash x \mathbin{\hat{:}} o} \quad y \mathbin{\hat{:}} o \multimap a\,\hat{}\,x!? \vdash y \mathbin{\hat{:}} o \multimap a\,\hat{}\,x!?}{x \mathbin{\hat{:}} o, y \mathbin{\hat{:}} o \multimap a\,\hat{}\,x \vdash y\,\hat{}\,x : a\,\hat{}\,x}}{y \mathbin{\hat{:}} o \multimap a\,\hat{}\,x \vdash \hat{\lambda}x.y\,\hat{}\,x : o \multimap a\,\hat{}\,x}$$

leaves y stranded away from the variable x that is required to make sense of y's type, because of the linear context splitting. There is no evident escape from the fact that x is linear, and yet needs to be used in two different ways: if we gave the resource x to allow y to be well-formed, then we wouldn't be able to use it to account for y's argument x in the first place.

Now if we used $\rightarrow \mathsf{type}$ instead of $\multimap \mathsf{type}$ to stay within LLF, we would have the type family

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

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

It is nonetheless possible, as Cervesato and Pfenning [CP02] did, to use a variant of (4) to encode a correct *proof* of the linear cut admissibility theorem, but only by representing object-language linear hypotheses 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:

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

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:

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D}_2 & \mathcal{D}_1\\ \Delta_{21} \vdash A_1 & \Delta_1, A_1 \vdash A_2\end{array}}{\Delta_1, \Delta_{21} \vdash A_2}\, cut \qquad \begin{array}{c}\mathcal{D}_3\\ \Delta_{22}, A_2 \vdash C\end{array}}{\Delta_1, \Delta_{21}, \Delta_{22} \vdash C}\, cut$$

This reasoning is encoded as follows, where *prems* and p are instances of the general currying encoding of $\otimes$ as alluded to above, with *prems* playing the role of t and p playing the role of *create*.

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

The long prefix of Π bindings 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

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

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

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

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

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

into

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

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.

2.7 Towards HLF

Let us recapitulate what appears to be the fundamental problem with using the type theory of LLF as a language for the statement of metatheorems, in particular the linear cut admissibility theorem: that it is on the one hand *necessary* to describe how stateful features of the deductive system being studied (in the example, the linear contexts) interact with one another, but on the other hand it is difficult to do so in LLF because the means that are available for describing properties, predicates, relations, interactions on data — dependent types — do not interact suitably well with the means that are available for describing stateful things — linear types.

The solution I propose is a deeper reconciliation between dependent and linear types, by way of decomposing a linear hypothesis into its two roles as a resource, and as an index to a type family. It extends the language of LLF to enable convenient *explicit* description and manipulation of the information implicit in the fact that Δ_1 and Δ_2 are distinct collections of linear hypotheses in the statement

$$\text{If } \Delta_1 \vdash A \text{ and } \Delta_2, A \vdash C, \text{ then } \Delta_1, \Delta_2 \vdash C$$

of linear cut admissibility, which are then combined in its conclusion. It does this by making patterns of resource use first-class objects in the type theory.

Specifically, I propose an extension of LF (which will turn out to be effectively an extension of LLF as well) 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 for the fragment of linear logic used in LLF. One can introduce a notion of resource labels subject not to a Kripke semantics bearing an accessibility relation, but rather to an algebraic structure, not unlike the phase semantics for linear logic [Gir95] or the resource semantics for BI given by Galmiche and Méry [GM03]. Because they nonetheless behave somewhat like Kripke worlds inasmuch as it is convenient to speak of propositions being located at them, I use 'world' and 'label' more or less interchangeably in the sequel.

A brief digression on the relationship to other semantics is merited here. One may note that even in such semantics one frequently also finds a relation in addition to the algebraic structure, whether an accessibility relation or in the case of relevance logics, a ternary relation that captures at once a combination of modal accessibility and algebraic properties posed relationally.

However, no such relations are required in the present work. The purpose of borrowing here the term "hybrid" is to emphasize the presence of labels for *some* semantic entity that can be quantified over, referred to, and otherwise manipulated in the calculus itself and not to specifically suggest that the exact apparatus attached to them is the same as is found in other hybrid calculi.

The question may reasonably be asked *why* merely the algebraic structure suffices in our case, and we suggest that the answer (as discussed slightly further below) is in large part that the representation language we propose is a *constructive* logic: for when a classical semantics is given for an intuitionistic substructural logic, some notion of Kripke modal treatment is naturally present merely to account for the gap between intuitionistic and classical, and further an algebraic structure for the substructural properties of it. Since

our representation language is itself constructive, we are able to focus on the substructural characteristics in isolation.

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 sections I motivate the design and key ideas of this new system by beginning with of basic considerations about tracking resource consumption.

2.7.1 Resource Counting

We now wish to explain how to make patterns of resource use into sensible first-class objects in a type theory. Note that 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. Suppose we want to more generally *count* how many uses a given variable sees.

Consider specifically 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

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

If we wanted certain variables in the system to be used linearly, this could perhaps 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 introduction rule for $\multimap$, in particular, since it introduces a new linear variable, could enforce this restriction as follows:

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

Are these U the appropriate notion of 'pattern of resource use' that we want? On the face of it, a U does not seem to be any easier to talk about or manipulate than a linear context itself, since it contains an arbitrary collection of variables. If we were to have variables standing for usage specifications U, they would be variables ranging over entire contexts, which introduces a sort of impredicativity which is not necessarily desirable. Any evident notion of well-formedness of usage specification Us 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.

But there is a more abstract representation of the same information found in usage specifications U, which naturally has the same level of generality in terms of what resource uses it can describe. Let there be a separate syntactic class of *worlds*

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

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 $p * \varepsilon$, $\varepsilon * p$, and p are considered equal.

Worlds can be used to encode information about how often variables are used in a 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 a 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, what we would have written immediately above as $x_1 \mapsto k_1, \ldots, x_n \mapsto k_n$.

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:

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

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

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

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

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

In order to explicate the previously unspecified 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 $[p]$, and has the effect of locating a hypothesis at a world p:

$$A ::= \cdots \mid A \mathbin{@} p$$

One can give the following natural-deduction introduction and elimination rules for @.

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

To construct a object that is marked as from world p, we go to that world and construct it; to use something marked as from world p, we go to that world to use it, in both cases regardless of which world q we started at. Note that neither of these rules introduce any new term constructs: the type $A \mathbin{@} p$ is merely a *refinement* of the type A.

From the point of view of labelling linear assumptions in this way we can give introduction and elimination rules for $\multimap$ as well.

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

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 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.

It is likewise due to the world annotations that it is still acceptable for the hypothesis rule above to not require, as it would in linear logic, that the context is a singleton. In HLF, the context contains representations of all linear hypotheses that have ever been hypothesized along the current branch of the proof-tree, but the current world controls which can actually be used.

2.7.2 Hybrid Type Operators

The reason for teasing out the resource use information into a separate syntactic object is to be able to manipulate it with other logical connectives, and corresponding type operators. Let us first consider the most common connectives found in the literature on hybrid logic besides @, the universal quantifier $\forall \alpha.A$ over worlds, and the local binding operator $\downarrow \alpha.A$, which binds a variable α to the *current* world at the time it is analyzed. The proposition $\downarrow \alpha.A$ can be thought of as meaning 'if the current world is α, then proposition A (which mentions the variable α) holds'. Both $\downarrow$ and $\forall$ are merely refinement types in the same sense as @: they do not introduce any term constructors in either their introduction or elimination rules.

The typing rules for $\downarrow \alpha.A$ allows access to the current world of the judgment by substituting it for the bound variable α.

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

In both the introduction and elimination rules, the current world is captured, and substituted for the bound variable.

The typing rules for $\forall$ are thoroughly standard for a universal quantifier.

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

When introducing a universal, one must carry out the proof under the assumption of a fresh parameter, and when eliminating it, one must supply an arbitrary well-formed instantiation of the variable.

Moreover, it is reasonable to add universal quantification to the language of kinds:

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

Note that 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. Therefore this extension to the language of kinds involves far less complication than would be required by adding $\multimap$ in the same place. The ability to quantify over worlds in kinds will be central to the techniques described below for encoding statements of metatheorems in HLF.

Before getting to that, however, it is useful to note that the extra connectives are already useful for providing (together with ordinary function types) a decomposition of $\multimap$. We can consider $A \multimap B$ as a *defined* connective, rather than a primitive of the language, treating it as a macro for

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

(the scope of $\forall$ and $\downarrow$ are generally both meant to extend as far to the right as possible)

The correctness of this definition of $\multimap$ is taken up formally in Section 4.1, but for an intuition of why why this definition of $\multimap$ works, notice that for every derivation

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

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

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

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

2.7.3 Π vs. $\forall$

In the above suggestive description, worlds are quantified by a $\forall$ that acts as a refinement, in that a proof term that has type $\forall \alpha.A$ is the same proof term as one that has type A — it is simply required to be well-formed for every α.

In fact for many reasons it will become useful instead to take the more explicit representation of universal quantification as a dependent function type, in this case with an argument that is itself a world. That is, instead of $\forall\alpha.A$, we may consider $\Pi\alpha{:}\mathsf{w}.A$, whose introduction and elimination rules would be written

$$\frac{\Gamma, \alpha : \mathsf{w} \vdash M : A[p]}{\Gamma \vdash \lambda\alpha.M : (\Pi\alpha{:}\mathsf{w}.A)[p]} \qquad \frac{\Gamma \vdash M : (\Pi\alpha{:}\mathsf{w}.A)[p] \qquad \Gamma \vdash q : \mathsf{w}}{\Gamma \vdash M\ q : (\{q/\alpha\}A)[p]}$$

the difference being that now to make something of a Π-type one must λ-*abstract* over a world variable, and to use such an expression, one must *apply* it to a world expression. Likewise, to the kind-level $\forall$ there corresponds an analogous Π.

The advantages of using Π instead of $\forall$ are found in simplifications of the design and implementation of algorithms for type reconstruction, unification, and coverage checking, where worlds behave more like ordinary LF expressions (except for their equational theory). The primary disadvantage is that the relationship between HLF and LLF is less straightforward, as a consequence of how $\top$ allows discarding of linear resources, and how the use of Π tracks *where* they are discarded, while $\forall$ does not. This is discussed further in Section 4.1. Our approach will be to initially include both Π and $\forall$ in the language, and cut down to the subsystem that only includes one or the other as appropriate.

2.7.4 Linear Cut Admissibility in HLF

We can now sketch the solution to the running example of representing linear cut elimination. The type family that captures the relation we want is

$$\begin{aligned} ca : \Pi\alpha_1{:}\mathsf{w}.\Pi\alpha_2{:}\mathsf{w}.&\\ &\mathit{conc}\ A\ @\ \alpha_1\\ &\to (\mathit{hyp}\ A \multimap \mathit{conc}\ C)\ @\ \alpha_2\\ &\to (\mathit{conc}\ C)\ @\ (\alpha_1 * \alpha_2)\\ &\to \mathsf{type} \end{aligned}$$

Observe that we can now directly and succinctly express the fact that one derivation is meant to use one collection of resources, the second derivation another set of resources, and the third derivation uses their combination. The world variables α_1, α_2 represent precisely the contexts Δ_1, Δ_2 in the informal statement of the theorem, and yet we are also able to use $\multimap$ in standard higher-order abstract syntax style to conveniently express the addition of one new linear hypothesis to the context. We return to this example to describe the complete proof in Section 6.1.

2.8 Related Work

2.8.1 Substructural Dependent Types

Work by Ishtiaq, Pym, et al. on RLF [IP98, Ish99] has investigated the possibility of a function space that is at once dependent and substructural, where type families are allowed to depend on arguments in a 'linear' way, as was claimed above to be difficult at best. However, their system is actually modelled on relevant (also sometimes called 'strict') logic, which permits assumptions to be used more than once, as long as they are used at least once. This therefore addresses the fact that linear hypotheses appearing as type family arguments seem to need to be used multiple times if they are used at all, once in the term, and once in the type. To the best of our knowledge no such programme has been carried out successfully with index objects that are actually *linear* in the sense of Girard's linear logic.

2.8.2 Linear Metalogical Frameworks

An unpublished paper by McCreight and Schürmann [MS03] begins with the same general aim as the present work, to make a metalogical framework capable of reasoning about linear encodings. Their approach differs in that they devise a metalogic that explicitly refers to, and quantifies over entire contexts, as opposed to HLF, which does the same to elements of the algebra of abstract identifiers that can be seen to play the role of contexts. This leads to significant complications with operations on contexts: well-formedness of contexts depends on well-formedness of (dependent!) types within them.

The world labels in HLF effectively stand for object-language contexts in some examples, but they themselves are not representation-language contexts, and are therefore are of a much more predicative nature. What worlds *are*, and when they are well-formed can be explained solely in terms of simple algebraic properties — in this case, 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. In addition they more generally can be used as representations of other object language features that have a commutative monoid structure, and lack the extra trappings of linear logic contexts.

2.8.3 Substructural Encodings in LF

An approach to encoding linearity in pure LF was proposed by Crary [Cra09]. In his technique, one does not address the linear context as such, but instead ensures that each variable in isolation is used exactly once. When a linear variable is bound, an explicit witness is required to establish that that variable appears once in the scope of the binding, and this witness is constructed out of constants that define the predicate of 'occurs exactly once' by using ordinary LF encoding techniques. Crary develops this idea with an

example of an adequate encoding of linear natural deduction terms, and a proof of weak normalization of well-typed terms.

This has the clear benefit of not requiring the addition of any new machinery to LF. However, as Crary notes, the idea does not evidently extend to other substructural logics such as ordered logic [Pol01], in which the restriction the substructural discipline imposes has to do with the *interaction* of hypotheses with one another. Also because the context is not reified it seems unclear how to represent in his technique the metatheorems we describe below whose statement is concerned with the behavior of entire contexts of linear assumptions.

Fundamentally we consider as an important benefit of our approach that one need not separately specify the predicate of linear occurrence of variables on a constant-by-constant basis: the entirety of the contract as to how substructural variables are to be used is contained in the types of the substructural constants themselves. One might make the argument that greater flexibility is afforded by being *able* to separately define how variables are allowed to occur within the logical framework itself (and indeed Crary also captures modal logic by similar techniques) but since the present system is an extension of LF, nothing prevents the user from also using his method, perhaps even in combination with the encoding techniques allowed by the hybrid features of HLF.

Chapter 3

A Hybrid Logical Framework

3.1 Definition of HLF

In this section we provide a complete formal definition of the type theory HLF. We begin with the syntax.

3.1.1 Core Syntax

As in LF, there is a notion of signature, in which are declared constants and constant type families.

$$\text{Signatures } \Sigma ::= \cdot \mid \Sigma, c : A \mid \Sigma, a : K$$

There are contexts, which as usual contain bindings of ordinary variables x, and also feature world variables α.

$$\text{Contexts } \Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, \alpha : \mathsf{w}$$

The symbol w is just a marker that the variable preceding it is a world variable, and not a metavariable standing for anything other than itself.

The syntax of expressions is as follows.

$$\begin{array}{rrl}
\text{Worlds} & p, q, r & ::= \alpha \mid p * p \mid \varepsilon \\
\text{Kinds} & K & ::= \Pi x{:}A.K \mid \Pi\alpha{:}\mathsf{w}.K \mid \forall\alpha.K \mid \mathsf{type} \\
\text{Types} & A, B & ::= \Pi x{:}A.B \mid \Pi\alpha{:}\mathsf{w}.B \mid \forall\alpha.A \mid \downarrow\alpha.B \mid A \mathbin{@} p \mid A \mathbin{\&} B \mid \top \mid b \\
\text{Base Types} & b & ::= a \cdot S \\
\text{Normal Terms} & M, N & ::= \lambda x.M \mid \lambda\alpha.M \mid R \mid \langle M_1, M_2 \rangle \mid \langle\rangle \\
\text{Atomic Terms} & R & ::= H \cdot S \\
\text{Heads} & H & ::= x \mid c \\
\text{Spines} & S & ::= () \mid (M; S) \mid (p; S) \mid (\pi_1; S) \mid (\pi_2; S) \\
\hline
\text{Syntactic Object} & X & ::= M \mid R \mid S \mid A \mid K \mid \Gamma \mid \Sigma \mid p
\end{array}$$

World expressions consist of world variables α and the distinguished empty world ε, possibly combined with the operation $*$. Kinds, in addition to the usual dependent function kinds and base kind 'type', allow for dependent quantification $\Pi\alpha{:}\mathsf{w}.K$ over worlds. Likewise there is a dependent type constructor over worlds $\Pi\alpha{:}\mathsf{w}.B$. Additions to the language of types relative to LF also include the pair type $A \mathbin{\&} B$ and its unit $\top$, as well as hybrid type operators $\forall\alpha.A$, $\downarrow\alpha.A$ and $A \mathbin{@} p$.

The syntax of terms is arranged so that every term is automatically in normal form: β-redices are ruled out by the grammar. A normal term is either a lambda abstraction, a pair, a unit expression, or else an atomic expression R. An atomic expression consists of a head (a variable x or constant c) applied to a *spine*, [CP97] a data structure capturing a sequence of uses of elimination rules, consisting of generally a mixture of function arguments and projections. In the syntax of spines, we often omit the final $()$ and treat $;$ associatively as spine concatenation and write for instance the spine $(M_1; (M_2; (M_3; ())))$ without introducing any ambiguity as $(M_1; M_2; M_3)$.

A reader not familiar with syntax in spine form can think of

$$x \cdot (M_1; M_2; M_3)$$

as an alternate way of writing the iterated application

$$((x\ M_1)\ M_2)\ M_3$$

Spines have the advantage of exposing the head variable (or constant) of the series of applications, which makes the definition of substitution (which cares decisively about whether the head variable is the same as or different from the variable being substituted for) more pleasant. The catch-all class X of arbitrary syntactic expressions is also useful below for describing substitution in a uniform way.

3.1.2 Core Typing Judgments

The central typing judgments of the theory are as follows, grouped suggestively according to the level (context, kind, type, term) they describe.

Signature formation	$\vdash \Sigma : \mathsf{sgn}$
Context formation	$\vdash_\Sigma \Gamma : \mathsf{ctx}$
Kind formation	$\Gamma \vdash_\Sigma K : \mathsf{kind}$
Type formation	$\Gamma \vdash_\Sigma A : \mathsf{type}$
Spine kinding	$\Gamma \vdash_\Sigma S : K > \mathsf{type}$
World formation	$\Gamma \vdash_\Sigma p : \mathsf{w}$
Type checking	$\Gamma \vdash_\Sigma M \Leftarrow A[p]$
Type synthesis	$\Gamma \vdash_\Sigma R \Rightarrow A[p]$
Spine typing	$\Gamma \vdash_\Sigma S : A[p] > C[r]$

Nearly every judgment is parametrized by a signature Σ, but in the sequel we assume frequently that we have fixed a particular Σ, and avoid writing it explicitly.

The spine typing judgment $\Gamma \vdash S : A[p] > C[r]$ is read as saying that for any head of type A at world p, if it is applied to spine S, the result has type C at world r. The distinction between normal and atomic terms is compatible with a *bidirectional* typing discipline as is to be expected: the type system is able to determine whether a normal term has a type A at world p if A and p are provided, but an atomic term *synthesizes* its type and world as outputs. The spine typing judgment takes Γ, S, A, p as input and yields C, r as outputs.

3.1.3 Auxiliary Definitions

A few definitions besides the ones already outlined are required.

Simple Types

Because the theory features dependent types, typing of terms depends on the result of carrying out substitutions of terms for variables found in types. Early definitions of LF simply performed substitution in a direct way, leaving β-redices in its wake, requiring a subsequent notion of definitional equality of expressions. More recent work on logical frameworks has benefited from the idea, introduced in work on CLF, of defining a *hereditary* substitution operation. Hereditary substitution carries out all reductions arising from the initial substitution, and resolves all redices arising from those reductions, and so on, thus the 'hereditary' in the name. It then always yields a term in canonical (β-normal and η-long) form.

To ensure termination of the hereditary substitution operations, they are indexed by the *simple type* (effectively introduced in [HHP87], sometimes called the *approximate type* [Ell89]) of the object being substituted or β-reduced, because it serves effectively as the high-priority part of a lexicographic termination order: every substitution recurses on substitution into a smaller term at the same simple type, or else at a substitution of a smaller simple type. As far as this work is concerned, simple types are those that arise from erasing *all* dependency information from ordinary types, as well as (this choice being made differently in some other definitions of simple types in the literature) the choice of which type family the base type belongs to. All that is remains is the shape of the type in terms of function arrows, products, and so on.

It will also help the staging of the metatheory below to have judgments that hold when a term *is* simply-typed, an approximation to the full judgment that the term is well-typed.

Simple types and contexts of simply-typed hypotheses are given by the grammar

$$\begin{aligned} \text{Simple Type } \tau &::= \bullet \mid \tau \to \tau \mid \mathsf{w} \to \tau \mid \tau \,\&\, \tau \mid \top \mid \star\tau \\ \text{Simple Context } \gamma &::= \cdot \mid \gamma, x : \tau \mid \gamma, \alpha : \mathsf{w} \end{aligned}$$

and the operations and judgments pertaining to them are

$$\begin{array}{rl}
\text{Type Simplification} & A^- = \tau \\
\text{Kind Simplification} & K^- = \tau \\
\text{Context Simplification} & \Gamma^- = \gamma \\
\hline
\text{Simple type formation} & \gamma \vdash A : \mathsf{type} \\
\text{Simple kind formation} & \gamma \vdash K : \mathsf{type} \\
\text{Simple kind formation} & \gamma \vdash \Gamma : \mathsf{ctx} \\
\hline
\text{Simple type checking} & \gamma \vdash M \Leftarrow \tau \\
\text{Simple type synthesis} & \gamma \vdash R \Rightarrow \tau \\
\text{Simple spine typing} & \gamma \vdash S : \tau > \tau'
\end{array}$$

The phrase 'simple type formation' is not to be taken to mean the rules for formation *of* simple types, but rather a check on the formation of full dependent types that nonetheless only cares about the *simple* typing of arguments to base types.

The simple type corresponding to the type A (the 'simplification of A') is written A^- (respectively the kind K yields the simple type K^-, the context Γ, the simple context Γ^-) and is defined as follows.

$$\begin{array}{rcl}
(\Pi x{:}A.B)^- & = & A^- \to B^- \\
(\Pi x{:}A.K)^- & = & A^- \to K^- \\
(\Pi \alpha{:}\mathsf{w}.B)^- & = & \mathsf{w} \to B^- \\
(\Pi \alpha{:}\mathsf{w}.K)^- & = & \mathsf{w} \to K^- \\
(\forall \alpha.K)^- & = & K^- \\
(b)^- & = & \bullet \\
(\mathsf{type})^- & = & \bullet \\
(\forall \alpha.B)^- & = & \star B^- \\
(\downarrow \alpha.B)^- & = & \star B^- \\
(A \mathbin{@} p)^- & = & \star A^- \\
(A \mathbin{\&} B)^- & = & A^- \mathbin{\&} B^- \\
\top^- & = & \top \\
\cdot^- & = & \cdot \\
(\Gamma, x : A)^- & = & \Gamma^-, x : A^- \\
(\Gamma, \alpha : \mathsf{w})^- & = & \Gamma^-, \alpha : \mathsf{w}
\end{array}$$

The symbol $\star$ is used uniformly for the simplification of the HLF refinement operations $\forall, \downarrow, @$ that do not affect proof terms. We might have dropped $\star$ from the language and had said $(\forall \alpha.B)^- = B^-$ and $(\downarrow \alpha.B)^- = B^-$ and $(A \mathbin{@} p)^- = A^-$ except that its presence simplifies the induction metric in several proofs below.

All of the simple typing rules are given below in section 3.1.7.

World Equality

At the boundary between type checking and type synthesis, there are two types, one coming from the output of type synthesis, and one coming from the input of type checking. We

must compare the two types there for equality, and for the same reason compare the two *worlds* involved for the appropriate notion of equality. Foreshadowing the encoding of linear logic, in which the collection of linear hypotheses is a multiset, the free algebra for a commutative monoid, we take equality on worlds to be the equivalence (symmetric, reflexive, transitive) relation $\equiv_{\mathsf{acu}}$ axiomatized by the laws of a commutative monoid, that is, associative, commutative, and unit laws for $*$ and ε.

$$\frac{}{p \equiv_{\mathsf{acu}} p} \qquad \frac{p \equiv_{\mathsf{acu}} q}{q \equiv_{\mathsf{acu}} p} \qquad \frac{p \equiv_{\mathsf{acu}} q \qquad q \equiv_{\mathsf{acu}} r}{p \equiv_{\mathsf{acu}} r} \qquad \frac{p \equiv_{\mathsf{acu}} p' \qquad q \equiv_{\mathsf{acu}} q'}{p * q \equiv_{\mathsf{acu}} p' * q'}$$

$$\frac{}{(p * q) * r \equiv_{\mathsf{acu}} p * (q * r)} \qquad \frac{}{p * q \equiv_{\mathsf{acu}} q * p} \qquad \frac{}{p * \varepsilon \equiv_{\mathsf{acu}} p}$$

Also, since other expressions types may include world expressions, we take the basic equality judgment $X_1 = X_2$ on general expressions to mean that X_1 and X_2 have exactly the same structure *up to* $\equiv_{\mathsf{acu}}$ on worlds appearing in them in parallel locations, in addition to being agnostic up to α-equivalence as is customarily assumed for comparing canonical forms of expressions with binding.

Generally, when other inference rules repeat a variable, the requirement is that one instantiates that variable the *same* way throughout the rule as usual, but we mean 'the same' up to this notion of $=$. For example,

$$\vdash () : o \mathbin{@} (\varepsilon * \varepsilon) > o \mathbin{@} \varepsilon$$

is a legitimate instance of the empty spine typing rule below, having in mind a signature containing o : type.

3.1.4 Hereditary Substitution

Armed with the notion of simple type above, we can give the operations that constitute the hereditary substitution algorithm as

Term Substitution	$\{M/x\}^{\tau} X = X$
World Substitution	$\{p/\alpha\}^{\mathsf{w}} X = X$
Reduction	$[M \mid S]^{\tau} = R$

The substitution operations $\{p/\alpha\}^{\mathsf{w}}$ (substitute the world p for the world variable α) and $\{M/x\}^{\tau}$ (substitute the term M for the variable x at simple type τ) are partial functions taking terms to terms, types to types, and so on. In particular $\{M/x\}^{\tau}$ signifies the fully β-normalized result of capture-avoiding substitution of M for x. The reduction operator $[M \mid S]^{\tau}$ is a partial function that yields an atomic term. It computes the result of hereditarily substituting all of the arguments S to the variables bound by the function M.

The partiality of these operations can be seen to arise from the fact that a function could be given more or fewer arguments than it expects, and the definition of reduction simply fails to have any clauses except those that match up arguments for λ-bindings

exactly. We will see in Section 3.2.3 that it is useful to make a separate definition of a notion of reduction that errs in just one direction, of allowing too few arguments to be given to a function, but not too many.

For the definition of substitution, let σ abbreviate either $\{p/\alpha\}^{\mathsf{w}}$ or $\{M/x\}^{\tau}$. We assume the usual conventions about tacit α-renaming to ensure distinct variable names when descending under binders.

Substitution on Kinds

$$\sigma(\Pi x{:}A.K) = \Pi x{:}(\sigma A).(\sigma K)$$
$$\sigma(\Pi \alpha{:}\mathsf{w}.K) = \Pi \alpha{:}\mathsf{w}.(\sigma K)$$
$$\sigma(\forall \alpha.K) = \forall \alpha.(\sigma K)$$
$$\sigma\,\mathsf{type} = \mathsf{type}$$

Substitution on Types

$$\sigma(\Pi x{:}A.B) = \Pi x{:}(\sigma A).(\sigma B)$$
$$\sigma(a \cdot S) = a \cdot (\sigma S)$$
$$\sigma(\Pi \alpha{:}\mathsf{w}.B) = \Pi \alpha{:}\mathsf{w}.(\sigma B)$$
$$\sigma(\forall \alpha.B) = \forall \alpha.(\sigma B)$$
$$\sigma(\downarrow \alpha.B) = \downarrow \alpha.(\sigma B)$$
$$\sigma(A \mathbin{@} p) = (\sigma A) \mathbin{@} (\sigma p)$$
$$\sigma(A \mathbin{\&} B) = \sigma A \mathbin{\&} \sigma B$$
$$\sigma \top = \top$$

Substitution on Terms

$$\sigma(\lambda x.M) = \lambda x.(\sigma M)$$
$$\sigma\langle M_1, M_2\rangle = \langle \sigma M_1, \sigma M_2\rangle$$
$$\sigma\langle\rangle = \langle\rangle$$
$$\sigma\varepsilon = \varepsilon$$
$$\sigma(p * q) = \sigma p * \sigma q$$
$$\sigma(c \cdot S) = c \cdot (\sigma S)$$
$$\{N/x\}^{\tau}(x \cdot S) = [N \mid \{N/x\}^{\tau} S]^{\tau}$$
$$\sigma(x \cdot S) = x \cdot (\sigma S) \qquad (\text{if } \sigma \text{ not a subst. for } x)$$
$$\{p/\alpha\}^{\mathsf{w}}\alpha = p$$
$$\sigma\alpha = \alpha \qquad (\text{if } \sigma \text{ not a subst. for } \alpha)$$

$$\sigma() = ()$$

$$\sigma(M; S) = (\sigma M; \sigma S)$$

$$\sigma(p; S) = (\sigma p; \sigma S)$$

$$\sigma(\pi_i; S) = (\pi_i; \sigma S)$$

Reduction

Substitution calls reduction when it would potentially create a β-redex by directly replacing a variable with a normal term, and the job of reduction is to eliminate that β-redex, and hereditarily all redices that that reduction creates. As reduction decomposes its argument, its simple type gets smaller — this guarantees termination.

$$\begin{array}{rcl}
[\lambda x.M \mid (N; S)]^{\tau_1 \to \tau_2} & = & [\{N/x\}^{\tau_1} M \mid S]^{\tau_2} \\
[\lambda \alpha.M \mid (p; S)]^{\mathsf{w} \to \tau} & = & [\{p/\alpha\}^{\mathsf{w}} M \mid S]^{\tau} \\
[\langle M_1, M_2 \rangle \mid (\pi_i; S)]^{\tau_1 \& \tau_2} & = & [M_i \mid S]^{\tau_i} \\
[M \mid S]^{\star \tau} & = & [M \mid S]^{\tau} \\
[R \mid ()]^{\bullet} & = & R
\end{array}$$

3.1.5 η-Expansion

Another centrally important operation, also indexed by simple types, η-expansion is what allows one to turn variables into terms in canonical form. In general write the η-expansion of a atomic term R at the simple type τ as $\mathsf{ex}_\tau(R)$, defined as follows:

$$\begin{array}{rcl}
\mathsf{ex}_\bullet(R) & = & R \\
\mathsf{ex}_{\mathsf{w} \to \tau}(H \cdot S) & = & \lambda \alpha.\mathsf{ex}_\tau(H \cdot (S; \alpha)) \\
\mathsf{ex}_{\tau_1 \to \tau_2}(H \cdot S) & = & \lambda x.\mathsf{ex}_{\tau_2}(H \cdot (S; \mathsf{ex}_{\tau_1}(x))) \\
\mathsf{ex}_{\tau_1 \& \tau_2}(H \cdot S) & = & \langle \mathsf{ex}_{\tau_1}(H \cdot (S; \pi_1)), \mathsf{ex}_{\tau_2}(H \cdot (S; \pi_2)) \rangle \\
\mathsf{ex}_{\top}(H \cdot S) & = & \langle\rangle \\
\mathsf{ex}_{\star \tau}(R) & = & \mathsf{ex}_\tau(R)
\end{array}$$

To concisely express the η-expansion of a variable or constant, we sometimes write $\mathsf{ex}_\tau(H)$ as an abbreviation for $\mathsf{ex}_\tau(H \cdot ())$.

3.1.6 Typing Rules

In this section we give the rules that inductively define the typing judgments of HLF.

Signature Formation

A signature is well-formed if every constant declared in it is assigned a well-formed type, and every constant type family is assigned a well-formed kind.

$$\frac{}{\vdash \cdot : \mathsf{sgn}} \qquad \frac{\vdash \Sigma : \mathsf{sgn} \qquad \vdash_\Sigma A : \mathsf{type}}{\vdash (\Sigma, x : A) : \mathsf{sgn}} \qquad \frac{\vdash \Sigma : \mathsf{sgn} \qquad \vdash_\Sigma K : \mathsf{kind}}{\vdash (\Sigma, a : K) : \mathsf{sgn}}$$

Context Formation

For a context to be well-formed, every type in it must be well-formed in the prefix of the context preceding it. As usual, it is implicit in the way we write contexts and combinations of them that all variables in a context are distinct.

$$\frac{}{\vdash \cdot : \mathsf{ctx}} \qquad \frac{\vdash \Gamma : \mathsf{ctx} \qquad \Gamma \vdash A : \mathsf{type}}{\vdash (\Gamma, x : A) : \mathsf{ctx}} \qquad \frac{\vdash \Gamma : \mathsf{ctx}}{\vdash (\Gamma, \alpha : \mathsf{w}) : \mathsf{ctx}}$$

Kind Formation

A kind is well-formed if every variable bound by a Π is assigned a well-formed type in the appropriate context.

$$\frac{\Gamma \vdash A : \mathsf{type} \qquad \Gamma, x : A \vdash K : \mathsf{kind}}{\Gamma \vdash \Pi x{:}A.K : \mathsf{kind}} \qquad \frac{\Gamma, \alpha : \mathsf{w} \vdash K : \mathsf{kind}}{\Gamma \vdash \Pi \alpha{:}\mathsf{w}.K : \mathsf{kind}} \qquad \frac{}{\Gamma \vdash \mathsf{type} : \mathsf{kind}}$$

Type Formation

$$\frac{\Gamma \vdash A : \mathsf{type} \qquad \Gamma, x : A \vdash B : \mathsf{type}}{\Gamma \vdash \Pi x{:}A.B : \mathsf{type}} \qquad \frac{\Gamma, \alpha : \mathsf{w} \vdash B : \mathsf{type}}{\Gamma \vdash \Pi \alpha{:}\mathsf{w}.B : \mathsf{type}}$$

$$\frac{\Gamma, \alpha : \mathsf{w} \vdash B : \mathsf{type}}{\Gamma \vdash \forall \alpha.B : \mathsf{type}}$$

$$\frac{a : K \in \Sigma \qquad \Gamma \vdash S : K > \mathsf{type}}{\Gamma \vdash a \cdot S : \mathsf{type}}$$

$$\frac{\Gamma, \alpha : \mathsf{w} \vdash B : \mathsf{type}}{\Gamma \vdash \downarrow \alpha.B : \mathsf{type}} \qquad \frac{\Gamma \vdash A : \mathsf{type} \qquad \Gamma \vdash p : \mathsf{w}}{\Gamma \vdash A \mathbin{@} p : \mathsf{type}}$$

$$\frac{\Gamma \vdash A : \mathsf{type} \qquad \Gamma \vdash B : \mathsf{type}}{\Gamma \vdash A \mathbin{\&} B : \mathsf{type}} \qquad \frac{}{\Gamma \vdash \top : \mathsf{type}}$$

Type Spine Kinding

$$\frac{}{\Gamma \vdash () : \mathsf{type} > \mathsf{type}}$$

$$\frac{\Gamma \vdash M \Leftarrow A[\varepsilon] \qquad \Gamma \vdash S : \{M/x\}^{A^-} K > \mathsf{type}}{\Gamma \vdash (M; S) : \Pi x{:}A.K > \mathsf{type}}$$

$$\frac{\Gamma \vdash p : \mathsf{w} \qquad \Gamma \vdash S : \{p/\alpha\}^{\mathsf{w}} K > \mathsf{type}}{\Gamma \vdash (p; S) : \Pi \alpha{:}\mathsf{w}.K > \mathsf{type}}$$

$$\frac{\Gamma \vdash p : \mathsf{w} \qquad \Gamma \vdash S : \{p/\alpha\}^{\mathsf{w}} K > \mathsf{type}}{\Gamma \vdash S : \forall \alpha.K > \mathsf{type}}$$

World Formation

$$\frac{\alpha : \mathsf{w} \in \Gamma}{\Gamma \vdash \alpha : \mathsf{w}} \qquad \frac{\Gamma \vdash p : \mathsf{w} \qquad \Gamma \vdash q : \mathsf{w}}{\Gamma \vdash p * q : \mathsf{w}} \qquad \frac{}{\Gamma \vdash \varepsilon : \mathsf{w}}$$

Type Checking

$$\frac{\Gamma \vdash R \Rightarrow b[p] \qquad b = b' \qquad p \equiv_{\mathsf{acu}} p'}{\Gamma \vdash R \Leftarrow b'[p']}$$

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

$$\frac{\Gamma, \alpha : \mathsf{w} \vdash M \Leftarrow B[p]}{\Gamma \vdash M \Leftarrow \forall \alpha.B[p]}$$

$$\frac{\Gamma \vdash M \Leftarrow (\{p/\alpha\}^{\mathsf{w}} B)[p]}{\Gamma \vdash M \Leftarrow \downarrow \alpha.B[p]}$$

$$\frac{\Gamma \vdash M \Leftarrow A[q]}{\Gamma \vdash M \Leftarrow A \mathbin{@} q[p]}$$

$$\frac{\Gamma \vdash M_1 \Leftarrow A_1[p] \qquad \Gamma \vdash M_2 \Leftarrow A_2[p]}{\Gamma \vdash \langle M_1, M_2 \rangle \Leftarrow A_1 \mathbin{\&} A_2[p]}$$

$$\frac{}{\Gamma \vdash \langle\rangle \Leftarrow \top[p]}$$

Type Synthesis

Write $\Gamma \vdash H : A$ to mean that either H is a variable $x : A$ in Γ, or else a constant $c : A$ in the signature.

$$\frac{\Gamma \vdash H : A \qquad \Gamma \vdash S : A[\varepsilon] > C[r]}{\Gamma \vdash H \cdot S \Rightarrow C[r]}$$

Term Spine Typing

$$\frac{}{\Gamma \vdash () : A[p] > A[p]}$$

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

$$\frac{\Gamma \vdash p : \mathsf{w} \qquad \Gamma \vdash S : (\{p/\alpha\}^{\mathsf{w}} B)[q] > C[r]}{\Gamma \vdash (p; S) : (\Pi \alpha{:}\mathsf{w}.B)[q] > C[r]}$$

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

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

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

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

3.1.7 Simple Typing

The rules for simple typing are essentially the same as the ordinary typing rules, but with all dependency information erased. For completeness, we nonetheless list them in full.

Simple Type Formation

$$\frac{\gamma \vdash A : \mathsf{type} \qquad \gamma, x : A^- \vdash B : \mathsf{type}}{\gamma \vdash \Pi x{:}A.B : \mathsf{type}} \qquad \frac{\gamma, \alpha : \mathsf{w} \vdash B : \mathsf{type}}{\gamma \vdash \Pi \alpha{:}\mathsf{w}.B : \mathsf{type}}$$

$$\frac{\gamma, \alpha : \mathsf{w} \vdash B : \mathsf{type}}{\gamma \vdash \forall \alpha.B : \mathsf{type}}$$

$$\frac{a : K \in \Sigma \qquad \gamma \vdash S : K^- > \bullet}{\gamma \vdash a \cdot S : \mathsf{type}} \qquad \frac{}{\gamma \vdash \mathsf{type} : \mathsf{type}}$$

$$\frac{\gamma, \alpha : \mathsf{w} \vdash B : \mathsf{type}}{\gamma \vdash \downarrow \alpha.B : \mathsf{type}} \qquad \frac{\gamma \vdash A : \mathsf{type} \qquad \gamma \vdash p : \mathsf{w}}{\gamma \vdash A \mathbin{@} p : \mathsf{type}}$$

$$\frac{\gamma \vdash A : \mathsf{type} \qquad \gamma \vdash B : \mathsf{type}}{\gamma \vdash A \mathbin{\&} B : \mathsf{type}} \qquad \frac{}{\gamma \vdash \top : \mathsf{type}}$$

Simple Type Checking

$$\frac{\gamma \vdash R \Rightarrow \bullet}{\gamma \vdash R \Leftarrow \bullet} \qquad \frac{\gamma \vdash M \Leftarrow \tau}{\gamma \vdash M \Leftarrow \star\tau}$$

$$\frac{\gamma, x : \tau_1 \vdash M \Leftarrow \tau_2}{\gamma \vdash \lambda x.M \Leftarrow \tau_1 \to \tau_2} \qquad \frac{\gamma, \alpha : \mathsf{w} \vdash M \Leftarrow \tau}{\gamma \vdash \lambda\alpha.M \Leftarrow \mathsf{w} \to \tau}$$

$$\frac{\gamma \vdash M_1 \Leftarrow \tau_1 \qquad \gamma \vdash M_2 \Leftarrow \tau_2}{\gamma \vdash \langle M_1, M_2 \rangle \Leftarrow \tau_1 \,\&\, \tau_2}$$

$$\frac{}{\gamma \vdash \langle\rangle \Leftarrow \top}$$

Simple Type Synthesis

Similar to the convention mentioned in type synthesis above, write $\gamma \vdash H : \tau$ to mean that either H is a variable $x : \tau$ in γ, or else a constant $c : A$ in the signature such that $A^- = \tau$.

$$\frac{\gamma \vdash H : \tau \qquad \gamma \vdash S : \tau > \tau'}{\gamma \vdash H \cdot S \Rightarrow \tau'}$$

Simple Spine Typing

$$\frac{}{\gamma \vdash () : \tau > \tau}$$

$$\frac{\gamma \vdash S : \tau > \tau'}{\gamma \vdash S : \star\tau > \tau'}$$

$$\frac{\gamma \vdash M \Leftarrow \tau_1 \qquad \gamma \vdash S : \tau_2 > \tau'}{\gamma \vdash (M; S) : \tau_1 \to \tau_2 > \tau'}$$

$$\frac{\gamma \vdash p : \mathsf{w} \qquad \gamma \vdash S : \tau > \tau'}{\gamma \vdash (p; S) : \mathsf{w} \to \tau > \tau'}$$

$$\frac{\gamma \vdash S : \tau_i > \tau'}{\gamma \vdash (\pi_i; S) : \tau_1 \,\&\, \tau_2 > \tau'}$$

3.2 Fundamental Properties of HLF

In this section we prove the standard results of an LF-family logical framework that demonstrate the system is suitably well-behaved. The two major results are the *substitution* and *identity* properties. These are dual, and their relationship aligns with many other central dualities of logics and type theories:

Substitution	Variable Use
β-reduction	η-expansion
Soundness	Completeness
Cut Elimination	Identity
Local Reduction	Local Expansion

Through the lens of the canonical forms of a logical framework, the essential fact about the substitution property is that a use of a variable can be *eliminated* provided a term of that variable's type. This fact is no longer trivial when attention is restricted to β-normal terms, because straightforward replacement of a variable with a term can create redices. Conversely the meaning of the identity theorem says that a variable can be used to *create* a term of the same type. This fact is not trivial because of the requirement of η-longness: a variable only becomes a well-formed normal term when it is suitably η-expanded.

3.2.1 Properties of Simple Typing

The plan for proving these results is to first establish them for the simple type discipline described above, and subsequently to prove them for the full theory. This staging allows us to pay attention only to simply well-typed terms in the second stage, which simplifies reasoning about well-definedness of substitutions, for substitution is total on simply well-typed arguments.

This fact is captured by the following lemma.

Lemma 3.2.1 (Simply Typed Substitution) *Suppose $\gamma \vdash N \Leftarrow \tau'$. Let σ be an abbreviation for some substitution $\{N/z\}^{\tau'}$ or $\{p/\alpha\}^{\mathsf{w}}$. In all of the following entailments, the conclusion implicitly includes the assertion that the substitution or reduction mentioned in it is well-defined.*

1. *If $\gamma \vdash S : \tau' > \bullet$, then $\gamma \vdash [N \mid S]^{\tau'} \Rightarrow \bullet$.*
2. *If $\gamma, z : \tau', \gamma' \vdash M \Leftarrow \tau$, then $\gamma, \gamma' \vdash \sigma M \Leftarrow \tau$.*
3. *If $\gamma, z : \tau', \gamma' \vdash R \Rightarrow \tau$, then $\gamma, \gamma' \vdash \sigma R \Rightarrow \tau$.*
4. *If $\gamma, z : \tau', \gamma' \vdash S : \tau_1 > \tau_2$, then $\gamma, \gamma' \vdash \sigma S : \tau_1 > \tau_2$.*
5. *If $\gamma, z : \tau', \gamma' \vdash A : \mathsf{type}$, then $\gamma, \gamma' \vdash \sigma A : \mathsf{type}$.*
6. *If $\gamma, z : \tau', \gamma' \vdash K : \mathsf{type}$, then $\gamma, \gamma' \vdash \sigma K : \mathsf{type}$.*

Proof By straightforward lexicographic induction first on τ' and subsequently on the typing derivation. The cases of this theorem are similar to (and where they differ, since

there are no substitutions carried out on simple types, simpler than) the cases of the main substitution result below. ■

Toward a proof of the simply typed identity property, we first observe that the typing of spines behaves roughly like function types.

Lemma 3.2.2 (Simply Typed Spine Composition) *If* $\Gamma \vdash S_1 : \tau > \tau'$ *and* $\Gamma \vdash S_2 : \tau' > \tau''$*, then* $\Gamma \vdash (S_1; S_2) : \tau > \tau''$*, where* ; *here is used to denote the evident concatenation of the two spines.*

Proof By straightforward induction on the typing derivation of S_1, observing that no rule but the rule that types the nil spine does anything but pass the output type along. ■

With this we are able to prove

Lemma 3.2.3 (Simply Typed Identity) *If* $\gamma \vdash R \Rightarrow \tau$*, then* $\gamma \vdash \mathsf{ex}_\tau(R) \Leftarrow \tau$

Proof By induction on τ.

Case: $\tau = \tau_1 \to \tau_2$. In this case, $R = H \cdot S$ with $\gamma \vdash H : \tau'$ and $\gamma \vdash S : \tau' > \tau_1 \to \tau_2$. By two applications of the induction hypothesis, we can construct a derivation as follows:

$$\dfrac{\dfrac{\gamma \vdash H : \tau' \qquad \dfrac{\gamma \vdash S : \tau' > \tau_1 \to \tau_2 \qquad \dfrac{\dfrac{}{y : \tau_1 \vdash y \cdot () \Rightarrow \tau_1}}{y : \tau_1 \vdash \mathsf{ex}_{\tau_1}(y) \Leftarrow \tau_1}\, i.h. \qquad \dfrac{}{\vdash () : \tau_2 > \tau_2}}{y : \tau_1 \vdash (\mathsf{ex}_{\tau_1}(y)) : \tau_1 \to \tau_2 > \tau_2}}{\gamma, y : \tau_1 \vdash (S; \mathsf{ex}_{\tau_1}(y)) : \tau' > \tau_2}\,\text{Lemma 3.2.2}}{\dfrac{\gamma, y : \tau_1 \vdash H \cdot (S; \mathsf{ex}_{\tau_1}(y)) \Rightarrow \tau_2}{\gamma, y : \tau_1 \vdash \mathsf{ex}_{\tau_2}(H \cdot (S; \mathsf{ex}_{\tau_1}(y))) \Leftarrow \tau_2}\, i.h.}}{\gamma \vdash \lambda y.\mathsf{ex}_{\tau_2}(H \cdot (S; \mathsf{ex}_{\tau_1}(y))) \Leftarrow \tau_1 \to \tau_2}$$

Case: $\tau = \mathsf{w} \to \tau_0$. A simplification of the previous case.

