Modern Logical Frameworks Design

Agata Anna Murawska

Advisor: Carsten Schürmann
Submitted: August 2017

IT University of Copenhagen

Abstract

Throughout the years, logical frameworks have been a successful tool for prototyping and developing a number of logics and programming languages. However, to use the full power of the LF approach, the behaviour of variables in the system being mechanised must match the behaviour of those used in the LF variant. While this is not a problem when working with more standard logical systems, it severely limits the usefulness of LF-like frameworks for less typical applications. A more practical consideration is that many of the existing frameworks do not have implementations, and even those that do lack support for reasoning about, or programming with, the mechanised systems.

Our main motivation is to eventually make it possible to model and reason about complex concurrent systems and protocols. No matter the application, be it the development of a logic for multiparty session types or a cryptographic protocol used in a voting system, we need the ability to model and reason about both the building blocks of these systems and the intricate connections between them.

To this end, this dissertation is an investigation into LF-based formalisms that might help address the aforementioned issues. We design and provide the meta-theory of two new frameworks, HyLF and Lincx. The former aims to extend the expressiveness of LF to include proof irrelevance and some user-defined behaviours, using ideas from hybrid logics. The latter is a showcase for an easier to implement framework, while also allowing more properties of the mechanised system to be expressed.

Resumé

Gennem årene har »logical frameworks« været et succesfuldt værktøj til at udvikle mange logikker og programmeringssprog, samt at skabe prototyper til disse. For at kunne benytte hele styrken bag LF-metodikken er det dog nødvendigt, at variabler i systemet der mekaniseres har adfærd, der passer til LF-variantens egne. Selv om dette ikke er noget problem når man arbejder med almindelige logiske systemer, er det en voldsom begrænsning når det gælder mindre udbredte anvendelser. Der er også visse praktiske problemer: de fleste frameworks mangler implementeringer, og de få, der er blevet implementeret, understøtter ikke at man ræsonnerer over, eller programmerer med, det mekaniserede system.

Vores primære mål med dette arbejde er at muliggøre, at man kan mekanisere og ræsonnere over komplekse parallelle systemer og protokoller. Uanset om formålet er at udvikle en logik for »multiparty session types« eller en kryptografisk protokol benyttet i et valgsystem, skal vi kunne mekanisere og ræsonnere over både disse systemers isolerede bestanddele og deres indviklede forbindelser.

Med dette formål er denne afhandling en undersøgelse af LF-baserede formalismer, der måske kan være med til at løse de ovennævnte problemer. Vi designer og præsenterer metateorier for to nye frameworks, nemlig HyLF og Lincx. Det første har som mål at udvide LFs udtryksfuldhed med »proof irrelevance« og visse brugerdefinerede opførelser, baseret på idéer fra hybride logikker. Det andet er et eksempel på et framework, der er nemmere at implementere, samtidig med at det muliggør, at flere af det mekaniserede systems egenskaber kan udtrykkes.

Acknowledgements

Throughout the writing process for this dissertation, I kept repeating to myself that "I do like writing". I do think this remains true now that the thesis is finished, and it would definitely not be the case without the support of some.

First, my supervisor, Carsten Schürmann, deserves gratitude for his ongoing support, as well as his patience with me, and allowing me the freedom to work on things that I found interesting.
Second, a big thanks to Nicolas Guenot, with whom I worked throughout the first two and a half years of my PhD, for introducing me to the wonderful world of proof theory, and teaching me the more magical parts of LaTeX. None of this would have been well-typeset without his help!
I am also extremely grateful to Brigitte Pientka for her warm welcome in Montreal during my research stay abroad. Not only was the collaboration there very fruitful, I consider my visit to McGill one of the most amazing learning experiences. *And* I got to write some OCaml!

My co-authors, colleagues and friends from Copenhagen, Montreal, Wrocław and elsewhere are too numerous to list (I'm sure I'd forget some, and be *really* embarrassed about it), but you know who you are, and I know that I owe you!

My eternal gratitude goes to Alec Faithfull, for his love, patience and support. Not to mention spellchecking and proof reading the entire thesis, and finding countless little ways to keep me (relatively) sane and well-rested in between writing and more writing.
Alec also translated the abstract of this thesis to Danish, and Taus Brock-Nannestad made sure it actually sounded good enough for a native Danish speaker *and* an LF expert.

Thank you.

Contents

Prelude

The ambition to obtain correct and well-behaved systems was present from the early days of computer science. Equally early it was understood that strong guarantees of correctness would come not from pencil-and-paper proofs, but rather from careful mechanization efforts using dedicated tools for theorem proving. As the programs we want to model and the properties we want to verify about them become more complex and intricate, the tools for modelling and reasoning about them must also become more adaptive and flexible to keep up with the growing complexity of the systems being modelled. Theorem provers like Agda and Coq are general enough to encode almost arbitrarily complex systems, although this generality comes at a cost. Fitting a system into such a framework often requires a heavy administrative burden, and with each change in a system whose properties we want to mechanise, these administrative steps have to be repeated and potentially even re-designed.