Case: $\tau = \star\tau_0$. Straightforward.

Case: $\tau = \top$. Straightforward.

Case: $\tau = \tau_1 \,\&\, \tau_2$. In this case, $R = H \cdot S$ with $H : \tau'$ and $\gamma \vdash S : \tau' > \tau_1 \,\&\, \tau_2$. By two applications of the induction hypothesis, we can construct a derivation as follows:

$$\dfrac{\dfrac{H : \tau' \qquad \dfrac{\gamma \vdash S : \tau' > \tau_1 \,\&\, \tau_2 \qquad \dfrac{\dfrac{}{\gamma \vdash () : \tau_1 > \tau_1}}{\gamma \vdash (\pi_1) : \tau_1 \,\&\, \tau_2 > \tau_1}}{\gamma \vdash (S; \pi_1) : \tau' > \tau_1}\,\text{Lemma 3.2.2}}{\dfrac{\gamma \vdash H \cdot (S; \pi_1) \Rightarrow \tau_1}{\gamma \vdash \mathsf{ex}_{\tau_1}(H \cdot (S; \pi_1)) \Leftarrow \tau_1}\, i.h.} \qquad \dfrac{\dfrac{}{\gamma \vdash () : \tau_1 > \tau_1}}{\gamma \vdash (\pi_1) : \tau_1 \,\&\, \tau_2 > \tau_1}}{\gamma \vdash \langle \mathsf{ex}_{\tau_1}(H \cdot (S; \pi_1)), \mathsf{ex}_{\tau_2}(H \cdot (S; \pi_2)) \rangle \Leftarrow \tau_1 \,\&\, \tau_2}\,(sym.)$$

Case: $\tau = \bullet$. Immediate.

■

In the sequel we will tacitly assume that all variables intrinsically carry their simple type, and that all objects, substitutions, and reductions are simply well-typed. To say that a substitution $\{M/x\}^\tau$ is simply well-typed is to say that $\gamma \vdash M \Leftarrow \tau$, and that a reduction $[M \mid S]^\tau$ is simply well-typed means that $\gamma \vdash M \Leftarrow \tau$ and $\gamma \vdash S : \tau > \bullet$.

3.2.2 Substitution

First we will establish some lemmas toward proving the substitution property. A basic fact about all typing judgments is that they can be weakened to a larger context and still hold.

Let J stand the right side (following the turnstile) of any judgment.

Lemma 3.2.4 (Weakening) *If $\Gamma, \Gamma' \vdash J$, then $\Gamma, x : A, \Gamma' \vdash J$. By the usual variable conventions, x is required to not already be in Γ or Γ'.*

Proof By induction on the typing derivation. ■

Let $FV(X)$ be the set of free variables of X. Then a substitution for a variable that does not occur has no effect.

Lemma 3.2.5 (Vacuous Substitutions)

- *If $x \notin FV(X)$, then $\{M/x\}^\tau X = X$.*
- *If $\alpha \notin FV(X)$, then $\{p/\alpha\}^\tau X = X$.*

Proof By induction on X. ■

Next we show how consecutive substitutions can commute with one another. This is essential to the general substitution theorem, because typing dependent function application itself requires performing a substitution, and we must show that this substitution is compatible with a substitution applied to the entire expression.

Lemma 3.2.6 (Substitutions Commute) *Let σ be a substitution of the form $\{N/y\}^{\tau'}$ or $\{q/\beta\}^{\mathsf{w}}$. Suppose $x \neq y$ and $\alpha \neq \beta$, and furthermore $x \neq FV(N)$ and $\alpha \notin FV(q)$. Then*

1. $\sigma(\{p/\alpha\}^{\mathsf{w}} X) = \{\sigma p/\alpha\}^{\mathsf{w}} \sigma X$
2. $\sigma(\{M/x\}^\tau X) = \{\sigma M/x\}^\tau \sigma X$
3. $\sigma[M \mid S]^\tau = [\sigma M \mid \sigma S]^\tau$

Proof By lexicographic induction first on the size of the types involved (see following), and subsequently on the structure of X, taking advantage of the assumption that everything in sight is simply well-typed.

In case 1, the pertinent simple type is τ'. In cases 2 and 3, the metric is the formal commutative sum $\tau + \tau'$, for which we say $\tau_1 + \tau_1' < \tau_2 + \tau_2'$, if either $\tau_1 < \tau_2$ and $\tau_1' \leq \tau_2'$, or else $\tau_1 < \tau_2'$ and $\tau_2 \leq \tau_1'$. The reason this induction metric is required can be seen if one follows around the types through a substitution that leads to a reduction that leads to another substitution: the types involved get smaller, except for the fact that τ and τ' switch places.

For equal τ, τ', case 3 is considered smaller than case 2.

We give a selection of representative cases.

Case: Part 2, $X = (M_0; S)$. Then

$\sigma(\{M/x\}^\tau(M_0; S)) = (\sigma(\{M/x\}^\tau M_0); \sigma(\{M/x\}^\tau S))$
$= ((\{\sigma M/x\}^\tau \sigma M_0); (\{\sigma M/x\}^\tau \sigma S))$ i.h. $(M_0, S < (M_0; S))$
$= \{\sigma M/x\}^\tau \sigma(M_0; S)$

Case: Part 2, $X = x \cdot S$. Then

$\sigma(\{M/x\}^\tau(x \cdot S)) = \sigma([M \mid \{M/x\}^\tau S]^\tau)$
$= [\sigma M \mid \sigma\{M/x\}^\tau S]^\tau$ i.h. (Case 3 < Case 2)
$= [\sigma M \mid \{\sigma M/x\}^\tau \sigma S]^\tau$ i.h. $(S < x \cdot S)$
$= \{\sigma M/x\}^\tau(x \cdot \sigma S)$
$= \{\sigma M/x\}^\tau \sigma(x \cdot S)$ $x \neq y$

Case: Part 2, $X = y \cdot S$. Then

$\sigma(\{M/x\}^\tau(y \cdot S)) = \sigma(y \cdot \{M/x\}^\tau S)$ $x \neq y$
$= [N \mid \sigma\{M/x\}^\tau S]^{\tau'}$
$= [N \mid \{\sigma M/x\}^\tau \sigma S]^{\tau'}$ i.h. $(S < x \cdot S)$
$= [\{\sigma M/x\}^\tau N \mid \{\sigma M/x\}^\tau \sigma S]^{\tau'}$ $x \notin FV(N)$
$= \{\sigma M/x\}^\tau [N \mid \sigma S]^{\tau'}$ i.h. (Case 3 < Case 2)
$= \{\sigma M/x\}^\tau \sigma(y \cdot S)$

Case: Part 3, $\tau = \tau_1 \to \tau_2$. By assumptions about simple typing, M is of the form $\lambda x.M_0$, and S is of the form $(M'; S')$.

$\sigma[M \mid S]^\tau = \sigma[\lambda x.M_0 \mid (M'; S')]^{\tau_1 \to \tau_2}$
$= \sigma[\{M'/x\}^{\tau_1} M_0 \mid S']^{\tau_2}$
$= [\sigma\{M'/x\}^{\tau_1} M_0 \mid \sigma S']^{\tau_2}$ i.h. $(\tau_2 + \tau' < \tau + \tau')$
$= [\{\sigma M'/x\}^{\tau_1} \sigma M_0 \mid \sigma S']^{\tau_2}$ i.h. $(\tau_1 + \tau' < \tau + \tau')$
$= [\lambda x.\sigma M_0 \mid (\sigma M'; \sigma S')]^{\tau_1 \to \tau_2}$
$= [\sigma(\lambda x.M_0) \mid \sigma(M'; S')]^{\tau_1 \to \tau_2}$
$= [\sigma M \mid \sigma S]^\tau$

■

We can now prove the substitution property for HLF.

Theorem 3.2.7 (Substitution Property)

1. *Suppose* $\Gamma \vdash p \Leftarrow \mathsf{w}$. *Let* σ *be an abbreviation for the substitution* $\{q/\beta\}^{\mathsf{w}}$, *with* β *being a variable that does not occur free in* Γ.
   (a) *If* $\Gamma, \beta : \mathsf{w}, \Gamma' \vdash M \Leftarrow A[p]$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma M \Leftarrow \sigma A[\sigma p]$.
   (b) *If* $\Gamma, \beta : \mathsf{w}, \Gamma' \vdash R \Rightarrow A[p]$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma R \Rightarrow \sigma A[\sigma p]$.
   (c) *If* $\Gamma, \beta : \mathsf{w}, \Gamma' \vdash S : A[p] > C[q]$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma S : \sigma A[\sigma p] > \sigma C[\sigma q]$.
   (d) *If* $\Gamma, \beta : \mathsf{w}, \Gamma' \vdash S : K > \mathsf{type}$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma S : \sigma K > \mathsf{type}$.
   (e) *If* $\Gamma, \beta : \mathsf{w}, \Gamma' \vdash A : \mathsf{type}$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma A : \mathsf{type}$.
   (f) *If* $\Gamma, \beta : \mathsf{w}, \Gamma' \vdash K : \mathsf{kind}$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma K : \mathsf{kind}$.
   (g) *If* $\Gamma, \beta : \mathsf{w}, \Gamma' \vdash p : \mathsf{w}$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma p : \mathsf{w}$.

(h) *If* $\vdash \Gamma, \beta : \mathsf{w}, \Gamma' : \mathsf{ctx}$, *then* $\vdash \Gamma, \sigma\Gamma' : \mathsf{ctx}$.

2. *Suppose* $\Gamma \vdash N \Leftarrow B[\varepsilon]$. *Let* σ *be an abbreviation for the substitution* $\{N/z\}^{B^-}$, *with* z *being a variable that does not occur free in* Γ *or* B.
   (a) *If* $\Gamma, z : B, \Gamma' \vdash M \Leftarrow A[p]$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma M \Leftarrow \sigma A[p]$.
   (b) *If* $\Gamma, z : B, \Gamma' \vdash R \Rightarrow A[p]$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma R \Rightarrow \sigma A[p]$.
   (c) *If* $\Gamma, z : B, \Gamma' \vdash S : A[p] > C[q]$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma S : \sigma A[p] > \sigma C[q]$.
   (d) *If* $\Gamma, z : B, \Gamma' \vdash S : K > \mathsf{type}$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma S : \sigma K > \mathsf{type}$.
   (e) *If* $\Gamma, z : B, \Gamma' \vdash A : \mathsf{type}$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma A : \mathsf{type}$.
   (f) *If* $\Gamma, z : B, \Gamma' \vdash K : \mathsf{kind}$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma K : \mathsf{kind}$.
   (g) *If* $\Gamma, z : B, \Gamma' \vdash p : \mathsf{w}$, *then* $\Gamma, \sigma\Gamma' \vdash \sigma p : \mathsf{w}$.
   (h) *If* $\vdash \Gamma, z : B, \Gamma' : \mathsf{ctx}$, *then* $\vdash \Gamma, \sigma\Gamma' : \mathsf{ctx}$.
   (i) *If* $\Gamma \vdash S : B[q] > A[p]$ *and* $\Gamma \vdash M \Leftarrow B[q]$ *then* $\Gamma \vdash [M|S]^{B^-} \Rightarrow A[p]$.

Proof By lexicographic induction on first the simple type B^-, next on the case (where case (i) is ordered less than all the remaining cases), and finally (for cases $(a) - (h)$) on the structure of the typing derivation. We show some representative cases.

Case: Part 2i, with M of the form $\lambda x.M_0$. Then the typing derivation of M must be of the form

$$\frac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma, x : B_1 \vdash M_0 \Leftarrow B_2[q]\end{array}}{\Gamma \vdash \lambda x.M_0 \Leftarrow \Pi x{:}B_1.B_2[q]}$$

Since we know that B is $\Pi x{:}B_1.B_2$, the typing derivation of S must look like

$$\frac{\begin{array}{cc}\mathcal{D}_2 & \mathcal{D}_3 \\ \Gamma \vdash M_1 \Leftarrow B_1[\varepsilon] & \Gamma \vdash S_1 : \{M_1/x\}^{B_1^-} B_2[q] > A[p]\end{array}}{\Gamma \vdash (M_1; S_1) : \Pi x{:}B_1.B_2[q] > A[p]}$$

with S being $(M_1; S_1)$. By the induction hypothesis (part 2a) on the smaller simple type B_1^- and the derivations $\mathcal{D}_1$ and $\mathcal{D}_2$, we find that

$$\Gamma \vdash \{M_1/x\}^{B_1^-} M_0 \Leftarrow \{M_1/x\}^{B_1^-} B_2[q] \qquad (*)$$

By induction hypothesis (part 2i) on the smaller simple type B_2^-, the fact $(*)$, and the derivation $\mathcal{D}_3$, deduce

$$\Gamma \vdash [\{M_1/x\}^{B_1^-} M_0 | S_1]^{B_2^-} \Rightarrow A[p]$$

But by definition of reduction we can read off that

$$[M|S]^{B^-} = [(\lambda x.M_0)|(M_1; S_1)]^{B_1^- \to B_2^-} = [\{M_1/x\}^{B_1^-} M_0 | S_1]^{B_2^-}$$

so we are done.

Case: Part 2i, with M atomic, of the form R. By inversion we have a typing derivation

$$\Gamma \vdash R \Leftarrow b[q] \tag{$*$}$$

That is, B is b. The only typing rule that would conclude $S : b > A$ is

$$\frac{}{\Gamma \vdash () : b > b}$$

so S must be empty, and A is also b. Therefore

$$[M|S]^\bullet = [R|()]^\bullet = R$$

but we already have a derivation that $\Gamma \vdash R \Leftarrow b[q]$, namely $(*)$.

Case: Part 2a, with

$$\mathcal{D} = \frac{\mathcal{D}'}{\Gamma, z : B, \Gamma' \vdash \lambda x.M_0 \Leftarrow \Pi x{:}A_0.B_0[p]} \qquad \Gamma, z : B, \Gamma', x : A_0 \vdash M_0 \Leftarrow B_0[p]$$

By the induction hypothesis on $\mathcal{D}'$ we know $\Gamma, \sigma\Gamma', x : \sigma A_0 \vdash \sigma M_0 \Leftarrow \sigma B_0[p]$. By rule application we obtain $\Gamma, \sigma\Gamma' \vdash \lambda x.\sigma M_0 \Leftarrow \Pi x{:}\sigma A_0.\sigma B_0[p]$ as required.

Case: Part 2a, with

$$\mathcal{D} = \frac{\mathcal{D}'}{\Gamma, z : B, \Gamma' \vdash R \Leftarrow b[p]} \qquad \Gamma, z : B, \Gamma' \vdash R \Rightarrow b'[p'] \qquad b = b'$$

By the induction hypothesis on $\mathcal{D}'$ we obtain $\Gamma, \sigma\Gamma' \vdash \sigma R \Rightarrow \sigma b'[p']$. By rule application we obtain $\Gamma, \sigma\Gamma' \vdash \sigma R \Leftarrow \sigma b[p]$ as required.

Case: Part 1a, with

$$\mathcal{D} = \frac{\mathcal{D}'}{\Gamma, z : B, \Gamma' \vdash R \Leftarrow b[p]} \qquad \Gamma, z : B, \Gamma' \vdash R \Rightarrow b'[p'] \qquad b = b' \qquad p \equiv_{\mathsf{acu}} p'$$

By the induction hypothesis on $\mathcal{D}'$ we obtain $\Gamma, \sigma\Gamma' \vdash \sigma R \Rightarrow \sigma b'[\sigma p']$. The relation $\equiv_{\mathsf{acu}}$ is evidently stable under substitution, so we also have $\sigma p \equiv_{\mathsf{acu}} \sigma' p'$. By rule application we obtain $\Gamma, \sigma\Gamma' \vdash \sigma R \Leftarrow \sigma b[\sigma p]$ as required.

Case: Part 2b, with

$$\mathcal{D} = \frac{\mathcal{D}'}{\Gamma, z : B, \Gamma' \vdash x \cdot S \Rightarrow A[p]} \qquad x : A_0 \in (\Gamma, z : B, \Gamma') \qquad \Gamma, z : B, \Gamma' \vdash S : A_0[\varepsilon] > A[p]$$

We split on three subcases depending on the location of $x \in \Gamma, z : B, \Gamma'$.

Subcase: $x \in \Gamma$. In this case z is not in scope in the type A_0 of x. Thus $\sigma A_0 = A_0$ by Lemma 3.2.5. By the induction hypothesis (part 2c) on $\mathcal{D}'$ we obtain $\Gamma, \sigma\Gamma' \vdash \sigma S : \sigma A_0 > \sigma A[p]$. By rule application we get $\Gamma, \sigma\Gamma' \vdash x \cdot \sigma S \Rightarrow \sigma A[p]$ as required.

Subcase: x is in fact z. In this case $A_0 = B$, and the term $\sigma(x \cdot S)$ we aim to type is $[N|\sigma S]^{B^-}$. By assumption $\Gamma \vdash N \Leftarrow B[\varepsilon]$, and by using Lemma 3.2.4 repeatedly we can obtain $\Gamma, \sigma\Gamma' \vdash N \Leftarrow B[\varepsilon]$. By Lemma 3.2.5 (because necessarily $x = z \notin FV(B)$) and the induction hypothesis (part 2c), we know $\Gamma, \sigma\Gamma' \vdash \sigma S : B[\varepsilon] > \sigma A[p]$. Use the induction hypothesis (part 2i: this is licensed because it is ordered as less than the other cases, and the simple type B^- has remained the same) to obtain the required derivation of $\Gamma, \sigma\Gamma' \vdash [N|\sigma S]^{B^-} \Rightarrow \sigma A[p]$.

Subcase: $x \in \Gamma'$. By the induction hypothesis (part 2c) $\Gamma, \sigma\Gamma' \vdash \sigma S : \sigma A_0[\varepsilon] > \sigma A[p]$. By definition of substitution on contexts $x : \sigma A_0 \in \Gamma, \sigma\Gamma'$ so it follows by rule application that $\Gamma, \sigma\Gamma' \vdash x \cdot \sigma S \Rightarrow \sigma A[p]$.

Case: Part 2c, with

$$\mathcal{D} = \frac{}{\Gamma, z : B, \Gamma' \vdash () : b[p] > b[p]}$$

Immediate.

Case: Part 2c, with

$$\mathcal{D} = \frac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Gamma, z : B, \Gamma' \vdash M \Leftarrow A_0[\varepsilon] \quad \Gamma, z : B, \Gamma' \vdash S : \{M/x\}^{A_0^-} A[p] > C[q]}{\Gamma, z : B, \Gamma' \vdash (M; S) : \Pi x{:}A_0.A[p] > C[q]}$$

Because of the substitution $\{M/x\}^{A_0^-}$, this case depends critically on Lemma 3.2.6, the property of commuting substitutions.

By the induction hypothesis (part 2a) we know $\Gamma, \sigma\Gamma' \vdash \sigma M \Leftarrow \sigma A_0[\varepsilon]$. Observe that N (remember σ is $\{N/z\}^{B^-}$) has no free occurrence of z, so by Lemma 3.2.6 infer that $\sigma\{M/x\}^{A_0^-} A = \{\sigma M/x\}^{A_0^-} \sigma A$. By the induction hypothesis (part 2c) we know $\Gamma, \sigma\Gamma' \vdash \sigma S : \sigma\{M/x\}^{A_0^-} A[p] > \sigma C[q]$, which is the same thing as $\Gamma, \sigma\Gamma' \vdash \sigma S : \{\sigma M/x\}^{A_0^-} \sigma A[p] > \sigma C[q]$. By rule application we obtain $\Gamma, \sigma\Gamma' \vdash (\sigma M; \sigma S) : \Pi x{:}\sigma A_0.\sigma A[p] > \sigma C[q]$.

Case: Part 1a, with

$$\mathcal{D} = \frac{\mathcal{D}_1}{\Gamma, \beta : \mathsf{w}, \Gamma' \vdash M : (\{p/\alpha\}^{\mathsf{w}} A_0)[p]}{\Gamma, \beta : \mathsf{w}, \Gamma' \vdash M : {\downarrow}\alpha.A_0[p]}$$

Because of the substitution $\{p/\alpha\}^{\mathsf{w}}$, this case depends critically on Lemma 3.2.6, the property of commuting substitutions.

By the induction hypothesis (part 1a) on $\mathcal{D}_1$, we know $\Gamma, \sigma\Gamma' \vdash \sigma M \Leftarrow \sigma\{p/\alpha\}A_0[\sigma p]$. Observe that q (remember σ is $\{q/\beta\}^{\mathsf{w}}$) has no free occurrence of α, which was just introduced, so by Lemma 3.2.6 infer that $\sigma\{p/\alpha\}^{\mathsf{w}} A_0 = \{\sigma p/\alpha\}^{\mathsf{w}} \sigma A_0$. Therefore we have $\Gamma, \sigma\Gamma' \vdash \sigma M \Leftarrow \{\sigma p/\alpha\}\sigma A_0[\sigma p]$, and by rule application $\Gamma, \sigma\Gamma' \vdash \sigma M \Leftarrow {\downarrow}\alpha.\sigma A_0[\sigma p]$.

■

3.2.3 Identity Property

The complement to the substitution property is the identity property, which expresses that the type theory enjoys an internal, global completeness: that any variable of type A has an η-expansion to a canonical term of type A that acts as a unit with respect to substitution. This is nontrivial because variables themselves are not terms. Because canonical forms are η-long, variables must be η-expanded.

Not only it is necessary to show that η-expansion is possible syntactically, but also that the expansion has the appropriate type. Because of dependent types, the proof of typing depends on also showing that η-expansion behaves correctly (namely, that it acts as a two-sided identity) with respect to substitution.

Furthermore to carry out the appropriate induction it is necessary to speak of intermediate stages of construction of the η-expansion, in other words of *partially* applied spines, in contrast to the requirement (precisely to enforce η-longness) in the type theory that all heads are fully applied, that they are supplied with the complete list of arguments.

First, we define *lax reduction* $[M \| S]^\tau$ by

$$\begin{array}{rcl} [\lambda x.M \| (N; S)]^{\tau_1 \to \tau_2} & = & [\{N/x\}^{\tau_1} M \| S]^{\tau_2} \\ [\lambda \alpha.M \| (p; S)]^{\mathsf{w} \to \tau} & = & [\{p/\alpha\}^{\mathsf{w}} M \| S]^{\tau} \\ [\langle M_1, M_2 \rangle \| (\pi_i; S)]^{\tau_1 \& \tau_2} & = & [M_i \| S]^{\tau_i} \\ [M \| S]^{\star \tau} & = & [M \| S]^{\tau} \\ [M \| ()]^{\tau} & = & M \end{array}$$

This is identical to ordinary reduction $[M \mid S]^\tau$ except (because of the clause $[M\|()]^\tau = M$ that is more general than $[R \mid ()]^\bullet = R$) that it allows functions to be applied to *fewer* arguments than there are λ abstractions, or terms of product type to be projected fewer times than there are pair constructors.

For the sake of the tacit assumption that everything in sight is simply well-typed, we say $[M\|S]^\tau$ is simply well-typed if $\gamma \vdash M \Leftarrow \tau$ and $\gamma \vdash S : \tau > \tau'$. It is straightforward to check the following simple typing fact, by induction on τ.

Lemma 3.2.8 *If* $\gamma \vdash M \Leftarrow \tau$ *and* $\gamma \vdash S : \tau > \tau'$, *then* $\gamma \vdash [M\|S]^\tau \Leftarrow \tau'$.

Note that the existing type system is already sufficiently expressive to describe typing of incomplete spines: it is only at the boundary between type synthesis and type checking that we impose the restriction that the resulting type is a base type.

We first establish a few lemmas regarding lax reduction and η-expansion.

Lemma 3.2.9 (Lax Reduction is Associative) $[M\|(S; S')]^\tau = [[M\|S]^\tau \| S']^{\tau'}$

Proof By induction on τ. To be definite about simple types, suppose $\gamma \vdash M \Leftarrow \tau$ and $\gamma \vdash S : \tau > \tau'$ and $\gamma \vdash S' : \tau' > \tau''$.

Case: $\tau = \bullet$. Both spines must be empty for simple typing to hold, so the result immediately follows.

Case: $\tau = \tau_1 \to \tau_2$. In this case S is of the form $(M_0; S_0)$ and M is of the form $\lambda y.N$. We have also the typing $\gamma \vdash S_0 : \tau_2 \to \tau'$.

$$\begin{aligned}
[M\|(S;S')]^{\tau} &= [\lambda y.N\|(M_0;S_0;S')]^{\tau_1\to\tau_2} \\
&= [\{M_0/y\}^{\tau_1}N\|(S_0;S')]^{\tau_2} \\
&= [[\{M_0/y\}^{\tau_1}N\|S_0]^{\tau_2}\|S']^{\tau'} && \text{by i.h.} \\
&= [[\lambda y.N\|(M_0;S_0)]^{\tau_1\to\tau_2}\|S']^{\tau'} \\
&= [[M\|S]^{\tau}\|S']^{\tau'}
\end{aligned}$$

Case: $\tau = \mathsf{w} \to \tau_0$. In this case S is of the form $(p; S_0)$ and M is of the form $\lambda\alpha.N$. We have also the typing $\gamma \vdash S_0 : \tau_0 \to \tau'$.

$$\begin{aligned}
[M\|(S;S')]^{\tau} &= [\lambda\alpha.N\|(p;S_0;S')]^{\mathsf{w}\to\tau_0} \\
&= [\{p/\alpha\}^{\mathsf{w}}N\|(S_0;S')]^{\tau_0} \\
&= [[\{p/\alpha\}^{\mathsf{w}}N\|S_0]^{\tau_0}\|S']^{\tau'} && \text{by i.h.} \\
&= [[\lambda\alpha.N\|(p;S_0)]^{\mathsf{w}\to\tau_0}\|S']^{\tau'} \\
&= [[M\|S]^{\tau}\|S']^{\tau'}
\end{aligned}$$

Case: $\tau = \top$. Cannot occur.

Case: $\tau = \tau_1 \,\&\, \tau_2$. In this case S is of the form $(\pi_i; S_0)$ and M is of the form $\langle N_1, N_2\rangle$.

$$\begin{aligned}
[M\|(S;S')]^{\tau} &= [\langle N_1,N_2\rangle\|(\pi_i;S_0;S')]^{\tau_1\&\tau_2} \\
&= [N_i\|(S_0;S')]^{\tau_i} \\
&= [[N_i\|S_0]^{\tau_i}\|S')]^{\tau'} && \text{by i.h.} \\
&= [[\langle N_1,N_2\rangle\|(\pi_i;S_0)]^{\tau_1\&\tau_2}\|S')]^{\tau'} \\
&= [[M\|S]^{\tau}\|S']^{\tau'}
\end{aligned}$$

Case: $\tau = \star\tau_0$.

$$\begin{aligned}
[M\|(S;S')]^{\tau} &= [M\|(S;S')]^{\star\tau_0} \\
&= [M\|(S;S')]^{\tau_0} \\
&= [[M\|S]^{\tau_0}\|S')]^{\tau'} && \text{by i.h.} \\
&= [[M\|S]^{\star\tau_0}\|S')]^{\tau'} \\
&= [[M\|S]^{\tau}\|S']^{\tau'}
\end{aligned}$$

■

This lemma shows that η-expansion is compatible with substitution.

Lemma 3.2.10 *If $x \neq H$, then $\{M/x\}^{\tau}\mathsf{ex}_{\tau'}(H \cdot S) = \mathsf{ex}_{\tau'}(H \cdot \{M/x\}^{\tau}S)$*

Proof By induction on τ'.

Case: $\tau' = \bullet$. Immediate by definition of ex and substitution.

Case: $\tau' = \tau'_1 \to \tau'_2$. Then

$$\begin{aligned}
&\{M/x\}^{\tau}\mathsf{ex}_{\tau'}(H \cdot S) \\
&= \{M/x\}^{\tau}\lambda z.\mathsf{ex}_{\tau'_2}(H \cdot (S;\mathsf{ex}_{\tau'_1}(z))) \\
&= \lambda z.\{M/x\}^{\tau}\mathsf{ex}_{\tau'_2}(H \cdot (S;\mathsf{ex}_{\tau'_1}(z))) \\
&= \lambda z.\mathsf{ex}_{\tau'_2}(H \cdot \{M/x\}^{\tau}(S;\mathsf{ex}_{\tau'_1}(z))) && \text{by i.h. on } \tau'_2 \\
&= \lambda z.\mathsf{ex}_{\tau'_2}(H \cdot (\{M/x\}^{\tau}S;\{M/x\}^{\tau}\mathsf{ex}_{\tau'_1}(z)))
\end{aligned}$$

$= \lambda z.\mathsf{ex}_{\tau'_2}(H \cdot (\{M/x\}^\tau S; \mathsf{ex}_{\tau'_1}(z \cdot \{M/x\}^\tau())))$ by i.h. on τ'_1
$= \lambda z.\mathsf{ex}_{\tau'_2}(H \cdot (\{M/x\}^\tau S; \mathsf{ex}_{\tau'_1}(z)))$
$= \mathsf{ex}_{\tau'}(H \cdot (\{M/x\}^\tau S))$

Case: $\tau' = \mathsf{w} \to \tau'_0$. Then
$\{M/x\}^\tau \mathsf{ex}_{\tau'}(H \cdot S)$
$= \{M/x\}^\tau \lambda\alpha.\mathsf{ex}_{\tau'_0}(H \cdot (S; \alpha))$
$= \lambda\alpha.\{M/x\}^\tau \mathsf{ex}_{\tau'_0}(H \cdot (S; \alpha))$
$= \lambda\alpha.\mathsf{ex}_{\tau'_0}(H \cdot \{M/x\}^\tau (S; \alpha))$ by i.h. on τ'_0
$= \lambda\alpha.\mathsf{ex}_{\tau'_0}(H \cdot (\{M/x\}^\tau S; \alpha))$
$= \mathsf{ex}_{\tau'}(H \cdot (\{M/x\}^\tau S))$

Case: $\tau' = \tau'_1 \,\&\, \tau'_0$. Then
$\{M/x\}^\tau \mathsf{ex}_{\tau'}(H \cdot S)$
$= \{M/x\}^\tau \langle \mathsf{ex}_{\tau'_1}(H \cdot (S; \pi_1)), \mathsf{ex}_{\tau'_2}(H \cdot (S; \pi_2))\rangle$
$= \langle \{M/x\}^\tau \mathsf{ex}_{\tau'_1}(H \cdot (S; \pi_1)), \{M/x\}^\tau \mathsf{ex}_{\tau'_2}(H \cdot (S; \pi_2))\rangle$
$= \langle \mathsf{ex}_{\tau'_1}(H \cdot \{M/x\}^\tau (S; \pi_1)), \mathsf{ex}_{\tau'_2}(H \cdot \{M/x\}^\tau (S; \pi_2))\rangle$ by i.h. on τ'_1, τ'_2
$= \langle \mathsf{ex}_{\tau'_1}(H \cdot (\{M/x\}^\tau S; \pi_1)), \mathsf{ex}_{\tau'_2}(H \cdot (\{M/x\}^\tau S; \pi_2))\rangle$
$= \mathsf{ex}_{\tau'}(H \cdot (\{M/x\}^\tau S))$

Case: $\tau' = \top$. Immediate.
Case: $\tau' = \star\tau_0$. Then
$\{M/x\}^\tau \mathsf{ex}_{\tau'}(H \cdot S)$
$= \{M/x\}^\tau \mathsf{ex}_{\star\tau_0}(H \cdot S)$
$= \{M/x\}^\tau \mathsf{ex}_{\tau_0}(H \cdot S)$
$= \mathsf{ex}_{\tau_0}(H \cdot \{M/x\}^\tau S)$ by i.h. on τ_0
$= \mathsf{ex}_{\star\tau_0}(H \cdot \{M/x\}^\tau S)$
$= \mathsf{ex}_{\tau'}(H \cdot \{M/x\}^\tau S)$

■

For the following lemma — which essentially expresses the fact mentioned above that η-expansions act as identities — we make use of the evident notion of variable-for-variable substitution $\{x/y\}$, which unlike hereditary substitution needs not perform any reduction.

Lemma 3.2.11 (Identity Laws)

1. $\{\mathsf{ex}_\tau(x)/y\}^\tau X = \{x/y\}X$
2. $[\mathsf{ex}_\tau(x \cdot S) \mid S']^\tau = x \cdot (S; S')$
3. *If* $x \notin FV(S)$ *then* $\{M/x\}^{\tau'} \mathsf{ex}_\tau(x \cdot S) = [M \| S]^{\tau'}$.

Proof By lexicographic induction on τ, the case 1–3, (considering later cases to be smaller for equal τ) and the object X.

1. Split cases on the structure of X. The reasoning is straightforward except when $X = y \cdot S'$. Then we must show, by the definition of substitution,

$$[\mathsf{ex}_\tau(x) \mid \{\mathsf{ex}_\tau(x)/y\}^\tau S']^\tau = x \cdot (\{x/y\}S')$$

We get $\{\mathsf{ex}_\tau(x)/y\}^\tau S' = \{x/y\}S'$ from the i.h. part 1, on the smaller expression S'. Then appeal to the i.h. part 2 (with $S = ()$) on the same simple type τ to see that

$$[\mathsf{ex}_\tau(x) \mid \{x/y\}S']^\tau = x \cdot (\{x/y\}S')$$

2. Split cases on τ.

Case: $\tau = \bullet$. Immediate from definitions, noting that by simple typing, we must have $S' = ()$.

Case: $\tau = \tau_1 \to \tau_2$. We have that S' must be of the form $(M_0; S_0)$.

$$\begin{aligned}
&[\mathsf{ex}_\tau(x \cdot S) \mid S']^\tau \\
&= [\lambda y.\mathsf{ex}_{\tau_2}(x \cdot (S; \mathsf{ex}_{\tau_1}(y))) \mid (M_0; S_0)]^\tau \\
&= [\{M_0/y\}^{\tau_1}\mathsf{ex}_{\tau_2}(x \cdot (S; \mathsf{ex}_{\tau_1}(y))) \mid S_0]^{\tau_2} \\
&= [\mathsf{ex}_{\tau_2}(x \cdot \{M_0/y\}^{\tau_1}(S; \mathsf{ex}_{\tau_1}(y))) \mid S_0]^{\tau_2} && \text{by Lemma 3.2.10} \\
&= [\mathsf{ex}_{\tau_2}(x \cdot (S; \{M_0/y\}^{\tau_1}\mathsf{ex}_{\tau_1}(y))) \mid S_0]^{\tau_2} && y \notin FV(S) \\
&= [\mathsf{ex}_{\tau_2}(x \cdot (S; [M_0 \| ()]^\tau)) \mid S_0]^{\tau_2} && \text{by i.h. 3 on } \tau_1 \\
&= [\mathsf{ex}_{\tau_2}(x \cdot (S; M_0)) \mid S_0]^{\tau_2} \\
&= x \cdot (S; M_0; S_0) && \text{by i.h. 2 on } \tau_2 \\
&= x \cdot (S; S')
\end{aligned}$$

Case: $\tau = \mathsf{w} \to \tau_0$. We have that S' must be of the form $(p; S_0)$.

$$\begin{aligned}
&[\mathsf{ex}_\tau(x \cdot S) \mid S']^\tau \\
&= [\lambda\alpha.\mathsf{ex}_{\tau_0}(x \cdot (S; \alpha)) \mid (p; S_0)]^\tau \\
&= [\{p/\alpha\}^{\mathsf{w}}\mathsf{ex}_{\tau_0}(x \cdot (S; \alpha)) \mid S_0]^{\tau_0} \\
&= [\mathsf{ex}_{\tau_0}(x \cdot \{p/\alpha\}^{\mathsf{w}}(S; \alpha)) \mid S_0]^{\tau_0} && \text{by Lemma 3.2.10} \\
&= [\mathsf{ex}_{\tau_0}(x \cdot (S; \{p/\alpha\}^{\mathsf{w}}\alpha)) \mid S_0]^{\tau_0} && \alpha \notin FV(S) \\
&= [\mathsf{ex}_{\tau_0}(x \cdot (S; p)) \mid S_0]^{\tau_0} \\
&= x \cdot (S; p; S_0) && \text{by i.h. 2 on } \tau_0 \\
&= x \cdot (S; S')
\end{aligned}$$

Case: $\tau = \tau_1 \,\&\, \tau_2$. We know S' is of the form $(\pi_i; S_0)$.

$$\begin{aligned}
&[\mathsf{ex}_\tau(x \cdot S) \mid S']^\tau \\
&= [\langle \mathsf{ex}_{\tau_1}(x \cdot (S; \pi_1)), \mathsf{ex}_{\tau_2}(x \cdot (S; \pi_2))\rangle \mid (\pi_i; S_0)]^\tau \\
&= [\mathsf{ex}_{\tau_i}(x \cdot (S; \pi_i)) \mid S_0]^{\tau_i} \\
&= x \cdot (S; \pi_i; S_0) && \text{by i.h. 2 on } \tau_i \\
&= x \cdot (S; S')
\end{aligned}$$

Case: $\tau = \top$. Impossible.

Case: $\tau = \star\tau_0$.

$$\begin{aligned}
&[\mathsf{ex}_\tau(x \cdot S) \mid S']^\tau \\
&= [\mathsf{ex}_{\star\tau_0}(x \cdot S) \mid S']^{\star\tau_0} \\
&= [\mathsf{ex}_{\tau_0}(x \cdot S) \mid S']^{\tau_0} \\
&= x \cdot (S; S') && \text{by i.h. 2 on } \tau_0
\end{aligned}$$

3. Split cases on τ.

Case: $\tau = \bullet$. Immediate.

Case: $\tau = \tau_1 \to \tau_2$. Note first of all that whenever $\gamma \vdash N \Leftarrow \tau_1 \to \tau_2$ we have

$$\lambda y.[N\|(\mathsf{ex}_{\tau_1}(y))]^{\tau_1 \to \tau_2} = N$$

This is because N, by inversion, must be of the form $\lambda z.N_0$, in which case we have

$\lambda y.[\lambda z.N_0\|(\mathsf{ex}_{\tau_1}(y))]^{\tau}$
$= \lambda y.[\{\mathsf{ex}_{\tau_1}(y)/z\}^{\tau_1} N_0\|()]^{\tau_2}$
$= \lambda y.\{\mathsf{ex}_{\tau_1}(y)/z\}^{\tau_1} N_0$
$= \lambda y.\{y/z\}N_0$ by i.h. 1 on τ_1
$= N$ α-equivalence

Having made this observation, compute

$\{M/x\}^{\tau'}\mathsf{ex}_{\tau}(x \cdot S)$
$= \{M/x\}^{\tau'}\lambda y.\mathsf{ex}_{\tau_2}(x \cdot (S; \mathsf{ex}_{\tau_1}(y)))$
$= \lambda y.\{M/x\}^{\tau'}\mathsf{ex}_{\tau_2}(x \cdot (S; \mathsf{ex}_{\tau_1}(y)))$
$= \lambda y.[M\|(S; \mathsf{ex}_{\tau_1}(y))]^{\tau'}$ by i.h. 3 on τ_2
$= \lambda y.[[M\|S]^{\tau'}\|(\mathsf{ex}_{\tau_1}(y))]^{\tau_1}$ associativity of lax reduction
$= [M\|S]^{\tau'}$ by above observation

Case: $\tau = \mathsf{w} \to \tau_0$. Similar to the above case.

Case: $\tau = \tau_1 \,\&\, \tau_2$. First of all observe that $[M\|S]^{\tau'}$, since it has simple type $\tau_1 \,\&\, \tau_2$, is of the form $\langle N_1, N_2\rangle$. Having made this observation, compute

$\{M/x\}^{\tau'}\mathsf{ex}_{\tau}(x \cdot S)$
$= \{M/x\}^{\tau'}\langle \mathsf{ex}_{\tau_1}(x \cdot (S; \pi_1)), \mathsf{ex}_{\tau_2}(x \cdot (S; \pi_2))\rangle$
$= \langle \{M/x\}^{\tau'}\mathsf{ex}_{\tau_1}(x \cdot (S; \pi_1)), \{M/x\}^{\tau'}\mathsf{ex}_{\tau_2}(x \cdot (S; \pi_2))\rangle$
$= \langle [M\|(S; \pi_1)]^{\tau'}, [M\|(S; \pi_2)]^{\tau'}\rangle$ i.h. twice, on τ_1, τ_2
$= \langle [[M\|S]^{\tau'}\|(\pi_1)]^{\tau}, [[M\|S]^{\tau'}\|(\pi_2)]^{\tau}\rangle$ associativity of lax reduction
$= \langle [\langle N_1, N_2\rangle\|(\pi_1)]^{\tau}, [\langle N_1, N_2\rangle\|(\pi_2)]^{\tau}\rangle$
$= \langle [N_1\|()]^{\tau_1}, [N_2\|()]^{\tau_2}\rangle$
$= \langle N_1, N_2\rangle$
$= [M\|S]^{\tau'}$

Case: $\tau = \star\tau_0$. Then compute

$\{M/x\}^{\tau'}\mathsf{ex}_{\tau}(x \cdot S)$
$= \{M/x\}^{\tau'}\mathsf{ex}_{\star\tau_0}(x \cdot S)$
$= \{M/x\}^{\tau'}\mathsf{ex}_{\tau_0}(x \cdot S)$
$= [M\|S]^{\tau'}$ i.h. on τ_0

Case: $\tau = \top$. Immediate

■

The following lemma, a generalization of Lemma 3.2.2, allows us to compose spines together, construing $\Gamma \vdash S : A[p] > B[q]$ as a sort of function from $A[p]$ to $B[q]$.

Lemma 3.2.12 (Spine Composition) *If* $\Gamma \vdash S_1 : A[p] > B[q]$ *and* $\Gamma \vdash S_2 : B[q] > C[r]$, *then* $\Gamma \vdash (S_1; S_2) : A[p] > C[r]$.

Proof By straightforward induction on the typing derivation of S_1, observing that no rule but the rule that types the nil spine does anything but pass the output type along. ∎

Now we have enough tools to prove that any atomic term that synthesizes a type (and world) can be η-expanded to a normal term that checks at that type (and that world). As a corollary, any variable, being trivially an atomic term at the type at which it is declared in the context by application to the empty spine, can be fully expanded to a normal term at its type and the empty world.

Theorem 3.2.13 (η-Expansion) *Suppose* $\Gamma^- \vdash A : \mathsf{type}$. *If* $\Gamma \vdash R \Rightarrow A[p]$, *then* $\Gamma \vdash \mathsf{ex}_\tau(R) \Leftarrow A[p]$, *where* $\tau = A^-$.

Proof By induction on the simplification of A. Say $R = H \cdot S$, so that by inversion $H : C$ and $\Gamma \vdash S : C[\varepsilon] > A[p]$. This latter fact will typically be used in appeals to Lemma 3.2.12.

Case: $A = b$. Immediate.

Case: $A = \Pi y{:}A_1.A_2$. In the induction hypotheses used here, the simple type A^- always decreases. Reason as follows:

$\Gamma^- \vdash \Pi y{:}A_1.A_2 : \mathsf{type}$	Assumption.
$\Gamma^- \vdash A_1 : \mathsf{type}$	Inversion
$\Gamma^-, y : A_1^- \vdash A_2 : \mathsf{type}$	Inversion
$\Gamma, y : A_1 \vdash y \Rightarrow A_1[\varepsilon]$	Rule
$\Gamma, y : A_1 \vdash \mathsf{ex}_{A_1^-}(y) \Leftarrow A_1[\varepsilon]$	i.h. at A_1^-
$\Gamma, y : A_1 \vdash () : \{\mathsf{ex}_{A_1^-}(y)/y\}^{A_1^-} A_2[p] > A_2[p]$	Lemma 3.2.11
$\Gamma, y : A_1 \vdash (\mathsf{ex}_{A_1^-}(y)) : \Pi y{:}A_1.A_2[p] > A_2[p]$	Rule
$\Gamma, y : A_1 \vdash (S; \mathsf{ex}_{A_1^-}(y)) : C[\varepsilon] > A_2[p]$	Lemma 3.2.12
$\Gamma, y : A_1 \vdash H \cdot (S; \mathsf{ex}_{A_1^-}(y)) \Rightarrow A_2[p]$	Rule
$\Gamma^- \vdash \{\mathsf{ex}_{A_1^-}(y)/y\}^{A_1^-} A_2 : \mathsf{type}$	Lemma 3.2.1
$\Gamma^- \vdash A_2 : \mathsf{type}$	Lemma 3.2.11
$\Gamma, y : A_1 \vdash \mathsf{ex}_{A_2^-}(H \cdot (S; \mathsf{ex}_{A_1^-}(y))) \Leftarrow A_2[p]$	i.h. at A_2^-
$\Gamma \vdash \lambda y.\mathsf{ex}_{A_2^-}(H \cdot (S; \mathsf{ex}_{A_1^-}(y))) \Leftarrow \Pi y{:}A_1.A_2[p]$	Rule

Case: $A = \Pi\alpha{:}\mathsf{w}.A_0$. Reason as follows:

$\Gamma^- \vdash \Pi\alpha{:}\mathsf{w}.A_0 : \mathsf{type}$	Assumption.
$\Gamma^-, \alpha : \mathsf{w}^- \vdash A_0 : \mathsf{type}$	Inversion
$\Gamma, \alpha : \mathsf{w} \vdash (\alpha) : \Pi\alpha{:}\mathsf{w}.A_0[p] > A_0[p]$	Rule
$\Gamma, \alpha : \mathsf{w} \vdash (S; \alpha) : C[\varepsilon] > A_0[p]$	Lemma 3.2.12
$\Gamma, \alpha : \mathsf{w} \vdash H \cdot (S; \alpha) \Rightarrow A_0[p]$	Rule
$\Gamma, \alpha : \mathsf{w} \vdash \mathsf{ex}_{A_0^-}(H \cdot (S; \alpha)) \Leftarrow A_0[p]$	i.h. at A_0^-
$\Gamma \vdash \lambda\alpha.\mathsf{ex}_{A_0^-}(H \cdot (S; \alpha)) \Leftarrow \Pi\alpha{:}\mathsf{w}.A_0[p]$	Rule

In this and the following case we are implicitly using the fact that the effect of the substitution $\{\alpha/\alpha\}^{\mathsf{w}}$ is the identity. This is straightfoward, in contrast to the previous case, because world variables are not η-expanded.

Case: $A = \forall\alpha.A_0$. Reason as follows:

$\Gamma^- \vdash \forall\alpha.A_0 : \mathsf{type}$	Assumption.
$\Gamma^-, \alpha : \mathsf{w}^- \vdash A_0 : \mathsf{type}$	Inversion
$\Gamma, \alpha : \mathsf{w} \vdash () : \forall\alpha.A_0[p] > A_0[p]$	Rule
$\Gamma, \alpha : \mathsf{w} \vdash S : C[\varepsilon] > A_0[p]$	Lemma 3.2.12
$\Gamma, \alpha : \mathsf{w} \vdash H \cdot S \Rightarrow A_0[p]$	Rule
$\Gamma, \alpha : \mathsf{w} \vdash \mathsf{ex}_{A_0^-}(H \cdot S) \Leftarrow A_0[p]$	i.h. at A_0^-
$\Gamma \vdash \mathsf{ex}_{A_0^-}(H \cdot S) \Leftarrow \forall\alpha.A_0[p]$	Rule

Case: $A = A_1 \mathbin{\&} A_2$. Reason as follows: (for both $i \in \{1,2\}$ when i occurs — use inversion to see $\Gamma \vdash A_i : \mathsf{type}$ as required for the induction hypothesis uses)

$\Gamma \vdash (\pi_i) \Rightarrow A_1 \mathbin{\&} A_2[p] > A_i[p]$	Rule
$\Gamma \vdash (S;\pi_i) \Rightarrow C[\varepsilon] > A_i[p]$	Lemma 3.2.12
$\Gamma \vdash H \cdot (S;\pi_i)) \Rightarrow A_i[p]$	Rule
$\Gamma \vdash \mathsf{ex}_{A_i^-}(H \cdot (S;\pi_i)) \Leftarrow A_i[p]$	i.h. at A_i^-
$\Gamma \vdash \langle \mathsf{ex}_{A_1^-}(H \cdot (S;\pi_1)), \mathsf{ex}_{A_2^-}(H \cdot (S;\pi_2))\rangle \Leftarrow A_1 \mathbin{\&} A_2[p]$	Rule

Case: $A = \top$. Immediate.

Case: $A = \downarrow\alpha.A_0$.

$\Gamma \vdash () : \downarrow\alpha.A_0[p] > \{p/\alpha\}^{\mathsf{w}}A_0[p]$	Rule
$\Gamma \vdash S : C[\varepsilon] > \{p/\alpha\}^{\mathsf{w}}A_0[p]$	Lemma 3.2.12
$\Gamma \vdash H \cdot S \Rightarrow \{p/\alpha\}^{\mathsf{w}}A_0[p]$	Rule
$\Gamma \vdash \mathsf{ex}_{\tau}(H \cdot S) \Leftarrow \{p/\alpha\}^{\mathsf{w}}A_0[p]$	i.h. at A_0^-
$\Gamma \vdash \mathsf{ex}_{\tau}(H \cdot S) \Leftarrow \downarrow\alpha.A_0[p]$	Rule

Case: $A = A_0 \mathbin{@} q$.

$\Gamma \vdash () : A_0 \mathbin{@} q[p] > A_0[q]$	Rule
$\Gamma \vdash S : C[\varepsilon] > A_0[q]$	Lemma 3.2.12
$\Gamma \vdash H \cdot S \Rightarrow A_0[q]$	Rule
$\Gamma \vdash \mathsf{ex}_{\tau}(H \cdot S) \Leftarrow A_0[q]$	i.h. at A_0^-
$\Gamma \vdash \mathsf{ex}_{\tau}(H \cdot S) \Leftarrow A_0 \mathbin{@} q[p]$	Rule

■

Corollary 3.2.14 (Identity Property) *If* $x : A \in \Gamma$, *then* $\Gamma \vdash \mathsf{ex}_{A^-}(x) \Leftarrow A[\varepsilon]$.

Chapter 4

Embeddings and Elaboration

Moving beyond the basic properties already discussed, which are essential to any similar logical framework, we treat in the next two sections some results that are particular to HLF's representational capabilities. Following that, we describe in the following chapter the design of algorithms built on top of its type theory that are useful for automated reasoning: unification and coverage checking.

It is useful as a preliminary step to carve out two subsets of the syntax of HLF as we have described it, call them HLF_Π and $\text{HLF}_\forall$. HLF_Π removes from the language the constructs $\forall\alpha.K$ and $\forall\alpha.A$, leaving $\Pi\alpha{:}\mathsf{w}.K$ and $\Pi\alpha{:}\mathsf{w}.A$ as the only forms of universal quantification over worlds. $\text{HLF}_\forall$ conversely removes $\Pi\alpha{:}\mathsf{w}.K$ and $\Pi\alpha{:}\mathsf{w}.A$, leaving $\forall\alpha.K$ and $\forall\alpha.A$. As a historical note, we first developed the system $\text{HLF}_\forall$, until it became clear that type reconstruction, unification, and coverage checking were much more well-behaved in HLF_Π. Section 4.1 is the account of the one purpose for which we found $\text{HLF}_\forall$ still useful, namely the fact that $\text{HLF}_\forall$ is a strict generalization of LLF, and the fact that it serves as a useful stepping-stone in the proof that HLF_Π is a generalization of LLF without $\top$.

The reason the previous chapter includes both subsets in one subsuming framework for the sake of uniformity of proofs. HLF_Π, however, is the language that is fundamentally the subject of our thesis statement, which we believe to be a good foundation for metatheory in general. In the sections following Section 4.1, we say simply HLF to mean HLF_Π.

4.1 Embedding LLF

In this section our goal is to explain the relationship between HLF and the linear logical framework LLF introduced by Cervesato and Pfenning [CP02].

We begin the story by trying to show (towards an illustrative failure) that HLF_Π is a simple generalization of LLF, by translating LLF's linear function type $\multimap$ as

$$B_1 \multimap B_2 \equiv \Pi\alpha{:}\mathsf{w}.{\downarrow}\beta.(B_1 \mathbin{@} \alpha) \to (B_2 \mathbin{@} (\beta * \alpha)) \qquad (*)$$

For clarity, we sometimes write $A^{\multimap}$ below to emphatically indicate the replacement of every $B_1 \multimap B_2$ occurring in A with the right-hand side of $(*)$, but otherwise we imagine

that $\multimap$ is identified with its macro-expansion.

The additive unit type $\top$, whose sole inhabitant $\langle\rangle$ is able to absorb arbitrary resources, causes problems with this attempt, for HLF_Π can be seen to distinguish in its terms the ways that different copies of $\langle\rangle$ divide up the linear context so absorbed, whereas LLF does not. For instance, supposing a signature Σ containing the types $o : \mathsf{type}$ and $\iota : \mathsf{type}$ and the constant $c : \top \multimap \top \multimap \iota$, in HLF_Π (taking the above definition $(*)$ of the meaning of $\multimap$) there are two terms of type $o \multimap \iota$ in the empty context, at world ε, written not in spine form to avoid cluttering the notation:

$$\vdash_{HLF} \lambda\alpha.\lambda x.c\ \alpha\ \langle\rangle\ \varepsilon\ \langle\rangle : o \multimap \iota[\varepsilon]$$

$$\vdash_{HLF} \lambda\alpha.\lambda x.c\ \varepsilon\ \langle\rangle\ \alpha\ \langle\rangle : o \multimap \iota[\varepsilon]$$

While in LLF, there is only one term of type $o \multimap \iota$ in the same context:[1]

$$\vdash_{LLF} \lambda x.c\ \langle\rangle\ \langle\rangle : o \multimap \iota$$

We can, however, recover the the faithful embedding into HLF_Π of LLF *without* $\top$, and see moreover that in the presence of $\top$, HLF provides a finer equational theory on terms, which arguably reflects the structure of linear logic *differently* from LLF, but no less (or more) canonically. Essentially, HLF captures more naturally the structure of *derivations* in linear logic, which do track the exact flow of resources, even into the additive unit $\langle\rangle$, whereas LLF is more closely connected to the linear λ-calculus viewed as a restriction of the ordinary λ-calculus to functions that happen to use their argument according to a linear discipline.

We show that this embedding of LLF without $\top$ into HLF_Π is correct in two steps. The first is the result, of independent interest, that $\text{HLF}_\forall$ faithfully embeds all of LLF. It will be seen that $\text{HLF}_\forall$ omits in terms exactly the same resource information that LLF does with regard to $\top$. This serves to underscore the fact that the hybrid approach *per se* is not inadequate for representing LLF — it is merely that our choice of equational theory for terms in HLF (which is chosen to make later algorithms such as type reconstruction, unification, and coverage more tractable) is *different* from the effective equational theory of terms in LLF. The second step is to see that $\text{HLF}_\forall$ and HLF_Π are equivalent in the absence of $\top$. This is easier to show than directly comparing LLF and HLF_Π, because both $\text{HLF}_\forall$ and HLF_Π are formulated in hybrid style, and one is able to talk about, for instances, uniqueness of *worlds* (see Lemma 4.1.6) and the resource use they represent, independently of other collateral features of contexts.

4.1.1 LLF

In this section we show that $\text{HLF}_\forall$ embeds a certain presentation of LLF, namely one given in canonical forms style, with the the syntax for linear and unrestricted lambda

[1]For the reader more familiar with the syntax of LLF in the literature as opposed to the syntax we used below, this would be written $\vdash_{LLF} \hat{\lambda}x.c \hat{} \langle\rangle \hat{} \langle\rangle : o \multimap \iota$

abstractions conflated, and only distinguished by their typing. Neither of these differences from LLF as it has been published is essential for the meaning of the system, but the choices we make here make the proofs considerably simpler. For our purposes, the syntax of LLF types is

$$A, B ::= a \cdot S \mid A \multimap B \mid \Pi x{:}A.B \mid A \mathbin{\&} B \mid \top$$

and its term language is identical to that of $\mathrm{HLF}_\forall$, namely

$$\begin{array}{rrl} \text{Normal Terms} & M, N & ::= \lambda x.M \mid R \mid \langle M_1, M_2 \rangle \mid \langle\rangle \\ \text{Atomic Terms} & R & ::= H \cdot S \\ \text{Heads} & H & ::= x \mid c \\ \text{Spines} & S & ::= () \mid (M; S) \mid (\pi_1; S) \mid (\pi_2; S) \end{array}$$

Its central typing judgments are:

$$\begin{array}{l} \Gamma; \Delta \vdash_{LLF} M \Leftarrow A \\ \Gamma; \Delta \vdash_{LLF} R \Rightarrow A \\ \Gamma; \Delta \vdash_{LLF} S : A > C \end{array}$$

where Δ is a context of *linear hypotheses* $x\hat{:}A$, subject to exchange but not contraction or weakening.

Its typing rules are

$$\frac{\Gamma; \Delta \vdash_{LLF} R \Rightarrow a \cdot S \qquad S =_\alpha S'}{\Gamma; \Delta \vdash_{LLF} R \Leftarrow a \cdot S'}$$

$$\frac{\Gamma, x : A; \Delta \vdash_{LLF} M \Leftarrow B}{\Gamma; \Delta \vdash_{LLF} \lambda x.M \Leftarrow \Pi x{:}A.B}$$

$$\frac{\Gamma; \Delta, x\hat{:}A \vdash_{LLF} M \Leftarrow B}{\Gamma; \Delta \vdash_{LLF} \lambda x.M \Leftarrow A \multimap B}$$

$$\frac{\Gamma; \Delta \vdash_{LLF} M_1 \Leftarrow A_1 \qquad \Gamma; \Delta \vdash_{LLF} M_2 \Leftarrow A_2}{\Gamma; \Delta \vdash_{LLF} \langle M_1, M_2 \rangle \Leftarrow A_1 \mathbin{\&} A_2}$$

$$\frac{}{\Gamma; \Delta \vdash_{LLF} \langle\rangle \Leftarrow \top}$$

$$\frac{c : A \in \Sigma \qquad \Gamma; \Delta \vdash_{LLF} S : A > C}{\Gamma; \Delta \vdash_{LLF} c \cdot S \Rightarrow C}$$

$$\frac{x : A \in \Gamma \qquad \Gamma; \Delta \vdash_{LLF} S : A > C}{\Gamma; \Delta \vdash_{LLF} x \cdot S \Rightarrow C} \qquad \frac{\Gamma; \Delta \vdash_{LLF} S : A > C}{\Gamma; \Delta, x\hat{:}A \vdash_{LLF} x \cdot S \Rightarrow C}$$

$$\frac{}{\Gamma; \cdot \vdash_{LLF} () : a \cdot S > a \cdot S}$$

$$\frac{\Gamma; \cdot \vdash_{LLF} M \Leftarrow A \qquad \Gamma; \Delta \vdash_{LLF} S : \{M/x\}^{A^-} B > C}{\Gamma; \Delta \vdash_{LLF} (M; S) : \Pi x{:}A.B > C}$$

$$\frac{\Gamma; \Delta_1 \vdash_{LLF} M \Leftarrow A \qquad \Gamma; \Delta_2 \vdash_{LLF} S : B > C}{\Gamma; \Delta_1, \Delta_2 \vdash_{LLF} (M; S) : A \multimap B > C}$$

$$\frac{\Gamma; \Delta \vdash_{LLF} S : A_i > C}{\Gamma; \Delta \vdash_{LLF} (\pi_i; S) : A_1 \,\&\, A_2 > C}$$

4.1.2 Soundness of embedding LLF into $\text{HLF}_\forall$

In this subsection, we are concerned exclusively with $\text{HLF}_\forall$, and so we change Π to $\forall$ in $(*)$ and instead use the encoding of $\multimap$ as

$$B_1 \multimap B_2 \equiv \forall \alpha. \downarrow \beta. (B_1 \mathbin{@} \alpha) \to (B_2 \mathbin{@} (\beta * \alpha)) \qquad (**)$$

Note that by inversion on the typing derivations of $\text{HLF}_\forall$, this is essentially equivalent to adding the typing rules

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

$$\frac{\Gamma \vdash q \Leftarrow \mathsf{w} \qquad \Gamma \vdash M \Leftarrow A[q] \qquad \Gamma \vdash S : B[p * q] > C[r]}{\Gamma \vdash (M; S) : A \multimap B[p] > C[r]}$$

Note that there is an element of nondeterministic choice of the world appearing above the inference line in the spine rules, since it no longer appears in the expression being checked. We continue below to identify $\text{HLF}_\forall$ and LLF types by virtue of this definition, and the direct identification of $\&, \top, \to$, and so on. We say 'A is an LLF type' if it is in the image of this identification, and require until the end of Section 4.1 that all types are such.

We next give a pair of definitions that realize a linear context as an $\text{HLF}_\forall$ context, and restrict an LLF context to a smaller context, provided a $\text{HLF}_\forall$ world that describes the set of linear resources that still remain.

Definition Given an LLF context Δ, we define the $\text{HLF}_\forall$ context $\Delta^{@}$ as follows:

$$(x_1 \mathbin{\hat{:}} A_1, \ldots, x_n \mathbin{\hat{:}} A_n)^{@} =$$

$$(\alpha_{x_1} : \mathsf{w}, x_1 : (A_1 \mathbin{@} \alpha_{x_1}), \ldots, \alpha_{x_n} : \mathsf{w}, x_n : (A_n \mathbin{@} \alpha_{x_n}))$$

A distinct, fresh world variable is invented for each term variable, and the term variable is 'located' at that world by use of $@$. For Δ an LLF context, we also define $\Delta\lfloor_p$ to be the LLF context

$$\Delta\lfloor_p = (x_{i_1} \mathbin{\hat{:}} A_{i_1}, \ldots, x_{i_m} \mathbin{\hat{:}} A_{i_m})$$

whenever $p \equiv_{\mathsf{acu}} \alpha_{x_{i_1}} \cdots \alpha_{x_{i_m}}$ for distinct $i_1, \ldots, i_m$, such that $x_{i_k} : A_{i_k} \in \Delta$ for every $k \in 1 \ldots m$. This definition makes sense if we imagine that the world variable α_x is chosen once and for all to uniquely correspond to the term variable x.

Remark The definition of $\Delta\lfloor_p$ is well-defined up to commutativity and associativity for world concatenation, because we have exchange and associativity on contexts.

There is an important lemma pertaining to the role of spine typings. Since a spine being typed at $A[p] > C[r]$ is intuitively the ability to produce a C that uses resources r from an A that has *already used* resources p, the resources used by S itself are the *difference* between the resources r that are ultimately used, and the resources p that have already been used. The lemma establishes that one can constructively find the 'gap' s between r and p.

Lemma 4.1.1 (Factorization) *Suppose A is an LLF type. If $\Gamma, \Delta^{@} \vdash S : A[p] > C[r]$, then there exists a world s such that $r \equiv_{\mathsf{acu}} p * s$.*

Proof By induction on the typing derivation.

Case: Linear spine cons:

$$\frac{\Gamma, \Delta^{@} \vdash q : \mathsf{w} \qquad \Gamma, \Delta^{@} \vdash M \Leftarrow A[q] \qquad \Gamma, \Delta^{@} \vdash S : B[p * q] > C[r]}{\Gamma, \Delta^{@} \vdash (M; S) : A \multimap B[p] > C[r]}$$

By induction hypothesis, r factors as $(p * q) * s'$ for some s'. Therefore by associativity it also factors as $p * (q * s')$. Set $s = q * s'$.

Case: Ordinary spine cons:

$$\frac{\Gamma, \Delta^{@} \vdash M \Leftarrow A[\varepsilon] \qquad \Gamma, \Delta^{@} \vdash S : \{M/x\}^{A} B[p] > C[r]}{\Gamma, \Delta^{@} \vdash (M; S) : \Pi x{:}A.B[p] > C[r]}$$

By induction hypothesis, r factors as $p * s$ for some s, and we done.

Case: Nil:

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

Set $s = \varepsilon$.

Case: Projection:

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

By induction hypothesis, r factors as $p * s$ for some s, and we are done.

■

We also observe that since the algebraic structure of $\varepsilon, *$ is the free commutative monoid over the world variables appearing in the context, we obtain a cancellativity property.

Lemma 4.1.2 (Cancellativity) *If $p * q \equiv_{\mathsf{acu}} p * r$, then $q \equiv_{\mathsf{acu}} r$.*

Proof A world expression p in a context of world variables $\alpha_1, \ldots, \alpha_n$ is characterized by the number of times each α_i appears in p: all equational laws preserve these counts, and any two such expressions with the same counts are $\equiv_{\mathsf{acu}}$ to one another. Appeal to the corresponding cancellativity property of addition in the module $\mathbb{N}^n$ over the ring $\mathbb{N}$, (and ultimately to cancellativity of addition in $\mathbb{N}$ itself) that if $v + w = v + u$, then $w = u$, for $v, w, u \in \mathbb{N}^n$. ■

Now we can prove that the above embedding is sound and complete, that the set of $\text{HLF}_\forall$ terms at a given type are exactly the same as the LLF terms at that type.

Lemma 4.1.3 (Soundness) *Suppose Δ is a valid LLF context, and A and C are valid LLF types. Assume p and r are each products of distinct world variables.*

1. *If $\Gamma, \Delta^@ \vdash M \Leftarrow A[p]$ then $\Gamma; \Delta\lfloor_p \vdash_{LLF} M \Leftarrow A$.*
2. *If $\Gamma, \Delta^@ \vdash S : A[p] > C[r]$ and $r \equiv_{\mathsf{acu}} p * q$ for some q, then $\Gamma; \Delta\lfloor_q \vdash_{LLF} S : A > C$.*

Proof By induction on the typing derivation.

Case: Linear lambda:

$$\frac{\Gamma, \Delta^@, \alpha_x : \mathsf{w}, x : A \mathbin{@} \alpha_x \vdash M \Leftarrow B[p * \alpha_x]}{\Gamma, \Delta^@ \vdash \lambda x.M \Leftarrow A \multimap B[p]}$$

By the induction hypothesis, we get $\Gamma; \Delta\lfloor_p, x : A \vdash_{LLF} M \Leftarrow B$. By rule, we get $\Gamma; \Delta\lfloor_p \vdash_{LLF} \lambda x.M \Leftarrow A \multimap B$.

Case: Regular lambda:

$$\frac{\Gamma, \Delta^@, x : A \vdash M \Leftarrow B[p]}{\Gamma, \Delta^@ \vdash \lambda x.M \Leftarrow \Pi x{:}A.B[p]}$$

By the induction hypothesis, we get $\Gamma, x : A; \Delta\lfloor_p \vdash_{LLF} M \Leftarrow B$. By rule, we get $\Gamma; \Delta\lfloor_p \vdash_{LLF} \lambda x.M \Leftarrow \Pi x{:}A.B$.

Case: Linear variable:

$$\frac{x : A \mathbin{@} \alpha_x \in \Delta^@ \qquad \dfrac{\Gamma, \Delta^@ \vdash S : A[\alpha_x] > C[r]}{\Gamma, \Delta^@ \vdash S : A \mathbin{@} \alpha_x[\varepsilon] > C[r]}}{\Gamma, \Delta^@ \vdash x \cdot S : C[r]}$$

By lemma, r factors as $q * \alpha_x$ for some q, and by induction hypothesis

$$\Gamma; \Delta\lfloor_q \vdash_{LLF} S : A > C$$

By rule, (since $x : A \mathbin{@} \alpha_x \in \Delta^@$ means we must have had $x : A \in \Delta$)

$$\Gamma; \Delta\lfloor_q, x : A \vdash_{LLF} x \cdot S : C$$

as required, because $\Delta\lfloor_r = \Delta\lfloor_{q * \alpha_x} = \Delta\lfloor_q, x : A$.

Case: Ordinary variable:

$$\frac{x : A \in \Gamma \qquad \Gamma, \Delta^@ \vdash S : A[\varepsilon] > C[r]}{\Gamma, \Delta^@ \vdash x \cdot S : C[r]}$$

By induction hypothesis

$$\Gamma; \Delta\lfloor_r \vdash_{LLF} S : A > C$$

By rule,

$$\Gamma; \Delta\lfloor_r \vdash_{LLF} x \cdot S \Rightarrow C$$

as required.

Case: Pairing:

$$\frac{\Gamma, \Delta^{@} \vdash M_1 \Leftarrow A_1[p] \qquad \Gamma, \Delta^{@} \vdash_{LLF} M_2 \Leftarrow A_2[p]}{\Gamma, \Delta^{@} \vdash \langle M_1, M_2 \rangle \Leftarrow A_1 \mathbin{\&} A_2[p]}$$

By induction hypothesis, we have $\Gamma; \Delta\lfloor_p \vdash_{LLF} M_i \Leftarrow A_i$, so by rule, $\Gamma; \Delta\lfloor_p \vdash_{LLF} \langle M_1, M_2 \rangle \Leftarrow A_1 \mathbin{\&} A_2$.

Case: Unit:

$$\frac{}{\Gamma, \Delta^{@} \vdash \langle\rangle \Leftarrow \top[p]}$$

Immediately by rule we have $\Gamma, \Delta\lfloor_p \vdash_{LLF} \langle\rangle \Leftarrow \top$

Case: Linear spine cons:

$$\frac{\Gamma, \Delta^{@} \vdash q \Leftarrow \mathsf{w} \qquad \Gamma, \Delta^{@} \vdash M \Leftarrow A[q] \qquad \Gamma, \Delta^{@} \vdash S : B[p * q] > C[r]}{\Gamma, \Delta^{@} \vdash (M; S) : A \multimap B[p] > C[r]}$$

By lemma, r factors as $p * q * s$ for some s. By induction hypothesis

$$\Gamma; \Delta\lfloor_q \vdash_{LLF} M \Leftarrow A$$

$$\Gamma; \Delta\lfloor_s \vdash_{LLF} S : A > C$$

By assumption $r \equiv_{\mathsf{acu}} p * q'$ for some q'. But we know then that $p * q * s \equiv_{\mathsf{acu}} p * q'$, so by cancellativity of $*$, we infer $q' \equiv_{\mathsf{acu}} q * s$. We have that $q * s$ and $p * q$ are both products of distinct world variables because they are both factors of $p * q' \equiv_{\mathsf{acu}} r$, which had the same property by assumption. By rule,

$$\Gamma; \Delta\lfloor_{q*s} \vdash_{LLF} (M; S) : A \multimap B > C$$

as required.

Case: Ordinary spine cons:

$$\frac{\Gamma, \Delta^{@} \vdash M \Leftarrow A[\varepsilon] \qquad \Gamma, \Delta^{@} \vdash S : \{M/x\}^{A^-} B[p] > C[r]}{\Gamma, \Delta^{@} \vdash (M; S) : \Pi x{:}A.B[p] > C[r]}$$

Let q be such that $p * q \equiv_{\mathsf{acu}} r$. By induction hypothesis

$$\Gamma; \Delta\lfloor_q \vdash_{LLF} S : \{M/x\}^{A^-} B > C$$

$$\Gamma; \cdot \vdash_{LLF} M \Leftarrow A$$

by rule,

$$\Gamma; \Delta\lfloor_q \vdash_{LLF} (M; S) : \Pi x{:}A.B > C$$

as required.

Case: Projection:

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

Let q be such that $p * q \equiv_{\mathsf{acu}} r$. By induction hypothesis, $\Gamma; \Delta\lfloor_q \vdash_{LLF} S : A_i > C$. By rule, $\Gamma; \Delta\lfloor_q \vdash_{LLF} (\pi_i; S) : A_1 \mathbin{\&} A_2 > C$, as required.

Case: Nil:

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

The only q such that $p * q \equiv_{\mathsf{acu}} p$ is ε. By rule,

$$\Gamma; \cdot \vdash_{LLF} () : a \cdot S > a \cdot S$$

as required.

■

4.1.3 Completeness of embedding LLF into HLF$_\forall$

Here we prove the converse direction, that every LLF proof has a counterpart in HLF$_\forall$.

Definition Let α_Δ be the concatenation of all worlds in $\Delta^@$,

Lemma 4.1.4 (Completeness)

1. *If* $\Gamma; \Delta \vdash_{LLF} M \Leftarrow A$, *then* $\Gamma, \Delta^@ \vdash M \Leftarrow A[\alpha_\Delta]$
2. *If* $\Gamma; \Delta \vdash_{LLF} R \Rightarrow A$, *then* $\Gamma, \Delta^@ \vdash R \Rightarrow A[\alpha_\Delta]$
3. *If* $\Gamma; \Delta \vdash_{LLF} S : A > C$, *then* $\Gamma, \Delta' \vdash S : A[p] > C[\alpha_\Delta * p]$, *for any* Δ' *that extends* $\Delta^@$ *and any* p *such that* $\Delta' \vdash p \Leftarrow \mathsf{w}$.

Proof By induction on the derivation.

Case:

$$\frac{\Gamma; \Delta \vdash_{LLF} R \Rightarrow a \cdot S \qquad a \cdot S = a' \cdot S'}{\Gamma; \Delta \vdash_{LLF} R \Leftarrow a' \cdot S'}$$

Immediate by applying rule to induction hypothesis, since $\alpha_\Delta \equiv_{\mathsf{acu}} \alpha_\Delta$.

Case:

$$\frac{\Gamma, x : A; \Delta \vdash_{LLF} M \Leftarrow B}{\Gamma; \Delta \vdash_{LLF} \lambda x.M \Leftarrow \Pi x{:}A.B}$$

By induction hypothesis, $\Gamma, x : A, \Delta^@ \vdash M \Leftarrow B[\alpha_\Delta]$. By exchange and rule application, $\Gamma, \Delta^@ \vdash \lambda x.M \Leftarrow \Pi x{:}A.B[\alpha_\Delta]$.

Case:

$$\frac{\Gamma; \Delta, x\hat{:}A \vdash_{LLF} M \Leftarrow B}{\Gamma; \Delta \vdash_{LLF} \lambda x.M \Leftarrow A \multimap B}$$

By induction hypothesis, $\Gamma, \Delta^@, \alpha_x : \mathsf{w}, x : A \mathbin{@} \alpha_x \vdash M \Leftarrow B[\alpha_\Delta * \alpha_x]$. By rule application, $\Gamma, \Delta^@ \vdash \lambda x.M \Leftarrow A \multimap B[\alpha_\Delta]$.

Case:

$$\frac{\Gamma; \Delta \vdash_{LLF} M_1 \Leftarrow A_1 \qquad \Gamma; \Delta \vdash_{LLF} M_2 \Leftarrow A_2}{\Gamma; \Delta \vdash_{LLF} \langle M_1, M_2 \rangle \Leftarrow A_1 \mathbin{\&} A_2}$$

By induction hypothesis, $\Gamma, \Delta^@ \vdash M_i \Leftarrow A_i[\alpha_\Delta]$. By rule,

$$\Gamma, \Delta^@ \vdash \langle M_1, M_2 \rangle \Leftarrow A_1 \mathbin{\&} A_2[\alpha_\Delta]$$

Case:

$$\frac{}{\Gamma; \Delta \vdash_{LLF} \langle\rangle \Leftarrow \top}$$

Immediately by rule, $\Gamma, \Delta^{@} \vdash \langle\rangle \Leftarrow \top[\alpha_\Delta]$

Case:

$$\frac{x : A \in \Gamma \qquad \Gamma; \Delta \vdash_{LLF} S : A > C}{\Gamma; \Delta \vdash_{LLF} x \cdot S \Rightarrow C}$$

By induction hypothesis, (choosing $p = \varepsilon$) we have $\Gamma, \Delta^{@} \vdash S : A[\varepsilon] > C[\alpha_\Delta]$. By rule application (since $x : A \in \Gamma$) we get $\Gamma, \Delta^{@} \vdash x \cdot S \Rightarrow C[\alpha_\Delta]$.

Case:

$$\frac{\Gamma; \Delta \vdash_{LLF} S : A > C}{\Gamma; \Delta, x \hat{:} A \vdash_{LLF} x \cdot S \Rightarrow C}$$

$\Gamma, \Delta^{@}, \alpha_x : \mathsf{w}, x : A \mathbin{@} \alpha_x \vdash S : A[\alpha_x] > C[\alpha_\Delta * \alpha_x]$ i.h.
$\Gamma, \Delta^{@}, \alpha_x : \mathsf{w}, x : A \mathbin{@} \alpha_x \vdash S : A @ \alpha_x[\varepsilon] > C[\alpha_\Delta * \alpha_x]$ by rule
$\Gamma, \Delta^{@}, \alpha_x : \mathsf{w}, x : A \mathbin{@} \alpha_x \vdash x \cdot S \Rightarrow C[\alpha_\Delta * \alpha_x]$ by rule.

Case:

$$\frac{}{\Gamma; \cdot \vdash_{LLF} () : a \cdot S > a \cdot S}$$

Immediate. Here $\alpha_\Delta = \varepsilon$, and $\Gamma \vdash () : a \cdot S[p] > a \cdot S[p]$ and $p * \varepsilon \equiv_{\mathsf{acu}} p$.

Case:

$$\frac{\Gamma; \cdot \vdash_{LLF} M \Leftarrow A \qquad \Gamma; \Delta \vdash_{LLF} S : \{M/x\}^{A^-} B > C}{\Gamma; \Delta \vdash_{LLF} (M; S) : \Pi x{:}A.B > C}$$

Let Δ' extending $\Delta^{@}$ and p such that $\Delta' \vdash p \Leftarrow \mathsf{w}$ be given.
$\Gamma, \Delta' \vdash S : \{M/x\}^{A^-} B[p] > C[q]$ by i.h.
$(q \equiv_{\mathsf{acu}} \alpha_\Delta * p)$
$\Gamma \vdash M \Leftarrow A[\varepsilon]$ i.h.
$\Gamma, \Delta' \vdash M \Leftarrow A[\varepsilon]$ weakening.
$\Gamma, \Delta' \vdash (M; S) : \Pi x{:}A.B[p] > C[q]$ by rule.

Case:

$$\frac{\Gamma; \Delta_1 \vdash_{LLF} M \Leftarrow A \qquad \Gamma; \Delta_2 \vdash_{LLF} S : B > C}{\Gamma; \Delta_1, \Delta_2 \vdash_{LLF} (M; S) : A \multimap B > C}$$

Let Δ' extending $\Delta_1^{@}, \Delta_2^{@}$ and p such that $\Delta' \vdash p \Leftarrow \mathsf{w}$ be given.
$\Gamma, \Delta' \vdash S : B[p * \alpha_{\Delta_1}] > C[q]$ by i.h.
$(q \equiv_{\mathsf{acu}} \alpha_{\Delta_2} * (p * \alpha_{\Delta_1}))$
$\Gamma, \Delta_1^{@} \vdash M \Leftarrow A[\alpha_{\Delta_1}]$ i.h.
$\Gamma, \Delta' \vdash M \Leftarrow A[\alpha_{\Delta_1}]$ weakening.
$\Gamma, \Delta' \vdash (M; S) : A \multimap B[p] > C[q]$ by rule.

Case:

$$\frac{\Gamma; \Delta \vdash_{LLF} S : A_i > C}{\Gamma; \Delta \vdash_{LLF} (\pi_i; S) : A_1 \mathbin{\&} A_2 > C}$$

Let Δ' extending $\Delta_1^{@}, \Delta_2^{@}$ and p such that $\Delta' \vdash p \Leftarrow \mathsf{w}$ be given.

$\Gamma, \Delta^{@} \vdash S : A_i[p] > C[\alpha_\Delta * p]$ by i.h.
$\Gamma, \Delta^{@} \vdash (\pi_i; S) : A_1 \mathbin{\&} A_2[p] > C[\alpha_\Delta * p]$ by rule.

■

4.1.4 Equivalence of HLF_Π and $\mathrm{HLF}_\forall$ without $\top$

Having observed that $\mathrm{HLF}_\forall$ faithfully embeds all LLF, we wish to relate its $\top$-free fragment to that of HLF_Π. Toward that end, we can define a erasure from HLF_Π to $\mathrm{HLF}_\forall$ that eliminates every world abstraction and world application. For any HLF_Π term M (resp. spine S), let $M^\dagger$ (resp. $S^\dagger$) be the $\mathrm{HLF}_\forall$ expression defined by

$$\begin{array}{rcl}
(\lambda x.M)^\dagger & = & \lambda x.(M^\dagger) \\
(\lambda \alpha.M)^\dagger & = & M^\dagger \\
\langle M_1, M_2 \rangle^\dagger & = & \langle M_1^\dagger, M_2^\dagger \rangle \\
(H \cdot S)^\dagger & = & H \cdot (S^\dagger) \\
(M; S)^\dagger & = & (M^\dagger; S^\dagger) \\
(p; S)^\dagger & = & S^\dagger \\
(\pi_i; S)^\dagger & = & (\pi_i; S^\dagger)
\end{array}$$

and likewise for types we define a function that changes every Π of a world variable to a $\forall$, and applies the translation to the term arguments of type families.

$$\begin{array}{rcl}
(\Pi x{:}A.B)^\dagger & = & \Pi x{:}A^\dagger.B^\dagger \\
(\Pi \alpha{:}\mathsf{w}.B)^\dagger & = & \forall \alpha{:}\mathsf{w}.B^\dagger \\
(a \cdot S)^\dagger & = & a \cdot (S^\dagger) \\
(\downarrow \alpha.B)^\dagger & = & \downarrow \alpha.B^\dagger \\
(A \mathbin{@} p)^\dagger & = & A^\dagger \mathbin{@} p \\
(A \mathbin{\&} B)^\dagger & = & A^\dagger \mathbin{\&} B^\dagger \\
(\top)^\dagger & = & \top
\end{array}$$

and the translation is lifted in the evident way to kinds, context, and signatures.

Then we can straightforwardly show that

Lemma 4.1.5 *If* $\Gamma \vdash_{HLF_\Pi} M \Leftarrow A[p]$, *then* $\Gamma^\dagger \vdash_{HLF_\forall} M^\dagger \Leftarrow A^\dagger[p]$.

Proof By induction on the derivation. The only thing we must observe at a case like the cons onto a spine of a world application is that the presence of a p in the HLF_Π term is a witness to the existence of a world, which is sufficient evidence to use the corresponding $\mathrm{HLF}_\forall$ rule. ■

However, the converse is not true in the presence of $\top$. This is because dependent types mean that well-typedness of a term depends on equality, and translated terms that are equal in $\mathrm{HLF}_\forall$ were not necessarily equal in HLF_Π. Extending our previous example,

we may consider a signature $\Sigma =$

$$\begin{array}{l} c : \top \multimap \top \multimap \iota \\ a : (o \multimap \iota) \to \mathsf{type} \\ b : a\ (\lambda\alpha.\lambda x.c\ \alpha\ \langle\rangle\ \varepsilon\ \langle\rangle) \to \mathsf{type} \\ d : a\ (\lambda\alpha.\lambda x.c\ \varepsilon\ \langle\rangle\ \alpha\ \langle\rangle) \\ e : b\ d \end{array}$$

which is *not* well-formed in HLF_Π, but its translation $\Sigma^\dagger =$

$$\begin{array}{l} c : \top \multimap \top \multimap \iota \\ a : (o \multimap \iota) \to \mathsf{type} \\ b : a\ (\lambda x.c\ \langle\rangle\ \langle\rangle) \to \mathsf{type} \\ d : a\ (\lambda x.c\ \langle\rangle\ \langle\rangle) \\ e : b\ d \end{array}$$

is well-formed in $\mathrm{HLF}_\forall$. Note that here we are using $\multimap$ in different ways according to whether we are talking about HLF_Π or $\mathrm{HLF}_\forall$. In the former case, we mean $\multimap$ as the abbreviation given in $(*)$ in Section 4.1, and in the latter, the abbreviation given in $(**)$ in Section 4.1.2. Clearly the translation $(\text{—})^\dagger$ carries the one to the other.

Therefore let us restrict our attention to the respective fragments of HLF_Π and $\mathrm{HLF}_\forall$ that lack $\top$ — every result in the remainder of this section is under the assumption that no type mentioned contains $\top$. In that case we can first prove that a $\mathrm{HLF}_\forall$ term uniquely determines the world it is well-typed at, if any.

Lemma 4.1.6 *In $\mathrm{HLF}_\forall$ without $\top$ we have*

1. *If $\Gamma \vdash M \Leftarrow A[p]$ and $\Gamma \vdash M \Leftarrow A[p]$, then $p \equiv_{\mathsf{acu}} p'$.*
2. *If $\Gamma \vdash R \Rightarrow A[p]$ and $\Gamma \vdash R \Rightarrow A[p]$, then $p \equiv_{\mathsf{acu}} p'$.*
3. *If $\Gamma \vdash S : A[p] > C[r]$ and $\Gamma \vdash S : A[p'] > C[r']$, and $p \equiv_{\mathsf{acu}} p'$, then $r \equiv_{\mathsf{acu}} r'$.*

Proof By induction on the derivation. For example, consider the case of part 3 at the type $A \multimap B$, where we have

$$\frac{\Gamma \vdash q \Leftarrow \mathsf{w} \qquad \Gamma \vdash M \Leftarrow A[q] \qquad \Gamma \vdash S : B[p * q] > C[r]}{\Gamma \vdash (M; S) : A \multimap B[p] > C[r]}$$

$$\frac{\Gamma \vdash q' \Leftarrow \mathsf{w} \qquad \Gamma \vdash M \Leftarrow A[q'] \qquad \Gamma \vdash S : B[p' * q'] > C[r']}{\Gamma \vdash (M; S) : A \multimap B[p'] > C[r']}$$

Prima facie we needn't have that q and q' are the same, since we separately assume the existence of derivations of $\Gamma \vdash (M; S) : A \multimap B[p] > C[r]$ and $\Gamma \vdash (M; S) : A \multimap B[p'] > C[r']$. But the induction hypothesis on $\Gamma \vdash M \Leftarrow A[q]$ and $\Gamma \vdash M \Leftarrow A[q']$ gives us that $q \equiv_{\mathsf{acu}} q'$. Use the induction hypothesis on $\Gamma \vdash S : B[p * q] > C[r]$ and $\Gamma \vdash S : B[p' * q'] > C[r']$, which yields $r \equiv_{\mathsf{acu}} r'$, as required.

Note that our pervasive assumption that all the types involved are in LLF is critical here. The full generality of $\mathrm{HLF}_\forall$ types, even without including $\top$ explicitly, violates this uniqueness property. If $o : \mathsf{type}$, and we have the constant $c : \forall\alpha.o \mathbin{@} \alpha$. Then $c \cdot () \Leftarrow o[p]$

for any well-formed p at all. Certainly also the inclusion of $\top$ falsifies this result: likewise $\langle\rangle \Leftarrow \top[p]$ for any well-formed p.

On that note it is also worth carefully examining the additive pair introduction case, to see that it is only the nullary product introduction that causes problems, and not additive products in general. In this case we have

$$\frac{\Gamma \vdash M_1 \Leftarrow A_1[r] \qquad \Gamma \vdash M_2 \Leftarrow A_2[r]}{\Gamma \vdash \langle M_1, M_2\rangle \Leftarrow A_1 \,\&\, A_2[r]}$$

and

$$\frac{\Gamma \vdash M_1 \Leftarrow A_1[r'] \qquad \Gamma \vdash M_2 \Leftarrow A_2[r']}{\Gamma \vdash \langle M_1, M_2\rangle \Leftarrow A_1 \,\&\, A_2[r']}$$

and the equivalence $r \equiv_{\mathsf{acu}} r'$ follows from the induction hypothesis on *either* branch. So we can see that in this case we have two redundant pieces of information that suffice to push the case through, whereas in the case of $\top$ we have zero. ■

Because worlds are unique without $\top$, we are able to show that the translation $(\text{—})^\dagger$ is injective.

Lemma 4.1.7 (Injectivity) *Suppose all types involved are LLF types not using* $\top$.

1. *If* $\Gamma \vdash M_i \Leftarrow A[p_i]$ *for* $i \in \{1,2\}$, *and* $p_1 \equiv_{\mathsf{acu}} p_2$, *then* $M_1^\dagger = M_2^\dagger$ *implies* $M_1 = M_2$.
2. *If* $\Gamma \vdash S_i : A[p_i] > C[r_i]$ *for* $i \in \{1,2\}$, *and* $p_1 \equiv_{\mathsf{acu}} p_2$ *and* $r_1 \equiv_{\mathsf{acu}} r_2$, *then* $S_1^\dagger = S_2^\dagger$ *implies* $S_1 = S_2$.

Proof By induction on the derivations. The critical case is when we have

$$\frac{\Gamma \vdash q_i : \mathsf{w} \qquad \Gamma \vdash M_i \Leftarrow A[q_i] \qquad \Gamma \vdash S_i : B[p * q_i] > C[r]}{\Gamma \vdash (q_i; M_i; S_i) : (A \multimap B)[p] > C[r]}$$

in which case we apply the previous lemma to see that $q_1 \equiv_{\mathsf{acu}} q_2$, and then because of this fact we may use the induction hypothesis to see that $M_1 = M_2$ and $S_1 = S_2$. Altogether we then have $(q_1; M_1; S_1) = (q_2; M_2; S_2)$ as required. ■

Finally we can show the converse of Lemma 4.1.5 holds in the absence of $\top$.

Lemma 4.1.8 *If* $\Gamma^\dagger \vdash_{HLF_\forall} M^\dagger \Leftarrow A^\dagger[p]$, *then* $\Gamma \vdash_{HLF_\Pi} M \Leftarrow A[p]$.

Proof By induction on the derivation. Use injectivity throughout the proof to make the process of case analysis of derivations unambiguous: in any event there is only one M that is the preimage under $(\text{—})^\dagger$ of $M^\dagger$ we get by assumption. ■

4.2 Embedding a Fragment of Bunched Logic

Bunched logic [OP99], also known as the logic of bunched implications, or simply BI for short, is like linear logic, in that it is a substructural logic that enforces a resource discipline on hypotheses in the context. The particularities of its resource discipline, and the structure of its notion of context, however, are rather different. Instead of having two

zones as linear logic does, one containing only linear *resources*, and one containing only unrestricted *assumptions*, bunched logic has a single tree-structured context that expresses resource-like and assumption-like relationships of hypotheses to one another. Instead of two kinds of context, there are two notions of *joining* one context to another – that is, two binary operations on contexts that yield a context. One is multiplicative, corresponding to the resource-like *separating* conjunction that plays such an important role in BI's use in separation logic [Rey02], which expresses that the two underlying contexts hold respectively of some division of the available resources into two parts. The other is additive, which expresses the *simultaneous* truth of the two underlying contexts for the *same* collection of resources.

In this section we show one way how this resource discipline, as it manifests itself in a fragment of BI, can be expressed also with the hybrid logical connectives in HLF, by giving a translation of that fragment into HLF, and showing that provability is preserved. There is a difference worth noting between this embedding of BI and the embedding of LLF above, which is that while we were careful above to ensure that equality (and disequality) of LLF terms was preserved under translation, the embedding of BI is only concerned with translating *provability* of BI propositions as the *existence* of HLF terms of the appropriate type. We make no claims that any particular equational theory on BI derivations is carried to equality of HLF terms.

4.2.1 The Logic of Bunched Implications

We recall here a presentation of the *negative fragment* of BI, restricting our attention to the negative connectives: $\multimap, \mathbin{-\!\!\circ}, \wedge, \top$. Here we use the linear logic symbol rather than the conventional BI $-\!\!*$, to foreshadow the fact that its encoding in HLF will be identical to the encoding of linear logic's $\multimap$. Our use of the word 'negative' should be considered somewhat speculative until such time as a complete focusing calculus for BI is given, but the choices seem fairly well-motivated by analogy to focusing behavior in linear logic, and in any event our present purpose is not to tell the whole story of focusing for BI, but simply to show that a fragment of it can be embedded in HLF. That we give it the name 'negative' is not essential, but is at least highly suggestive.

The syntax of the language is

$$\begin{array}{rrcl} \text{Propositions } A, B & ::= & A \multimap A \mid A \mathbin{-\!\!\circ} A \mid A \wedge A \mid \top \mid \mathsf{a} \\ \text{Contexts } \Gamma, \Delta & ::= & A \mid 1_m \mid 1_a \mid \Gamma, \Gamma \mid \Gamma; \Gamma \end{array}$$

There are two implications, the multiplicative $\multimap$, which represents a function defined *separately* from its argument, and the additive $\mathbin{-\!\!\circ}$, which represents a function allowed to *share* resources with its argument. There is also the additive conjunction $\wedge$, and its unit $\top$. Propositional atoms are written a. Contexts are built from propositions, binary combinations of contexts, and units for those joins. The comma is the multiplicative join, and 1_m is its unit, and the semicolon is the additive join, and 1_a is its unit.

In the sequel, we identify contexts up to the congruence $\equiv$ defined by associativity,

commutativity, and unit laws for both kinds of context joining:

$$\begin{array}{cc} (\Gamma_1;\Gamma_2);\Gamma_3 \equiv \Gamma_1;(\Gamma_2;\Gamma_3) & (\Gamma_1,\Gamma_2),\Gamma_3 \equiv \Gamma_1,(\Gamma_2,\Gamma_3) \\ \Gamma_1;\Gamma_2 \equiv \Gamma_2;\Gamma_1 & \Gamma_1,\Gamma_2 \equiv \Gamma_2,\Gamma_1 \\ \Gamma;1_a \equiv \Gamma & \Gamma,1_m \equiv \Gamma \end{array}$$

$$\frac{}{\Gamma \equiv \Gamma} \quad \frac{\Gamma \equiv \Delta}{\Delta \equiv \Gamma} \quad \frac{\Gamma_1 \equiv \Gamma_2 \quad \Gamma_2 \equiv \Gamma_3}{\Gamma_1 \equiv \Gamma_3}$$

$$\frac{\Gamma \equiv \Delta \quad \Gamma' \equiv \Delta'}{\Gamma;\Gamma' \equiv \Delta;\Delta'} \quad \frac{\Gamma \equiv \Delta \quad \Gamma' \equiv \Delta'}{\Gamma,\Gamma' \equiv \Delta,\Delta'}$$

A sequent calculus for the judgment $\Gamma \vdash_{BI} A$ is given by the following rules, where $\Gamma(—)$ is a notion of a context-with-hole common in the BI literature, so that the expression $\Gamma(\Delta)$ denotes a context tree structure containing Δ as a subtree. When the surrounding $\Gamma(—)$ occurs multiple times in a rule, it denotes replacement of the subtree by different contexts. The import of this is that, for example, in the $-\!\circ\, L$ rule, we are allowed, in order to prove C from a given context, to find any occurrence of $A -\!\circ B$ deep within that context, so long as it occurs separated by a comma from Δ, and we are able to show that Δ entails A, and that C is entailed by the context obtained when the subtree $\Delta, A -\!\circ B$ that we started with is replaced by B.

$$\frac{}{A \vdash_{BI} A}\, hyp \qquad \frac{\Gamma(\Delta;\Delta) \vdash_{BI} C}{\Gamma(\Delta) \vdash_{BI} C}\, cont \qquad \frac{\Gamma(\Delta) \vdash_{BI} C}{\Gamma(\Delta;\Delta') \vdash_{BI} C}\, weak$$

$$\frac{\Gamma, A \vdash_{BI} B}{\Gamma \vdash_{BI} A -\!\circ B} -\!\circ R \qquad \frac{\Delta \vdash_{BI} A \qquad \Gamma(B) \vdash_{BI} C}{\Gamma(\Delta, A -\!\circ B) \vdash_{BI} C} -\!\circ L$$

$$\frac{\Gamma; A \vdash_{BI} B}{\Gamma \vdash_{BI} A \rightharpoonup B} \rightharpoonup R \qquad \frac{\Delta \vdash_{BI} A \qquad \Gamma(B) \vdash_{BI} C}{\Gamma(\Delta; A \rightharpoonup B) \vdash_{BI} C} \rightharpoonup L$$

$$\frac{\Gamma \vdash_{BI} A_1 \qquad \Gamma \vdash_{BI} A_2}{\Gamma \vdash_{BI} A_1 \wedge A_2} \wedge R \qquad \frac{\Gamma(A_1;A_2) \vdash_{BI} C}{\Gamma(A_1 \wedge A_2) \vdash_{BI} C} \wedge L$$

$$\frac{}{\Gamma \vdash_{BI} \top} \top R \qquad \frac{\Gamma(1_a) \vdash_{BI} C}{\Gamma(\top) \vdash_{BI} C} \top L$$

The rule *hyp* allows a hypothesis to be used to reach a conclusion of the same proposition. Like linear logic, hypotheses can only be used when the surrounding context is empty. The rules *cont* and *weak* establish how semicolon is different from comma: if we take the perspective of the proof search process, where time flows from the conclusion of inference rules towards the premise, we may say that it is by separating two copies of an existing context with a semicolon that we are allowed to perform contraction, and it is extra baggage Δ' attached somewhere deep in the context with a semicolon that we are allowed to weaken away. The rest of the rules give the meaning of the individual logical connectives.

Bunched logic satisfies a principle of cut admissibility, which means that the following rule is admissible:

$$\frac{\Gamma \vdash A \qquad \Delta(A) \vdash C}{\Delta(\Gamma) \vdash C}\ cut$$

That is, whenever there is a proof of $\Gamma \vdash A$ and $\Delta(A) \vdash C$, there is also a proof of $\Delta(\Gamma) \vdash C$. The cut rule is not actually a rule of the system — the fact that it is admissible as a rule means that any proof using it can be rewritten as a proof not using it.

4.2.2 Encoding

To map this logic into HLF, we can define a translation $(\text{—})^\dagger$ as follows:

$$\begin{array}{rcl}
(A \multimap B)^\dagger & = & \Pi\alpha : \mathsf{w}.{\downarrow}\beta.A^\dagger \mathbin{@} \alpha \to B^\dagger \mathbin{@} (\alpha * \beta) \\
(A \mathrel{-\!\!\circ} B)^\dagger & = & {\downarrow}\beta.A^\dagger \mathbin{@} \beta \to B^\dagger \\
(A \wedge B)^\dagger & = & A^\dagger \wedge B^\dagger \\
\top^\dagger & = & \top \\
\mathsf{a}^\dagger & = & \mathsf{a}
\end{array}$$

The multiplicative implication $\multimap$, exactly like the encoding of the linear arrow above, grabs the current world β and adds to it a fresh world α, locating the new hypothesis A at that α. The additive arrow instead simply grabs the current world β and puts its hypothesis at β. Its conclusion will naturally be at β as well without any need for $@$, due to the typing rule for the HLF function space. The additive binary and nullary conjunction in BI map directly onto the same conjunctions in HLF. Propositional atoms a are translated as base types — for each a that might appear in a BI sequent, we suppose that we have added $\mathsf{a} : \mathsf{type}$ to the HLF signature Σ.

The goal of the next few sections is thus to show that the proposition A has a proof in BI if and only if the HLF type $A^\dagger$ is inhabited. In the sequel when it is unambiguous we will drop for convenience the explicit translation $(\text{—})^\dagger$ and simply consider $\multimap$, $\mathrel{-\!\!\circ}$, etc. to be *abbreviations* for the content of their translation.

The reason the proof works is that the HLF connectives are able to accurately capture the meaning of the resource discipline of bunched logic: A BI multiplicative implication $A \mathrel{-\!\!*} B$ (which we are here writing $\multimap$ as already noted) is can be used for a proof of A that uses resources disjoint from B. This is captured by the quantification over a world variable, and the requirement that the proof of the body of the implication must be at a world that is the combination of this fresh variable with the existing set of resources. A BI additive implication $A \mathrel{-\!\!\circ} B$ instead allows for *sharing*: its argument A uses the *same* resources as the body, and this is captured by binding the current world, and specifying those same resources for the argument.

4.2.3 Labellings

The proof works by meaans of an auxiliary data structure that contains both the structural information of a bunched context, and the label information of an HLF context:

$$\text{Labellings} \quad L ::= 1_m \mid 1_a \mid (L, L) \mid (L; L) \mid x : A[p]$$

The function of these labellings is to connect BI's notion of sharing to HLF's notion of equality of worlds, and BI's notion of freshness and multiplicative apartness of resources to HLF's notion of resource combination.

Labellings are identified up to the same of associative, commutative, and unit equalities that bunched contexts are required to respect. Variable names x may be duplicated across additive context joins, i.e. those marked with a semicolon, but not across multiplicative joins, those marked with a comma.

The well-formedness of labellings is given by the relation $\mapsto$, which takes in a labelling and emits an HLF world that describes the resources that correspond to using the entire context. It is defined by

$$\frac{}{1_m \mapsto \varepsilon} \qquad \frac{}{1_a \mapsto p} \qquad \frac{L_1 \mapsto p_1 \qquad L_2 \mapsto p_2}{(L_1, L_2) \mapsto p_1 * p_2} \qquad \frac{L_1 \mapsto p \qquad L_2 \mapsto p}{(L_1; L_2) \mapsto p} \qquad \frac{}{(x : A[p]) \mapsto p}$$

Because of the rule for 1_a, the relation is not functional; the additive unit can be considered valid at any world. Note that a labelling that is a pair of ;-separated subtrees that correspond to different resources, for example $(x : A[p]; (x : A[p], y : B[q]))$, falls outside the domain of this partial function and fails to be related to any HLF world.

We can define two functions $\mathbf{h}, \mathbf{b}$ that project out from a labelling the HLF and BI contexts that correspond to it.

$$\begin{aligned}
\mathbf{h}(1_m) &:= \cdot & \mathbf{b}(1_m) &:= 1_m \\
\mathbf{h}(1_a) &:= \cdot & \mathbf{b}(1_a) &:= 1_a \\
\mathbf{h}(L_1, L_2) &:= \mathbf{h}(L_1), \mathbf{h}(L_2) & \mathbf{b}(L_1, L_2) &:= \mathbf{b}(L_1), \mathbf{b}(L_2) \\
\mathbf{h}(L_1; L_2) &:= \mathbf{h}(L_1) \cup \mathbf{h}(L_2) & \mathbf{b}(L_1; L_2) &:= \mathbf{b}(L_1); \mathbf{b}(L_2) \\
\mathbf{h}(x : A[p]) &:= p : \mathsf{w}, x : A \mathbin{@} p & \mathbf{b}(x : A[p]) &:= A
\end{aligned}$$

where by $p : \mathsf{w}$ we mean $\alpha_1 : \mathsf{w}, \ldots, \alpha_n : \mathsf{w}$, assuming $\{\alpha_1, \ldots, \alpha_n\}$ is the set of world variables appearing in p.

4.2.4 Completeness

In this section we show that the embedding is complete: every bunched logic proof can be mapped to an HLF term.

Observe first of all that sublabellings of well-formed labellings must also be well-formed.

Lemma 4.2.1 *If $L_1(L_2) \mapsto p$, then there exists q such that $L_2 \mapsto q$.*

Throughout the proof of completeness, we make frequent use also of the fact that we can swap one sublabelling of a labelling for another as long as they are well-formed at the same world, and have the well-formedness of the overall labelling be preserved.

Lemma 4.2.2 *If $L_1(L_2) \mapsto q$, and $L_2 \mapsto p$ and $L_2' \mapsto p$, then $L_1(L_2') \mapsto q$.*

The main result is as follows.

Proposition 4.2.3 (Completeness) *If $L \mapsto p$ and $\mathbf{b}(L) \vdash_{BI} A$, then there exists M such that $\mathbf{h}(L) \vdash M \Leftarrow A[p]$.*

Proof By induction on the derivation of $\mathbf{b}(L) \vdash_{BI} A$.

Case:

$$\frac{}{A \vdash_{BI} A}$$

Since $\mathbf{b}(L) = A$ and $L \mapsto p$, we know L is of the form $x : A[p]$, and so $\mathbf{h}(L) = p : \mathsf{w}, x : A \mathbin{@} p$. It follows from Lemma 3.2.14 that there is a term, namely $\mathsf{ex}_{A^-}(x)$, such that $\mathbf{h}(L) \vdash \mathsf{ex}_{A^-}(x) \Leftarrow A \mathbin{@} p[\varepsilon]$. By inversion on the typing rules, we must have had also $\mathbf{h}(L) \vdash \mathsf{ex}_{A^-}(x) \Leftarrow A[p]$, as required.

Case:

$$\frac{\Gamma(\Delta;\Delta) \vdash_{BI} C}{\Gamma(\Delta) \vdash_{BI} C}$$

We are given a labelling L such that $L \mapsto p$ and $\mathbf{b}(L) = \Gamma(\Delta)$. Hence L is of the form $L_1(L_2)$ for $\mathbf{b}(L_1) = \Gamma$ and $\mathbf{b}(L_2) = \Delta$. It is easy to then check that $L_1(L_2;L_2) \mapsto p$, and $\mathbf{b}(L_1(L_2;L_2)) = \Gamma(\Delta;\Delta)$, and $\mathbf{h}(L_1(L_2;L_2)) = \mathbf{h}(L_1(L_2))$. Therefore the induction hypothesis yields $\mathbf{h}(L_1(L_2)) \vdash M \Leftarrow A[p]$, as required.

Case:

$$\frac{\Gamma(\Delta) \vdash_{BI} C}{\Gamma(\Delta;\Delta') \vdash_{BI} C}$$

We are given a labelling L such that $L \mapsto p$ and $\mathbf{b}(L) = \Gamma(\Delta)$. Hence L is of the form $L_1(L_2;L_3)$ for $\mathbf{b}(L_1) = \Gamma$ and $\mathbf{b}(L_2) = \Delta$ and $\mathbf{b}(L_3) = \Delta'$. It is easy to then check that $L_1(L_2) \mapsto p$, and $\mathbf{b}(L_1(L_2)) = \Gamma(\Delta)$, and $\mathbf{h}(L_1(L_2))$ is a subset of the hypotheses in $\mathbf{h}(L_1(L_2;L_3))$. The induction hypothesis yields $\mathbf{h}(L_1(L_2)) \vdash M \Leftarrow A[p]$, and we can take advantage of the weakening property of HLF itself to obtain $\mathbf{h}(L_1(L_2;L_3)) \vdash M \Leftarrow A[p]$ as required.

Case:

$$\frac{\Gamma, A \vdash_{BI} B}{\Gamma \vdash_{BI} A \multimap B}$$

We are given a labelling L such that $L \mapsto p$ and $\mathbf{b}(L) = \Gamma$. Let L' be $L, x : A[\alpha]$ for a fresh x and α, and observe that $L' \mapsto p * \alpha$ and $\mathbf{b}(L') = \Gamma, A$ and $\mathbf{h}(L') = \mathbf{h}(L), \alpha : \mathsf{w}, x : A \mathbin{@} \alpha$. The induction hypothesis gives M such that $\mathbf{h}(L), \alpha : \mathsf{w}, x : A \mathbin{@} \alpha \vdash M \Leftarrow B[p * \alpha]$. By rule, $\mathbf{h}(L) \vdash \lambda\alpha.\lambda x.M \Leftarrow A \multimap B[p]$, as required.

Case:

$$\frac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Delta \vdash_{BI} A \quad \Gamma(B) \vdash_{BI} C}{\Gamma(\Delta, A \multimap B) \vdash_{BI} C}$$

We are given a labelling L such that $L \mapsto r$ and $\mathbf{b}(L) = \Gamma(\Delta, A \multimap B)$. Hence L is of the form $L_1(L_2, x : A \multimap B[p])$ such that $\mathbf{b}(L_1) = \Gamma$ and $\mathbf{b}(L_2) = \Delta$. Let q be such

that $L_2 \mapsto q$ by Lemma 4.2.1. By induction hypothesis on L_2 and $\mathcal{D}_1$, we obtain a term N such that $\mathbf{h}(L_2) \vdash N \Leftarrow A[q]$.
Let L_3 be the labelling $L_1(y : B[p * q])$ and observe that $L_3 \mapsto r$ and $\mathbf{b}(L_3) = \Gamma(B)$ and $\mathbf{h}(L_3) = \mathbf{h}(L_1), y : B \mathbin{@} (p * q)$. By the induction hypothesis on L_3 and $\mathcal{D}_2$, we obtain a term M such that $\mathbf{h}(L_1), y : B \mathbin{@} (p * q) \vdash M : C[r]$.
Now we can use the identity and substitution theorems for HLF to see that the term we seek is $M' = \{\mathsf{ex}_{B^-}(x \cdot (N))/y\}^{B^-} M$, which satisfies

$$\mathbf{h}(L_1), \mathbf{h}(L_2), x : A \multimap B \vdash M' \Leftarrow C[r]$$

Case:

$$\frac{\Gamma; A \vdash_{BI} B}{\Gamma \vdash_{BI} A \rightharpoonup B}$$

We are given a labelling L such that $L \mapsto p$ and $\mathbf{b}(L) = \Gamma$. Let L' be $L; x : A[p]$ for a fresh x, and observe that $L' \mapsto p$ and $\mathbf{b}(L') = \Gamma; A$ and $\mathbf{h}(L') = \mathbf{h}(L), x : A \mathbin{@} p$. The induction hypothesis gives M such that $\mathbf{h}(L), x : A \mathbin{@} p \vdash M \Leftarrow B[p]$. By rule, $\mathbf{h}(L) \vdash \lambda x.M \Leftarrow A \rightharpoonup B[p]$, as required.

Case:

$$\frac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Delta \vdash_{BI} A \quad \Gamma(B) \vdash_{BI} C}{\Gamma(\Delta; A \rightharpoonup B) \vdash_{BI} C}$$

We are given a labelling L such that $L \mapsto r$ and $\mathbf{b}(L) = \Gamma(\Delta; A \rightharpoonup B)$. Hence L is of the form $L_1(L_2; x : A \rightharpoonup B[p])$ such that $\mathbf{b}(L_1) = \Gamma$ and $\mathbf{b}(L_2) = \Delta$. Let q be such that $L_2 \mapsto q$ by Lemma 4.2.1. By induction hypothesis on L_2 and $\mathcal{D}_1$, we obtain a term N such that $\mathbf{h}(L_2) \vdash N \Leftarrow A[q]$.
Let L_3 be the labelling $L_1(y : B[p])$ and observe that $L_3 \mapsto r$ and $\mathbf{b}(L_3) = \Gamma(B)$ and $\mathbf{h}(L_3) = \mathbf{h}(L_1), y : B \mathbin{@} p$. By the induction hypothesis on L_3 and $\mathcal{D}_2$, we obtain a term M such that $\mathbf{h}(L_1), y : B \mathbin{@} p \vdash M : C[r]$.
Now we can use the identity and substitution theorems for HLF to see that the term we seek is $M' = \{\mathsf{ex}_{B^-}(x \cdot (N))/y\}^{B^-} M$, which satisfies

$$\mathbf{h}(L_1), \mathbf{h}(L_2), x : A \rightharpoonup B \vdash M' \Leftarrow C[r]$$

Case:

$$\frac{\Gamma \vdash_{BI} A_1 \qquad \Gamma \vdash_{BI} A_2}{\Gamma \vdash_{BI} A_1 \wedge A_2}$$

We have a labelling L such that $\Gamma = \mathbf{b}(L)$ and $L \mapsto p$. By induction hypothesis, there are M_1, M_2 such that $\mathbf{h}(L) \vdash M_i \Leftarrow A_i[p]$ for $i \in \{1, 2\}$. By rule, $\mathbf{h}(L) \vdash \langle M_1, M_2 \rangle \Leftarrow A_1 \wedge A_2[p]$.

Case:

$$\frac{\Gamma(A_1; A_2) \vdash_{BI} C}{\Gamma(A_1 \wedge A_2) \vdash_{BI} C}$$

We have a labelling L such that $\Gamma(A_1 \wedge A_2) = \mathbf{b}(L)$ and $L \mapsto p$. Hence L is of the form $L_0(x : A_1 \wedge A_2[q])$. But then consider $L' = L_0(x_1 : A_1[q]; x_2 : A_2[q])$ and observe $L' \mapsto p$ and $\mathbf{b}(L') = \Gamma(A_1; A_2)$ and $\mathbf{h}(L') = \mathbf{h}(L_0), x_1 : A_1 \mathbin{@} q, x_2 : A_2 \mathbin{@} q$.
By induction hypothesis, there is M such that $\mathbf{h}(L_0), x_1 : A_1 \mathbin{@} q, x_2 : A_2 \mathbin{@} q \vdash M \Leftarrow C[p]$. Let $M' = \{\mathsf{ex}_{A_1^-}(x \cdot (\pi_1))/x_1\}\{\mathsf{ex}_{A_2^-}(x \cdot (\pi_2))/x_2\}M$ and observe that

$$\mathbf{h}(L_0), x : (A_1 \wedge A_2) \mathbin{@} q \vdash M' \Leftarrow C[p]$$

Case:

$$\frac{}{\Gamma \vdash_{BI} \top}$$

By rule, $\mathbf{h}(L) \vdash \langle\rangle \Leftarrow \top[p]$.

Case:

$$\frac{\Gamma(1_a) \vdash_{BI} C}{\Gamma(\top) \vdash_{BI} C}$$

We have a labelling L such that $\Gamma(\top) = \mathbf{b}(L)$ and $L \mapsto p$. Hence L is of the form $L_0(x : \top[q])$. We can construct $L' = L_0(1_a)$ which also has the property that $L' \mapsto p$ and $\mathbf{b}(L') = \Gamma(1_a)$ and $\mathbf{h}(L') = \mathbf{h}(L_0)$. By induction hypothesis, there is M such that $\mathbf{h}(L_0) \vdash M \Leftarrow C[p]$. By the weakening property of HLF, we also have $\mathbf{h}(L_0), x : \top \mathbin{@} q \vdash M \Leftarrow C[p]$.

■

4.2.5 Soundness

In this section we show the converse soundness of the embedding: every HLF term of a type that arises from the translation of a BI proposition can in fact be mapped back to a BI proof.

For this direction we need a stronger induction hypothesis on the structure of contexts. A labelling L is said to be *simple* when every HLF type in it is in the image of the translation $(\text{—})^\dagger$ above, and also we can derive $L : \mathsf{simple}$ with the following rules.

$$\frac{}{1_m : \mathsf{simple}} \qquad \frac{L : \mathsf{simple} \qquad \alpha, x \notin L}{(L, x : A[\alpha]) : \mathsf{simple}} \qquad \frac{L : \mathsf{simple} \qquad L \mapsto p}{(L; x : A[p]) : \mathsf{simple}}$$

A *simple world* is a world expression $\alpha_1 * \cdots * \alpha_n$ where all of the α_i are distinct. We define a partial operation of *restriction*, written $L\lfloor_p$ that takes a simple labelling L and a simple world p and yields a simple labelling. It is defined by the following clauses.

$$\begin{array}{rll}
1_m\lfloor_\varepsilon = 1_m & & \\
(L, x : A[\alpha])\lfloor_{p*\alpha} = (L\lfloor_p), x : A[\alpha] & & \\
(L, x : A[\alpha])\lfloor_p = (L\lfloor_p) & & \alpha \notin p \\
(L; x : A[q])\lfloor_p = (L\lfloor_p); x : A[q] & & q \leq p \\
(L; x : A[q])\lfloor_p = (L\lfloor_p) & & q \not\leq p
\end{array}$$

where $q \leq p$ means that there exists r such that $q * r \equiv_{\mathsf{acu}} p$.

There are two important lemmas toward soundness. One relates the restriction to a product of worlds to the multiplicative conjunction of their respective restrictions.

Lemma 4.2.4 *Suppose L is a simple labelling. If $\mathbf{b}(L\lfloor_p), \mathbf{b}(L\lfloor_q) \vdash_{BI} A$, then $\mathbf{b}(L\lfloor_{p*q}) \vdash_{BI} A$.*

Proof By induction on the derivation of simplicity of L. The important observation is that the only BI hypotheses that are in $\mathbf{b}(L\lfloor_{p*q})$ and not in $\mathbf{b}(L\lfloor_p), \mathbf{b}(L\lfloor_q)$ are additively conjoined, and can be weakened away. ■

The other concerns variable appearances in simple labellings.

Lemma 4.2.5 *If L is a simple labelling and $x : A[p] \in L$, then $\mathbf{b}(L\lfloor_p) \vdash_{BI} A$.*

Proof By induction on the derivation of L's simplicity.

Case:

$$\frac{L_0 : \mathsf{simple} \qquad L_0 \mapsto p}{(L_0; x : A[p]) : \mathsf{simple}}$$

Construct the derivation

$$\frac{\dfrac{}{A \vdash_{BI} A}\, hyp}{\mathbf{b}(L_0\lfloor_p); A \vdash_{BI} A}\, weak$$

Case:

$$\frac{L_0 : \mathsf{simple} \qquad L_0 \mapsto q}{(L_0; y : B[q]) : \mathsf{simple}}$$

with $x \neq y$. Assume without loss that $q \leq p$, for if not, then it vanishes from the restriction and we can immediately apply the induction hypothesis. Construct the derivation

$$\frac{\dfrac{}{\mathbf{b}(L_0\lfloor_p) \vdash_{BI} A}\, i.h.}{\mathbf{b}(L_0\lfloor_p); B \vdash_{BI} A}\, weak$$

Case:

$$\frac{L_0 : \mathsf{simple} \qquad \alpha, x \notin L}{(L_0, x : A[\alpha]) : \mathsf{simple}}$$

Observe that $L_0\lfloor_\varepsilon = 1_m$ and so we are immediately done, by use of *hyp*.

Case:

$$\frac{L_0 : \mathsf{simple} \qquad \alpha, y \notin L_0}{(L_0, y : B[\alpha]) : \mathsf{simple}}$$

for $x \neq y$. Observe that $\alpha \notin p$, because $\alpha \notin L_0$ and p is found in L_0 by assumption. So y is discarded by restriction, and we can apply the induction hypothesis.

■

Now we can show that the embedding of negative BI into HLF is sound.

Proposition 4.2.6 (Soundness) *Suppose types A and C are in the image of the translation $(\text{—})^\dagger$, that L is a simple labelling, and that p, r are simple worlds.*

1. *If* $\mathbf{h}(L) \vdash M \Leftarrow A[p]$, *then* $\mathbf{b}(L\lfloor_p) \vdash_{BI} A$.
2. *If* $\mathbf{h}(L) \vdash S \Leftarrow A[p] > C[r]$ *and* $\mathbf{b}(L\lfloor_p) \vdash_{BI} A$, *then* $\mathbf{b}(L\lfloor_r) \vdash_{BI} C$.

Proof By induction on the typing derivation. The cases for the additive pair and unit are straightforward. The remaining cases are left and right rules for $\rightharpoonup$ and $\multimap$, the variable case, and the nil spine case.

Case:

$$\dfrac{x : A \mathbin{@} p \in \mathbf{h}(L) \quad \dfrac{\mathcal{D}}{\mathbf{h}(L) \vdash S : A[p] > C[r]}}{\mathbf{h}(L) \vdash S : A \mathbin{@} p[\varepsilon] > C[r]} \\ \hline \mathbf{h}(L) \vdash x \cdot S \Leftarrow C[r]$$

From $x : A \mathbin{@} p \in \mathbf{h}(L)$ we know that $x : A[p] \in L$, which by Lemma 4.2.5 gives $\mathbf{b}(L\lfloor_p) \vdash_{BI} A$. By the induction hypothesis on $\mathcal{D}$, we conclude $\mathbf{b}(L\lfloor_r) \vdash_{BI} C$, as required.

Case:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}}{\mathbf{h}(L), \alpha : \mathsf{w}, x : A \mathbin{@} \alpha \vdash M \Leftarrow B[\alpha * p]}}{\mathbf{h}(L), \alpha : \mathsf{w}, x : A \mathbin{@} \alpha \vdash M \Leftarrow B \mathbin{@} (\alpha * p)[p]}}{\mathbf{h}(L), \alpha : \mathsf{w} \vdash \lambda x.M \Leftarrow A \mathbin{@} \alpha \to B \mathbin{@} (\alpha * p)[p]}}{\mathbf{h}(L), \alpha : \mathsf{w} \vdash \lambda x.M \Leftarrow \downarrow\beta.A \mathbin{@} \alpha \to B \mathbin{@} (\alpha * \beta)[p]}}{\mathbf{h}(L) \vdash \lambda\alpha.\lambda x.M \Leftarrow \Pi\alpha : \mathsf{w}.\downarrow\beta.A \mathbin{@} \alpha \to B \mathbin{@} (\alpha * \beta)[p]} \\ \hline \mathbf{h}(L) \vdash \lambda\alpha.\lambda x.M \Leftarrow A \multimap B[p]$$

Because $(\mathbf{h}(L), x : A \mathbin{@} \alpha) = \mathbf{h}(L, x : A[\alpha])$ and $(L, x : A[\alpha])\lfloor_{p\alpha} = (L\lfloor_p), x : A[\alpha]$, the induction hypothesis can be applied to $\mathcal{D}$, yielding $\mathbf{b}(L\lfloor_p), A \vdash_{BI} B$. By rule, $\mathbf{b}(L\lfloor_p) \vdash_{BI} A \multimap B$, as required.

Case:

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{\mathbf{h}(L), x : A \mathbin{@} p \vdash M \Leftarrow B[p]}}{\mathbf{h}(L) \vdash \lambda x.M \Leftarrow A \mathbin{@} p \to B[p]}}{\mathbf{h}(L) \vdash \lambda x.M \Leftarrow A \rightharpoonup B[p]}$$

Because $(\mathbf{h}(L), x : A \mathbin{@} p) = \mathbf{h}(L), x : A[p]$ and $(L, x : A[\beta])\lfloor_p = (L\lfloor_p), x : A[p]$, the induction hypothesis can be applied to $\mathcal{D}$, yielding $\mathbf{b}(L\lfloor_p); A \vdash_{BI} B$. By rule, $\mathbf{b}(L\lfloor_p) \vdash_{BI} A \rightharpoonup B$, as required.

Case:

$$\dfrac{\mathbf{h}(L) \vdash q \Leftarrow \mathsf{w} \qquad \mathbf{h}(L) \vdash M \Leftarrow A[q] \quad \mathbf{h}(L) \vdash B[p * q] > C[r]}{\mathbf{h}(L) \vdash (q; M; S) : A \multimap B[p] > C[r]}$$

By the induction hypothesis on $\mathcal{D}_1$, we find $\mathbf{b}(L\lfloor_q) \vdash_{BI} A$, and by assumption we have $\mathbf{b}(L\lfloor_p) \vdash_{BI} A \multimap B$. Our aim is to show $\mathbf{b}(L\lfloor_r) \vdash_{BI} C$. By induction hypothesis

on $\mathcal{D}_2$, it suffices to show $\mathbf{b}(L\lfloor_{p*q}) \vdash_{BI} B$. But we can construct a derivation

$$\dfrac{\mathbf{b}(L\lfloor_p) \vdash_{BI} A \multimap B \qquad \dfrac{\mathbf{b}(L\lfloor_q) \vdash_{BI} A \quad \dfrac{}{B \vdash_{BI} B}\,hyp}{A \multimap B, \mathbf{b}(L\lfloor_q) \vdash_{BI} B}\,{\multimap}L}{\mathbf{b}(L\lfloor_p), \mathbf{b}(L\lfloor_q) \vdash_{BI} B}\,cut$$

and Lemma 4.2.4 says that this is sufficient. Here we are using the cut principle noted above to get the final derivation.

Case:

$$\dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\mathbf{h}(L) \vdash M \Leftarrow A[p] \quad \mathbf{h}(L) \vdash B[p] > C[r]}$$
$$\mathbf{h}(L) \vdash (M; S) : A \rightharpoonup B[p] > C[r]$$

By the induction hypothesis on $\mathcal{D}_1$, we find $\mathbf{b}(L\lfloor_p) \vdash_{BI} A$, and by assumption we have $\mathbf{b}(L\lfloor_p) \vdash_{BI} A \rightharpoonup B$. Our aim is to show $\mathbf{b}(L\lfloor_r) \vdash_{BI} C$. By induction hypothesis on $\mathcal{D}_2$, it suffices to show $\mathbf{b}(L\lfloor_p) \vdash_{BI} B$. But we can construct a derivation

$$\dfrac{\mathbf{b}(L\lfloor_p) \vdash_{BI} A \rightharpoonup B \qquad \dfrac{\mathbf{b}(L\lfloor_p) \vdash_{BI} A \quad \dfrac{}{B \vdash_{BI} B}\,hyp}{A \rightharpoonup B; \mathbf{b}(L\lfloor_p) \vdash_{BI} B}\,{\rightharpoonup}L}{\dfrac{\mathbf{b}(L\lfloor_p); \mathbf{b}(L\lfloor_p) \vdash_{BI} B}{\mathbf{b}(L\lfloor_p) \vdash_{BI} B}\,cont}$$

Case:

$$\dfrac{}{\mathbf{h}(L) \vdash () : A[p] > A[p]}$$

Immediate.

■

4.3 Elaboration

It will become useful for the description of further algorithms for HLF (in particular while describing unification) to reduce the number of different type operators in the type system, and focus on only a fragment of the full language. We will do so, however, in a way that does not fundamentally decrease the language's expressive power. Moreover, the trouble we have already gone to establishing the essential coherence of the full language shows that the constructors about to be eliminated, although they are extremely useful as syntactic sugar, are not *merely* ad hoc syntactic sugar. They make logical and type-theoretic sense in their own right, but it happens to be possible to translate them away into a simpler system.

The first targets are @ and ↓, which are convenient for defining other connectives, such as ⊸, in a *modular* way, but they are eliminable if we are willing to elaborate an entire type

expression at once. We now describe a translation from HLF into itself which eliminates uses of the @ and ↓. This can be thought of as proceeding in two stages. First observe that any use of these connectives can be pushed down to base types by use of identities such as

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

where '≡' means that the types have precisely the same set of terms inhabiting them. Then observe that every base type, which then has the form $(a \cdot S) \mathbin{@} p$, can be replaced by a type family $a \cdot (p; S)$ that simply incorporates the world as an extra argument, so long as we appropriately change the kind of a declared in the signature. This suggestive argument is made explicit by the development below.

Having done this, all typing judgments now uniformly take place at the empty world ε — we have internalized all reasoning about worlds into mere argument-passing in type families. The extra position in the judgment that tracks the world can be discarded, since now it never varies. We have arrived at a variant of HLF which lacks @ and ↓, but which retains $\Pi\alpha : \mathsf{w}$, and so it essentially is just LF with an additional special type, w, which is subject to $\equiv_{\mathsf{acu}}$, and a notion of product types. The latter, which are now entirely orthogonal to the substructural apparatus of the system, can be eliminated by currying, as described in Section 4.3.2. As an aside, negative occurrences of products could have been curried away *before* hybrid operators were compiled, since $A\&B \multimap C$ can be expressed as $\Pi\alpha{:}\mathsf{w}.\downarrow\beta.A \mathbin{@} \alpha \to B \mathbin{@} \alpha \to C \mathbin{@} (\alpha * \beta)$.

4.3.1 Elaborating Hybrid Operators

In this section we show how to elaborate away the hybrid type operators ↓ and @ by adding an extra world argument to type families in the signature. We define a function $A^{(p)}$ ('elaborate A at world p') that takes a type A and a world p and yields a type.

$$\begin{aligned}
(\Pi x{:}A.B)^{(p)} &= \Pi x{:}(A^{(\varepsilon)}).(B^{(p)}) \\
(\Pi\alpha{:}\mathsf{w}.B)^{(p)} &= \Pi\alpha{:}\mathsf{w}.(B^{(p)}) \\
(a \cdot S)^{(p)} &= a \cdot (p; S) \\
(\downarrow\alpha.B)^{(p)} &= (\{p/\alpha\}^{\mathsf{w}}B)^{(p)} \\
(A \mathbin{@} q)^{(p)} &= A^{(q)} \\
(A \mathbin{\&} B)^{(p)} &= A^{(p)} \mathbin{\&} B^{(p)} \\
(\top)^{(p)} &= \top
\end{aligned}$$

We extend this function to contexts $\Gamma^{(\varepsilon)}$, signatures $\Sigma^{(\varepsilon)}$, and kinds $K^{(\varepsilon)}$ by

$$\begin{aligned}
(\Gamma, x : A)^{(\varepsilon)} &= \Gamma^{(\varepsilon)}, x : A^{(\varepsilon)} \\
(\Gamma, \alpha : \mathsf{w})^{(\varepsilon)} &= \Gamma^{(\varepsilon)}, \alpha : \mathsf{w} \\
(\Sigma, c : A)^{(\varepsilon)} &= \Sigma^{(\varepsilon)}, c : A^{(\varepsilon)} \\
(\Sigma, a : K)^{(\varepsilon)} &= \Sigma^{(\varepsilon)}, a : \Pi\alpha{:}\mathsf{w}.(K^{(\varepsilon)}) \\
(\cdot)^{(\varepsilon)} &= \cdot \\
(\Pi x{:}A.K)^{(\varepsilon)} &= \Pi x{:}(A^{(\varepsilon)}).(K^{(\varepsilon)}) \\
(\Pi\alpha{:}\mathsf{w}.K)^{(\varepsilon)} &= \Pi\alpha{:}\mathsf{w}.(K^{(\varepsilon)}) \\
(\mathsf{type})^{(\varepsilon)} &= \mathsf{type}
\end{aligned}$$

All translations of contexts, signatures, and types are required to notionally take place at the world ε, (although we could have just as well used a different, less suggestive syntax for their translation) because hypotheses, signature declarations, and types are all intrinsically unrestricted. The correctness of the elaboration process is given by the following result.

Theorem 4.3.1 (Hybrid Elaboration)

1. $\Gamma^{(\varepsilon)} \vdash_{\Sigma(\varepsilon)} M \Leftarrow A^{(p)}[\varepsilon]$ *iff* $\Gamma \vdash_\Sigma M \Leftarrow A[p]$
2. $\Gamma^{(\varepsilon)} \vdash_{\Sigma(\varepsilon)} R \Rightarrow A^{(p)}[\varepsilon]$ *iff* $\Gamma \vdash_\Sigma R \Rightarrow A[p]$
3. $\Gamma^{(\varepsilon)} \vdash_{\Sigma(\varepsilon)} S : A^{(p)}[\varepsilon] > B^{(q)}[\varepsilon]$ *iff* $\Gamma \vdash_\Sigma S : A[p] > B[q]$
4. $\Gamma^{(\varepsilon)} \vdash_{\Sigma(\varepsilon)} S : K^{(\varepsilon)} > \mathsf{type}$ *iff* $\Gamma \vdash_\Sigma S : K > \mathsf{type}$
5. $\Gamma^{(\varepsilon)} \vdash_{\Sigma(\varepsilon)} A^{(\varepsilon)} : \mathsf{type}$ *iff* $\Gamma \vdash_\Sigma A : \mathsf{type}$
6. $\Gamma^{(\varepsilon)} \vdash_{\Sigma(\varepsilon)} K^{(\varepsilon)} : \mathsf{kind}$ *iff* $\Gamma \vdash_\Sigma K : \mathsf{kind}$

Proof By induction on the typing derivation. We give some representative cases.

Case: Part 1, with $M = \lambda x.M_0$ and $A = \Pi x{:}A_1.A_2$. By inversion, we are comparing derivations

$$\frac{\Gamma^{(\varepsilon)}, x : A_1^{(\varepsilon)} \vdash M_0 \Leftarrow A_2^{(p)}[\varepsilon]}{\Gamma^{(\varepsilon)} \vdash \lambda x.M_0 \Leftarrow (\Pi x{:}A_1.A_2)^{(p)}[\varepsilon]} \quad \Leftrightarrow \quad \frac{\Gamma, x : A_1 \vdash M_0 \Leftarrow A_2[p]}{\Gamma \vdash \lambda x.M_0 \Leftarrow \Pi x{:}A_1.A_2[p]}$$

We can get both implications by appeal to the induction hypothesis.

Case: Part 2, comparing the derivations

$$\frac{x : A^{(\varepsilon)} \in \Gamma^{(\varepsilon)} \quad \Gamma \vdash S : A^{(\varepsilon)}[\varepsilon] > C^{(r)}[\varepsilon]}{\Gamma^{(\varepsilon)} \vdash x \cdot S \Rightarrow C^{(r)}[\varepsilon]} \Leftrightarrow \frac{x : A \in \Gamma \quad \Gamma \vdash S : A[\varepsilon] > C[r]}{\Gamma \vdash x \cdot S \Rightarrow C[r]}$$

But this case follows by appearl to the induction hypothesis, part 3.

Case: Part 3, with $A = \downarrow\alpha.A_0$. Compare the following:

$$\frac{\Gamma^{(\varepsilon)} \vdash S : (\{p/\alpha\}A_0)^{(p)}[\varepsilon] > B^{(q)}[\varepsilon]}{\Gamma^{(\varepsilon)} \vdash S : (\downarrow\alpha.A_0)^{(p)}[\varepsilon] > B^{(q)}[\varepsilon]} \quad \Leftrightarrow \quad \frac{\Gamma \vdash S : \{p/\alpha\}A_0[p] > B[q]}{\Gamma \vdash S : \downarrow\alpha.A_0[p] > B[q]}$$

The derivation on the left is not even a real inference step; it just unpacks the definition of $(\downarrow\alpha.A_0)^{(p)}$.

Case: Part 3, with $A = A_0 \mathbin{@} r$. Compare the following:

$$\frac{\Gamma^{(\varepsilon)} \vdash S : A_0^{(r)}[\varepsilon] > B^{(q)}[\varepsilon]}{\Gamma^{(\varepsilon)} \vdash S : (A_0 \mathbin{@} r)^{(p)}[\varepsilon] > B^{(q)}[\varepsilon]} \quad \Leftrightarrow \quad \frac{\Gamma \vdash S : A[r] > B[q]}{\Gamma \vdash S : A_0 \mathbin{@} r[p] > B[q]}$$

The derivation on the left is not even a real inference step; it just unpacks the definition of $(A_0 \mathbin{@} r)^{(p)}$.

Case: Part 3, with $S = ()$. Observe that all of the following are equivalent:

- $(a \cdot S) \mathbin{@} p = (a \cdot S') \mathbin{@} p'$
- $S = S'$ and $p \equiv_{\mathsf{acu}} p'$
- $a \cdot (p; S) = a \cdot (p'; S')$

Case: Part 1, at the synthesis-checking boundary. Compare the following derivations:

$$\frac{\Gamma^{(\varepsilon)} \vdash R \Rightarrow a' \cdot (p'; S')[\varepsilon] \qquad a \cdot (p; S) = a' \cdot (p'; S') \qquad \varepsilon \equiv_{\mathsf{acu}} \varepsilon}{\Gamma^{(\varepsilon)} \vdash R \Leftarrow a \cdot (p; S)[\varepsilon]}$$

$$\Leftrightarrow$$

$$\frac{\Gamma \vdash R \Rightarrow (a' \cdot S')[p'] \qquad a \cdot S = a' \cdot S' \qquad p \equiv_{\mathsf{acu}} p'}{\Gamma \vdash R \Leftarrow (a \cdot S)[p]}$$

■

In the image of the translation above, the world part of the judgment remains constantly $[\varepsilon]$. Therefore in the sequel, we sometimes write simply

$$\Gamma \vdash M \Leftarrow A$$

instead of

$$\Gamma \vdash M \Leftarrow A[\varepsilon]$$

A clarifying note about the effect of this translation on the type level: one might be concerned about the fact that now various types that were different before elaboration are now the same. For instance, both $(\downarrow \alpha.o \mathbin{@} (\alpha * \alpha))@p$ and $o \mathbin{@} (p * p)$ both elaborate to $o \cdot (p * p)$, assuming o had kind type before elaboration, and therefore has type $\mathsf{w} \to \mathsf{type}$ afterwards.

Might it not be the case that some M was ill-typed prior to elaboration because equality between these two types failed, and *is* well-typed after elaboration, because the images of the two types are equal? The answer is no, and the reason it is — that is, the reason the above theorem holds — is essentially that typechecking of terms is only sensitive to equality of *base* types, and the worlds at which they are checked, and the translation is manifestly injective on the pair of the base type and world that it receives as input.

4.3.2 Elaborating Products

The goal is now to show that also products types ($\&$ and $\top$) can be eliminated, by a currying translation. Compared to elimination of hybrid operators, the translation is slightly more involved, because it requires changing terms as well as types. For this reason, it will be advantageous for the fact of isomorphism of the signature before and after translation to be shown by a more piece-by-piece argument. We will want to be able substitute suitably constructed *type functions* for declared type families in the signature, so that we can freely swap around curried and uncurried versions of pieces of signatures and see that such swapping operations are bijections.

Type Functions

In the history of work with LF and related systems, type-level functions — function expressions that compute a type as their output — have variously been included or left out. For our present purposes, they prove to be a convenient tool, especially if we restrict attention to those that return a base type (i.e. being of the form $\lambda x.\cdots\lambda x_n.b$) rather than allowing more general type functions of the form $\lambda x.\cdots A$. Since only base types will be substituted for type constants in the signature, techniques such as the on-the-fly η-expansion used by Nanevski, Morrisett and Birkedal [NMB06] to cope with more general polymorphism are not required.

Therefore we introduce

$$\text{Normal Type Expressions} \quad F ::= \lambda x.F \mid b$$

with the kinding judgment $\Gamma \vdash F \Leftarrow K$ defined by rules

$$\frac{\Gamma, x : A \vdash F \Leftarrow K}{\Gamma \vdash \lambda x.F \Leftarrow \Pi x{:}A.K} \qquad \frac{a : K \in \Sigma \qquad \Gamma \vdash S : K > \mathsf{type}}{\Gamma \vdash a \cdot S \Leftarrow \mathsf{type}}$$

and simple typing $\gamma \vdash F \Leftarrow \tau$ by

$$\frac{\gamma, x : \tau \vdash F \Leftarrow \tau'}{\gamma \vdash \lambda x.F \Leftarrow \tau \to \tau'} \qquad \frac{a : K \in \Sigma \qquad \gamma \vdash S : \tau > \bullet}{\gamma \vdash a \cdot S \Leftarrow \bullet}$$

Substitutions $\{F/a\}^\tau$ (where we presume $\gamma \vdash F \Leftarrow \tau$) can be carried out by recapitulating the definition of term substitutions on types, kinds, contexts, and signatures, except when we come to a base type instead say

$$\{F/a_1\}^\tau(a_2 \cdot S) = \begin{cases} [F \mid S]^\tau & \text{if } a_1 = a_2 \\ a_2 \cdot S & \text{if } a_1 \neq a_2 \end{cases}$$

Additionally, term substitutions are extended to apply to type functions F in the evident way. Reduction $[F \mid S]^\tau$ is defined by

$$\begin{aligned} [\lambda x.F \mid (N; S)]^{\tau_1 \to \tau_2} &= [\{N/x\}^{\tau_1} F \mid S]^{\tau_2} \\ [b \mid ()]^\bullet &= b \end{aligned}$$

Signature Substitution and Identity

By the same methods as above we get substitution and identity results for type functions, and for ordinary term constants in the signature.

Theorem 4.3.2 (Type Function Substitution) *Suppose* $\vdash_\Sigma F \Leftarrow K$. *Let* σ *be an abbreviation for the substitution* $\{F/a\}^{K^-}$, *with* a *being a type family not named in* Σ. *Let* $\Sigma_1 = \Sigma, a : K, \Sigma'$ *for some* Σ', *and let* $\Sigma_2 = \Sigma, \sigma\Sigma'$.

1. *If* $\Gamma \vdash_{\Sigma_1} M \Leftarrow A$, *then* $\sigma\Gamma \vdash_{\Sigma_2} M \Leftarrow \sigma A$.
2. *If* $\Gamma \vdash_{\Sigma_1} R \Rightarrow A$, *then* $\sigma\Gamma \vdash_{\Sigma_2} R \Rightarrow \sigma A$.
3. *If* $\Gamma \vdash_{\Sigma_1} S : A > C$, *then* $\sigma\Gamma \vdash_{\Sigma_2} S : \sigma A > \sigma C$.
4. *If* $\Gamma \vdash_{\Sigma_1} S : K > \mathsf{type}$, *then* $\sigma\Gamma \vdash_{\Sigma_2} S : \sigma K > \mathsf{type}$.
5. *If* $\Gamma \vdash_{\Sigma_1} A : \mathsf{type}$, *then* $\sigma\Gamma \vdash_{\Sigma_2} \sigma A : \mathsf{type}$.
6. *If* $\Gamma \vdash_{\Sigma_1} K : \mathsf{kind}$, *then* $\sigma\Gamma \vdash_{\Sigma_2} \sigma K : \mathsf{kind}$.
7. *If* $\vdash_{\Sigma_1} \Gamma : \mathsf{ctx}$, *then* $\vdash_{\Sigma_2} \sigma\Gamma : \mathsf{ctx}$.

Theorem 4.3.3 (Type Function Identity) *If* $a : K \in \Sigma$, *then* $\vdash \mathsf{ex}_{K^-}(a \cdot ()) \Leftarrow K$.

Theorem 4.3.4 (Constant Substitution) *Suppose* $\vdash_\Sigma M \Leftarrow A$. *Let* σ *be an abbreviation for the substitution* $\{M/c\}^{A^-}$, *with* c *being a type family not named in* Σ. *Let* $\Sigma_1 = \Sigma, c : A, \Sigma'$ *for some* Σ', *and let* $\Sigma_2 = \Sigma, \sigma\Sigma'$.

1. *If* $\Gamma \vdash_{\Sigma_1} M \Leftarrow A$, *then* $\sigma\Gamma \vdash_{\Sigma_2} \sigma M \Leftarrow \sigma A$.
2. *If* $\Gamma \vdash_{\Sigma_1} R \Rightarrow A$, *then* $\sigma\Gamma \vdash_{\Sigma_2} \sigma R \Rightarrow \sigma A$.
3. *If* $\Gamma \vdash_{\Sigma_1} S : A > C$, *then* $\sigma\Gamma \vdash_{\Sigma_2} \sigma S : \sigma A > \sigma C$.
4. *If* $\Gamma \vdash_{\Sigma_1} S : K > \mathsf{type}$, *then* $\sigma\Gamma \vdash_{\Sigma_2} \sigma S : \sigma K > \mathsf{type}$.
5. *If* $\Gamma \vdash_{\Sigma_1} A : \mathsf{type}$, *then* $\sigma\Gamma \vdash_{\Sigma_2} \sigma A : \mathsf{type}$.
6. *If* $\Gamma \vdash_{\Sigma_1} K : \mathsf{kind}$, *then* $\sigma\Gamma \vdash_{\Sigma_2} \sigma K : \mathsf{kind}$.
7. *If* $\vdash_{\Sigma_1} \Gamma : \mathsf{ctx}$, *then* $\vdash_{\Sigma_2} \sigma\Gamma : \mathsf{ctx}$.

Theorem 4.3.5 (Constant Identity) *If* $c : A \in \Sigma$, *then* $\vdash \mathsf{ex}_{A^-}(c \cdot ()) \Leftarrow A$.

Currying Operations

Now we are ready to begin describing the actual translation. To translate a signature, we will show that it is suitably isomorphic to a signature that does not use product types, and the translation is then the constructive content of the fact of isomorphism.

Specifically, we say that two types A_0 and A_1 are isomorphic, written $A_0 \cong A_1$, when there exist terms M_0, M_1 such that

1. $x_i : A_i \vdash M_j \Leftarrow A_j$
2. $\{M_i/x_i\}^{A_i^-} M_j = \mathsf{ex}_{A_j^-}(x_j \cdot ())$

for both $i \in \{0, 1\}$ with $j = 1 - i$. From a category-theoretic perspective, M_0, M_1 are just the appropriate pair of morphisms going back and forth between objects A_0 and A_1, which compose in both directions to identity morphisms. Similarly two kinds K_1 and K_2 are isomorphic, written $K_0 \cong K_1$, when there exist two type functions F_0, F_1, such that

1. $\vdash_{\Sigma, a_i : K_i} F_j \Leftarrow K_j$

2. $\{F_i/a_i\}^{K_i^-} F_j = \mathsf{ex}_{K_j^-}(a_i \cdot ())$

for both $i \in \{0,1\}$ with $j = 1 - i$.

Then we can easily establish many particular isomorphisms.

Lemma 4.3.6

1. $\Pi x{:}A.(B \mathbin{\&} C) \cong (\Pi x{:}A.B) \mathbin{\&} (\Pi x{:}A.C)$
2. $\Pi \alpha{:}\mathsf{w}.(B \mathbin{\&} C) \cong (\Pi \alpha{:}\mathsf{w}.B) \mathbin{\&} (\Pi \alpha{:}\mathsf{w}.C)$
3. $\Pi z{:}(A \mathbin{\&} B).C \cong \Pi x{:}A.\Pi y{:}B.\{\langle x, y\rangle/z\}^{A^- \& B^-} C$
4. $\Pi x{:}A.\top \cong \top$
5. $\Pi \alpha{:}\mathsf{w}.\top \cong \top$
6. $\Pi x{:}\top.C \cong \{\langle\rangle/x\}^{\top} C$
7. $\Pi z{:}(A \mathbin{\&} B).K \cong \Pi x{:}A.\Pi y{:}B.\{\langle x, y\rangle/z\}^{A^- \& B^-} K$
8. $\Pi x{:}\top.K \cong \{\langle\rangle/x\}^{\top} K$

Proof By straightforward construction of witness terms, and checking that they constitute an isomorphism. Following the types, one is scarcely able to guess any but the correct term. For example, ignoring η-expansion (which can and must be done, strictly speaking, but it clutters up the presentation) case 3 uses the terms

$$w : \Pi z{:}(A \mathbin{\&} B).C \vdash \lambda x.\lambda y.w \cdot (\langle x, y\rangle) \Leftarrow \Pi x{:}A.\Pi y{:}B.\{\langle x, y\rangle/z\}^{A^- \& B^-} C$$

$$u : \Pi x{:}A.\Pi y{:}B.\{\langle x, y\rangle/z\}^{A^- \& B^-} C \vdash \lambda z.u \cdot (z \cdot (\pi_1); z \cdot (\pi_2)) \Leftarrow \Pi z{:}(A \mathbin{\&} B).C$$

Use Lemma 3.2.11 to see that these compose correctly. ∎

We can also compose isomorphisms in the expected ways.

Lemma 4.3.7 *The following rules are admissible.*

$$\frac{A \cong A' \qquad B \cong B'}{\Pi x{:}A.B \cong \Pi x{:}A'.B'} \qquad \frac{B \cong B'}{\Pi \alpha{:}\mathsf{w}.B \cong \Pi \alpha{:}\mathsf{w}.B'} \qquad \frac{}{A \cong A} \qquad \frac{A \cong A'}{A' \cong A} \qquad \frac{A \cong A' \qquad A' \cong A''}{A \cong A''}$$

$$\frac{A \cong A' \qquad K \cong K'}{\Pi x{:}A.K \cong \Pi x{:}A'.K'} \qquad \frac{K \cong K'}{\Pi \alpha{:}\mathsf{w}.K \cong \Pi \alpha{:}\mathsf{w}.K'} \qquad \frac{}{K \cong K} \qquad \frac{K \cong K'}{K' \cong K} \qquad \frac{K \cong K' \qquad K' \cong K''}{K \cong K''}$$

Proof By construction of witness terms. ∎

We define *organic* ('additive-free') types and kinds, and weakly organic types, by the following grammar. A weakly organic type is allowed to have additives, but only at the top level — it is a finite conjunction of a collection of organic types.

$$\begin{array}{rr} \text{Organic Kinds} & \overline{K} ::= \Pi x{:}\overline{A}.\overline{K} \mid \Pi \alpha{:}\mathsf{w}.\overline{K} \mid \mathsf{type} \\ \text{Organic Types} & \overline{A}, \overline{B} ::= \Pi x{:}\overline{A}.\overline{B} \mid \Pi \alpha{:}\mathsf{w}.\overline{B} \mid b \\ \text{Weakly Organic Types} & P ::= \top \mid \overline{B} \mathbin{\&} P \end{array}$$

Lemma 4.3.8 *For any well-formed type A, there exists a type A' such that $A \cong A'$ and A' is weakly organic. For any well-formed kind K, there exists a kind K' such that $K \cong K'$ and K' is organic.*

Proof By induction on A or K as appropriate. For an example of how the reasoning goes, consider $A = \Pi x{:}A_1.A_2$. By the induction hypothesis, we get $P_i \cong A_i$ for both $i \in \{1, 2\}$. Suppose that $P_i = \overline{B}_{i1} \,\&\, \cdots \,\&\, \overline{B}_{in_i}$. Then

$$\begin{aligned}\Pi x{:}A_1.A_2 &\cong \Pi x{:}P_1.P_2 \\ &\cong \Pi x{:}\overline{B}_{11}.\ldots\Pi x{:}\overline{B}_{1n_1}.P_2 \\ &\cong (\Pi x{:}\overline{B}_{11}.\ldots\Pi x{:}\overline{B}_{1n_1}.\overline{B}_{21}) \,\&\, \cdots \,\&\, (\Pi x{:}\overline{B}_{11}.\ldots\Pi x{:}\overline{B}_{1n_1}.\overline{B}_{2n_2})\end{aligned}$$

which by inspection is weakly organic. ■

We can continue carrying out this program at the top-level signature level, taking constants $c : A$ and type families $a : K$ and replacing them first (in the typed constant case) with weakly organic versions of themselves, then splitting if necessary into many constant declarations. In this way we see that additive product types add no essential expressivity to the type theory, and we can content ourselves in principle with reasoning about systems that lack them.

4.4 Related Work

4.4.1 Type Functions

The original definition of LF [HHP93] included a notion of type-level λ abstraction, while the early logical-relations based canonical forms existence proofs of Harper and Pfenning [HP01, HP05] were easier to describe without them. Vanderwaart and Crary later showed [VC02] that the Harper-Pfenning proof could be extended to cope with them.

On the one hand it seems rather natural given the existence of pi-kinds $\Pi x{:}A.K$ to imagine inhabitants of them other than declared type family constants, but on the other, the practical utility of including them in the basic theoretical framework[2] has been traditionally viewed with some skepticism. Indeed Geuvers and Barendsen [GB99] showed that, strictly speaking, type functions are unnecessary for any encoding whose intended meaning is phrased in terms of canonical inhabitants of *types*, which is to say that type functions cannot infect the canonical forms of terms, a fact that is more or less straightforwardly observed in the present style where canonical forms are syntactically enforced.

4.4.2 Use-Counting Linear Functions

We may observe that the representation of $\multimap$ as

$$\Pi\alpha{:}\mathsf{w}.{\downarrow}\beta.(B_1 \mathbin{@} \alpha) \to (B_2 \mathbin{@} (\beta * \alpha))$$

[2]Despite the fact that type-level *abbreviations* are used with some frequency in implementations.

has a straightforward generalization to repeated occurrences of α on the right. That is, we may say

$$B_1 \multimap_n B_2 = \Pi\alpha{:}\mathsf{w}.{\downarrow}\beta.(B_1 \mathbin{@} \alpha) \to (B_2 \mathbin{@} (\beta * \overbrace{\alpha * \cdots * \alpha}^{n \text{ times}}))$$

to achieve a function space that requires its argument to be used *exactly* n times. This notion of n-use linear functions as a generalization of linear logic has been explored in its own right by Wright [WBF93] and Boudol [Bou93].

4.4.3 Semantic Approaches

On the semantic side, the use of a commutative monoid to model resource combination in substructural logics is thoroughly well-established. In classical linear logic, one need look no farther than Girard's phase semantics [Gir95].

The situation is similar for the semantics of BI, the logic of bunched implications [OP99, Pym99, Pym02]. Work by Pym, O'Hearn, and Yang [POY04], and Galmiche and Méry [GM03], investigates an 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 $x : A \mathbin{@} \alpha$.

Chapter 5

Reasoning about HLF

5.1 Unification

Unification is the task of solving a set of equations with variables standing for unknowns, which is to say, determining whether there is a substitution for those variables that leaves every equation true, and (as is often desired) if so finding a most general such substitution. The variables for which we seek substitutions are called *metavariables* to distinguish them from the other variables that may appear, such as λ-bound variables. Unification is a problem of widespread utility in programming languages and logical frameworks: it is used for type inference and reconstruction algorithms, for the execution of programs in logic programming style, and for reasoning about the behavior of such programs.

Type theories in the LF family, with features like higher-order function types and dependent types permit powerful representation techniques, but the design of unification algorithms for such languages is more complicated. Even restricting attention to the simply-typed λ-calculus, it is known that full higher-order unification is undecidable. This is perhaps to be expected: higher-order unification means asking whether equations can be solved by instantiating unknowns that are arbitrarily higher-order functions. The character of such problems is rather different from the first-order unification encountered in the operational semantics of languages such as Prolog.

However, searching for tractable subsets of the higher-order unification problem has proved fruitful. Higher-order matching, where one side of each equation is required to have no metavariables, was found to be decidable by Stirling [Sti06]. Another particularly well-behaved subset for our purposes arises from the notion of higher-order *pattern* [Mil91] identified by Dale Miller. The so-called *pattern fragment* requires that metavariables only appear, when they are of function type, applied to a sequence of distinct λ-bound variables. This restriction makes unification decidable and even guarantees the existence of a most general unifier when there is a unifier at all.

The pattern fragment as such is still somewhat more restrictive than appropriate for many applications. An empirical study of the limitations of the pattern fragment and the usefulness of going beyond it can be found in Michaylov and Pfenning [MP92]. They nonetheless also observed that the majority of unification problems encountered in practice still come close to being in the pattern fragment, in a 'dynamic' way. Though a unification problem as a whole might not be in the pattern fragment, certain parts of it may still be,

and we may fruitfully try to eagerly solve these currently tractable equations, and let the information we learn from them instantiate metavariables and simplify other equations. The approach approximates the undecidable problem of general unification by describing a sound collection of heuristics that (necessarily!) will not solve all problems presented to it, but may achieve a satisfying level of partial coverage of the space of problems encountered in practice.

The contract we wish such an algorithm to satisfy is that, given a unification problem P, in a finite amount of time it will yield a solution of P, report that P has no solution, or report that P falls outside the set of problems that it can solve. So by 'coverage[1]' we mean here the set of problems the algorithm yields a useful answer for — either a solution, or the definite knowledge that there is no solution. We know that we cannot achieve complete coverage since higher-order unification is undecidable, but we aim to cover as many problems as is feasible under the constraint that the algorithm is tractable to implement, understand, and prove correct.

A *constraint simplification* algorithm along these lines was suggested by Dowek et al. [DHKP96] as an extension to their algorithm for higher-order pattern unification presented in the same paper. However, no proof was given of the extension's correctness, and, as it happens, it fails to terminate on some inputs. (a counterexample to termination is given in Section 5.1.1) The way the algorithm can be coaxed into nontermination has to do with precisely how the algorithm postpones difficult work, specifically when it is uncertain how to resolve conflicting information about how metavariables of function type can depend on their arguments.

This section describes an algorithm for constraint simplification for unification problems in HLF. It achieves termination, while retaining a reasonably high level of coverage, by using a different technique for postponing difficult equations. This technique nonetheless emerges rather naturally out of existing ideas in the study of pattern unification, as do certain aspects of its correctness proof.

A significant challenge to designing correct unification algorithms in this setting the presence of dependent types. While we would ordinarily like to assume every equation is between two objects of the same type, solving equations over dependent types makes this invariant difficult to maintain, and generally creates the possibility of equations arising between terms of different types, or which are between terms that are not well-typed at all. This is because as we compare two function applications, the earlier arguments affect the type of the later arguments, and if the former are not equal, the latter will not be of the same type. Conal Elliott [Ell90] dealt with these issues in his PhD thesis, (as did Pym [Pym90] independently at roughly the same time) but in a Huet-style pre-unification algorithm, by using a rather complex invariant that equations can be partially ordered to exhibit how the solvability of one guarantees the well-typedness of another. The arguments presented below are still not entirely trivial, but we achieve some simplification by choosing the typing invariant to be more straightforwardly that all equations are well-typed *modulo*, in a suitable sense, the solvability of all equations. In this way we need not be concerned about how the equations are ordered to see that they remain well-formed as the algorithm

[1]Not to be confused with the notion of coverage checking of logic programs described below.

executes.

Not least of all, we must do some extra work to account for the novelties in HLF above and beyond LF. As argued above, we can reduce them to merely the addition of a special type w whose objects are understood to intrinsically respect the equational theory $\equiv_{\mathsf{acu}}$. Therefore we must describe a constraint-simplification algorithm for equational unification in that theory.

The remainder of this section is organized as follows. Section 5.1.1 describes the language in which we study unification. Section 5.1.2 describes what a unification problem is, Section 5.1.3 gives the constraint simplification algorithm itself, and Section 5.1.4 gives the proof of its correctness. Section 5.1.6 describes how we extend the algorithm to HLF.

5.1.1 Language Extensions

There are three extensions we wish to describe relative to a basic LF-like background. Initially, we entirely ignore the notion of worlds introduced by HLF. Even in Section 5.1.6, we are able to escape direct treatment the hybrid operators $\downarrow$ and $@$ by assuming the elaboration process described in Section 4.3 has already taken place.

First of all is the addition of features from the *contextual modal* type theory to handle the typing of metavariables for unification. Next we introduce a notion of 'placeholder' term which serves to compensate somewhat for the loss of coverage resulting from removing an unsafe (with regard to termination) unification step proposed by [DHKP96]. Finally, we add a class of *free variables* closely related to the hypotheses in contextual modal type theory, which are in some sense *modal* without being *contextual*. It seems likely that free variables as such are not strictly necessary for the theory to internally work, but prove to be a convenient tool at various points in the development to come, especially when describing how unification is used in practice in, for example, type reconstruction.

Contextual Modal Type Theory

The *contextual modal type theory* described by Pientka et al. [NPP05] provides a convenient and logically motivated language with which to describe the behavior of metavariables. It has two salient logical features: one, that it possesses a notion of hypotheses of a modal judgment of *categorical* truth (the traditional notion of 'necessarily' true in alethic modal logic) corresponding to the fact that in higher-order unification we are generally interested in *closed* instantiations for metavariables. Two, these categorical modal hypotheses are 'necessarily' true *with respect to a local context* of assumptions — they are 'contextual'. This feature makes it easy to describe in logical terms the common implementation practice of *lowering* function variables to base type by changing function application to a notion of substitution, for substitutions are effectively nothing other than maps from one context — the local context of a metavariable — to another — the context in which the containing expression is typed.

The extensions we make to the syntax of the language to accommodate contextual modal features are as follows. We note that these extensions are made not because the

system without them is fundamentally inadequate, but to arrive at a description of the unification algorithm that more perspicuous to an implementer's point of view.

$$\begin{aligned}
\text{Substitutions } \sigma &::= \cdot \mid (y/x).\sigma \mid (M/x)^{\tau}.\sigma \\
\text{Modal Contexts } \Delta &::= \cdot \mid \Delta, u :: (\Gamma \vdash b) \\
\text{Modal Substitutions } \theta &::= \cdot \mid \theta, (R/u) \\
\text{Atomic Terms } R &::= \cdots \mid u[\sigma]
\end{aligned}$$

We first of all will need a notion of first-class substitution. A substitution σ contains a list of variable-for-variable substitutions (y/x) (which are instrumental in the definition of pattern substitutions below in Section 5.1.3), and simply-typed term-for-variable substitutions $(M/x)^{\tau}$. We will drop the superscript simple type on substitutions in the sequel when it is uniquely inferrable from the surrounding context.

There is then a new sort of context, a modal context Δ, which contains declarations of modal variables u, whose declared type consists of a local context Γ (we will frequently also use the letter Ψ for local contexts) and a base type b. These variables u will be used for metavariables in unification problems. The typing should be taken to mean that the valid instantiations for u are *closed* terms — except they are allowed uses of variables in Γ — which have type b. The restriction to base types is entirely benign, precisely because modal variables have local contexts. Instead of directly considering metavariables at function type, we may follow the common practice of *lowering* them to base type and representing what would have been function arguments as substitutions for their local context Γ.

For convenience we define a first-class notion θ of substitution for modal variables as well. Since all modal variables have base type, the only sorts of terms we will substitute for them are atomic terms R.

We introduce a new form of atomic term, $u[\sigma]$, a *use* of a metavariable u, under a suspended explicit substitution σ.

We update the typing judgments to include a context Δ

Context formation	$\Delta \vdash \Gamma : \mathsf{ctx}$
Type formation	$\Delta; \Gamma \vdash A : \mathsf{type}$
Type checking	$\Delta; \Gamma \vdash M \Leftarrow A$
Type synthesis	$\Delta; \Gamma \vdash R \Rightarrow A$
Spine typing	$\Delta; \Gamma \vdash S : A > C$

All typing rules are updated accordingly by inserting the Δ in every judgment uniformly. In addition, we must provide a typing rule for the new atomic term $u[\sigma]$. It depends on a typing judgment $\Delta; \Gamma \vdash \sigma : \Psi$ for substitutions σ. The rule for $u[\sigma]$ is

$$\frac{u :: (\Psi \vdash b) \in \Delta \qquad \Delta; \Gamma \vdash \sigma : \Psi}{\Delta; \Gamma \vdash u[\sigma] \Rightarrow \{\sigma\}b}$$

and the rules for typing substitutions are

$$\frac{}{\Delta; \Gamma \vdash \cdot : \cdot} \qquad \frac{\Delta; \Gamma \vdash M \Leftarrow \{\sigma\}A \qquad \Delta; \Gamma \vdash \sigma : \Psi}{\Delta; \Gamma \vdash (M/x).\sigma : (\Psi, x : A)}$$

$$\frac{y : A' \in \Gamma \qquad \Delta; \Gamma \vdash \sigma : \Psi \qquad \{\sigma\}A = A'}{\Delta; \Gamma \vdash (y/x).\sigma : (\Psi, x : A')}$$

where the notation $\{\sigma\}X$ indicates the operation of actually applying all the individual substitutions in σ to the expression X, in contrast to the σ in $u[\sigma]$, which remains inert as a suspended substitution, waiting for u to be instantiated. In general throughout this section we use {braces} to indicate substitutions acting as functions, (parentheses) to indicate parts of substitutions considered as data structures unto themselves, and [brackets] always to indicate the suspended substitution attached to a metavariable. Although the definition and typing of substitutions σ are arranged to be meaningful as simultaneous substitutions, we may reuse our former definition of single substitutions and say that $\{\sigma\}$ applies its substitutions from right to left as they appear in σ. In other words, we define

$$\{\cdot\}X = X$$
$$\{(x/y), \sigma\}X = \{x/y\}\{\sigma\}X$$
$$\{(M/x), \sigma\}X = \{M/x\}\{\sigma\}X$$

The definition of hereditary substitution must be extended to the new case of atomic terms, and therefore to substitutions.

$$\{M/x\}^\tau(u[\sigma]) = u[\{M/x\}^\tau\sigma]$$
$$\{M/x\}^\tau((x/y).\sigma) = (M/y).(\{M/x\}^\tau\sigma)$$
$$\{M/x\}^\tau((z/y).\sigma) = (z/y).(\{M/x\}^\tau\sigma) \qquad (x \neq z)$$
$$\{M/x\}^\tau((N/y).\sigma) = (\{M/x\}^\tau N/y).(\{M/x\}\sigma)$$
$$\{M/x\}^\tau(\cdot) = \cdot$$

We also must specify when *modal substitutions* are well-formed, and how they operate. These are the substitutions of closed atomic expressions for modal variables. For our purposes, we allow variables in the modal context to have types depending on one another even when the graph of dependencies has cycles. This may seem rather exotic, but it is justifiable by thinking of the entire context Δ as assigning simple types first of all, and then refining these declarations with dependent types once all the (simply-typed) variables are in scope. This approach has the advantage of eliminating the need for reasoning about reordering of the modal context, and also directly reflects the typical implementation of unification, which uses (intrinsically unordered) imperative reference cells for metavariables, whose types can indeed in practice be cyclically dependent during unification.

It is worth noting that when the algorithm succeeds and returns a set of solutions, the variables that are still free may still have cyclically dependent types. If the intended application of unification prohibits this (such as the abstraction phase of type reconstruction in a logical framework) then one can simply check for cycles and report an error if they are still present.

The typing judgment for modal substitutions is

$$\Delta' \vdash \theta : \Delta$$

and it is defined by the rule

$$\frac{\theta = (\vec{R}/\vec{u}) \qquad u_i :: (\Psi_i \vdash b_i) \in \Delta \qquad \Delta'; \{\theta\}\Psi_i \vdash R_i \Rightarrow \{\theta\}b_i \qquad (\forall i \in 1 \ldots n)}{\Delta' \vdash \theta : \Delta}$$

(where $(\vec{R}/\vec{u})$ abbreviates $(R_1/u_1) \cdots (R_n/u_n)$) which requires all terms R_i to have the type declared in Δ for u_i, after θ has been applied to it. Carrying out the substitution θ before we even know it is fully well-typed is meaningful because at least we know by prior assumption that it is simply well-typed.

The operation of modal substitution θX is defined similarly to ordinary substitution above, in that it is simply homomorphic on nearly all cases, the exception being when we come to a use of a modal variable.

We define in that case

$$\frac{(R/u) \in \theta}{\theta(u[\sigma]) = \{\theta\sigma\}R}$$

The important feature of modal substitutions is that they are only constructed out of modal, not ordinary, hypotheses; and so they represent appropriately the role of closed instances of metavariables. We want to show that modal substitutions satisfy an appropriate substitution principle, which requires a lemma about modal substitutions commuting with ordinary substitutions.

Lemma 5.1.1 *Modal and ordinary substitutions commute.*

1. $\theta\{M/x\}^\tau N = \{\theta M/x\}^\tau \theta N$
2. $\theta[M \mid S]^\tau = [\theta M \mid \theta S]^\tau$

Proof By lexicographic induction on the simple type τ, the case, (with 2 being considered less than 1) and within case 1, the size of the subject N. The two most interesting cases are as follows.

Case: $N = u[\sigma]$. Suppose $(R/u) \in \theta$. Then

$$\begin{aligned}
&\theta\{M/x\}^\tau(u[\sigma]) \\
&= \theta u[\{M/x\}^\tau\sigma] \\
&= \{\theta\{M/x\}^\tau\sigma\}R \\
&= \{\{\theta M/x\}^\tau(\theta\sigma)\}R && \text{i.h.} \\
&= \{\{\theta M/x\}^\tau(\theta\sigma)\}\{\theta M/x\}^\tau R && x \notin FV(R) \\
&= \{\theta M/x\}^\tau\{\theta\sigma\}R && \text{Substitutions Commute} \\
&= \{\theta M/x\}^\tau\theta(u[\sigma])
\end{aligned}$$

Case: $N = x \cdot S$. Then

$$\begin{aligned}
&\theta\{M/x\}^\tau(x \cdot S) \\
&= \theta[M \mid \{M/x\}^\tau S]^\tau
\end{aligned}$$

$$
\begin{aligned}
&= [\theta M \mid \theta\{M/x\}^\top S]^\top && \text{i.h.}\\
&= [\theta M \mid \{\theta M/x\}^\top \theta S]^\top && \text{i.h.}\\
&= \{\theta M/x\}^\top (x \cdot \theta S)\\
&= \{\theta M/x\}^\top \theta (x \cdot S)
\end{aligned}
$$

■

The substitution principle for modal substitutions is now the following result.

Theorem 5.1.2 *For any judgment J, (e.g., $M \Leftarrow A$, $R \Rightarrow A$, etc.) if $\Delta' \vdash \theta \Leftarrow \Delta$ and $\Delta; \Gamma \vdash J$, then $\Delta'; \theta\Gamma \vdash \theta J$.*

Proof By induction on the derivation being substituted into. The interesting case is when we reach a modal variable:

Case: Suppose the derivation is

$$\frac{u :: (\Psi \vdash b) \in \Delta \qquad \Gamma \vdash \sigma : \Psi}{\Delta; \Gamma \vdash u[\sigma] \Rightarrow \{\sigma\}b}$$

In this case what we want to show is $\Delta'; \theta\Gamma \vdash \theta(u[\sigma]) \Rightarrow \theta\{\sigma\}b$. Suppose $(R/u) \in \theta$. Then $\theta(u[\sigma]) = \{\theta\sigma\}R$. By inversion on modal substitution typing, we know $\Delta'; \theta\Psi \vdash R \Rightarrow \theta b$. By the induction hypothesis, $\Delta'; \theta\Gamma \vdash \theta\sigma : \theta\Psi$. By ordinary substitution, $\Delta'; \theta\Gamma \vdash \{\theta\sigma\}R \Rightarrow \{\theta\sigma\}\theta b$, which is equivalent to our goal, by commutativity of modal with ordinary substitutions.

■

Placeholders

We begin by describing a counterexample to the termination of the constraint-simplification algorithm described by Dowek et al. [DHKP96] The example is made to conform to our notation.

Suppose o is a base type. Let Δ consist of the metavariables u, v, w all of type $(z : o \vdash o)$. Abbreviate $u[(R/z)]$ by simply $u[R]$, and similarly for v and w. We also take the liberty of writing $x \cdot ()$ as simply x. Let f be a constant of type $o \to o$.

Without setting up unification formally yet, consider the pair of equations

$$\lambda x.\lambda y.u[x] \doteq \lambda x.\lambda y.f \cdot (v[w[y]])$$

$$\lambda x.\lambda y.v[x] \doteq \lambda x.\lambda y.u[w[y]]$$

and suppose we are trying to find solutions to u, v, w that satisfy both of them. Note incidentally that both equations are well-typed at the type $o \to o \to o$.

Examine the first equation in particular. We notice that the function $\lambda x.\lambda y.u[x]$ on the left does not use its second argument y at all. This is true independently of the instantiation of u, because u must be instantiated by a *closed* term up to the substitution $[x]$ it receives. Therefore neither can the right side of this equation (once v and w are instantiated) mention y. However, since v is applied to an expression that itself has the

variable w in it, we do not know whether v or w projects away its argument, but we know at least one of them must. By 'v projects away the argument x' we mean that the instantiation for v does not mention the variable x.

Notice that because the first equation must hold, u's instantiation has to be of the form $f \cdot M$ for some term M that possibly mentions the variable z. We might hope therefore that we are making progress by creating a new variable $u' :: (z : o \vdash o)$ and carrying out the instantiation $u \leftarrow f \cdot (u'[z])$ — in fact, this is precisely what the algorithm suggested in [DHKP96] does. But if we do, we only arrive at the pair of equations

$$(\lambda x.\lambda y.f \cdot (u'[x]) \doteq \lambda x.\lambda y.f \cdot v[w[y]]) \wedge (\lambda x.\lambda y.v[x] \doteq \lambda x.\lambda y.f \cdot u'[w[y]])$$

which, after stripping the identical constants f from the first equation, leads only to

$$(\lambda x.\lambda y.u'[x] \doteq \lambda x.\lambda y.v[w[y]]) \wedge (\lambda x.\lambda y.v[x] \doteq \lambda x.\lambda y.f \cdot u'[w[y]])$$

Swapping the two equations and changing the names of the unification variables, this is identical to the pair we started with, and the algorithm (if it continues executing similar steps) loops forever.

To fix this problem at all is quite easy: simply disallow the algorithm from carrying out instantiations like $u \leftarrow f \cdot (u'[z])$ above. However, this seems rather drastic: we do in fact appear to have a significant piece of information when faced with an equation $u \leftarrow f \cdot (M)$, and we would prefer not to waste it.

Our approach is to directly embody in the language the intuition that the occurrence of the bound variable y is something that 'cannot survive' once *both* v and w have been instantiated, for it is not in the range of the substitution $[x]$ that occurs on the left. This idea can be found implicitly in Tobias Nipkow's algorithm [Nip93] for higher-order pattern unification over simple types. He makes use of the 'deBruijn index $-\infty$' after computing inverses of substitutions to stand for variables that do not occur in the range of the substitution. Although Nipkow says it 'smells of a hack,' we aim to show that its use can be theoretically justified.

We therefore introduce an explicit placeholder, written $\_$, for an expression that occurs somewhere in an argument to a unification variable, but for which we mean to require that every complete solution will project it away:

$$\text{Normal Terms } M, N ::= \cdots \mid \_$$

Considering simple types, '$\_$' is allowed to have any simple type τ. To look at it another way, especially if we maintain the notion that every term is intrinsically simply typed, there is a copy of $\_$, call it $\_^\tau$, at every simple type τ, though we typically drop the superscript τ when it is clear.

The definitions of hereditary substitution and reduction are extended by saying

$$\{M/x\}^\tau(\_) = \_ \qquad [\_ \mid S]^\tau = \_$$

With this idea we can transform the equation $\lambda x.\lambda y.u[x] \doteq \lambda x.\lambda y.f\ v[w[y]]$ by the instantiation $u \leftarrow f \cdot (v[w[\_]])$, which leads to the original pair of equations being turned into

$$\lambda x.\lambda y.f \cdot (v[w[\_]]) \doteq \lambda x.\lambda y.f \cdot (v[w[y]])$$

$$\lambda x.\lambda y.v[x] \doteq \lambda x.\lambda y.f \cdot (v[w[\_]])$$

It will turn out that we can reason about these equations using a form of the occurs-check, and correctly reject them as unsolvable.

While we will continue to use the symbol '=' below to mean strict syntactic equality of two expressions up to α-varying bound variables (and $\equiv_{\mathsf{acu}}$ on worlds when appropriate), it will be convenient to also define the relation $\equiv$ by saying that $X \equiv X'$ if $X = X'$ and also X, X' contain no occurrences of $\_$.

Free Modal Variables

The other extension we wish to make for describing unification is a notion of variables m which are *modal* like metavariables, but unlike them they are not *contextual*, and in discussion of unification they are considered to be *not subject to instantiation*. To say that they are modal means that they are declared in the modal context Δ and are allowed to be used inside an R in a modal substitution containing (R/u), and may occur in the types of metavariables, and other free variables. To say that they are not contextual means that instead of being used under a substitution for local variables, they take a spine of arguments, just as ordinary variables do. Not being subject to instantiation means that unification will be defined in terms of seeking a substitution for (i.e. that replaces) the collection of metavariables, which excludes the set of free modal variables.

We add therefore to the syntax of heads and modal contexts the following.

$$\begin{array}{c} \text{Heads } H ::= \cdots \mid m \\ \text{Modal Contexts } \Delta ::= \cdots \mid \Delta, m :: A \end{array}$$

One role of these variables is to represent *free* variables in a unification problem whose type may involve metavariables, but which is not meant to be instantiated during the course of unification. These arise naturally from wanting to use unification for type reconstruction in a dependent type theory. For example, if we encoded $n \times p$ matrices, and the operation of matrix transposition $M^\top$ and a theorem claiming that if $M_1^\top = M_2$, then $M_2^\top = M_1$, we might write something like

$$\begin{array}{l} \mathit{matrix} : \mathit{nat} \to \mathit{nat} \to \mathsf{type} \\ \mathit{transpose} : \mathit{matrix}\ \mathsf{N}\ \mathsf{P} \to \mathit{matrix}\ \mathsf{P}\ \mathsf{N} \to \mathsf{type} \\ \mathit{thm} : \mathit{transpose}\ \mathsf{M}_1\ \mathsf{M}_2 \to \mathit{transpose}\ \mathsf{M}_2\ \mathsf{M}_1 \to \mathsf{type} \end{array}$$

in which the free variables N, P, M_1, M_2 are understood as implicitly Π-bound, and whose types are to be determined by type reconstruction via unification. Our knowledge of the *type* of M_1 can be represented as $\mathit{matrix}\ u[\cdot]\ v[\cdot]$ for metavariables $u :: (\cdot \vdash \mathit{nat}), v :: (\cdot \vdash \mathit{nat})$, for unification will determine what u and v must be, but M_1 itself is not open for instantiation, and is represented therefore as a free variable.

The reason free variables take spines instead of substitutions so that they may be conveniently compared for equality: equality on spines is, as is typical for canonical-forms-only presentations of LF, a simple matter of traversing the syntax and ensuring everything is literally identical up to α-conversion. For substitutions, because of the presence of

variable-for-variable replacement (y/x), equality is complicated by the fact that such a replacement can also be represented as a term-for-variable replacement $(\mathsf{ex}_\tau y/x)$.

It is for this reason that in general we avoid as much as possible in the sequel posing directly the question of whether two substitutions are considered equal, with one exception in Section 5.1.6.

Now the need for comparing free variables for equality in the first place is that they *not* subject to instantiation (whereas metavariables are), and therefore still exist (and appear in types) after a putative solution-substitution of a unification problem is carried out; and of course typing depends on equality of base types at the synthesis-checking boundary, which may involve equality of terms that are arguments to base types, and those terms may involve free variables.

The other role of free variables is to allow us to conveniently state the correctness of an algorithm that works (as is common) by repeated small transformations the unification problem. We wish to say that a the solutions to a unification problem are ground instantiations of all of its variables, and that all transition steps preserve that set. However, some unification problems may be trivially solvable because some of the types of their variables are uninhabited. Because of dependent types, we cannot decide inhabitation. The fact that (modal) free variables are available at every type prevents exactly this sort of trivial solvability.

5.1.2 Unification Problems

A unification problem is defined to be a pair consisting of a modal context Δ and a set P of equations in that context, written $\Delta \vdash P$, where

$$\begin{aligned} \text{Equation Sets } P &::= \top \mid P \wedge Q \\ \text{Equations } Q &::= M \doteq M' \mid R \doteq R' \mid S \doteq S' \mid u \doteq R \mid u \leftarrow R \end{aligned}$$

The intended interpretation of $\Delta \vdash P$ is the question of whether there exist instantiations of all the metavariables in Δ that satisfy the conjunction of equations in P. The 'equation' $u \leftarrow R$ indicates that we have found an instantiation for u, and that it is R. It differs from the use of $u \doteq R$, in that in the latter, u may have other occurrences in R, preventing us (for reasons of circularity relevant to the **Occurs-Check**) from carrying out an instantiation. We often write just P instead of $\Delta \vdash P$ when it is unambiguous.

The *active metavariables* of P are the metavariables in P such that there are no assignments $u \leftarrow R$ to u in P. A modal substitution $\theta = (R_1/u_1) \ldots (R_n/u_n)$ is *ground* if there are no occurrences of metavariables in the R_i. Free variables m are still allowed. As stated before, this guarantees that unification problems are never trivially unsolvable because the types of their metavariables are uninhabited, because we always have the option of making up a free variable of the same type. A *solution* to $\Delta \vdash P$ is a ground modal substitution θ for all the metavariables in Δ such that

1. For every equation $X \doteq X' \in P$ we have $\theta X \equiv \theta X'$
2. For every $u \leftarrow R \in P$ we have $(\theta R/u) \in \theta$

3. All the terms in θ have no occurrence of '_'.

We write $\theta \models P$ if θ is a solution of P.

Let $\vec{u}$ be a subset of the metavariables in P. A *$\vec{u}$-solution* to P is a ground modal substitution for $\vec{u}$ that arises as the restriction of some solution of P to the variables $\vec{u}$. We write $\theta \models_{\vec{u}} P$ in this case, and the set of all such solutions is written $\mathsf{Sol}(\Delta \vdash P, \vec{u})$.

5.1.3 Algorithm

A state of the unification algorithm is either $\Delta \vdash P$ (a set of equations in context Δ) or the constant $\bot$, standing for failure. Extend Sol to account for $\bot$ with the clause $\mathsf{Sol}(\bot, \vec{u}) = \emptyset$.

A *pattern substitution* is a substitution σ that consists of only distinct variable-for-variable substitutions and placeholders. Formally:

$$\frac{}{\vdash \cdot\ \mathsf{pat}} \qquad \frac{\vdash \sigma\ \mathsf{pat} \qquad y \notin \mathrm{rng}\,\sigma}{\vdash \sigma, (y/x)\ \mathsf{pat}} \qquad \frac{\vdash \sigma\ \mathsf{pat}}{\vdash \sigma, (\_/x)\ \mathsf{pat}}$$

A *strong pattern substitution* is a pattern substitution that has no placeholders. We use ρ to denote a pattern substitution, and ξ to denote a strong pattern substitution.

We define the following auxiliary functions: ξ_Γ^{-1} computes the inverse of a strong pattern substitution whose codomain is the context Γ. It is defined by the clauses

$$\begin{array}{rcl} \xi_\cdot^{-1} & = & \cdot \\ \xi_{\Gamma,x}^{-1} & = & \xi_\Gamma^{-1}, \begin{cases} (y/x) & \text{if } (x/y) \in \xi \\ (\_/x) & \text{otherwise.} \end{cases} \end{array}$$

The function $\xi \cap \mathrm{id}$ replaces non-identity substitutions in ξ with the placeholder _, defined as follows.

$$\begin{array}{rcll} \cdot \cap \mathrm{id} & = & \cdot & \\ (\xi, (x/x)) \cap \mathrm{id} & = & (\xi \cap \mathrm{id}), (x/x) & \\ (\xi, (x/z)) \cap \mathrm{id} & = & (\xi \cap \mathrm{id}), (\_/z) & (\text{if } x \neq z) \end{array}$$

We need two operations for pruning variables out of substitutions and out of the types assigned to modal variables, respectively. The judgment $\rho_{\backslash x} = \rho'$ is defined by

$$\frac{\mathrm{dom}(\rho) \setminus \{x\} = x_1, \ldots, x_n \qquad (\_/x) \in \rho}{\rho_{\backslash x} = (x_1/x_1) \ldots (x_n/x_n)}$$

and the judgment $(\Gamma \vdash b)_{\backslash x} = (\Gamma')$ by

$$\frac{x \notin FV(b, \Gamma')}{(\Gamma, x : A, \Gamma' \vdash b)_{\backslash x} = (\Gamma, \Gamma')}$$

The purpose of the first of these two is to take a substitution replacing a chosen variable with a placeholder, and construct out of it the identity substitution on all variables except

for the chosen one. The second constructs a new context from the context of a modal type by removing one variable from it.

For any syntactic class $\mathcal{X}$ use $\hat{\mathcal{X}}\{Y\}$ to refer to an expression of syntactic class $\mathcal{X}$ with a hole in it, where the hole has been replaced by the expression Y, which may refer to variables bound in $\mathcal{X}$. Also take $\hat{\mathcal{X}}_{rig}\{Y\}$ to refer to a *rigid* context in which Y occurs, that is, Y's occurrence is not within the arguments σ of some metavariable occurrence $u[\sigma]$. Similarly $\hat{\mathcal{X}}_{srig}\{Y\}$ refers to a *strongly rigid context* in which Y occurs, that is, not within a substitution of a metavariable, nor within an argument to a bound variable x.

The algorithm consists of repeatedly applying the following transition rules:

Decomposition

$$
\begin{array}{l}
(\lambda x.M \doteq \lambda x.M') \wedge P \mapsto (M \doteq M') \wedge P \\
(H \cdot S \doteq H \cdot S') \wedge P \mapsto (S \doteq S') \wedge P \\
(H \cdot S \doteq H' \cdot S') \wedge P \mapsto \bot \qquad (\text{if } H \neq H') \\
((M; S) \doteq (M'; S')) \wedge P \mapsto (M \doteq M') \wedge (S \doteq S') \wedge P \\
(() \doteq ()) \wedge P \mapsto P \\
\hat{Q}_{rig}\{\_\} \wedge P \mapsto \bot
\end{array}
$$

Inversion

$$(u[\xi] \doteq R) \wedge P \mapsto (u \doteq [\xi^{-1}]R) \wedge P$$

Occurs-Check

$$
\begin{array}{l}
(u \doteq H \cdot \hat{S}\{u[\xi]\}) \wedge P \mapsto (u \doteq H \cdot \hat{S}\{\_\}) \wedge P \\
(u \doteq c \cdot \hat{S}_{srig}\{u[\sigma]\}) \wedge P \mapsto \bot
\end{array}
$$

Intersection

$$(u \doteq u[\xi]) \wedge P \mapsto \begin{cases} P & \text{if } \xi \cap \mathrm{id} = \xi \\ (u \doteq u[\xi \cap \mathrm{id}]) \wedge P & \text{otherwise} \end{cases}$$

Pruning

$$\frac{\rho_{\backslash x} = \rho' \qquad (\Gamma \vdash b)_{\backslash x} = (\Gamma') \qquad v \notin \Delta \cup \{u\}}{\begin{array}{l}(\Delta, u :: (\Gamma \vdash b) \vdash \hat{Q}_{rig}\{u[\rho]\} \wedge P) \mapsto \\ \quad (\Delta, u :: (\Gamma \vdash b), v :: (\Gamma' \vdash b) \vdash (u \doteq v[\rho']) \wedge \hat{Q}_{rig}\{u[\rho]\} \wedge P)\end{array}}$$

Instantiation

$$\frac{u \notin FV(R)}{(\Delta \vdash (u \doteq R) \wedge P) \mapsto (\{R/u\}\Delta \vdash (u \leftarrow R) \wedge \{R/u\}P))}$$

The algorithm may nondeterministically choose any of these steps, with the restriction that after **Pruning**, it must immediately take an **Instantiation** step on the freshly created equation $u \doteq v[\rho']$. This **Instantiation** could have been incorporated into the definition of **Pruning**, obviating such a side condition, but it is simpler for the proofs of correctness

below to take a conceptually separate treatment the substitutions that **Instantiation** carries out. Furthermore steps that would not cause the unification state to change at all (for example performing **Intersection** twice in a row on the same equation) are forbidden.

Note that the premise on **Instantiation** that u must not appear free in R is necessarily satisfied after **Pruning** takes place, because the pruning substitution ρ' is a pattern, and v is fresh.

The algorithm reports success whenever the current state consists of assignments $u \leftarrow R$ (with no R containing a placeholder), and also then reports the modal substitution induced by the assignments. It reports failure whenever the state reaches $\bot$. It reports that the problem can neither be solved or rejected when no transitions from the current state are possible.

5.1.4 Correctness

We proceed to show that the algorithm is correct. The three facts we wish to show are that it terminates, that it preserves solutions, and that it preserves well-formedness of states of the unification algorithm.

Termination

Theorem 5.1.3 *The algorithm always terminates, resulting in one of*

- *A solved state, where P is only assignments $u \leftarrow R$, and no '$\_$' appears in any R*
- *A stuck state, i.e., one on which no transition rule applies*
- *Failure $\bot$*

Proof By consideration of the lexicographic order on

1. The number of active metavariables.
2. The total size of the local contexts of the active metavariables.
3. The total size of the terms in all the equations in the unification problem, with $\_$ considered smaller than any other term.

All transitions that change the state at all decrease this metric. Most transitions decrease (3) and maintain (1) and (2). Pruning (including the required instantiation step following it as described above in section 5.1.3) replaces one metavariable with another one of a smaller context, decreasing (2) and maintaining (1). Instantiation reduces (1). ■

Preservation of Solutions

Next we show that, as the algorithm progresses, each transition rule neither creates nor destroys solutions. First we note some basic lemmas about inversion, whose proofs are straightforward inductions working from the definitions given.

Lemma 5.1.4 *If every ordinary variable in X is in* $\operatorname{rng}\xi$*, then* $\{\xi\}\{\xi^{-1}\}X = X$.

Lemma 5.1.5 $\{\xi^{-1}\}\{\xi\}X = X$.

Corollary 5.1.6 (Injectivity of pattern substitutions) *If* $\{\xi\}X = \{\xi\}X'$*, then* $X = X'$*.*

Now we can show the main result of this subsection, that every step preserves the set of solutions. The statement of the theorem may seem peculiar, in that we cannot obviously chain together uses of the theorem across many steps, because the set of variables $\vec{u}$ may change. This is addressed by appropriately considering restrictions of solution sets to smaller sets of variables; see the proof of Lemma 5.1.9.

Theorem 5.1.7 *If* $P_0 \mapsto P_1$ *then* $\mathsf{Sol}(P_0, \vec{u}) = \mathsf{Sol}(P_1, \vec{u})$*, where* $\vec{u}$ *is the set of metavariables of* P_0*.*

Proof By case analysis on which transition rule was taken.

Case: **Decomposition**

These cases are relatively easy. For example, $\theta(\lambda x.M) = \theta(\lambda x.M')$ iff $\theta M = \theta M'$, and similarly for homomorphic decomposition of the other constructs. If a placeholder appears in a rigid position, then no substitution will eliminate it, and no $\equiv$ relation (as occurs in the definition of Sol) can hold with placeholders in it.

Case: **Inversion**

In this case $P_0 \mapsto P_1$ is $u[\xi] \doteq R \wedge P \mapsto u \doteq \{\xi^{-1}\}R \wedge P$. In one direction, suppose θ_1 is a $\vec{u}$-solution of P_1, so we have $\theta_1 u = \theta_1\{\xi^{-1}\}R$. Thus $\theta_1\{\xi^{-1}\}R = \{\xi^{-1}\}\theta_1 R$ has no placeholders, so $\theta_1 R$ must only have variables from $\operatorname{rng}\xi$. Apply ξ to both sides, and we find $\theta_1 u = \theta_1\{\xi\}u = \{\xi\}\theta_1 u = \{\xi\}\{\xi^{-1}\}\theta_1 R = \theta_1 R$. Thus θ_1 is also a $\vec{u}$-solution of P_0.

In the other direction, suppose θ_0 is a $\vec{u}$-solution of P_0, which means $\theta_0(u[\xi]) = \{\xi\}(\theta_0 u) = \theta_0 R$.

We now want to show that $\theta_0 u = \theta_0\{\xi^{-1}\}R$. It will suffice to show, by injectivity of pattern substitutions, that $\theta_0 R = \{\xi\}\theta_0\{\xi^{-1}\}R$. But by the equality $\{\xi\}(\theta_0 u) = \theta_0 R$ derived above we already know that $\theta_0 R$ is in the range of ξ, so $\{\xi\}\theta_0\{\xi^{-1}\}R = \{\xi\}\{\xi^{-1}\}\theta_0 R = \theta_0 R$.

Case: **Occurs-Check**

There are two transitions. In both cases, observe that a subterm[2] $u[\sigma]$ of R, being atomic, has one of two fates after the substitutions of a putative solution $\theta \models P_0$ are carried out: either it is completely eliminated from the term by a substitution higher up in the expression tree that discards it, or else $\{\sigma\}R'$ occurs as a subterm of θR, where $(R'/u) \in \theta$. Thus the result of carrying out a substitution on, for example, a spine-with-hole to yield $(\theta\hat{S})$ is still an expression context, but it may not be linear even if the original $\hat{S}$ was.

Consider the first transition

$$(u \doteq H \cdot \hat{S}\{u[\xi]\}) \wedge P \mapsto (u \doteq H \cdot \hat{S}\{\_\}) \wedge P$$

Let a solution θ of P_0 be given, with $(R'/u) \in \theta$. We know then that $R' \equiv H \cdot (\theta\hat{S})\{\{\xi\}R'\}$. From this we can see that $(\theta\hat{S})$ must project out its argument, for

[2]Though in one case the substitution is written ξ because we know it to be a pattern assumption, here we are using σ to generally include both cases

otherwise R' is a larger term than itself, since $\{\xi\}R'$ is the same size as R', because ξ is a pattern. Therefore there is no difference between $\theta(\hat{S}\{u[\xi]\})$ and $\hat{S}\{\_\}$, and the latter state still has θ as a solution
Consider the other transition

$$(u \doteq c \cdot \hat{S}_{srig}\{u[\sigma]\}) \wedge P \mapsto \bot$$

Again let a solution θ of P_0 be given, with $(R'/u) \in \theta$.
We know $R' = c \cdot (\theta\hat{S}_{rig})\{\{\theta\sigma\}R'\}$. We may use this equation to expand its own right side again to see

$$R' = c \cdot (\theta\hat{S}_{srig})\{c \cdot (\{\theta\sigma\}\theta\hat{S}_{rig})\{\{\theta\sigma\}\{\theta\sigma\}R'\}\}$$

Since $\hat{S}_{srig}$ is strongly rigid, no substitution can project away its argument, and we can continue telescoping this expression to infer that R' has, for every n, more than n occurrences of the constant c, a contradiction.

Case: **Intersection**
Since pattern substitutions only do renaming, any solution of P_0 must refrain from using any variables that are not fixed by ξ. Thus any solution of P_0 is still a solution of P_1.

Case: **Pruning**
Clearly any solution of the latter state is also a solution of the former. To show that no solutions are lost, consider a solution θ to P_0. It assigns some term R to u. If R has a free occurrence of x, then $\theta(u[\xi]) = \{\xi\}R$ will have a placeholder in it, because $(\_/x) \in \xi$. Since $u[\xi]$ occurs rigidly, this cannot be projected away, and we have a contradiction. Therefore there is a term without occurrence of x, which can be substituted for v in P_1: the solution of P_1 consists of all of θ, plus this additional substitution for v.

Case: **Instantiation** Assuming $u \notin FV(R)$, we need to show $\mathsf{Sol}(\Delta \vdash (u \doteq R) \wedge P)$ is the same as $\mathsf{Sol}(\{R/u\}\Delta \vdash (u \leftarrow R) \wedge \{R/u\}P)$. Consider what it means for a putative solution θ to have the property $\theta \models P_0$. The unification state P_0 is for this case $(u \doteq R) \wedge P$, so we would need that

- (R'/u) belongs to θ for some R'
- $R' \equiv \theta R$
- $\theta \models P$.

To have θ satisfy P_1, which is $(u \leftarrow R) \wedge \{R/u\}P)$, we would need

- $(\theta R/u)$ belongs to θ
- $\theta \models \{R/u\}P$.

By the commutativity of modal and ordinary substitutions. we can see that these two sets of conditions are equivalent.

■

This has as an immediate consequence that if we reach a solved state $u_1 \leftarrow R_1 \wedge \cdots \wedge u_n \leftarrow R_n$, then $\theta = (R_1/u_1) \ldots (R_n/u_n)$ is a unifier, and indeed a most general unifier of the original problem. Every solution to the original problem is an instance of θ, because it is a solution of $u_1 \leftarrow R_1 \wedge \cdots \wedge u_n \leftarrow R_n$ by the theorem.

Typing Modulo

We come to the task of defining the typing invariant for the algorithm. There are two particular challenges we should take note of: one is that we want to be able work on equations in a fairly arbitrary order despite the fact that, because of dependent types, typing of one equation depends on solvability of another, and the other is the need to reason about typing of placeholders $\_$. To handle the first issue, we define what it means to be well-typed *modulo* a set of equations P.

We say $X \equiv_P X'$ ('X is equivalent to X' modulo P') if, for any ground θ that substitutes for the metavariables of X, X' that is a solution of P, we have $\theta X \equiv \theta X'$. This equivalence is only meant to be asked of X, X' that are placeholder-free, although P may have placeholders remaining in it. The relation $\equiv_P$ is a partial equivalence relation whose support is the set of placeholder-free simply-typed expressions. For all typing judgments $\Gamma \vdash J$, we define $\Gamma \vdash_P J$ by the same rules as for $\Gamma \vdash J$, except replacing the occurrences of $=$ in them with $\equiv_P$.

Generalizing to equivalence modulo P sometimes requires more subtle care about which parts of judgments are input and output. For example, we can obtain a generalization of the substitution principle for the typing judgment $\vdash_P$ of typing modulo P, namely

Lemma 5.1.8

1. *If* $\Gamma \vdash_P M \Rightarrow A$ *and* $\Gamma, x : B, \Gamma' \vdash_P J$ *and* $A \equiv_P B$*, then* $\Gamma, \{M/x\}\Gamma' \vdash_P \{M/x\}J$.
2. *If* $\Gamma \vdash_P R \Rightarrow b$ *and* $u : (\Psi \vdash b') \in \Delta$ *and* $b \equiv_P b'$ *and* $\Psi \vdash_P J$*, then* $\{R/u\}\Delta; \{R/u\}\Gamma' \vdash_P \{R/u\}J$.

which differs most notably in that for atomic terms, we need to 'slacken' to account for a possible equivalence modulo P rather than exact equality between b and b', because the type b is an output of the process of synthesizing a type for R, and may not already be $\equiv_P$ to b'.

From the fact that unification preserves solutions comes the fact that equivalence and typing modulo P do not change when the unification algorithm acts on P.

Lemma 5.1.9 *If* $P_0 \mapsto P_1$ *and* $A \equiv_{P_0} B$*, then* $A \equiv_{P_1} B$.

Proof Let θ be given such that $\theta \models P_1$. Suppose the set of metavariables of P_0 is $\vec{u}$: it may be smaller, but not bigger, than that of P_1. Thus $\theta|_{\vec{u}} \models_{\vec{u}} P_1$. By theorem 5.1.7, also $\theta|_{\vec{u}} \models_{\vec{u}} P_0$. By assumption that $A \equiv_{P_0} B$, we have $\theta|_{\vec{u}} A = \theta|_{\vec{u}} B$, and so $\theta A = \theta B$, as required. ∎

Corollary 5.1.10 *Suppose* $P_0 \mapsto P_1$*. Then*

1. *If* $\Delta; \Gamma \vdash_{P_0} M \Leftarrow A$*, then* $\Delta; \Gamma \vdash_{P_1} M \Leftarrow A$.
2. *If* $\Delta; \Gamma \vdash_{P_0} R \Rightarrow A$*, then there is* A' *such that* $\Delta; \Gamma \vdash_{P_1} R \Rightarrow A'$ *and* $A' \equiv_{P_0} A$.
3. *If* $\Delta; \Gamma \vdash_{P_0} S : A > C$*, then there is* C' *such that* $\Delta; \Gamma \vdash_{P_1} S : A > C'$ *and* $C' \equiv_{P_0} C$.

Proof By induction on the derivation of $\Delta; \Gamma \vdash_{P_0} J$. Most cases are simple appeals to the induction hypothesis on all components. A more interesting case is when we deal with the following rule:

$$\frac{\Delta; \Gamma \vdash_{P_0} R \Rightarrow A \qquad A \equiv_{P_0} B}{\Delta; \Gamma \vdash_{P_0} R \Leftarrow B}$$

By induction hypothesis, there is an A' such that $\Delta; \Gamma \vdash_{P_1} R \Rightarrow A'$ and $A \equiv_{P_0} A'$. By transitivity, we have $A' \equiv_{P_0} B$, and by Lemma 5.1.9, $A' \equiv_{P_1} B$. So we can form a derivation

$$\frac{\Delta; \Gamma \vdash_{P_1} R \Rightarrow A' \qquad A' \equiv_{P_1} B}{\Delta; \Gamma \vdash_{P_1} R \Leftarrow B}$$

as required. ■

Well-formedness of Unification States

We need a notion of well-formedness of a unification state P that has placeholders $\_$ in it. To summarize what we are about to do, P will be considered well-formed if it is appropriately related to some P' that is placeholder-free, and which is well-typed in a more straightforward way.

Define $X' \sqsupseteq X$ (pronounced "X' is a completion of X") to mean X' arises by replacing every $\_$ in X with some normal term, a different term being allowed for each $\_$. This means that if, during unification, we take some well-formed X' and simply replace a normal subterm of it with $\_$, the resulting term X will manifestly still be considered well-formed, because its immediately prior state $X' \sqsupseteq X$ is a witness to the fact that X is suitably related to a well-typed expression. This makes reasoning about the type preservation of the occurs-check and intersection transitions quite pleasantly simple. Completions only play a role in the theory, and do not (or at least need not) appear at all in the implementation.

We can now define the judgment $\Delta \vdash_{P'} P \ \mathsf{wf}$ (resp. $\Delta \vdash_{P'} Q \ \mathsf{wf}$) that the unification problem P (resp. equation Q) is well-formed modulo P':

$$\frac{\Delta; \Gamma \vdash_{P'} M_i' \Leftarrow A \qquad M_i' \sqsupseteq M_i \qquad (\forall i \in \{1,2\})}{\Delta \vdash_{P'} M_1 \doteq M_2 \ \mathsf{wf}}$$

$$\frac{\Delta; \Gamma \vdash_{P'} R_i' \Rightarrow A_i \qquad R_i' \sqsupseteq R_i \qquad A_1 \equiv_{P'} A_2 \qquad (\forall i \in \{1,2\})}{\Delta \vdash_{P'} R_1 \doteq R_2 \ \mathsf{wf}}$$

$$\frac{\Delta; \Gamma \vdash_{P'} S_i' : A > C_i \qquad S_i' \sqsupseteq S_i \qquad C_1 \equiv_{P'} C_2 \qquad (\forall i \in \{1,2\})}{\Delta \vdash_{P'} S_1 \doteq S_2 \ \mathsf{wf}}$$

$$\frac{u :: (\Gamma \vdash b) \in \Delta \qquad \Gamma \vdash_{P'} R' \Rightarrow b' \qquad R' \sqsupseteq R \qquad b' \equiv_{P'} b}{\Delta \vdash_{P'} u \doteq R \ \mathsf{wf}}$$

$$\frac{u :: (\Gamma \vdash b) \in \Delta \qquad \Gamma \vdash_{P'} R' \Rightarrow b' \qquad R' \sqsupseteq R \qquad b' \equiv_{P'} b}{\Delta \vdash_{P'} u \leftarrow R \ \mathsf{wf}}$$

$$\frac{\Delta' \sqsupseteq \Delta \qquad \Delta' \vdash_{P'} Q \ \mathsf{wf} \qquad \Delta \vdash_{P'} P \ \mathsf{wf}}{\Delta \vdash_{P'} Q \wedge P \ \mathsf{wf}}$$

$$\frac{}{\Delta \vdash_{P'} \top \ \mathsf{wf}}$$

A clarifying note on the first three rules above: the point is that there must exist a *single* Γ (and in the term and spine rules, a single A) that works for both $i \in \{1,2\}$.

The pertinent contexts and types do not appear in the unification problem itself, but the existence of suitable types is the content of the well-formedness judgments.

We say $\Delta \vdash P\ \mathsf{wf}$ if there exists an extension $\Delta' = \Delta, m_1 :: A_1, \ldots, m_n :: A_n$ of Δ by free variables such that $\Delta' \vdash_P P\ \mathsf{wf}$.

First of all it is easy to show that taking a step in the set of equations in the 'modulo' subscript on the turnstile preserves typing.

Lemma 5.1.11 *If $P_0 \mapsto P_1$ and $\Delta \vdash_{P_0} P\ \mathsf{wf}$, then $\Delta \vdash_{P_1} P\ \mathsf{wf}$.*

Proof By induction on the derivation of $\Delta \vdash_{P_0} P\ \mathsf{wf}$, using Corollary 5.1.10 to transfer typing judgments and equivalences forward. ■

Inversion Completion

Although, as noted above, the definitions are arranged to make reasoning about the introduction of _ during the occurs-check and intersection transitions easy, it still remains to justify why inversion — the only other transition that creates placeholders — preserves types. We need to construct a completion of ξ^{-1} that is free of placeholders. Assuming $\Gamma \vdash \xi : \Gamma'$, this is defined as follows, similarly to inversion, as

$$\xi^*_{\cdot} = \cdot$$

$$\xi^*_{\Gamma, x:A} = \xi^*_\Gamma, \begin{cases} (y/x) & \text{if } (x/y) \in \xi \\ (u[\xi^*_\Gamma]/x) & \text{otherwise, for a fresh } u :: (\Gamma \vdash A). \end{cases}$$

This definition is in fact essentially identical to the standalone definition of inversion given by Dowek et al. [DHKP96] when their aim is to merely show that pattern substitutions have a one-sided inverse. The important idea is that for every new placeholder we would have created by inversion, we insert a new metavariable of the correct type, so that that the resulting expression is still well-typed, and is a completion of inversion.

Since this definition is so close to inversion, it shares many of its properties, in particular being a one-sided inverse. Most importantly, however, the substitutions it outputs are well-typed for well-typed inputs.

Lemma 5.1.12 *If every ordinary variable in X is in $\operatorname{rng}\xi$, then $\{\xi\}\{\xi^*\}X = X$.*

Lemma 5.1.13 $\{\xi^*\}\{\xi\}X = X$.

We now want to show that ξ^* itself is well-typed. First we note a fact about the way that types of individual variables behave under pattern substitution:

Lemma 5.1.14 *Suppose $\Gamma \vdash \xi : \Gamma'$. If $(x/y) \in \xi$, and $x : A \in \Gamma$, then $y : B \in \Gamma'$ such that $\{\xi\}B = A$.*

Proof By induction on the typing of ξ. If $\xi = (x/y).\xi_0$, then we read the conclusion directly off the typing

$$\frac{x : \{\xi\}B \in \Gamma \qquad \Gamma \vdash \xi_0 : \Gamma'}{\Gamma \vdash (x/y).\xi_0 : (\Gamma', y : B)}$$

Otherwise simply apply the induction hypothesis. ■

Having said that, we can now prove

Lemma 5.1.15 *If* $\Gamma \vdash_P \xi : \Gamma'$*, then* $\Gamma' \vdash_P \xi^*_\Gamma : \Gamma$.

Proof By induction on Γ. The base case is easy. Of the two non-base cases, one is when the variable, say x, does not occur in $\operatorname{rng}\xi$, and $\xi^*_{\Gamma,x:A} = (u[\xi^*_\Gamma]/x).\xi^*_\Gamma$. We can use the derivation arising from the appeal to the induction hypothesis twice to get the derivation

$$\dfrac{\dfrac{u :: \Gamma \vdash A \in \Delta \qquad \Gamma' \vdash_P \xi^*_\Gamma : \Gamma}{\Gamma' \vdash_P u[\xi^*_\Gamma] \Rightarrow \{\xi^*_\Gamma\}A} \qquad \{\xi^*_\Gamma\}A \equiv_P \{\xi^*_\Gamma\}A}{\dfrac{\Gamma' \vdash_P u[\xi^*_\Gamma] \Leftarrow \{\xi^*_\Gamma\}A \qquad \Gamma' \vdash_P \xi^*_\Gamma : \Gamma}{\Gamma' \vdash_P (u[\xi^*_\Gamma]/x).\xi^*_\Gamma : (\Gamma, x : A)}}$$

In the other case x does actually occur in the range of ξ as $(x/y) \in \xi$, and so $\xi^*_{\Gamma,x:A} = (y/x).\xi^*_\Gamma$. By the previous lemma, pick a B such that $A = \{\xi\}B$ and $y : B \in \Gamma'$. Then $y : \{\xi^*_\Gamma\}A (= \{\xi^*_\Gamma\}\{\xi\}B = B) \in \Gamma'$, which together with a use of the induction hypothesis allows the derivation

$$\dfrac{y : \{\xi^*_\Gamma\}A \in \Gamma' \qquad \Gamma' \vdash \xi^*_\Gamma : \Gamma}{\Gamma' \vdash (y/x).\xi^*_\Gamma : (\Gamma, x : A)}$$

■

Lemma 5.1.16 *If* $\Gamma \vdash_P \xi : \Gamma'$*, then* $\Gamma' \vdash_P \xi^*_\Gamma : \Gamma$.

There remain only a few lemmas before we can show the main theorem of this section. The first is technical, but easy.

Lemma 5.1.17 *If* $(M_1; S_1) \doteq (M_2; S_2) \in P_0$*, and* $M'_i \sqsupseteq M_i$ *for both* $i \in \{1, 2\}$*, then* $M'_1 \equiv_{P_0} M'_2$.

Proof Let θ be given such that $\theta \models P_0$. In particular, $(\theta M_1; \theta S_1) \equiv (\theta M_2; \theta S_2)$. But then $\theta M_1 \equiv \theta M_2$, and this equation must be placeholder-free, so $\theta M'_1 \equiv \theta M'_2$. ■

Next, we show that equivalence modulo P respects substitutions in the following sense.

Lemma 5.1.18 *Assume* $M_1 \equiv_P M_2$ *and* $X_1 \equiv_P X_2$ *and* $S_1 \equiv_P S_2$*. Then*

1. $\{M_1/x\}^\tau X_1 \equiv_P \{M_2/x\}^\tau X_2$
2. $[M_1 \mid S_1]^\tau \equiv_P [M_2 \mid S_2]^\tau$

Proof By induction on simple type τ, and subsequently the structure of the term. ■

We can transfer typing judgments across equivalence modulo P.

Lemma 5.1.19 *Suppose* $A \equiv_P A'$ *and* $\Gamma \equiv_P \Gamma'$.

1. *If* $\Gamma \vdash_P M \Leftarrow A$*, then* $\Gamma' \vdash_P M \Leftarrow A'$.
2. *If* $\Gamma \vdash_P R \Leftarrow C$*, then there exists* C' *such that* $\Gamma' \vdash_P R \Leftarrow C'$ *and* $C \equiv_P C'$.
3. *If* $\Gamma \vdash_P S \Leftarrow A > C$*, then there exists* C' *such that* $\Gamma' \vdash_P S \Leftarrow A' > C'$ *and* $C \equiv_P C'$.

Proof By induction on the typing derivation. ■

Preservation of Types

We can now show the following result, that unification preserves types as it proceeds. This can be used to see that when unification terminates successfully, by reaching a state that consists only of a conjunction of assignments $u \leftarrow R$, that all the terms in that assignment are well-typed.

Theorem 5.1.20 *If* $\Delta_0 \vdash P_0$ wf *and* $(\Delta_0 \vdash P_0) \mapsto (\Delta_1 \vdash P_1)$*, then* $\Delta_1 \vdash P_1$ wf*.*

Proof By case analysis on the possible steps. If the step is not instantiation, we get the preservation of well-formedness of all the equations we didn't touch from Corollary 5.1.10, and what needs to be checked is just that the new equations are still well-typed.

Case:

$$(\lambda x.M_1 \doteq \lambda x.M_2) \wedge P \mapsto (M_1 \doteq M_2) \wedge P$$

By assumption there exist $\Gamma, \Pi x{:}A.B, M_1' \sqsupseteq M_1, M_2' \sqsupseteq M_2$ such that

$$\Delta; \Gamma \vdash_{P_0} \lambda x.M_1' \Leftarrow \Pi x{:}A.B$$

$$\Delta; \Gamma \vdash_{P_0} \lambda x.M_2' \Leftarrow \Pi x{:}A.B$$

By inversion, we have

$$\Delta; \Gamma, x : A \vdash_{P_0} M_1' \Leftarrow B$$

$$\Delta; \Gamma, x : A \vdash_{P_0} M_2' \Leftarrow B$$

so after pushing forward with Lemma 5.1.10, we have

$$\Delta \vdash_{P_1} M_1 \doteq M_2 \text{ wf}$$

Case:

$$(M_1; S_1) \doteq (M_2; S_2) \wedge P \mapsto (M_1 \doteq M_2) \wedge (S_1 \doteq S_2) \wedge P$$

By assumption there exist $\Gamma, \Pi x{:}A.B, C_i, M_i' \sqsupseteq M_i, S_i' \sqsupseteq S_i$ such that

$$\Delta; \Gamma \vdash_{P_0} (M_i'; S_i') \Leftarrow \Pi x{:}A.B > C_i$$

and $C_1 \equiv_{P_0} C_2$. By inversion, we have

$$\Delta; \Gamma \vdash_{P_0} M_1' \Leftarrow A \qquad \Delta; \Gamma \vdash_{P_0} S_1' \Leftarrow \{M_1'/x\}B > C_1$$

$$\Delta; \Gamma \vdash_{P_0} M_2' \Leftarrow A \qquad \Delta; \Gamma \vdash_{P_0} S_2' \Leftarrow \{M_2'/x\}B > C_2$$

At this stage we observe that the spine tails S_1' and S_2' are at different types because different arguments were substituted into them. The plan is now to take advantage of P_0 to bring them back together. By Lemma 5.1.17, since $(M_1; S_1) \doteq (M_2; S_2) \in P_0$, we know $M_1' \equiv_{P_0} M_2'$. Then $\{M_1'/x\}B \equiv_{P_0} \{M_2'/x\}B$ by Lemma 5.1.18. Finally, using Lemma 5.1.19 and $\Delta; \Gamma \vdash_{P_0} S_1' \Leftarrow \{M_1'/x\}B > C_1$, we get the existence of $C_3 \equiv_{P_0} C_1$ such that

$$\Delta; \Gamma \vdash_{P_0} S_1' \Leftarrow \{M_2'/x\}B > C_3$$

Using Lemma 5.1.10, push forward all the judgments from $\vdash_{P_0}$ to $\vdash_{P_1}$ to see

$$\Delta \vdash_{P_1} S_1 \doteq S_2 \text{ wf}$$

Case:

$$u[\xi] \doteq R \wedge P \mapsto u \doteq \{\xi^{-1}\}R \wedge P$$

Say $u :: (\Gamma' \vdash b) \in \Delta$. By assumption there exist $\Gamma, A, R' \sqsupseteq R$ such that

$$\Gamma \vdash_{P_0} R' \Rightarrow A$$

$$\Gamma \vdash_{P_0} \xi : \Gamma'$$

$$\{\xi\}b \equiv_{P_0} A$$

Note that we didn't have to consider $\xi' \sqsupseteq \xi$, because since ξ is a strong pattern substitution, it has no placeholders to fill in.
By Lemma 5.1.16, we get

$$\Gamma' \vdash_{P_0} \{\xi^*\}R' \Rightarrow \{\xi^*\}A$$

$$\{\xi^*\}\{\xi\}b \equiv_{P_0} \{\xi^*\}A$$

It is easy to see that $\{\xi^*\}R' \sqsupseteq \{\xi^{-1}\}R$. But $\{\xi^*\}\{\xi\}b \equiv_{P_0} b$ by Lemma 5.1.13.

Case: $(\Delta \vdash (u \doteq R) \wedge P) \mapsto (\{R/u\}\Delta \vdash (u \leftarrow R) \wedge \{R/u\}P)$, with the side-condition that $(u \notin FV(R))$.
We have $\Delta' \sqsupseteq \Delta$, $u :: (\Gamma \vdash b) \in \Delta'$, $P' \sqsupseteq P$, $R' \sqsupseteq R$ such that $\Delta'; \Gamma \vdash_{P_0} R' \Rightarrow A$, $b \equiv_{P_0} A$. So by the modal substitution theorem for $\vdash_{P_0}$, we can see

$$\{R/u\}\Delta \vdash_{P_0} (u \leftarrow R) \wedge \{R/u\}P \ \mathsf{wf}$$

which we can push forward with Lemma 5.1.10 to get

$$\{R/u\}\Delta \vdash_{P_1} (u \leftarrow R) \wedge \{R/u\}P \ \mathsf{wf}$$

■

5.1.5 Counterexamples

In this section we collect some examples that illustrate why certain choices in the design of the algorithm above were forced, and provide evidence for why we could not have made them otherwise. Assume for all examples a signature containing at least

$$o : \mathsf{type}$$
$$a : o \to \mathsf{type}$$
$$k : o$$

Occurs-Check

The first occurs-check transition rule must be constrained to pattern substitutions ρ. Otherwise, we might consider for $u :: (z : o \to o \vdash o)$, and $g : o \to o$,

$$u[g] \doteq g \cdot (u[\lambda y.k])$$

(where for compactness of presentation we write $[M]$ instead of $[M/z]$) has solution $u \leftarrow z \cdot k$. Replacing $u[\lambda y.k]$ in the unification problem with $u[\lambda y.y]$ would also have worked as a counterexample, with the same solution.

The second occurs-check transition rule must be constrained to strongly-rigid occurrences of u, because for $u :: (z : o \to o \vdash o)$,

$$\lambda f.u[f] \doteq \lambda f.c\ (f\ (u[\lambda x.k]))$$

in fact does have a solution, namely $u \leftarrow c\ (z\ (c\ k))$.

Weak patterns

It is not generally permissible to carry out inversion on weak patterns, that is, patterns that are a mixture of distinct bound variables and placeholders rather than exclusively variables. For a modal context containing $u :: (x_1 : o, y : a\ x_1, x_2 : o, z : (a\ x_2) \to o \vdash o)$, the unification problem

$$x : o, y' : a\ x, z' : a\ x \to o \vdash u[z'.\_.y'.\_] \doteq z'y'$$

(where we are similarly leaving out the names of variables substituted for, for brevity) is well-typed (because the left-hand side completes to $u[z'.x.y.x]$), but the result of inversion assigns $u \leftarrow z\ y$, which is not.

The Appearance of Cyclic Type Dependency

An example of how cyclic dependencies among types can arise is

$$w :: o, u :: (a\ w \vdash o), v : a\ w \vdash w \leftarrow u[v] \wedge P$$

which takes a step to

$$u :: (a\ (u[v]) \vdash o), v : a\ (u[v]) \vdash P$$

And indeed this unification problem still might have solutions. For instance, if the signature had $b : \Pi x{:}o.a\ x$, then a modal substitution $(k/u)(b \cdot (k)/v)$ would be well-typed.

5.1.6 World Unification

To extend this algorithm to HLF, we need to add clauses to several definitions to accommodate worlds. Syntactically, we create a notion of modal world variable, free world variable, and extend ordinary and modal substitutions to include worlds.

$$\begin{aligned}
\text{Substitutions } \sigma &::= \cdots \mid (p/\alpha)^{\mathsf{w}}.\sigma \\
\text{Modal Contexts } \Delta &::= \cdots \mid \Delta, \upsilon :: \Psi \vdash \mathsf{w} \mid \Delta, \mu :: \mathsf{w} \\
\text{Modal Substitutions } \theta &::= \cdots \mid \theta, (p/\upsilon) \\
\text{Worlds } p &::= \cdots \mid \upsilon[\sigma] \mid \mu
\end{aligned}$$

World substitutions are typed analogously to term substitutions:

$$\frac{\Delta;\Gamma \vdash p \Leftarrow \mathsf{w} \qquad \Delta;\Gamma \vdash \sigma : \Psi}{\Delta;\Gamma \vdash (p/\alpha).\sigma : (\Psi, \alpha : \mathsf{w})}$$

and so too are world metavariables and free variables.

$$\frac{\upsilon :: \Psi \vdash \mathsf{w} \in \Delta \qquad \Delta;\Gamma \vdash \sigma : \Psi}{\Delta;\Gamma \vdash \upsilon[\sigma] : \mathsf{w}}$$

$$\frac{\mu :: \mathsf{w} \in \Delta}{\Delta;\Gamma \vdash \mu : \mathsf{w}}$$

Pattern substitutions are extended in the evident way, requiring a list of distinct world variables.

We assume that the Ψ in any modal type $\upsilon :: \Psi \vdash \mathsf{w}$ consists only of world variable declarations. The notion of equation is extended in the evident way to allow equations between worlds, and instantiations of world metavariables:

$$\text{Equations } Q ::= \cdots \mid p \doteq p' \mid \upsilon \doteq p \mid \upsilon \leftarrow p$$

We define $\alpha \in p$ to mean that the world variable appears somewhere in p. This relation is well-defined up to $\equiv_{\mathsf{acu}}$, for associativity, commutativity and unit laws neither create nor destroy appearances of parameters. The relation $\alpha \in_{\mathsf{gnd}} p$ means that the variable α appears in the *ground part* of p, that is, not within the substitution of a metavariable. This is basically the same as the previous notion of strictly rigid occurrence, but specialized for worlds.

Creation of placeholders in world expressions is avoided by a side condition in inversion. Because the occurs-check does not play the same role in an equational theory with a unit (which allows for instance $\upsilon = \upsilon * p$ to have solutions even for nontrivial p, since both υ and p might both be ε) we did not find reasoning about placeholders to be required in practice, and would prefer to avoid the complications of reasoning about direct interactions between the placeholder and the ACU theory of worlds.

The extra unification steps added to the previously described algorithm to reason about worlds are as follows.

Decomposition

$$\frac{p_1 \equiv_{\mathsf{acu}} p_1' * \alpha \qquad p_2 \equiv_{\mathsf{acu}} p_2' * \alpha}{(p_1 \doteq p_2) \wedge P \mapsto (p_1' \doteq p_2') \wedge P}$$

$$\frac{p_1 \equiv_{\mathsf{acu}} p_1' * \upsilon[\sigma] \qquad p_2 \equiv_{\mathsf{acu}} p_2' * \upsilon[\sigma]}{(p_1 \doteq p_2) \wedge P \mapsto (p_1' \doteq p_2') \wedge P}$$

$$\frac{p_1 \equiv_{\mathsf{acu}} p_1' * \upsilon[\xi] \qquad p_2 \equiv_{\mathsf{acu}} \varepsilon}{(p_1 \doteq p_2) \wedge P \mapsto (\upsilon \leftarrow \varepsilon) \wedge (p_1' \doteq \varepsilon) \wedge P}$$

$$\frac{\alpha \in_{\mathsf{gnd}} p_1 \qquad \alpha \notin p_2}{(p_1 \doteq p_2) \wedge P \mapsto \bot}$$

$$\frac{\alpha \in_{\mathsf{gnd}} p_2 \qquad \alpha \notin p_1}{(p_1 \doteq p_2) \wedge P \mapsto \bot}$$

Pruning

$$\frac{\alpha \notin p, q \qquad (\Psi \vdash \mathsf{w})_{\setminus\beta} = \Psi' \qquad \xi_{\setminus\beta} = \xi'}{\begin{array}{l}\Delta, \upsilon :: (\Psi \vdash \mathsf{w}) \vdash (\alpha * p \doteq \upsilon[\xi] * q) \wedge P \mapsto \\ \quad \Delta, \upsilon :: (\Psi \vdash \mathsf{w}), \upsilon' :: (\Psi' \vdash \mathsf{w}) \vdash (\alpha * p \doteq \upsilon[\xi] * q) \wedge (\upsilon \doteq \beta * \upsilon'[\xi']) \wedge P\end{array}}$$

Inversion

$$\frac{\_ \notin \{\xi^{-1}\}p}{(\upsilon[\xi] \doteq p) \wedge P \mapsto (\upsilon \doteq \{\xi^{-1}\}p) \wedge P}$$

Instantiation

$$\frac{\upsilon \notin FV(p)}{(\Delta \vdash (\upsilon \doteq p) \wedge P) \mapsto (\{p/\upsilon\}\Delta \vdash (\upsilon \leftarrow p) \wedge \{p/\upsilon\}P))}$$

where the substitution restriction operation on world variables in the pruning rule is defined by

$$\frac{\mathrm{dom}(\rho) \setminus \{\beta\} = \alpha_1, \ldots, \alpha_n \qquad (\alpha/\beta) \in \rho}{\rho_{\setminus\alpha} = (\alpha_1/\alpha_1) \ldots (\alpha_n/\alpha_n)}$$

We generalize the definition of $\vec{u}$-solutions as follows. For any $\vec{u}$ a subset of the ordinary term metavariables in P and $\vec{v}$ a subset of P's world metavariables, a $\vec{u}, \vec{v}$-*solution* to P is a ground modal substitution for $\vec{u}$ and $\vec{v}$ that arises as the restriction of some solution of P to the variables $\vec{u}, \vec{v}$. We write $\theta \models_{\vec{u},\vec{v}} P$ in this case, and the set of all such solutions is written $\mathsf{Sol}(P, \vec{u}, \vec{v})$.

As before, we wish to show that solutions are preserved under all unification steps. Preservation of types for the extension is simple, because all new steps uniformly take a world equation to another world equation.

Theorem 5.1.21 *If* $P_0 \mapsto P_1$ *then* $\mathsf{Sol}(P_0, \vec{u}, \vec{v}) = \mathsf{Sol}(P_1, \vec{u}, \vec{v})$*, where* $\vec{u}$ *is the set of metavariables of* P_0 *and* $\vec{v}$ *the set of its world metavariables.*

Proof Most of the **Decomposition** cases are again quite easy. In the case of

$$\frac{p_1 \equiv_{\mathsf{acu}} p_1' * \upsilon[\xi] \qquad p_2 \equiv_{\mathsf{acu}} \varepsilon}{(p_1 \doteq p_2) \wedge P \mapsto (\upsilon \leftarrow \varepsilon) \wedge (p_1' \doteq \varepsilon) \wedge P}$$

we rely on ξ being a pattern to see that the only modal substitution for υ that could produce ε is ε itself. Otherwise, $\upsilon[\xi]$ could be for instance $\upsilon[\varepsilon/x]$ and υ could be instantiated with the local variable x, and still yield ε.

If a parameter occurs in the ground part of one side and not at all on the other, clearly no substitution can reconcile the difference, and we can fail.

The **Pruning** rule is somewhat unlike the pruning rule for ordinary terms. It relies on exactly one rigid occurrence of a world variable α on one side of the equation, and exactly one occurrence on the other side, in a pattern substitution attached to a metavariable υ. By reasoning about counts of occurrences of world variables after instantiation, we can see that the instantiation of υ must use its local variable that corresponds to α exactly once, allowing us to split of (as υ') a pruned metavariable that accounts for the remainder of υ's instantiation. For the sake of termination, we make the same requirement that an instantiation step immediately follow this **Pruning** rule.

The **Inversion** step is essentially a special case of **Inversion** for ordinary terms, and the same reasoning applies *a fortiori*. **Instantiation** also follows the same reasoning as in Theorem 5.1.7. ■

Remark that absent from the above set of rules is any analogue to **Intersection**. This is because the commutativity in the equational theory on worlds makes naïvely applying such a step unsound. In the case of ordinary terms, rather than worlds, observe that **Intersection** would transform

$$u \doteq u[(y/x).(x/y)]$$

to

$$u \doteq u[\_.\_]$$

which then allows **Pruning** to instantiate u with a fresh metavariable that depends on neither of its arguments. In contrast, the world unification problem

$$\upsilon \doteq \upsilon[(\beta/\alpha).(\alpha/\beta)]$$

potentially has (and indeed, if that is the only equation in the unification problem, definitely does have) solutions that use the local variables α and β, for example $\upsilon \leftarrow \alpha * \beta$, since $\alpha * \beta \equiv_{\mathsf{acu}} \beta * \alpha$.

In this particular case, we could reason that since υ's instantiation must be invariant under permutation of its two local variables, its instantiation must be some multiple of $\alpha * \beta$, licensing a step

$$\upsilon \doteq \upsilon[(\beta/\alpha).(\alpha/\beta)] \mapsto \upsilon \leftarrow \upsilon'[\alpha * \beta]$$

for some fresh υ' with one local world variable. Although this step alone would be sound, it is plainly rather ad hoc. We leave the task of finding a suitable generalization of this sort of permutative reasoning for future work, not least because intersections do not seem to arise often enough in the examples we have looked at so far to motivate heroic measures for solving them.

5.2 Coverage Checking

Coverage checking is a central part of checking the correctness of metatheorems. Recall that a metatheorem is written in the form of its constructive content: a logic program that pattern-matches against the possible forms of its inputs, which are the premises of the statement of the metatheorem. To coverage-check a metatheorem is to determine that every possible input is covered by a clause of the proof. Or, to be more precise, this process just described is referred to as *input* coverage, since a notion of *output* coverage also exists, wherein the pattern matching against the outputs returned by calls to subgoals (in proofs, the extraction of results coming out of appeals to lemmas or to the induction hypothesis) is checked to be sufficiently general. However, in this section we will be exclusively concerned with input coverage checking.

We give a brief overview below of how Twelf input coverage checks a proof of a metatheorem. A more complete explanation can be found in [SP03]. The aim of describing it here is to provide enough background to make some sense of specific extensions to it that we propose to coverage check metatheorems in HLF. Because of the elaboration passes described in Section 4.3, coverage checking in HLF can proceed mostly in the same way as coverage checking for ordinary LF, since we can treat worlds as if they were merely expressions of type w, subject to the HLF-centric extensions to unification already described. However, this approach to coverage checking appears to be too weak to successfully coverage check the more interesting of the examples we have studied so far. We propose two additional forms of automated reasoning about coverage, described below, which make it possible to successfully determine that coverage holds for a larger class of metatheorems.

The first is a finitely-branching case split when a world-unification equation has no most general solution, but can be seen to have finitely many solutions which, taken together, are sufficiently general. A related transformation, called simply finitary splitting, is already used when coverage checking LF, as described in [SP03], but it is applied when we know that an entire type has only finitely many inhabitants, in particular (as is often useful) when we know it has zero or one. The finitary world-splitting we propose differs in that it analyzes a finite set of places in an expression where a given world variable can be used, rather than the set of inhabitants of the type of worlds more globally.

The other extension is a form of reasoning that takes advantage of a notion of *monotonicity* of resources as used in linear logic and related substructural logics. This notion of monotonicity should not be confused with the sense in which linear logic is a 'nonmonotonic logic', namely that a linear assumption, once made, is capable of being consumed and later unusable, in contrast to ordinary unrestricted hypotheses. Rather the thing that

is monotone in the present discussion is the fact that, after a resource has been consumed, it cannot be *un*consumed: the fact of its use cannot be revoked. When this property holds, as it will for types arising from translations of LLF as described, it enables a very useful *strengthening* transformation on coverage goals, in which we can under appropriate circumstances determine that a variable in the context cannot occur in a term, because, considered as a resource, that variable has already been irrevocably consumed.

We now proceed to sketch existing work on coverage and splitting in ordinary LF.

5.2.1 The Meaning of Coverage in LF

For greater clarity concerning the meaning of coverage itself without getting too mired in technical details, we initially give all definitions without the more refined concepts of free modal and metavariables introduced above, and briefly note at the end how to connect up coverage with the unification algorithm.

The goal of this subsection is to explain the concepts of coverage *goal* and coverage *clause*, both of which will take the form of a base-type-in-context $\Gamma \vdash b$. While they have the same shape, coverage goals and clauses, in particular their contexts, are thought of differently. The variables in a coverage goal represent the unknowns in a theorem that must be accounted for, and so they are thought of as universally quantified. The variables in a coverage clause represent the freedom a clause has to be instantiated in various ways to account for instances of the goal, and so they are thought of as existentially quantified.

More concretely, consider the example of the proof of *plus_assoc* from Section 2.2.3. The statement of the theorem is represented as the type family, and its proof is represented by two constants that form inhabitants of that type family.

$$
\begin{aligned}
&\mathit{plus\_assoc} : \mathit{plus}\ N_1\ N_{23}\ M \to \mathit{plus}\ N_1\ N_2\ N_{12} \to \mathit{plus}\ N_2\ N_3\ N_{23} \\
&\qquad \to \mathit{plus}\ N_{12}\ N_3\ M \to \mathit{type} \\
&\mathit{pa/z} : \mathit{plus\_assoc}\ \mathit{plus\_z}\ \mathit{plus\_z}\ P\ P. \\
&\mathit{pa/s} : \mathit{plus\_assoc}\ P_1\ P_2\ P_3\ P_4 \\
&\qquad \to \mathit{plus\_assoc}\ (\mathit{plus\_s}\ P_1)\ (\mathit{plus\_s}\ P_2)\ P_3\ (\mathit{plus\_s}\ P_4).
\end{aligned}
$$

The first three arguments to *plus_assoc* shown are intended as inputs, and the last as an output. This means that the relation *plus_assoc* is to be interpreted as a function that transforms a triple containing

- A derivation that $N_1 + N_{23} = M$
- A derivation that $N_1 + N_2 = N_{12}$
- A derivation that $N_2 + N_3 = N_{13}$

into a derivation that $N_{12} + N_3 = M$, which essentially means that $N_1 + (N_2 + N_3) = (N_1 + N_2) + N_3$.

We first wish to describe the coverage *goal* for this theorem. Since we are only considering input coverage checking, we may without loss drop the final (output) argument of

plus_assoc throughout the signature, and obtain instead

$$\begin{array}{l} \mathit{plus\_assoc} : \mathit{plus}\ N_1\ N_{23}\ M \to \mathit{plus}\ N_1\ N_2\ N_{12} \to \mathit{plus}\ N_2\ N_3\ N_{23} \to \mathit{type} \\ \mathit{pa/z} : \mathit{plus\_assoc}\ \mathit{plus\_z}\ \mathit{plus\_z}\ P. \\ \mathit{pa/s} : \mathit{plus\_assoc}\ P_1\ P_2\ P_3 \\ \qquad \to \mathit{plus\_assoc}\ (\mathit{plus\_s}\ P_1)\ (\mathit{plus\_s}\ P_2)\ P_3. \end{array}$$

We then define the coverage goal $\mathrm{goal}(a : K)$ given a type family $a : K$, if $K = \Pi x_1{:}A_1 \cdots \Pi x_n{:}A_n.\,\mathsf{type}$, to be

$$\mathrm{goal}(a : K) = x_1{:}A_1 \cdots x_n{:}A_n \vdash a\ x_1\ \cdots\ x_n$$

Here we are being slightly informal in not using spine form, and failing to mention the required η-expansion for the variables x_i. More precisely, we would say $x_1{:}A_1 \cdots x_n{:}A_n \vdash a \cdot (\mathsf{ex}_{A_1^-}(x_1); \cdots; \mathsf{ex}_{A_n^-}(x_n))$.

The coverage goal for *plus_assoc*, for example, including including the implicit arguments not mentioned above in the signature, is

$$\begin{array}{l} N_1 : \mathit{nat}, N_{23} : \mathit{nat}, N_2 : \mathit{nat}, N_3 : \mathit{nat}, N_{12} : \mathit{nat}, M : \mathit{nat}, \\ PL_1 : \mathit{plus}\ N_1\ N_{23}\ M, PL_2 : \mathit{plus}\ N_1\ N_2\ N_{12}, PL_3 : \mathit{plus}\ N_2\ N_3\ N_{23} \\ \vdash \mathit{plus\_assoc}\ PL_1\ PL_2\ PL_3 \end{array} \qquad (\mathit{goal})$$

Now we come to the clauses of the proof. In general a clause is structured as a constant of function type

$$c : A \text{ where } A = \Pi x_1{:}A_1 \ldots \Pi x_k{:}A_k.b_1 \to \cdots \to b_n \to b_\star$$

where the type $b_\star$ is called the *head* of the clause, and the b_i are the *subgoals*. Input coverage checking, because it is meant to correspond to a guarantee of success of the initial pattern matching of a logic programming query against some clause, is only concerned with the head of each clause, and how the variables $x_1 \cdots x_k$ can be instantiated to make the head match a given coverage goal. So drop all subgoals, and instantiate the Π-bound variables with fresh variables in a context to obtain the clause corresponding to the original type A, written

$$\mathrm{clause}(A) = x_1 : A_1, \cdots, x_k : A_k \vdash b_\star$$

For example, the coverage clause obtained in this way from the constant pa/z is

$$x_1 : \mathit{nat}, x_2 : \mathit{nat}, x_3 : \mathit{nat}, x : \mathit{plus}\ x_1\ x_2\ x_3 \vdash \mathit{plus\_assoc}\ \mathit{plus\_z}\ \mathit{plus\_z}\ x \qquad (\mathit{pa/z})$$

and from pa/s we get

$$\begin{array}{l} x_1 : \mathit{nat}, \cdots, x_6 : \mathit{nat}, \\ p_1 : \mathit{plus}\ x_1\ x_2\ x_3, p_2 : \mathit{plus}\ x_1\ x_4\ x_5, p_3 : \mathit{plus}\ x_4\ x_6\ x_2 \\ \vdash \mathit{plus\_assoc}\ (\mathit{plus\_s}\ p_1)\ (\mathit{plus\_s}\ p_2)\ p_3 \end{array} \qquad (\mathit{pa/s})$$

The question of whether a single clause can, just by itself, establish coverage of a coverage goal is the content of the definition of *immediate coverage*.

Definition We say that a goal $\Gamma_G \vdash b_G$ is *immediately covered* by a clause $\Gamma_C \vdash b_C$ if there exists a substitution $\Gamma_G \vdash \sigma : \Gamma_C$ such that $b_G = \{\sigma\}b_C$.

The ultimate criterion by which we judge coverage is whether in fact every individual possible input is covered by some member of a collection of clauses given by constants from the the signature. Therefore we make the following definitions. Say that the coverage goal $\cdot \vdash b'$ is a *ground instance* of a coverage goal $\Gamma \vdash b$ when b' is of the form $\{\sigma\}b$ for some substitution $\cdot \vdash \sigma : \Gamma$.

Definition A goal $\Gamma_G \vdash b_G$ is *covered* by a finite set of clauses $\{\Gamma_i \vdash b_i\}_{i \in I}$ if for every ground instance $\cdot \vdash b'$ of $\Gamma_G \vdash b_G$, there is some i such that $\Gamma_i \vdash b_i$ immediately covers $\cdot \vdash b'$.

In the case of our example, neither the clause (pa/z) nor (pa/s) immediately cover $(goal)$, but taken together, it can be seen (for instance by splitting as below) that (pa/z) and (pa/s) cover $(goal)$.

We note that immediate coverage implies coverage.

Lemma 5.2.1 *If $\Gamma \vdash b$ is immediately covered by $\Gamma' \vdash b'$, then it is covered by any collection of clauses that includes $\Gamma' \vdash b'$.*

Proof Let a ground instance $\cdot \vdash b_0$ of $\Gamma \vdash b$ be given, via the substitution $\cdot \vdash \sigma : \Gamma$ that has the property that $\{\sigma\}b = b_0$. Since $\Gamma \vdash b$ is immediately covered by $\Gamma' \vdash b'$ by assumption, let $\Gamma \vdash \sigma' : \Gamma'$ be such that $\{\sigma'\}b' = b$. Then the substitution $\{\sigma\}\sigma'$ witnesses $\{\{\sigma\}\sigma'\}b' = b_0$, as required. ■

5.2.2 Splitting

The initial coverage goal for a type family is very often not immediately covered by any of the clauses of a proof, for a proof usually works by analyzing the many possible cases of some input. Just as unification proceeded by transforming a set of equations step-by-step, preserving the set of solutions, coverage checking proceeds by transforming a collection of coverage goals step-by-step, beginning with the singleton set of the initial coverage coal.

The repeated step in the case of coverage, analogous to the various transformation steps in unification, is called *splitting*, which analyzes a coverage goal apart into a finite collection of coverage goals that, taken together, have the same instances as the original goal. That is, splitting transforms a coverage goal $\Gamma \vdash b$ into a set $\mathcal{G}$ of coverage goals. We require, for the splitting algorithm to be considered correct, that if $\mathcal{G}$ is covered by a collection of clauses $\mathcal{C}$, then so too $\Gamma \vdash b$ is covered by $\mathcal{C}$. In other words, $\mathcal{G}$ must be a refinement of $\Gamma \vdash b$ into an exhaustive set of possible special cases thereof.

Splitting of the goal $\Gamma \vdash b$ is accomplished by furthermore choosing a variable $x : A$ in Γ on which to split cases, and considering the possible forms of closed expressions that could have type A. To simplify the following presentation and highlight the main ideas, we describe only splitting of variables of base type. Splitting of variables of higher function types is nonetheless important in applications, and we refer the reader to the definitions in [SP03].

We borrow a variant of the notions of unifier and most general unifier suited to the current notational setting in which we use entirely ordinary variables instead of the modal variables used in the previous section.

Definition A substitution $\Gamma \vdash \sigma : \Psi$ is a *unifier* of $\Psi \vdash b_1 \doteq b_2 \wedge x \doteq M$ (with $x \in \Psi$) if $\{\sigma\}b_1 = \{\sigma\}b_2$ and $\sigma(x) = \{\sigma\}M$. Say σ is a *most general unifier* if every other unifier $\Gamma' \vdash \sigma' : \Psi$ factors through it, where by 'σ' factors through σ' we mean there exists $\Gamma' \vdash \sigma_\star : \Gamma$ such that $\sigma' = \{\sigma_\star\}\sigma$.

Definition Let a coverage goal $\Gamma \vdash b$ be given, with $x : b_x \in \Gamma$. We will build up a set $\mathcal{G}$ of new coverage goals as a replacement for the original goal, by *splitting on* x. Consider each $c : B \in \Sigma$ in turn, supposing that clause$(B) = \Psi_c \vdash b_c$ and $\Psi_c = z_1 : C_1, \cdots, z_m : C_m$.

- If there is a most general unifier $\Gamma' \vdash \sigma : \Gamma, \Psi_c$ of

$$\Gamma, \Psi_c \vdash b_x \doteq b_c \wedge x \doteq c\ z_1\ \cdots\ z_m$$

  then add $\Gamma' \vdash \{\sigma\}b$ to $\mathcal{G}$
- If there is no unifier, then add nothing to $\mathcal{G}$.

If we can not establish for some particular $c : B$ that it has a most general unifier, nor that it has no unifier at all, then splitting fails.

Continuing with the example of associativity of plus, splitting applied to the coverage goal (*goal*) above, splitting on the variable N_1, yields the two coverage goals

$$\begin{array}{l} N_{23} : nat, N_2 : nat, N_3 : nat, N_{12} : nat, M : nat, \\ PL_1 : plus\ z\ N_{23}\ M, PL_2 : plus\ z\ N_2\ N_{12}, PL_3 : plus\ N_2\ N_3\ N_{23} \\ \vdash plus\_assoc\ PL_1\ PL_2\ PL_3 \end{array} \qquad (goal_1)$$

$$\begin{array}{l} N_1' : nat, N_{23} : nat, N_2 : nat, N_3 : nat, N_{12} : nat, M : nat, \\ PL_1 : plus\ (s\ N_1')\ N_{23}\ M, PL_2 : plus\ (s\ N_1')\ z\ N_2\ N_{12}, PL_3 : plus\ N_2\ N_3\ N_{23} \\ \vdash plus\_assoc\ PL_1\ PL_2\ PL_3 \end{array} \qquad (goal_2)$$

where the end result is that in $(goal_1)$, the variable N_1 has been replaced by z, and in $(goal_2)$ it has been replaced by $s\ N_1'$. This splitting does not result in either goal being immediately covered. If we had instead chosen to split on PL_1, and then subsequently PL_2, we would obtain two coverage goals, each of which would be immediately covered by (and would in fact be α-equivalent to) (pa/z) and (pa/s), respectively.

Actual implementation of coverage checking looks to *how* immediate coverage checking fails (when it does fail) to make intelligent guesses about which variable it is likely to be effective to split on. This process is described in [SP03]. However, we take a more abstract view of the coverage checking algorithm and simply consider that every possible splitting is nondeterministically allowed.

In other words, the nondeterministic coverage checking algorithm is this: To test that a single coverage goal $\Gamma \vdash b$ is covered in the signature Σ, first check if it is immediately covered by clause(A) for some $c : A \in \Sigma$. If so, report success. Otherwise, choose a variable in Γ and split on it, yielding a set of goals $\{\Gamma_i \vdash b_i\}_{i \in I}$. Finally, recursively test in the same way that $\Gamma_i \vdash b_i$ is covered, for all i.

The result to prove to confirm that this algorithm is sound is the following.

Lemma 5.2.2 *Let $\mathcal{C} = \{\text{clause}(A) \mid c : A \in \Sigma\}$ be the coverage clauses for all constants in the signature. If splitting $\Gamma \vdash b$ on the variable $x \in \Gamma$ yields $\{\Gamma_i \vdash b_i\}_{i \in I}$, and for all $i \in I$ we have that $\Gamma_i \vdash b_i$ is covered by $\mathcal{C}$, then $\Gamma \vdash b$ is covered by $\mathcal{C}$.*

Proof Let a ground instance $\cdot \vdash b_0$ of $\Gamma \vdash b$ be given, with $\cdot \vdash \sigma : \Gamma$ such that $\{\sigma\}b = b_0$. Consider the replacement $\sigma(x)$ that σ assigns to x. It is easy to check that well-typed substitutions assign appropriately well-typed expressions to variables: if $x : b_x \in \Gamma$, we must have $\cdot \vdash \sigma(x) \Rightarrow \{\sigma\}b_x$. The term $\sigma(x)$ must be of the form $c\ M_1 \cdots\ M_n$ for some constant $c : B \in \sigma$. Recalling the definition of splitting, which included a case for every constant in the signature, in particular c, suppose clause(B) takes the form$\Psi_c \vdash b_c$ and $\Psi_c = z_1 : C_1, \cdots, z_m : C_m$. Notice then that, by typing of $c\ M_1 \cdots\ M_n$, we must have

$$\{\sigma\}b_x = \{M_1/z_1\} \cdots \{M_n/z_n\}b_c \tag{$\dagger$}$$

In other words, the combined substitution $\sigma(M_1/z_1) \cdots (M_n/z_n)$ (call it σ') is a unifier of

$$\Gamma, \Psi_c \vdash b_x \doteq b_c \wedge x \doteq c\ z_1\ \cdots\ z_m \tag{$*$}$$

By the definition of splitting, we included the coverage goal

$$\Gamma' \vdash \{\sigma_0\}b$$

Where $\Gamma' \vdash \sigma_0 : \Gamma, \Psi_c$ is a most general unifier of $(*)$. Since σ_0 is most general, σ' must factor through it, i.e. there exists $\cdot \vdash \sigma_\star : \Gamma'$ such that $\sigma' = \{\sigma_\star\}\sigma_0$.

Recall that by assumption we have that since $\Gamma' \vdash \{\sigma_0\}b$ is one of the goals that resulted from splitting, it is covered by $\mathcal{C}$, and any ground instance of it is immediately covered: but now $\cdot \vdash \{\sigma_\star\}\{\sigma_0\}b = \cdot \vdash \{\sigma'\}b = \cdot \vdash \{\sigma\}b$ is a ground instance of it, satisfying our goal of showing that $\cdot \vdash \{\sigma\}b$ is immediately covered by some clause in $\mathcal{C}$. ■

5.2.3 World Splits

Unification as described above operates by keeping a single collection of equations and transforming it while maintaining the set of solutions. One could alternatively imagine keeping track of a state of knowledge about a unification problem that involves disjunctions across multiple sets of equations. Disjunctions can be seen to naturally arise from equations that are not in the pattern fragment by virtue of duplicated variables in arguments to metavariables. For instance when searching for a u such that the equation

$$\lambda x.u\ x\ x \doteq \lambda x.x$$

holds, there are exactly two possibilities, either

$$u \leftarrow \lambda x_1.\lambda x_2.x_1 \text{ or } u \leftarrow \lambda x_1.\lambda x_2.x_2$$

Generalization to disjunctions throughout the process of unification is a significant complication, but the already disjunctive nature of splitting provides a natural motivation for

incorporating just a little bit of disjunctive reasoning at the boundary between unification and coverage checking *when* we can determine that a unification problem has not a single most general unifier, but rather a set of unifiers which, when taken together, are as general as the original query.

We can take advantage of the following scenario. Suppose we are in a situation when unification would otherwise report failure to find a most general unifier or to refute the unification problem as unsolvable. Although it failed, because it is formulated as as a transition system, it is naturally able to provide the final set of equations on which the transition system got stuck. If this final set of equations — which unification could not find a most general unifier for — can be determined to have a most general *set* of unifiers, then coverage checking can still make headway. The notion of most general set of unifiers we mean is the following generalization of the above definition of most general unifier.

Definition A set of substitutions $\sigma_1, \ldots, \sigma_n$ (each with $\Gamma_i \vdash \sigma_i : \Psi$) is a *most general set of unifiers* if every other unifier $\Gamma' \vdash \sigma' : \Psi$ factors through some substitution in it, i.e. there exists $i \in 1 \ldots n$ and $\Gamma' \vdash \sigma_0 : \Gamma_i$ such that $\sigma' = \{\sigma_0\}\sigma_i$. Observe that having an empty most general set of unifiers is the same thing as a unification problem having no solutions.

Subsequently, the definition of splitting can take advantage of sets of most general unifiers.

Definition Let a coverage goal $\Gamma \vdash b$ be given, with $x : b_x \in \Gamma$. We will build up a set $\mathcal{G}$ of new coverage goals as a replacement for the original goal. Consider each $c : B \in \Sigma$ in turn, supposing that $\mathrm{clause}(B) = \Psi_c \vdash b_c$ and $\Psi_c = z_1 : C_1, \cdots, z_m : C_m$.

If there is a most general set of unifiers $\{\Gamma'_i \vdash \sigma_i : \Gamma, \Psi_c\}_{i \in I}$ of

$$\Gamma, \Psi_c \vdash b_x \doteq b_c \wedge x \doteq c\ z_1\ \cdots\ z_m$$

then add $\{\Gamma'_i \vdash \{\sigma_i\}b_x\}_{i \in I}$ to $\mathcal{G}$. If for any $c : B$ we fail to find a set of most general set of unifiers, then splitting fails.

With this definition of splitting, Lemma 5.2.2 again holds. The proof works in exactly the same way, except that when we use the assumption that a most general unifier exists and extract from it a factoring substitution from which to generate evidence that the original goal is covered, we use instead the assumption that a most general *set* of unifiers exists, and extract from it one of the substitutions in the set, and subsequently a factoring substitution.

We can make use of this extra generality when splitting whenever we can detect that a unification problem naturally has multiple solutions. This is effective for dealing with a form of equation on worlds that occurs frequently in checking the examples in Chapter 6. To give an example of it, we must return to using the notation from the previous section, for we need to talk about world metavariables that depend on other worlds, and we have not introduced in the language of world expressions any lambda-abstraction over worlds. So suppose we are seeking instantiations of world metavariables υ_1 and υ_2, both of type $\alpha : \mathsf{w} \vdash \mathsf{w}$, to satisfy the equation

$$p * \alpha \doteq \upsilon_1[\alpha/\alpha] * \upsilon_2[\alpha/\alpha] \qquad (\dagger)$$

where p is some world expression not mentioning α. No rule proposed above for world unification can make any progress on this equation, but we can straightforwardly ascertain in this case that either υ_1 or υ_2 must not use their respective argument α.

A higher-level picture of why this sort of equation arises can be given by consideration of metatheorems about linear logic derivations, such as cut admissibility. In analyzing the structure of a linear logic derivation that ends with an inference rule that divides the context such as the $\otimes$ right rule,

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}$$

If we are led to consider an arbitrary linear hypothesis found in the conclusion $\Gamma, \Delta \vdash A \otimes B$ of the rule, then it is unknown a priori whether that hypothesis is to be found in Γ, or in Δ. This typically appears at the HLF level in the form of a world variable α, which represents the linear hypothesis that is known to appear somewhere in the combined context. The Γ which might contain α corresponds to $\upsilon_1[\alpha/\alpha]$, and the role of Δ is played by $\upsilon_2[\alpha/\alpha]$. The role played by p is just the remainder of Γ, Δ once the chosen linear hypothesis is removed. The claim that α can not be used in both υ_1 and υ_2 amounts to the fact that no linear hypothesis can go into *both* Γ and Δ, for linear contexts are disjoint.

The typical way that proofs by hand proceed in this situation is to simply consider both cases, and that is precisely what we intend to formally capture with an analogous notion of case split during coverage checking.

When performing coverage checking, we could simply look for instances of literally (perhaps up to α-variation) the equation (†) in the output of unification when searching for a most general unifier when attempting to split on some variable, but we would like to generalize it slightly, to better capture the underlying principle. There is nothing necessary about having two metavariables that mention α on the right, nor that they are the *only* constituents of the right-hand side. In general, if we have a single top-level occurrence of a world variable on one side of an equation, and a number of metavariables on the other which have access to that variable, only one of those metavariables can actually use it. Formally, we have the following lemma:

Lemma 5.2.3 *Let Q be the equation*

$$p * \alpha \doteq q * \upsilon_1[\xi_1] * \cdots * \upsilon_n[\xi_n] \tag{$*$}$$

Suppose that α does not occur in p or q, and occurs exactly once in each of $\xi_1, \ldots, \xi_n$. Specifically, suppose $(\alpha/\beta_i) \in \xi_i$ for each i, and suppose the type of each υ_i is $\Psi_i \vdash \mathsf{w}$. Then the set of solutions of Q is equal to the union for all $i \in \{1, \ldots, n\}$ of the solutions to

$$p * \alpha \doteq q * \upsilon'_1[\xi'_1] * \cdots * \upsilon_i[\xi_i] * \cdots * \upsilon'_n[\xi'_n] \tag{$**$}$$

this being the same equation as $(*)$, except for every $j \neq i$, we have replaced $\upsilon_j[\xi_j]$ by $\upsilon'_j[\xi'_j]$, where υ'_j is a fresh world metavariable of the pruned type $(\Gamma_j)_{\backslash\beta_j} \vdash \mathsf{w}$ and ξ'_j is the pruned substitution $(\xi_j)_{\backslash x}$.

Proof The easier direction is seeing that every solution of one of the $(**)$ is a solution of $(*)$, for $(**)$ is merely a pruned instance of $(*)$ that a fortiori imposes that some metavariables can not mention some of their arguments.

In the other direction, suppose we have a solution θ to $(*)$. Since α does not occur at all in p, there is exactly one occurrence in $\theta(p * \alpha)$. Therefore there must be exactly one occurrence of α in $\theta(q * \upsilon_1[\xi_1] * \cdots * \upsilon_n[\xi_n])$. By assumption α does not occur in q, so there is one occurrence of α in $\theta(\upsilon_1[\xi_1] * \cdots * \upsilon_n[\xi_n])$. Subsequently we know that there must be some i such that there is one occurrence of α in $\theta(\upsilon_i[\xi_i])$, and no occurrences of α in $\theta(\upsilon_j[\xi_j])$ for all $j \neq i$. Thus θ must replace each such υ_j with a world expression not including β_j, so there is a solution θ' that merely assigns the erstwhile replacement of υ_j instead to υ'_j and is still well-formed, and is a solution to $(**)$ (for the particular i that was given) as required. ■

5.2.4 Lowering and Raising

As noted, some of the presentation here is in terms of ordinary variables of (in general) function type, whereas unification was described with modal variables whose type is always a base type in a local context. The appropriate notion of impedance-matching between the two is called *lowering* or *raising* depending on which direction one goes: to make a modal variable that has a base type out of a variable that has a function type is to *lower* its type to a base type, and conversely one can *raise* the result of unification featuring remaining metavariables back into an ordinary function type.

These transformations are conceptually fairly straightforward — there is not an essential change to the number or type of arguments to any variable, but rather a change in the interpretation of a variable as instantiatable or not, and the different direction associativity present in substitutions as opposed to spines — but can be somewhat notationally involved. For example, the idea of lowering works in the following way. Supposing we are searching for instantiations for all of the (ordinary) variables in $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ that satisfy the equations P. Then we want to call the unification algorithm above with an input $\Delta \vdash P$ where Δ consists of metavariables $u_i :: \Psi_i \vdash b_i$ that correspond to the x_i. We can begin by setting $\Psi_i = y_1 : B_{i1}, \ldots, y_{m_i} : B_{im_i}$, assuming that A_i is of the form

$$\Pi y_1 : B_{i1} \ldots \Pi y_{m_i} : B_{im_i} . b_i$$

This conveys the basic idea, but we must also replace every occurrence of x_i with the corresponding u_i, and convert the spine of arguments to the function x_i to a substitution for the local variables of u_i. We refer the reader to [NPP05] for further discussion.

5.2.5 Monotonicity

We now return to the idea that in linear logic, a resource that has consumed stays forever consumed. Specifically, note that in a series of sequent left (or, in natural deduction style,

elimination) rules, the linear context always grows *larger* as we proceed from the premises of an inference rule to its conclusion. In HLF, this corresponds to the current *world* growing larger (where we think of $p * q$ as larger than either p or q) from top to bottom of the appropriate parts of a derivation, which are the rules that construct spines. If this were always the case — which in HLF in general it is not, although in the image of translation of LLF it is, as we will see below — it would permit a useful sort of subformula property, in that we know any derivation of $\Gamma \vdash M \Leftarrow A[p]$ cannot use any $x : A \mathbin{@} \alpha$ when α does not appear in p, and we would be able to use this to narrow down the possible form of coverage goals, and succeed in coverage checking of more metatheorems.

For example, if during coverage checking consider which terms M could possibly exist that would satisfy

$$\alpha : \mathsf{w}, x : b \mathbin{@} \alpha, \beta : \mathsf{w}, y : b' \vdash M : b''[\alpha]$$

we would like to conclude that M contains no occurrence of β or of y. From the point of view of LLF, the linear assumption y, guarded by the resource name β, has already been spent elsewhere, and cannot be used in M. This reasoning is called *strengthening* because, in making a context smaller, it is naturally a sort of opposite to the general principle of weakening, which makes the context larger. The intent is to change the coverage algorithm to include the following tactic: When trying to check (in HLF) coverage of the goal $\Gamma \vdash b$, consider a variable $x : \ldots \Pi\beta{:}\mathsf{w}.\Pi y{:}A \mathbin{@} \beta \ldots (b_x \mathbin{@} p) \in \Gamma$. If we can determine that no term of type $b_x \mathbin{@} p$ can possibly mention β or y, it then suffices to check coverage of the goal $\Gamma' \vdash b$, where Γ' is Γ with the type of x changed to elide $\Pi\beta{:}\mathsf{w}.\Pi y{:}A \mathbin{@} \beta$.

There is a strong analogy between this process of monotonicity strengthening and existing form of strengthening of coverage goals licensed by *subordination* [Vir99]. (Indeed our approach can be viewed as simply a more general notion of subordination that is sensitive to modal worlds.) The subordination relation is a binary relation on types, where A is said to be subordinate to type B if a term of type A can appear as a subterm of a term of type B. A reasonable approximation to the subordination relation can be inferred from a signature by observing which type families ever occur in the arguments of constructors that create terms of other type families. For another example, if in ordinary LF we are seeking a term M that satisfies

$$x : exp \vdash M : nat$$

and the only constants defining the type nat in the signature are, as we might expect, $z : nat$ and $s : nat \to nat$, then hereditarily there is no way for a variable of any other type such as exp to 'infect' terms of type nat, and we may strengthen x away: the only M such that $x : exp \vdash M : nat$ are M such that $\cdot \vdash M : nat$.

Just as subordination needs to perform an analysis on the signature to justify its strengthening, the world-based strengthening we wish to do relies on making sure the constants in the signature are *monotone* in an appropriate sense on worlds in an appropriate sense. To see why this is required, consider an HLF signature

$$o : \mathsf{type}$$
$$c : \Pi\alpha.o \mathbin{@} \alpha \to o \mathbin{@} \varepsilon$$

In this signature we could write a term

$$\alpha : \mathsf{w}, x : o \mathbin{@} \alpha \vdash c\ \alpha\ x : o[\varepsilon]$$

which is well-formed at the empty world ε — and so appears as if it should be well-formed in the empty linear context — and yet it mentions the resource label α we expect therefore to be unused and the variable x whose type is associated with α, which we think of as already consumed, or at least otherwise unavailable, because α does not appear in ε.

We lay the blame for this failure on the type of the constant c: it is unlike any type that arises from $\to$, $\multimap$, $\&$, $\top$, or $\rightharpoonup$, in that it quantifies over a world, which is used as the label of the argument of c, but then it is not used on c's result, which is available at the empty world. This can be seen as taking an argument that intuitively has already consumed the resource α, and yielding an expression that *has not* consumed it. We can see by contrast that the linear function space encoding $\forall\alpha.{\downarrow}\beta.A \mathbin{@} \alpha \to B \mathbin{@} (\alpha * \beta)$ outputs something at a world $\alpha * \beta$ that is at least as big as its input's world α.

In this section we describe a monotonicity condition on signatures which, when satisfied, justifies the strengthening described above. Moreover, we show that every type arising from the embeddings of LLF described above satisfies this condition.

First we wish to restrict our attention, without loss of generality, to the fragment of HLF types that remain after part of the hybrid elaboration phase has completed, where $@$ is pushed down to base types, and $\downarrow$ is eliminated. The judgment $\Gamma \vdash A\,\mathsf{elab}$ ('A elaborated') captures this condition, and is defined as follows.

$$\frac{\Gamma, \alpha : \mathsf{w} \vdash A\,\mathsf{elab}}{\Gamma \vdash \Pi\alpha{:}\mathsf{w}.A\,\mathsf{elab}} \qquad \frac{\Gamma \vdash A\,\mathsf{elab} \qquad \Gamma, x : A \vdash B\,\mathsf{elab}}{\Gamma \vdash \Pi x{:}A.B\,\mathsf{elab}} \qquad \frac{}{\Gamma \vdash (b \mathbin{@} p)\,\mathsf{elab}}$$

We generally assume all types A below satisfy $\Gamma \vdash A\,\mathsf{elab}$.

The additional restriction we wish to impose on the types that are assigned to constants in the signature is that they are *monotone*. In words, the restriction is that every negatively-occurring (in the usual sense of occurring under an odd number of left-hand sides of function types) type expression of the form $\Pi\alpha{:}\mathsf{w}.\cdots b \mathbin{@} p$, α must appear in p. It will be easy to show that every LLF type satisfies this property, for among them, *every* type expression, whether positively or negatively occurring, will see every Π-bound world-variable have the appropriate occurrence. We make the more fine-grained restriction with an eye to future work, where perhaps different encoding strategies other than those in the image of translations from known systems may still benefit from the reasoning principles that monotonicity allows.

Formally, the monotonicity property is defined as two mutually recursive properties, which alternate as we pass under the left-hand side of function types. They are written $\Gamma \vdash A\,\mathsf{m}^+$, for indicating that the type A is *monotone occurring positively*, and $\Gamma \vdash A\,\mathsf{m}^-$, for indicating that A is *monotone occurring negatively*. There is an additional predicate on variables, written $\alpha \in_s A$, means α has a what is called a *strict* occurrence in A. In our case, it amounts simply to α occurring in the base type at the end of the prefix of Π bindings in A but we use the word 'strict' in allusion to its use in other work [PS98] to suggest the similarity of the important property of strict occurrences at work here, which

is that no substitution of function arguments for the Π-bound variables can cause a strict occurrence of a variable to disappear. We do also use the symbol $\in$ without the subscript s to mean the usual sense of 'occurs at all in'.

Monotone types and strict occurrences are defined by the following inference rules.

$$\frac{\Gamma, \alpha : \mathsf{w} \vdash A\,\mathsf{m}^+}{\Gamma \vdash \Pi\alpha{:}\mathsf{w}.A\,\mathsf{m}^+} \qquad \frac{\Gamma \vdash A\,\mathsf{m}^- \quad \Gamma, x : A \vdash B\,\mathsf{m}^+}{\Gamma \vdash \Pi x{:}A.B\,\mathsf{m}^+} \qquad \frac{}{\Gamma \vdash (b \mathbin{@} p)\,\mathsf{m}^+}$$

$$\frac{\Gamma, \alpha : \mathsf{w} \vdash A\,\mathsf{m}^- \qquad \alpha \in_s A}{\Gamma \vdash \Pi\alpha{:}\mathsf{w}.A\,\mathsf{m}^-} \qquad \frac{\Gamma \vdash A\,\mathsf{m}^+ \quad \Gamma, x : A \vdash B\,\mathsf{m}^-}{\Gamma \vdash \Pi x{:}A.B\,\mathsf{m}^-} \qquad \frac{}{\Gamma \vdash (b \mathbin{@} p)\,\mathsf{m}^-}$$

$$\frac{\beta \in_s A}{\beta \in_s \Pi\alpha{:}\mathsf{w}.A} \qquad \frac{\beta \in_s B}{\beta \in_s \Pi x{:}A.B} \qquad \frac{\beta \in p}{\beta \in_s b \mathbin{@} p}$$

One important property about strict occurrences is the following.

Lemma 5.2.4 *If* $\Gamma \vdash S : A[\varepsilon] > b[q]$ *and* $\alpha \in_s A$*, then* $\alpha \in q$.

Proof Straightforward induction, observing in the spine cons cases that elab and $\in_s$ are preserved by substitution. ■

Extend the notion of m^- in the evident way to contexts and signatures: $\vdash \Gamma\,\mathsf{m}^-$ iff every type in Γ is m^-, and likewise $\vdash \Sigma\,\mathsf{m}^-$ the type of every constant in Σ is m^-. We can then see that when every type is monotone, variables at worlds that are not part of the currently available set of resources cannot be used. (The fact that ε is used inductively as the current world of term type-checking may seem counterintuitively over-restrictive, but it still suffices because of our assumption that all types satisfy elab and have all world operations pushed down to base types)

Lemma 5.2.5 *Let some* Γ *be given such that it is of the form (up to some permutation)* $\Gamma_0, \alpha : \mathsf{w}, x : A$*, with* $\alpha \in_s A$*, and* α *not occurring anywhere in* Γ_0*. Let* B, p *be given with* $\alpha \notin FV(B)$ *and* $\alpha \notin p$*. Suppose that* $\vdash \Gamma\,\mathsf{m}^-$ *and* $\vdash \Sigma\,\mathsf{m}^-$.

- *If* $\Gamma \vdash M \Leftarrow B[\varepsilon]$ *and* $\Gamma \vdash B\,\mathsf{m}^+$*, then* $\alpha, x \notin FV(M)$.
- *If* $\Gamma \vdash R \Rightarrow b[p]$*, then* $\alpha, x \notin FV(R)$.
- *If* $\Gamma \vdash S : B[\varepsilon] > b[p]$*, and* $\Gamma \vdash B\,\mathsf{m}^-$*, then* $\alpha, x \notin FV(S)$.

Proof By induction on the derivation.

Case: M is of the form $\lambda x.M_0$, with a type of the form $\Pi x{:}B_1.B_2$ and inversion on the derivation of $\Pi x{:}B_1.B_2\,\mathsf{m}^+$ gives $\Pi x{:}B_1\,\mathsf{m}^-$, allowing us to maintain the invariant that $\vdash \Gamma\,\mathsf{m}^-$. We depend on $\alpha \notin B_1$ to maintain that $\alpha \notin \Gamma_0$.

Case: M of the form R with typing derivation

$$\frac{\Gamma \vdash R \Rightarrow b[p] \qquad b = b' \qquad p \equiv_{\mathsf{acu}} p'}{\dfrac{\Gamma \vdash R \Leftarrow b'[p']}{\Gamma \vdash R \Leftarrow (b' \mathbin{@} p')[\varepsilon]}}$$

The proof proceeds by immediate appeal to induction hypothesis, noting that $\equiv_{\mathsf{acu}}$ cannot eliminate or create any occurrence of the world variable α, which does not occur in p' by assumption.

Case: S of the form $(p; S_0)$ with typing derivation

$$\frac{\Gamma \vdash q : \mathsf{w} \qquad \Gamma \vdash S_0 : (\{q/\beta\}^{\mathsf{w}} B_0)[\varepsilon] > b[p]}{\Gamma \vdash (q; S_0) : (\Pi\beta{:}\mathsf{w}.B_0)[\varepsilon] > b[p]}$$

Suppose towards a contradiction that $\alpha \in q$. Then notice by inversion on $\Gamma \vdash \Pi\beta{:}\mathsf{w}.B_0\,\mathsf{m}^-$ that we get $\beta \in_s B_0$. In that case, clearly then $\alpha \in_s \{q/\beta\}B_0$. By Lemma 5.2.4 this would mean $\alpha \in p$, contradicting an assumption. So $\alpha \notin q$. Seeing that $\alpha, x \notin S_0$ follows by appeal to the induction hypothesis, after observing that the inductive invariant $\Gamma \vdash B\,\mathsf{m}^-$ is preserved by world substitution, and the invariant $\alpha \notin B$ is preserved by substitution of a world *not containing* α, a fact that we do indeed now know about q.

Case: $y \cdot S$ for some $y \neq x$ with typing derivation

$$\frac{y : B \in \Gamma \qquad \Gamma \vdash S : B[\varepsilon] > b[p]}{\Gamma \vdash y \cdot S \Rightarrow b[p]}$$

That α, x do not occur in S follows by induction hypothesis. We have $\Gamma \vdash B\,\mathsf{m}^-$ because $\vdash \Gamma\,\mathsf{m}^-$, and $\alpha \notin B$ because y is not x, and therefore in the part of the context forbidden from mentioning α.

Case: $x \cdot S$ with typing derivation

$$\frac{x : A \in \Gamma \qquad \Gamma \vdash S : A[\varepsilon] > b[p]}{\Gamma \vdash x \cdot S \Rightarrow b[p]}$$

This case cannot arise. For by assumption, $\alpha \in_s A$. Lemma 5.2.4 then entails that $\alpha \in p$, contradicting the assumption that $\alpha \notin p$. ■

5.2.6 LLF is Monotone

To see that every LLF type is monotone in both positive and negative occurrences, we first define a slight variant of the translation $(\text{—})^{(p)}$ in 4.3.1 that refrains from internalizing @ at base types, and so arrives exactly at types satisfying elab, by replacing the existing clauses for base types and type family declarations

$$\begin{aligned}(a \cdot S)^{(p)} &= a \cdot (p; S)\\ (\Sigma, a : K)^{(\varepsilon)} &= \Sigma^{(\varepsilon)}, a : \Pi\alpha{:}\mathsf{w}.(K^{(\varepsilon)})\end{aligned}$$

with the following:

$$\begin{aligned}(a \cdot S)^{(p)} &= (a \cdot S)\ @\ p\\ (\Sigma, a : K)^{(\varepsilon)} &= \Sigma^{(\varepsilon)}, a : K^{(\varepsilon)}\end{aligned}$$

First we make a simple observation about LLF types and strict occurrences:

Lemma 5.2.6 *For any LLF type A, we have $\alpha \in_s A^{(\alpha * p)}$*

Proof Straightforward induction. ■

We can then show the following result. Recall that the superscript $\multimap$ is meant to emphasize that we are expanding out the LLF type $A \multimap B$ as

$$\Pi\alpha{:}\mathsf{w}.{\downarrow}\beta.A \mathbin{@} \alpha \to B \mathbin{@} (\alpha * \beta)$$

Lemma 5.2.7 *If A is an LLF type in the context Γ_0, then $\Gamma \vdash (A^{\multimap})^{(p)}\,\mathsf{m}^+$ and $\Gamma \vdash (A^{\multimap})^{(p)}\,\mathsf{m}^-$, where $\Gamma = \Gamma_0^{(\varepsilon)}$.*

Proof By induction on the structure of A. The interesting case is $A = B_1 \multimap B_2$. In that case, $(A^{\multimap})^{(p)}$ is

$$\Pi\alpha{:}\mathsf{w}.(B_1^{\multimap})^{(\alpha)} \to (B_2^{\multimap})^{(\alpha * p)}$$

By induction hypothesis, we have

$$\Gamma, \alpha : \mathsf{w} \vdash (B_1^{\multimap})^{(\alpha)}\,\mathsf{m}^{\pm} \qquad \Gamma, \alpha : \mathsf{w}, x : (B_1^{\multimap})^{(\alpha)} \vdash (B_2^{\multimap})^{(\alpha * p)}\,\mathsf{m}^{\pm}$$

for both $\pm \in \{+, -\}$. From there, we need only build derivations

$$\dfrac{\dfrac{\dfrac{i.h.}{\Gamma, \alpha : \mathsf{w} \vdash (B_1^{\multimap})^{(\alpha)}\,\mathsf{m}^-} \quad \dfrac{i.h.}{\Gamma, \alpha : \mathsf{w}, x : (B_1^{\multimap})^{(\alpha)} \vdash (B_2^{\multimap})^{(\alpha * p)}\,\mathsf{m}^+}}{\Gamma, \alpha : \mathsf{w} \vdash (B_1^{\multimap})^{(\alpha)} \to (B_2^{\multimap})^{(\alpha * p)}\,\mathsf{m}^+}}{\Gamma \vdash \Pi\alpha{:}\mathsf{w}.(B_1^{\multimap})^{(\alpha)} \to (B_2^{\multimap})^{(\alpha * p)}\,\mathsf{m}^+}$$

and

$$\dfrac{\dfrac{\dfrac{\textit{Lemma } 5.2.6}{\alpha \in_s (B_2^{\multimap})^{(\alpha * p)}}}{\alpha \in_s (B_1^{\multimap})^{(\alpha)} \to (B_2^{\multimap})^{(\alpha * p)}} \quad \dfrac{\dfrac{i.h.}{\Gamma, \alpha : \mathsf{w} \vdash (B_1^{\multimap})^{(\alpha)}\,\mathsf{m}^+} \quad \dfrac{i.h.}{\Gamma, \alpha : \mathsf{w}, x : (B_1^{\multimap})^{(\alpha)} \vdash (B_2^{\multimap})^{(\alpha * p)}\,\mathsf{m}^-}}{\Gamma, \alpha : \mathsf{w} \vdash (B_1^{\multimap})^{(\alpha)} \to (B_2^{\multimap})^{(\alpha * p)}\,\mathsf{m}^-}}{\Gamma \vdash \Pi\alpha{:}\mathsf{w}.(B_1^{\multimap})^{(\alpha)} \to (B_2^{\multimap})^{(\alpha * p)}\,\mathsf{m}^-}$$

■

5.3 Related Work

5.3.1 Constraint Domains

It is reasonable 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 treating LLF is just a special case. This would certainly be 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 $\forall$ or Π) and that the equational theory on hypothetical worlds can be defined equationally.

In particular unification modulo axioms such as associativity, commutativity, and unit laws, among others such as idempotence has been studied extensively [LC94]. Unfortunately not as much work seems to have been done combining these axiomatic theories with higher-order functions. Boudet comes perhaps the closest to our setting of the algebraic unification problem interacting with higher-order unification by studying AC-unification of higher-order patterns [BBC97].

5.3.2 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 [WCPW03a, WCPW03b] also supports a logic programming language, LolliMon [LPPW05]. Andreoli's early work with focusing also inspired a language LO he developed with Pareschi [AP90], which, like its logical counterpart in the original focusing 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 worlds — ideally they might turn out to be isomorphic, providing a alternative logical explanation for why their algorithm works.

Chapter 6

Applications

This chapter is devoted to a selection of example applications of the HLF type theory. They demonstrate on how HLF enables representation and checking of metatheorems about stateful deductive systems. Partly as a reminder that the examples are automatically machine-checkable, and partly for compactness of presentation, they are given in concrete syntax, for the extension to the Twelf system that we have implemented.

Here is a brief glossary of how Twelf syntax, including the extensions we have made to it, relates to the mathematical notation used so far.

Mathematical	Twelf
$\Pi x{:}A.B$	`{x : A} B`
$\lambda x.M$	`[x] M`
$\Pi\alpha{:}\mathsf{w}.B$	`{a : w} B`
$\lambda\alpha.M$	`[a : w] M`
$A \multimap B$	`A -o B`
$A \to B$	`A -> B`
$A \mathbin{@} p$	`A @ p`
ε	`e`
type	`type`

Lambda abstractions `[x] M` may also come equipped with a type ascription on the variable, written `[x : A] M`, which helps type reconstruction determine the type of the whole expression. Comments, and pragma declarations particular to Twelf, such as fixity declarations and instructions to check coverage, are preceded by a percent sign `%`.

The current implementation does not yet directly support the additive conjunction and unit &, $\top$, but so far it has proved rather simple to simply manually carry out the elaboration phase described above where & and $\top$ would have been used. We nonetheless expect it to be quite easy to add this elaboration pass to the front-end as a convenience. In the encodings below, we occasionally make a note of how the encoding could be made slightly

more compact were these connectives available, using for them the following notation.

Mathematical	Twelf
$\&$	`&`
$\top$	`t`

6.1 Linear Cut Admissibility and Identity

In this section we show how our running example of the linear cut admissibility theorem can be treated in HLF. Much is owed to the original LLF encoding of the proof of the same theorem described by Cervesato and Pfenning [CP02]. The gain here is that we are able to state the theorem itself in a precise way, accounting for the linear contexts in a way not directly available in LLF.

An adequacy result about the encoding itself is given by Lemma 6.1.1. By 'adequacy' we mean the existence of a bijection between canonical forms of expressions of LF, and an already well-understood set of mathematical structures. These bijections are also typically expected to be *compositional*, in the sense that they commute with substitutions.

Lemma 6.1.1 (Adequacy of Linear Encoding) *Sequent derivations in intuitionistic linear logic are in bijective correspondence with HLF terms of the type family* conc *in the signature below, in the following sense:*

There is a representation map $\ulcorner - \urcorner$ *which*

- *takes linear logic propositions to HLF terms of type* o
- *takes linear logic proofs of* $A_1, \ldots, A_n \vdash C$ *to HLF terms of type*

$$\Pi\alpha_1{:}\mathsf{w}.\mathsf{hyp}\ \ulcorner A_1 \urcorner\ @\ \alpha_1 \rightarrow \cdots \Pi\alpha_n{:}\mathsf{w}.\mathsf{hyp}\ \ulcorner A_n \urcorner\ @\ \alpha_n \rightarrow \mathsf{conc}\ \ulcorner C \urcorner\ @\ (\alpha_1 * \cdots * \alpha_n)$$

Furthermore, $\ulcorner - \urcorner$ *is a bijection.*

Proof By straightforward induction over linear logic sequent deductions and HLF terms. It should be apparent from the constants that inhabit the type family conc that there is one such constant for each inference rule of linear logic. ■

By way of comparison, *an* adequacy theorem for the encoding also holds for LLF, but it is not the same result as above, because of the difference in how LLF and HLF treat $\top$. Instead of faithfully corresponding to sequent calculus *derivations* as HLF, LLF terms over a similar signature would correspond to a certain form of normal natural deduction terms arising from sequent derivations.

An annotated description of the encoding follows, divided into sections by proposition, to illustrate the proof's natural modularity.

6.1.1 Basic Definitions

The most basic declarations to be made are of the type o of propositions, and the types of sequent conclusions and hypotheses.

```
o : type. %name o A.
conc : o -> @type.
hyp : o -> @type.
```

We write `@type` as a hint to the implementation that terms of type conc A and hyp A for some $A : o$ can be used as linear hypotheses, or used in the presence of linear hypotheses. The kind of the type `o`, by contrast, is simply given as `type`, for under no circumstances do we ever make a linear hypothesis *that* something is a proposition, nor is the fact of something being a proposition every subject to the existence of particular linear hypotheses. We surmise that such things might perhaps be sensible in a linear second-order logic. We emphasize that distinction between `type` and `@type` is not really an essential type-theoretic distinction, but merely an implementation approach to make type inference easier.

We now come to the statement of cut admissibility metatheorem.

```
%%% Cut admissibility

ca : {A : o}
      conc A @ G
      -> (hyp A -o conc C) @ D
      -> conc C @ (G * D)
      -> type.

%mode (ca +A +D +E -F).
```

As mentioned before, this directly reflects the structure of quantification over contexts of the usual theorem proved by hand. We quantify explicitly over the cut formula A at the beginning so that we can later mention it in the induction measure when we ask the system to check that the cut elimination procedure is terminating. The mode declaration on the final line asserts that the cut formula and the first two derivations, of type `conc A @ G` and `(hyp A -o conc C) @ D`, are to be treated as inputs, while the last argument to `ca`, of type `conc C @ (G * D)`, is to be an output of the relation.

The one structural rule of the system allows a hypothesis to be used as a conclusion.

```
% Init rule

init : hyp A -o conc A.
```

The existence of this rule already requires two cases in the proof of the cut admissibility theorem. If we name the derivations that are the premises by saying that we are cutting a derivation $\mathcal{D}$ of $\Gamma \vdash A$ into a derivation $\mathcal{E}$ of $\Gamma, A \vdash C$, then there is one case for when the init rule is used as the last rule of $\mathcal{D}$, and one for when it is the last rule of $\mathcal{E}$.

```
%% Init Conversions

ca_init_l : ca A (init ^ H) E (E ^ H).

ca_init_r : ca A D init D.
```

The caret symbol `^` is in the implementation made to be a synonym for underscore `_` (no relation to the 'this term cannot arise' underscore described in the section on unification

above) which tells the type and term reconstruction engine in Twelf to figure out via unification what should go in its place. Because linear function space in HLF expands to a Π-binder on a world variable followed by an ordinary term-level function space, a constant like init that has a linear function type actually takes two arguments: the world and the term. It is sometimes tedious as a user of the system to have to figure out what its world argument should be, but term reconstruction is most often capable of determining it in cases such as immediately above. The choice of `^` as an additional way to ask for term reconstruction simply serves to make such an elision resemble the traditional way of writing linear application of a function f to an argument x as $f\,\hat{}\,x$.

6.1.2 Additive Conjunction

Here are the defining inference rules for the additive conjunction &.

```
and    : o -> o -> o.  %infix right 11 and.

%%% Inference Rules

andl1 : (hyp (A and B) -o conc C)
        o- (hyp A -o conc C).
andl2 : (hyp (A and B) -o conc C)
        o- (hyp B -o conc C).

andr : {a : w}
        conc (A and B) @ a
        <- conc B @ a
        <- conc A @ a.
```

The right rule andr is displayed in its form following manually applying the elaboration translation that eliminates additives. With them, we could have equally well written

```
andr : conc (A and B)
         o- (conc A & conc B).
```

where `&` here denotes HLF's notion of additive conjunction, which would have translated to what we already have written above.

The cut admissibility cases for & (as are the cases for all other connectives below) are divided into *principal*, $\mathcal{D}$*-commutative*, and $\mathcal{E}$*-commutative* cases. Again supposing that we are cutting a derivation $\mathcal{D}$ of $\Gamma \vdash A$ into a derivation $\mathcal{E}$ of $\Gamma, A \vdash C$, the principal cases are those for which both derivations $\mathcal{D}$ and $\mathcal{E}$ ends with introducing & into the cut formula $A = A_1 \,\&\, A_2$, where $\mathcal{D}$ introduces it with a right rule, and $\mathcal{E}$ with a left rule. The $\mathcal{D}$-commutative (resp. $\mathcal{E}$-commutative) cases are when a rule for & is the last rule used to make the derivation $\mathcal{D}$, (resp. $\mathcal{E}$) when it does *not* affect the cut formula A.

The notation `[x :^ A]` is shorthand for a 'linear lambda', which expands to

```
[a : w][x :^ (A @ a)]
```

the successive binding of a world variable followed by an ordinary variable at that world.

```
%%% Principal Cases

ca/and1 : ca (A1 and A2) (andr ^ D1 D2)
           ([h:^ (hyp (A1 and A2))] andl1 ^ E ^ h) F
           <- ca A1 D1 E F.


ca/and2 : ca (A1 and A2) (andr ^ D1 D2)
           ([h :^ (hyp (A1 and A2))] andl2 ^ E ^ h) F
           <- ca A2 D2 E F.


%%% D-Commutative Cases

cad/andl1  : ca A (andl1 ^ D1 ^ H) E (andl1 ^ D1' ^ H)
              <- {a : w} {h1: hyp B1 @ a} ca A (D1 ^ h1) E (D1' ^ h1).


cad/andl2  : ca A (andl2 ^ D2 ^ H) E (andl2 ^ D2' ^ H)
              <- {a : w} {h2:hyp B2 @ a} ca A (D2 ^ h2) E (D2' ^ h2).

%%% E-Commutative Cases

cae/andr : ca A D ([h:^ (hyp A)] andr ^ (E1 ^ h) (E2 ^ h))
                  (andr ^ E1' E2')
            <- ca A D E1 E1'
            <- ca A D E2 E2'.

cae/andl1 : ca A D
         ([a0:w] [h:hyp A @ a0]
             andl1 ^ ([a1:w] [h1:hyp A3 @ a1] E1 ^ h ^ h1) HP H)
                     (andl1 ^ E1' ^ H)
             <- ({a1:w}{h1 : hyp A3 @ a1} ca A D ([a0:w] [h:hyp A @ a0]
                      E1 a0 h a1 h1) (E1' ^ h1)).

cae/andl2 : ca A D
         ([a0:w] [h:hyp A @ a0]
             andl2 ^ ([a1:w] [h1:hyp A3 @ a1] E2 ^ h ^ h1) HP H)
                    (andl2 ^ E2' ^ H)
             <- ({a1:w}{h1 : hyp A3 @ a1} ca A D ([a0:w] [h:hyp A @ a0]
                      E2 a0 h a1 h1) (E2' ^ h1)).
```

6.1.3 Additive unit

The encoding of $\top$ is little more than the evident 0-ary version of the binary additive conjunction just described. Its one inference rule, also translated to eliminate HLF-level additive type constructors, can be straightforwardly read as: in any linear logic context, $\top$ is provable in that context. As for the cut elimination cases, there is only one, since $\top$ possesses only a single left rule, and no right rule that could possibly create principal or $\mathcal{D}$-commutative cases to consider.

```
top : o.

%%% Inference Rules

topr : {a:w} conc top @ a.

%%% Principal Cases

%%% D-Commutative Cases

%%% E-Commutative Cases

cae/topr : {D:conc A @ P} ca A D ([a:w][h: (hyp A) @ a] topr (Q * a))
                          (topr (Q * P)).
```

If the implementation had its own internal additive unit `t`, we could have written instead

```
topr : t -o conc top.
```

which would have elaborated to what we wrote above.

6.1.4 Linear Implication

The following section give the rules and cut admissibility cases for linear implication $\multimap$. Following that, we treat tensor $\otimes$, disjunction $\oplus$, and their units 1 and 0, which contain no essential novelties.

```
lol  : o -> o -> o.  %infix right 10 lol.

%%% Inference Rules

loll : (hyp (A lol B) -o conc C)
        o- (hyp B -o conc C)
        o- conc A.

lolr : conc (A lol B)
           o- (hyp A -o conc B).
```

```
id/lol : id (A lol B)
          ([hfunc:^(hyp (A lol B))]
             lolr ^ ([harg:^(hyp A)] loll ^ (DA ^ harg) ^ DB ^ hfunc))
          <- id A DA
          <- id B DB.

%%% Principal Cases

ca/lol  : ca (A1 lol A2) (lolr ^ D2)
           ([h:^ (hyp (A1 lol A2))] loll ^ E1 ^ E2 ^ h) F
           <- ca A1 E1 D2 D2'
           <- ca A2 D2' E2 F.

%%% D-Commutative Cases

cad/loll   : ca A (loll ^ D1 ^ D2 ^ H) E (loll ^ D1 ^ D2' ^ H)
      <- ({a : w} {h2:hyp B2 @ a} ca A (D2 ^ h2) E (D2' ^ h2)).

%%% E-Commutative Cases

cae/lolr : ca A D ([h:^(hyp A)] lolr ^ (E2 ^ h)) (lolr ^ E2')
    <- ({a1:w}{h1:hyp B1 @ a1} ca A D
               ([a2:w][h2:hyp A @ a2] E2 a2 h2 a1 h1) (E2' a1 h1)).


cae/loll2 : ca A D ([h:^ (hyp A)] loll ^ E1 ^ (E2 ^ h) ^ H)
                                       (loll ^ E1 ^ E2' ^ H)
    <- ({a2 : w}{h2:hyp B2 @ a2} ca A D ([a1:w][h1: hyp A @ a1]
                 E2 a1 h1 a2 h2) (E2' a2 h2)).

car/loll1 : ca A D ([h:^ (hyp A)] loll ^ (E1 ^ h) ^ E2 ^ H)
                                       (loll ^ E1' ^ E2 ^ H)
    <- ca A D E1 E1'.
```

6.1.5 Multiplicative Conjunction

```
tensor  : o -> o -> o.  %infix right 11 tensor.

%%% Inference Rules

tensorl : (hyp (A tensor B) -o conc C)
   o- (hyp A -o hyp B -o conc C).
```

```
tensorr : conc (A tensor B)
   o- conc A
   o- conc B.

%%% Principal Cases

ca/tensor1 : ca (B1 tensor B2) (tensorr ^ D2 ^ D1)
      ([h:^ (hyp (B1 tensor B2))] tensorl ^ E ^ h) F
      <- ({a2 : w}{h2 : hyp B2 @ a2} ca B1 D1
    ([a1 : w][h1 : hyp B1 @ a1] E a1 h1 a2 h2) (E' a2 h2))
      <- ca B2 D2 ([a2 : w][h2 : hyp B2 @ a2] E' a2 h2) F.


%%% D-Commutative Cases

cad/tensorl  : ca A (tensorl ^ D ^ H) E (tensorl ^ D' ^ H)
<- ({a1 : w} {h1 : hyp _ @ a1}
    {a2 : w} {h2 : hyp _ @ a2}
      ca A (D a1 h1 a2 h2) E (D' a1 h1 a2 h2)).

%%% E-Commutative Cases

cae/tensorl  : ca A D
([a : w] [h : hyp _ @ a] tensorl ^ (E a h) ^ H)
(tensorl ^ E' ^ H)
<- ({a1 : w} {h1 : hyp _ @ a1}
    {a2 : w} {h2 : hyp _ @ a2}
      ca A D
      ([a : w] [h : hyp _ @ a] E a h a1 h1 a2 h2)
      (E' a1 h1 a2 h2)).

cae/tensorr1 : ca A D
       ([h :^ (hyp A)] tensorr ^ (E1 ^ h) ^ E2)
       (tensorr ^ E1' ^ E2)
       <- ca A D E1 E1'.

cae/tensorr2 : ca A D
       ([h :^ (hyp A)] tensorr ^ E1 ^ (E2 ^ h))
       (tensorr ^ E1 ^ E2')
       <- ca A D E2 E2'.
```

6.1.6 Multiplicative Unit

```
one : o.

%%% Inference Rules

oner : conc one.
onel : (hyp one -o conc C) o- conc C.

%%% Principal Cases

ca/one : ca one oner ([a:w][h : hyp one @ a] onel ^ D ^ h) D.

%%% D-Commutative Cases

cad/onel : ca A (onel ^ D ^ H) E (onel ^ F ^ H)
      <- ca A D E F.

%%% E-Commutative Cases

cae/onel: ca A D ([a] [x:hyp A @ a] onel ^ (E a x) ^ H) (onel ^ F ^ H)
    <- ca A D E F.
```

6.1.7 Disjunction

```
plus : o -> o -> o. %infix right 11 plus.

%%% Inference Rules

plusl : {a:w} ((hyp A -o conc C) @ a)
 -> ((hyp B -o conc C) @ a)
 -> ((hyp (A plus B) -o conc C) @ a).

plusr1 : conc A -o conc (A plus B).
plusr2 : conc B -o conc (A plus B).

%%% Principal Cases

ca/plus1 : ca (A1 plus A2) (plusr1 ^ D1)
   ([h:^ (hyp (A1 plus A2))] plusl ^ E1 E2 ^ h) F
   <- ca A1 D1 E1 F.
```

```
ca/plus2 : ca (A1 plus A2) (plusr2 ^ D2)
   ([h:^ (hyp (A1 plus A2))] plusl ^ E1 E2 ^ h) F
   <- ca A2 D2 E2 F.

%%% D-Commutative Cases

cad/plusl  : ca A (plusl ^ D1 D2 ^ H)
      E
      (plusl ^ E1' E2' ^ H)
      <- ({a:w} {h:hyp A1 @ a} ca A (D1 ^ h) E (E1' ^ h))
      <- ({a:w} {h:hyp A2 @ a} ca A (D2 ^ h) E (E2' ^ h)).

%%% E-Commutative Cases

cae/plusl :
     ca X D ([x:w][h: (hyp X) @ x] plusl ^ (E1 x h) (E2 x h) ^ H)
     (plusl ^ E1' E2' ^ H)
     <- ({a:w}{ha : hyp A1 @ a} ca X D ([x:w][hx: (hyp X) @ x]
                E1 x hx a ha) (E1' a ha))
     <- ({a:w}{ha : hyp A2 @ a} ca X D ([x:w][hx: (hyp X) @ x]
                E2 x hx a ha) (E2' a ha)).

cae/plusr1 :
     ca X D ([x:w][h: (hyp X) @ x] plusr1 ^ (E x h)) (plusr1 ^ F)
      <- ca X D E F.

cae/plusr2 :
     ca X D ([x:w][h: (hyp X) @ x] plusr2 ^ (E x h)) (plusr2 ^ F)
      <- ca X D E F.
```

6.1.8 Disjunctive Unit

```
zero : o.

%%% Inference Rules

zerol : {a:w} {b:w} hyp zero @ a -> conc C @ a * b.

%%% Principal Cases

%%% D-Commutative Cases
```

```
cad/zerol  : ca A (zerol ^ ^ H) E (zerol ^ ^ H).

%%% E-Commutative Cases

cae/zerol : ca A D ([h:^ (hyp A)] zerol ^ ^ H) (zerol ^ ^ H).
```

6.1.9 Checking Cut Admissibility

To check the correctness of the proof of cut admissibility, we specify to form of the context in which derivations are allowed to be constructed. This notion of *regular world* assumption is explained in Schürmann's thesis [Sch00b].

```
%block b : some {A : o} block {a : w} {x : hyp A @ a}.
%worlds (b) (ca _ _ _ _).
```

Here we describe the structure of how object language contexts are represented: we have said that the HLF context must consist of a series of repeated blocks of variables, each one of which consists of a world variable α : w, followed by a term variable x : *hyp* A @ α at that world, for some type A that may be different from one block to the next.

```
%total {A [D E]} (ca A D E F).
```

This *totality declaration* checks, by using coverage checking as we have described, as well as termination checking, that ca is a total relation. Specifically, by the mode declaration made earlier for ca, this means that for every A, D, and E of the appropriate type, there exists an F and a derivation M of type *ca* A D E F (made out of the constants declared immediately above) that is a witness that A, D, E, F together actually belongs to the relation ca. By Lemma 6.1.1, this amounts to the fact that cut admissibility actually holds for the linear sequent calculus.

6.2 Type Preservation in MiniML

Another example of using HLF is to verify type preservation of a simple programming language with updateable reference cells. Its syntax and type system are given taking advantage of standard LF encoding techniques, and its operational semantics by using substructural features as found in LLF and HLF.

For this result, we can use without any significant modification (we have changed the symbol used for product types from * to ** to avoid clash with the binary operation of world combination) the syntax and type system encodings described by [CP02]. We include them here.

6.2.1 Syntax

```
%% Types

tp : type.  %name tp T.

nat   : tp.
1     : tp.              % unit
**     : tp -> tp -> tp.  %infix right 10 **. % product
=>    : tp -> tp -> tp.  %infix right 9 =>. % function
rf    : tp -> tp.        % cells

exp  : type.  %name exp E.
val  : type.  %name val V.
cell : type.  %name cell C.
final: type.  %name final W.

%%% Expressions
% Natural Numbers
z    : exp.
s    : exp -> exp.
case : exp -> exp -> (val -> exp) -> exp.

% Unit
unit : exp.

% Pairs
pair : exp -> exp -> exp.
fst  : exp -> exp.
snd  : exp -> exp.

% Functions
lam  : (val -> exp) -> exp.
app  : exp -> exp -> exp.

% References
ref  : exp -> exp.
deref: exp -> exp.
assign: exp -> exp -> exp.
seq  : exp -> exp -> exp.

% Let and Polymorphism
let  : exp -> (val -> exp) -> exp.
letv : val -> (val -> exp) -> exp.

% Recursion
```

```
fix  : (exp -> exp) -> exp.

% Values
vl   : val -> exp.

%%% Values
z*   : val.
s*   : val -> val.
unit*: val.
pair*: val -> val -> val.
lam* : (val -> exp) -> val.
ref* : cell -> val.

%%% Final state
val* : val -> final.
new* : (cell -> final) -> final.
loc* : cell -> val -> final -> final.
```

6.2.2 Type system

```
%% Typing Judgments

ofc : cell -> tp -> type.        %name ofc Oc.
ofe : exp -> tp -> type.         %name ofe Oe.
ofv : val -> tp -> type.         %name ofv Ov.
off : final -> tp -> type.       %name off Of.
ofi : inst -> tp -> type.        %name ofi Oi.
ofk : cont -> tp -> tp -> type.  %name ofk Ok.

%%% Expressions

% Natural Numbers
ofe_z : ofe z nat.
ofe_s : ofe (s E) nat
 <- ofe E nat.
ofe_case: ofe (case E1 E2 E3) S
   <- ofe E1 nat
   <- ofe E2 S
   <- ({x:val} ofv x nat -> ofe (E3 x) S).

% Unit
ofe_unit : ofe unit 1.
```

```
% Pairs
ofe_pair : ofe (pair E1 E2) (T1 ** T2)
   <- ofe E1 T1
   <- ofe E2 T2.
ofe_fst  : ofe (fst E) T1
    <- ofe E (T1 ** T2).
ofe_snd  : ofe (snd E) T2
    <- ofe E (T1 ** T2).

% Functions
ofe_lam  : ofe (lam E) (T1 => T2)
    <- ({x:val} ofv x T1 -> ofe (E x) T2).
ofe_app  : ofe (app E1 E2) T1
    <- ofe E1 (T2 => T1)
    <- ofe E2 T2.

% References
ofe_ref  : ofe (ref E) (rf T)
    <- ofe E T.
ofe_deref : ofe (deref E) T
        <- ofe E (rf T).
ofe_assign : ofe (assign E1 E2) 1
        <- ofe E1 (rf T)
        <- ofe E2 T.
ofe_seq : ofe (seq E1 E2) T2
   <- ofe E1 T1
   <- ofe E2 T2.

% Let
ofe_let : ofe (let E1 E2) T2
   <- ofe E1 T1
   <- ({x:val} ofv x T1 -> ofe (E2 x) T2).

ofe_letv : ofe (letv V1 E2) T2
    <- ofe (E2 V1) T2.

% Recursion
ofe_fix : ofe (fix E) T
   <- ({u:exp} ofe u T -> ofe (E u) T).

% Values
ofe_vl  : ofe (vl V) T
   <- ofv V T.
```

```
%%% Values

ofv_z   : ofv (z*) nat.
ofv_s   : ofv (s* V) nat
   <- ofv V nat.
ofv_unit : ofv (unit*) 1.
ofv_pair : ofv (pair* V1 V2) (T1 ** T2)
    <- ofv V1 T1
    <- ofv V2 T2.
ofv_lam  : ofv (lam* E) (T1 => T2)
      <- ({x:val} ofv x T1 -> ofe (E x) T2).

ofv_ref  : ofv (ref* C) (rf T)
    <- ofc C T.

%%% Final States

off_val  : off (val* V) T
    <- ofv V T.
off_new  : off (new* W) T2
    <- ({c:cell} {d:ofc c T} off (W c) T2).
off_loc  : off (loc* C V W) T2
    <- ofc C T1
    <- ofv V T1
    <- off W T2.

%%% Abstract Machine Instructions

ofi_ev : ofi (ev E) T
  <- ofe E T.
ofi_return : ofi (return V) T
      <- ofv V T.

ofi_case1 : ofi (case1 V1 E2 E3) T
     <- ofv V1 nat
     <- ofe E2 T
     <- ({x:val} ofv x nat -> ofe (E3 x) T).

ofi_pair1 : ofi (pair1 V1 E2) (T1 ** T2)
     <- ofv V1 T1
     <- ofe E2 T2.

ofi_fst1 : ofi (fst1 V) T1
    <- ofv V (T1 ** T2).
```

```
ofi_snd1 : ofi (snd1 V) T2
    <- ofv V (T1 ** T2).

ofi_app1 : ofi (app1 V1 E2) T1
    <- ofv V1 (T2 => T1)
    <- ofe E2 T2.
ofi_app2 : ofi (app2 V1 V2) T1
    <- ofv V1 (T2 => T1)
    <- ofv V2 T2.

ofi_ref1 : ofi (ref1 V) (rf T)
    <- ofv V T.
ofi_deref1 : ofi (deref1 V) T
         <- ofv V (rf T).
ofi_assign1 : ofi (assign1 V1 E2) 1
         <- ofv V1 (rf T)
         <- ofe E2 T.
ofi_assign2 : ofi (assign2 V1 V2) 1
         <- ofv V1 (rf T)
         <- ofv V2 T.

ofi_let1 : ofi (let1 V1 E2) T2
    <- ofv V1 T1
    <- ({x:val} ofv x T1 -> ofe (E2 x) T2).

%%% Continuations
ofk_init : ofk init T T.
ofk_;    : ofk (K ; I) T1 T3
    <- ({x:val} ofv x T1 -> ofi (I x) T2)
    <- ofk K T2 T3.
```

6.2.3 Operational Semantics

None of the above uses any encoding techniques not already available in plain LF. The encoding of the operational semantics of the language, however, represents the imperative update of reference cells by making the fact that a reference cell currently holds a value into a linear hypothesis, so that when its value changes, the linear hypothesis for the old value is consumed, and a new one is hypothesized in its place.

The evaluation judgments are as follows.

```
ceval    : exp -> final -> @type.          %name ceval Ev.
exec     : cont -> inst -> final -> @type. %name exec Ex.
close    : final -> final -> @type.        %name close Ec.
contains : cell -> val -> @type.           %name contains Et.
```

```
read     : cell -> val -> @type.            %name read Ed.
```

The type family ceval is the top-level interface to evaluation of a closed expression. Given an expression, it outputs a term of type final, which consists of the returned value, together with bindings for the state of all reference cells at the end of the computation. exec is called to carry out execution of an instruction with respect to a continuation stack, yielding a final result. close is called to wrap up all the remaining reference-cell bindings at the end of a computation to yield a closed term of type final. The type contains is the type of which linear hypotheses are made to represent the fact of a reference cell containing a value.

The type family read reads the current value from a reference cell. It does this via rd, the only constant that inhabits read:

```
rd : {a:w} {b:w} (contains C V) @ b -> (read C V) @ (a * b).
```

This can be read in the following way. Suppose that we know the fact that reference cell C contains value V, and that this is associated with the world β. For any state of the entire collection of reference cells that can be partitioned into the form $\alpha * \beta$ — that is, for any state of the store that includes the fact that C contains V — attempting to read the value out of reference cell C will in fact succeed in reporting V as its value.

With the additive unit `t`, this could have been written as

```
rd : t -o contains C V -o read C V.
```

for the additive unit would then successfully consume all extra linear facts about the store that are not the fact that C contains V.

The clauses that define evaluation are as follows.

```
%%% Execution

% Natural Numbers
ex_z : exec K (ev z) W
 o- exec K (return z*) W.

ex_s : exec K (ev (s E1)) W
 o- exec (K ; [x1] return (s* x1)) (ev E1) W.

ex_case : exec K (ev (case E1 E2 E3)) W
    o- exec (K ; [x1] case1 x1 E2 E3) (ev E1) W.

ex_case1_z : exec K (case1 z* E2 E3) W
       o- exec K (ev E2) W.

ex_case1_s : exec K (case1 (s* V1) E2 E3) W
       o- exec K (ev (E3 V1)) W.

% Unit
ex_unit : exec K (ev (unit)) W
   o- exec K (return unit*) W.
```

```
% Pairs
ex_pair : exec K (ev (pair E1 E2)) W
    o- exec (K ; [x1] pair1 x1 E2) (ev E1) W.

ex_pair1 : exec K (pair1 V1 E2) W
     o- exec (K ; [x2] return (pair* V1 x2)) (ev E2) W.

ex_fst : exec K (ev (fst E1)) W
   o- exec (K ; [x1] fst1 x1) (ev E1) W.

ex_fst1 : exec K (fst1 (pair* V1 V2)) W
    o- exec K (return V1) W.

ex_snd : exec K (ev (snd E1)) W
   o- exec (K ; [x1] snd1 x1) (ev E1) W.

ex_snd1 : exec K (snd1 (pair* V1 V2)) W
    o- exec K (return V2) W.

% Functions
ex_lam : exec K (ev (lam E1)) W
   o- exec K (return (lam* E1)) W.

ex_app : exec K (ev (app E1 E2)) W
   o- exec (K ; [x1] app1 x1 E2) (ev E1) W.

ex_app1 : exec K (app1 V1 E2) W
    o- exec (K ; [x2] app2 V1 x2) (ev E2) W.

ex_app2 : exec K (app2 (lam* E1) V2) W
    o- exec K (ev (E1 V2)) W.

% References
ex_ref : exec K (ev (ref E1)) W
   o- exec (K ; [x1] ref1 x1) (ev E1) W.

ex_ref1 : exec K (ref1 V1) (new* W)
   o- ({c : cell} contains c V1
 -o exec K (return (ref* c)) (W c)).

ex_deref : exec K (ev (deref E1)) W
       o- exec (K ; [x1] deref1 x1) (ev E1) W.
```

```
ex_deref1 : {a:w}
     read C V1 @ a
       -> exec K (return V1) W @ a
     -> exec K (deref1 (ref* C)) W @ a.

ex_assign : exec K (ev (assign E1 E2)) W
       o- exec (K ; [x1] assign1 x1 E2) (ev E1) W.

ex_assign1 : exec K (assign1 V1 E2) W
        o- exec (K ; [x2] assign2 V1 x2) (ev E2) W.

ex_assign2 : exec K (assign2 (ref* C1) V2) W
        o- contains C1 V1
        o- (contains C1 V2 -o exec K (return unit*) W).

ex_seq : exec K (ev (seq E1 E2)) W
  o- exec (K ; [x1] ev E2) (ev E1) W.

% Let
ex_let : exec K (ev (let E1 E2)) W
   o- exec (K ; [x1] let1 x1 E2) (ev E1) W.

ex_let1 : exec K (let1 V1 E2) W
    o- exec K (ev (E2 V1)) W.

ex_letv : exec K (ev (letv V1 E2)) W
   o- exec K (ev (E2 V1)) W.

% Recursion
ex_fix : exec K (ev (fix E1)) W
   o- exec K (ev (E1 (fix E1))) W.

% Values
ex_vl : exec K (ev (vl V)) W
    o- exec K (return V) W.

ex_return : exec (K ; C) (return V) W
    o- exec K (C V) W.

ex_init : exec init (return V) W
   o- close (val* V) W.

%%% Collecting the final state
```

```
% In LLF close_done should come last and will only
% succeed if there are no "contains" assumptions
% left in the state.
close_done : close W W.

close_loc : close W1 W
     o- contains C1 V1
     o- close (loc* C1 V1 W1) W.

%%% Top-level evaluation

cev : ceval E V
       o- exec init (ev E) V.
```

The most interesting ones from the perspective of using linearity to represent state change are the following three.

```
ex_ref1 : exec K (ref1 V1) (new* W)
           o- ({c : cell} contains c V1 -o
                  exec K (return (ref* c)) (W c)).

ex_deref1 : {a:w}
             read C V1 @ a
             -> exec K (return V1) W @ a
             -> exec K (deref1 (ref* C)) W @ a.

ex_assign2 : exec K (assign2 (ref* C1) V2) W
              o- contains C1 V1
              o- (contains C1 V2 -o exec K (return unit*) W).
```

The constant `ex_ref1` creates a new reference cell by means of the nested Π-quantifier `{c : cell}`, and assigns it a new value by linearly hypothesizing `contains c V1`. Dereferencing takes place in `ex_deref1`, where the current world is effectively copied and passed along into two subgoals, one which reads out the current value of the reference cell by calling `read`, and the other which continues the rest of the computation, using that value. Assignment works by consuming the resource used to prove `contains C1 V1`, which is to say, whatever the old value of `C1` was, and hypothesizes in the remainder of the computation a new value as a linear hypothesis `contains C1 V2`.

6.2.4 Type Preservation

This section contains the proof of type preservation of MiniML, that if an expression is well-typed, and evaluates to a result, then that result is well-typed, and in fact has the same type as the original expression.

The main theorems to show are the following:

```
tpev : ceval E W @ P -> ofe E T -> off W T -> type.
tpex : {P:w} exec K I W @ P -> ofk K T S -> ofi I T -> off W S -> type.
tpcl : close W W' @ P -> off W T -> off W' T -> type.
tpct : contains C V @ P -> ofc C T -> ofv V T -> type.
tprd : read C V @ P -> ofc C T -> ofv V T -> type.

%mode (tpev +Ev +Oe -Of).
%mode (tpex +P +Ex +Ok +Oi -Of).
%mode (tpcl +Ec +Of -Of').
%mode (tpct +Et -Oc -Ov).
%mode (tprd +Ed -Oc -Ov).
```

Each one says something of the following form: for every possible store, represented by the world expression `P`, if some form evaluation (be it of a closed expression, stack machine state, etc.) takes place, then it preserves types.

There is an important lemma to show as well, which is essentially a claim of uniqueness of typing of reference cells and the values they hold during the course of execution.

```
eqlemma : {C : cell} ofv V T1 -> ofc C T1 -> ofc C T2 -> ofv V T2 -> type.
%mode (eqlemma +C +Ov1 +Oc1 +Oc2 -Ov2).
```

Specifically, if a reference cell is statically known to have type `T1`, and is *also* statically known to have type `T2`, then the two types must be the same, and we can therefore transfer knowledge that a value has type `T1` to the knowledge that it has type `T2`.

```
%%% Reading the state without affecting it

tprd_rd :  tprd (rd ^ ^ CONT) OFC Ov
     <- tpct CONT OFC Ov.

%%% Execution

% Natural Numbers
tpex_z : tpex ^ (ex_z ^ Ex1) Ok (ofi_ev (ofe_z)) Of
  <- tpex ^ Ex1 Ok (ofi_return (ofv_z)) Of.

tpex_s : tpex ^ (ex_s ^ Ex1) Ok (ofi_ev (ofe_s Oe1)) Of
  <- tpex ^ Ex1 (ofk_; Ok ([x1:val] [ov1:ofv x1 nat]
   ofi_return (ofv_s ov1)))
          (ofi_ev Oe1) Of.

tpex_case : tpex ^ (ex_case ^ Ex1) Ok (ofi_ev (ofe_case Oe3 Oe2 Oe1)) Of
     <- tpex ^ Ex1 (ofk_; Ok ([x1:val] [ov1:ofv x1 nat]
      ofi_case1 Oe3 Oe2 ov1))
     (ofi_ev Oe1) Of.

tpex_case1_z : tpex ^ (ex_case1_z ^ Ex1) Ok (ofi_case1 Oe3 Oe2 (ofv_z)) Of
```

```
<- tpex ^ Ex1 Ok (ofi_ev Oe2) Of.

tpex_case1_s : tpex ^ (ex_case1_s ^ Ex1) Ok (ofi_case1 Oe3 Oe2 (ofv_s Ov1)) Of
<- tpex ^ Ex1 Ok (ofi_ev (Oe3 V1 Ov1)) Of.

% Unit
tpex_unit : tpex ^ (ex_unit ^ Ex1) Ok (ofi_ev (ofe_unit)) Of
       <- tpex ^ Ex1 Ok (ofi_return (ofv_unit)) Of.

% Pairs
tpex_pair : tpex ^ (ex_pair ^ Ex1) Ok (ofi_ev (ofe_pair Oe2 Oe1)) Of
     <- tpex ^ Ex1 (ofk_; Ok ([x1:val] [ov1:ofv x1 T1]
      ofi_pair1 Oe2 ov1))
     (ofi_ev Oe1) Of.

tpex_pair1 : tpex ^ (ex_pair1 ^ Ex1) Ok (ofi_pair1 Oe2 Ov1) Of
      <- tpex ^ Ex1 (ofk_; Ok ([x2:val] [ov2:ofv x2 T2]
       ofi_return (ofv_pair ov2 Ov1)))
      (ofi_ev Oe2) Of.

tpex_fst : tpex ^ (ex_fst ^ Ex1) Ok (ofi_ev (ofe_fst Oe1)) Of
    <- tpex ^ Ex1 (ofk_; Ok ([x1:val] [ov1:ofv x1 (T1 ** T2)]
     ofi_fst1 ov1))
            (ofi_ev Oe1) Of.

tpex_fst1 : tpex ^ (ex_fst1 ^ Ex1) Ok (ofi_fst1 (ofv_pair Ov2 Ov1)) Of
     <- tpex ^ Ex1 Ok (ofi_return Ov1) Of.

tpex_snd : tpex ^ (ex_snd ^ Ex1) Ok (ofi_ev (ofe_snd Oe1)) Of
    <- tpex ^ Ex1 (ofk_; Ok ([x1:val] [ov1:ofv x1 (T1 ** T2)]
     ofi_snd1 ov1))
            (ofi_ev Oe1) Of.

tpex_snd1 : tpex ^ (ex_snd1 ^ Ex1) Ok (ofi_snd1 (ofv_pair Ov2 Ov1)) Of
     <- tpex ^ Ex1 Ok (ofi_return Ov2) Of.

% Functions
tpex_lam : tpex ^ (ex_lam ^ Ex1) Ok (ofi_ev (ofe_lam Oe1)) Of
      <- tpex ^ Ex1 Ok (ofi_return (ofv_lam Oe1)) Of.

tpex_app : tpex ^ (ex_app ^ Ex1) Ok (ofi_ev (ofe_app Oe2 Oe1)) Of
      <- tpex ^ Ex1 (ofk_; Ok ([x1:val] [ov1:ofv x1 (T2 => T1)]
       ofi_app1 Oe2 ov1))
              (ofi_ev Oe1) Of.
```

```
tpex_app1 : tpex ^ (ex_app1 ^ Ex1) Ok (ofi_app1 Oe2 Ov1) Of
       <- tpex ^ Ex1 (ofk_; Ok ([x2:val] [ov2:ofv x2 T2]
        ofi_app2 ov2 Ov1))
       (ofi_ev Oe2) Of.

tpex_app2 : tpex ^ (ex_app2 ^ Ex1) Ok (ofi_app2 Ov2 (ofv_lam Oe1)) Of
       <- tpex ^ Ex1 Ok (ofi_ev (Oe1 V2 Ov2)) Of.

% References

tpex_ref : tpex ^ (ex_ref ^ Ex1) Ok (ofi_ev (ofe_ref Oe1)) Of
      <- tpex ^ Ex1 (ofk_; Ok ([x1:val] [ov1:ofv x1 T1] ofi_ref1 ov1))
       (ofi_ev Oe1) Of.

tpex_ref1 : tpex ^ (ex_ref1 ^ Ex1) Ok
                 (ofi_ref1 (Ov1 : ofv V1 T1)) (off_new Of)
       <- ({c:cell} {dc:ofc c T1} {a:w} {et : contains c V1 @ a}
     tpct et dc Ov1 ->
     ({V : val} {Ov : ofv V T1} eqlemma c Ov dc dc Ov) ->
      tpex ^ (Ex1 c ^ et) Ok (ofi_return (ofv_ref dc)) (Of c dc)).

tpex_deref : tpex ^ (ex_deref ^ Ex1) Ok (ofi_ev (ofe_deref Oe1)) Of
         <- tpex ^ Ex1 (ofk_; Ok ([x1:val] [Ov1:ofv x1 (rf T1)]
          ofi_deref1 Ov1))
         (ofi_ev Oe1) Of.

tpex_deref1 : tpex ^ (ex_deref1 ^ Rd1 Ex1) Ok
                           (ofi_deref1 (ofv_ref Oc')) Of
       <- tprd Rd1 Oc Ov
       <- eqlemma C Ov Oc Oc' Ov'
       <- tpex ^ Ex1 Ok (ofi_return Ov') Of.

tpex_assign : tpex ^ (ex_assign ^ Ex1) Ok (ofi_ev (ofe_assign Oe2 Oe1)) Of
         <- tpex ^ Ex1 (ofk_; Ok ([x1:val] [Ov1:ofv x1 (rf T)]
  ofi_assign1 Oe2 Ov1))
    (ofi_ev Oe1) Of.

tpex_assign1 : tpex ^ (ex_assign1 ^ Ex1) Ok (ofi_assign1 Oe2 Ov1) Of
  <- tpex ^ Ex1 (ofk_; Ok ([x2:val] [Ov2:ofv x2 T]
   ofi_assign2 Ov2 Ov1))
          (ofi_ev Oe2) Of.

tpex_assign2 : tpex ^ (ex_assign2 ^ Ex2 ^ Et1) Ok
```

```
                    (ofi_assign2 Ovnew (ofv_ref OFC)) Of
<- tpct Et1 OFC' Ovold
<- eqlemma _ Ovnew OFC OFC' Ovnew'
  <- ({a:w} {et2:contains C Vnew @ a}
        tpct et2 OFC' Ovnew'
        -> tpex ^ (Ex2 ^ et2) Ok (ofi_return (ofv_unit)) Of).

tpex_seq : tpex ^ (ex_seq ^ Ex1) Ok (ofi_ev (ofe_seq Oe2 Oe1)) Of
    <- tpex ^ Ex1 (ofk_; Ok ([x1:val] [Ov1:ofv x1 T1]
     ofi_ev Oe2))
            (ofi_ev Oe1) Of.

% Let
tpex_let : tpex ^ (ex_let ^ Ex1) Ok (ofi_ev (ofe_let Oe2 Oe1)) Of
    <- tpex ^ Ex1 (ofk_; Ok ([x1:val] [ov1:ofv x1 T1]
     ofi_let1 Oe2 ov1))
            (ofi_ev Oe1) Of.

tpex_let1 : tpex ^ (ex_let1 ^ Ex2) Ok (ofi_let1 Oe2 Ov1) Of
     <- tpex ^ Ex2 Ok (ofi_ev (Oe2 V1 Ov1)) Of.

tpex_letv : tpex ^ (ex_letv ^ Ex2) Ok (ofi_ev (ofe_letv Oe2)) Of
     <- tpex ^ Ex2 Ok (ofi_ev Oe2) Of.

% Recursion
tpex_fix  : % {Oe: {u:exp} ofe u T -> ofe (E u) T}
     tpex ^ (ex_fix ^ Ex1) Ok (ofi_ev (ofe_fix Oe)) Of
     <- tpex ^ Ex1 Ok (ofi_ev (Oe (fix E) (ofe_fix Oe))) Of.

% Values
tpex_vl   : tpex ^ (ex_vl ^ Ex1) Ok (ofi_ev (ofe_vl Ov)) Of
     <- tpex ^ Ex1 Ok (ofi_return Ov) Of.

tpex_return : {Ov:ofv _ T1}
       tpex ^ (ex_return ^ Ex1) (ofk_; Ok Oc) (ofi_return Ov) Of
       <- tpex ^ Ex1 Ok (Oc V Ov) Of.

tpex_init : tpex ^ (ex_init ^ Ec1) (ofk_init) (ofi_return Ov) Of
     <- tpcl Ec1 (off_val Ov) Of.

%%% Collecting the final state

tpcl_done : tpcl (close_done) Of Of.
```

```
tpcl_loc  : tpcl (close_loc ^ Ec2 ^ Et1) Of Of'
     <- tpct Et1 Oc Ov1
     <- tpcl Ec2 (off_loc Of Ov1 Oc) Of'.

%%% Evaluation

tpev_cev : tpev (cev ^ Ex) Oe Of
    <- tpex ^ Ex (ofk_init) (ofi_ev Oe) Of.
```

6.2.5 Checking Type Preservation

The end of the development again contains of block declarations describing the structure required of the context when checking totality of the relations that constitute the metatheorems we are interested in. These declarations are more involved than those for cut admissibility, because we need to cases for the metatheorems tpct and eqlemma that deal explicitly with hypotheses of type contains representing stateful reference cell values as they are introduced.

```
%block b :
some {V1 : val} {T1 : tp} {Ov1 : ofv V1 T1}
block {c:cell} {dc : ofc c T1} {a:w}
    {et : contains c V1 @ a}
    {d : tpct et dc Ov1}
    {d' : {V : val} {Ov : ofv V T1} eqlemma c Ov dc dc Ov}.

%block b' :
some  {T2 : tp} {C : cell} {V2 : val} {Oc : ofc C T2} {Ov2 : ofv V2 T2}
block {a:w}
     {et2 : contains C V2 @ a}
     {d : tpct et2 Oc Ov2}.

%worlds (b | b')
 (tpct _ _ _)
 (tpcl _ _ _)
 (tpex _ _ _ _ _)
 (tprd _ _ _)
 (tpev _ _ _)
 (eqlemma _ _ _ _ _).
```

Following that, the implementation can check that eqlemma and the mutually recursive type-preservation metatheorems are total as relations: type preservation for MiniML holds.

```
%total C (eqlemma C _ _ _ _).
```

```
%total (Et Ec Ex Ed Ev)
 (tpct Et _ _)
 (tpcl Ec _ _)
 (tpex _ Ex _ _ _)
 (tprd Ed _ _)
 (tpev Ev _ _).
```

Chapter 7

Conclusion

We have presented the type theory of HLF, an extension of LF with hybrid type operators, which admits an encoding methodology similar to that of LF, where the derivations of a deductive system are represented as the possible forms of well-typed expressions over a signature of typed constants, and metatheorems about such a deductive system can be represented as total, functional relations between the types of derivations, implemented in the style of logic programming. The newly added hybrid type operators, since they provide a decomposition and explanation of the linear function type, mean that we can recover many of the representational felicities of linear logic the same way that LLF does: representations of deductive systems of a naturally stateful character can be achieved with greater simplicity and elegance than in plain LF, by describing an isolated state change as the consumption of resources that constitute the old state.

The chief advantage of HLF over linear LF is that its hybrid type operators let us be more explicit about specific states when it is necessary. Since forming a kind that classifies a type family in the signature is the type-theoretic content of describing the assumptions and conclusions of a metatheorem, HLF's richer language of kinds allows the effective statement of more metatheorems, by using kind-level quantifiers over world variables to express universal quantification over object-language states. It may even be understating our case to say 'more', since it seems the majority of informal metatheorems encountered in practice *need* to mention in some form or another 'the current state' or entire substructural contexts as first-class objects. Hybrid type constructors make this mode of description possible, and quite natural.

7.1 Future Work

There are several directions in which we might like to proceed, taking this work as a foundation. We discuss here the possibilities of other encoding techniques that seem newly possible in HLF, the challenge of incorporating positive logical connectives into the type theory, and the issues involved in representing other substructural resources disciplines besides the one underlying linear logic.

7.1.1 Encoding Substructural Names

Joint work with Robert Harper has suggested that there is an application of HLF to the representation of the notion of *names* as found in, for example, nominal logic. Nominal logic [Pit03] offers a set of logical tools for manipulating a notion of names meant to be invariant under appropriate renamings. The specific claim about nominal logic that we wish to address is that it offer a more convenient way to reason when one requires positive evidence of *disequality* of names, compared to, say, higher order abstract syntax. This claim is made and illustrated with several examples by Cheney and Urban [CU06].

Observe that the resource discipline in substructural logics such as linear logic also latently possesses the means for describing apartness of different hypotheses. For in the data given to a linear function of many arguments, linearity requires that we may not use a hypothesis more than once. In this way, we can see that the arguments possess a notion of disjointness from one another, in that it is legitimate to consume arguments A and B if A and B are different, but it is not legitimate to consume A and A, for this consumes A twice. The goal is to turn this observation into an encoding technique, in which requiring apartness of names is as straightforward and simple as it is nominal logic.

One challenge is the fact that one immediately runs into the necessity of combining the resource-consuming behavior of linear types with dependent types. The relation # (for which $x\#y$ means 'x apart from y') appears to want to be a type family whose kind is something like $\mathit{name} \multimap \mathit{name} \multimap \mathsf{type}$ — since we mean to treat names somehow linearly.

HLF's separation of resource use from type dependency offers a solution to this problem in a similar way to the statement of linear cut admissibility. By using HLF's hybrid type operators for labelling specific linear resources with worlds and quantifying over worlds. Just as $\multimap$ was encoded using hybrid type operators, we can provide definitions for several convenient function types, so that the apartness relation can given a suitable HLF kind and axiomatized quite straightforwardly by the declaration of a single constant.

Just as we treated the linear function space $A \multimap B$ is a macro that expands to

$$\Pi\alpha{:}\mathsf{w}.{\downarrow}\beta.A@\alpha \to B@(\alpha * \beta)$$

we define the following types as abbreviations.

- The function space $A \not\multimap B$ consists of functions that promise to use their argument exactly *zero* times. This is accomplished by imitating the definition of the linear function space, but instead of putting one copy of the world α that labels the hypothesis into the world of the conclusion, we omit it. The definition is

  $$A \not\multimap B = \Pi\alpha{:}\mathsf{w}.A@\alpha \to B$$

  The fact that no α appears in B makes it possible to also extend this to a definition of a kind constructor $A \not\multimap K$, defined in a similar way

  $$A \not\multimap K = \Pi\alpha{:}\mathsf{w}.A@\alpha \to K$$

- A linear Π, written $\Pi x\hat{:}A.B$, as

$$\Pi x\hat{:}A.B = \Pi\alpha{:}\mathsf{w}.{\downarrow}\beta.\Pi x\hat{:}A@\alpha.B@(\alpha * \beta)$$

of which $\multimap$ is a trivialization when x does not appear in B.

With these, nominal logic's notion of apartness becomes quite simply definable in an HLF signature by declaring

$$\begin{array}{l} \mathit{name} : \mathit{type}. \\ \# : \mathit{name} \not\multimap \mathit{name} \not\multimap \mathsf{type}\,. \\ \mathit{irrefl} : \Pi x\hat{:}\mathit{name}.\Pi x'\hat{:}\mathit{name}.\top \multimap \# \hat{} x \hat{} x' \end{array}$$

With these declarations, # simply is the apartness relation on names; that is, in an HLF context consisting of a collection of linear assumptions of names, by which we mean one of the form

$$\Gamma = \alpha_1 : \mathsf{w}, n_1 : \mathit{name} \mathbin{@} \alpha_1, \cdots, \alpha_n : \mathsf{w}, x_n : \mathit{name} \mathbin{@} \alpha_n$$

there exists a term M such that $\Gamma \vdash M \Leftarrow \# \hat{} x_i \hat{} x_j[\alpha_1 * \cdots * \alpha_n]$ if and only if x_i and x_j are distinct. In point of fact, that term will be a use of the irreflexivity axiom $\mathit{irrefl} \hat{} x_i \hat{} x_j \langle\rangle\ q$ where q is the unique world such that $q * x_i * x_j \equiv_{\mathsf{acu}} \alpha_1 * \cdots * \alpha_n$.

Apart from the technical details, the essential point to take away from this sketch is again that HLF appears to allow a rich set of substructural function spaces to exist and to interact in useful ways with dependent types, which hold the possibility of capturing type-theoretic features like names that on their surface had little to do with substructural logic.

7.1.2 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.

An obstacle for a similar, relatively straightforward extension of HLF with positive connectives is the fact that we have already shown how to simulate the additive arrow in BI as

$${\downarrow}\alpha.A@\alpha \to B@\alpha$$

for there are quite reasonable claims [O'H03] based on category-theoretic reasoning that the full versions of bunched and linear logic should be 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.

7.1.3 Other Algebras, Other Logics

There does seem to be another possible approach, however, one that incorporates both a full range of logical connectives both negative and positive, and seems to be parametrizable over an algebraic equational theory. In joint work with Frank Pfenning [RP09] we have begun to investigate a logic (not yet a logical framework) which has a similar notion of worlds, and also a dual notion of frames, which are related to positive propositions in the same way that worlds are related to negative propositions. The problem suggested above is avoided because the encoding of bunched logic's additive arrow is no longer valid in the presence of (linear) positive connectives, but the entirety of bunched logic can be separately encoded more directly, by axiomatizing an algebraic and order-theoretic structure on the worlds and frames in the representation language. In fact, it is quite possible that these ideas eventually naturally converge on a level of generality comparable to that of display logic [Bel82], except that they replace display logic's intrinsically substructural context (just as we replaced linear logic's intrinsically substructural context) with a resource discipline enforced by separate world labels.

The story we hope to eventually tell with this direction of work is one where common ideas in the semantics of substructural logics can be detached from their usual classical and model-theoretic surroundings, and be rehabilitated in a purely constructive setting — and what's more, as the results in [RP09] show, we get not only a constructive semantics, but also one that is compatible with focusing [And92]. And while 'constructive semantics' may sound like an oxymoron to anyone firmly committed to that which either word in the phrase connotes, there are clear benefits to be gained from each that we believe can be achieved simultaneously.

Constructivity means compatibility with judgments-as-types, and a reasonable notion of proofs as a tractable data structure, and the analysis of logical connectives into simpler algebraic operations typical of semantic approaches is what we have used to make substructural types more deeply compatible with dependent types, and made it possible to continue using the successful logical framework methodologies discussed above. The modern landscape of substructural logics is extremely rich in variety of ideas and useful applications, and we believe that the study of logical frameworks can and should benefit from it.

Bibliography

[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.

[BBC97] Alexandre Boudet, Re Boudet, and Evelyne Contejean. Ac-unification of higher-order patterns. In *In G. Smolka (Ed), Proc. Principles and Practice of Constraint Programming – CP'97, Lecture*, pages 267–281. Springer-Verlag, 1997.

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

[Bel82] Nuel Belnap. Display logic. *Journal of philosophical logic*, 11:375–417, 1982.

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

[Bou93] Gérard Boudol. The lambda-calculus with multiplicities. Technical Report 2025, INRIA Sophia, 1993.

[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.

[CP97] 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.

[Cra09] Karl Crary. Higher-order representation of substructural logics. Technical report, Carnegie Mellon University, 2009. http://www.cs.cmu.edu/ crary/papers/2009/substruct.pdf.

[CU06] James Cheney and Christian Urban. Nominal logic programming, 2006.

[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.

[Ell89] Conal Elliott. Higher-order unification with dependent types. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, 1989.

[Ell90] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1990. Available as Technical Report CMU-CS-90-134.

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

[GB99] Herman Geuvers and Erik Barendsen. Some logical and syntactical observations concerning the first-order dependent type system λp. *Mathematical. Structures in Comp. Sci.*, 9(4):335–359, 1999.

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

[Gir95] J.-Y. Girard. Linear logic: Its syntax and semantics. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic (Proc. of the Workshop on Linear Logic, Cornell University, June 1993)*. Cambridge University Press, 1995.

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

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[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 equivalence and canonical forms in the lf type theory. Technical report, ACM Transactions on Computational Logic, 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.

[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.

[LC94] Patrick Lincoln and Jim Christian. Adventures in associative-commutative unification. *Journal of Symbolic Computation*, 8:393–416, 1994.

[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.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[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.

[MP92] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Proceedings of the Workshop on the Prolog Programming Language*, pages 257–271, 1992.

[MS03] Andrew McCreight and Carsten Schürmann. A meta linear logical frame-

work. Draft manuscript, 2003.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In *Proceedings of Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 64–74, 1993.

[NMB06] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 62–73, New York, NY, USA, 2006. ACM.

[NPP05] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual Modal Type Theory. *ACM Transactions on Computational Logic*, 2005.

[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.

[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.

[Pit03] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

[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.

[PS98] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 179–193, Kloster Irsee, Germany, March 1998. Springer-Verlag LNCS 1657.

[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.

[Pym90] D.J. Pym. *Proofs, search and computation in general logic*. PhD thesis, University of Edinburgh, 1990.

[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.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, 2002.

[RP09] Jason Reed and Frank Pfenning. A constructive approach to the resource semantics of substructural logics. Unpublished manuscript. Available at http://www.cs.cmu.edu/∼jcreed/papers/rp-substruct.pdf, 2009.

[Sch00a] 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.

[Sch00b] Carsten Schürmann. Automating the meta-theory of deductive systems. Technical Report CMU-CS-00-146, Department of Computer Science, Carnegie Mellon University, 2000.

[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.

[Sti06] Colin Stirling. A game-theoretic approach to deciding higher-order matching. In *ICALP*, pages 348–359, 2006.

[VC02] J.C. Vanderwaart and K. Crary. A simplified account of the metatheory of linear LF. *Electronic Notes in Theoretical Computer Science*, 70(2):11–28, 2002.

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

[WBF93] D.A. Wright and C.A. Baker-Finch. Usage Analysis with Natural Reduction Types. In *Proceedings of the Third International Workshop on Static Analysis*, pages 254–266. Springer-Verlag London, UK, 1993.

[WCPW03a] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, 2003.

[WCPW03b] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, 2003.