The logical framework approach to modelling and reasoning shifts some of this burden from the user of the system to its developer. There is, however, a trade-off: a concrete LF-like framework is well-suited only for a very specific subclass of problems. It is probably for this reason that the LF methodology seems to be lagging behind a little when compared to the development of Coq or Agda: each new development in the LF *world* is a separate, self-contained instance, rather than a module that might be combined with existing tools and frameworks. Perhaps it would be of value for the LF community to focus part of their efforts on adding more expressive power to the frameworks being built, potentially even at the cost of moving part of the maintenance burden to the user.

To this end, the work presented here is an investigation into possible solutions which, while still adhering to the LF methodology, allow modelling larger classes of systems. Additionally, a more uniform way of reasoning within different logical frameworks with some shared properties is of interest.

One concrete area in which these developments might be appreciated is concurrency theory, in particular reasoning about voting systems. For instance, the DemTech project is interested in, amongst other things, developing trustworthy protocols for elections. This can clearly be considered as a concurrent, multi-party system. On some level, it is also an example of resource-aware computation, where each ballot that has been cast must necessarily be counted – precisely once. (The latter constraint holds even when modelling non-electronic protocols considered in voting.) Finally, one might be interested in formally reasoning about the trust that we put in such a system, using epistemic logics. The eventual answers to these questions may well be beyond the scope of this dissertation, but they should be kept in mind as part of the motivation for this work.

Synopsis

We begin this dissertation by covering the background material important for both of the subsequent parts. This includes an introduction to the logical framework methodology and to linear logic. A more thorough description of the Linear Logical Framework LLF is then presented, as both the HyLF and Lincx frameworks, described in subsequent parts, rely crucially on ideas introduced in LLF.

The next part is dedicated to HyLF, Hybrid Logical Framework. It begins by presenting the design of HLF, a framework for linear logic using ideas from hybrid logic to encode linearity; these ideas were, in turn, a conceptual foundation for HyLF. The second chapter goes on to present a slightly revised version of our LFMTP 2014 paper which introduced HyLF.

The last part investigates a more general approach to logical frameworks, which is that of Contextual Modal LF. Both this variant of LF and its implementation in the Beluga system are discussed in the first chapter of this part. The next chapter presents our ESOP 2017 paper, which introduced a linear contextual framework Lincx.

Part I

Background

Chapter 1

Logical Framework Methodology

Modelling logics, programming languages and other complex systems to verify their properties or reason formally about them has been an area of ongoing research for several decades, leading to the development of proof assistants and model checkers. The approach we are interested in in this thesis is that of logical frameworks, the first implementation of which was the Automath system [De Bruijn, 1980]. The name of the particular style of formalism studied in the thesis comes from a later work, namely the Edinburgh Logical Framework LF [Harper et al., 1993]. The original paper introducing LF has, at the time of writing, over 1700 citations – it is hard to argue against it being influential and interesting to the scientific community.

The logical framework methodology is based on the concept of using an expressive *meta-logic* to model whole classes of object logics. The meta-logic is typically higher-order, and thanks to the Curry-Howard correspondence, provability of certain formulae of the object logic can be reduced to an inhabitation problem in the higher-order meta-logic encoding. A crucial property of the meta-logic is, therefore, the decidability of type checking, since checking the type of an object is equivalent to checking the correctness of a proof.

1.1 Kinds, Types and Terms

In the case of standard LF, the meta-logic in question is $\lambda\Pi$-calculus (described as part of the lambda cube in [Barendregt and Hemerik, 1990]), a first-order dependent type system where a Π type may only depend on a simply-typed term. This is typically expressed as an explicit syntactic distinction between kinds and types, as we do not allow higher-order dependencies. Grammars for kinds, types and terms (as presented in the original LF system) can be given as:

$$\begin{array}{lll} \text{Kinds} & K & ::= \ \mathsf{type} \mid \Pi x{:}A.K \\ \text{Type Families} & A, B & ::= \ a \mid \Pi x{:}A.B \mid A\ M \\ \text{Terms} & M, N & ::= \ c \mid x \mid \lambda x : A.M \mid M\ N \end{array}$$

A *term* in LF can be a lambda abstraction, an application, a variable or a constant declared in the signature Σ.

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

A *type family* can again be a constant, a dependent type or an application of a dependent type to a term. On both the kind and the type level, non-dependent functions will be denoted using $\rightarrow$, so $\Pi x{:}A.B \equiv A \rightarrow B$ if B does not depend on x.

The presentation of types and terms differs, depending on the variant of the framework considered. A pre-canonical presentation by [Cervesato and Pfenning, 1996] distinguishes atomic and normal terms, allowing only η-long forms of the latter. A more recent canonical presentation, studied by [Watkins et al., 2002] and later by [Harper and Licata, 2007], ensures that only canonical forms are typeable. Further, variants using spines instead of applications have been investigated, starting with [Cervesato and Pfenning, 2003]. Finally, a different base logic with its own connectives can be considered, like in [Cervesato and Pfenning, 1996, Watkins et al., 2002, Reed, 2009]. Even with all that variety, the separation of kinding and typing is, in general, preserved.

When discussing variants of LF in the later chapters, more modern presentations will be given; however, for the purpose of this general introduction to the methodology of logical frameworks, a presentation close to the original one will suffice, as we focus our attention on the uses of the framework, rather than on the meta-theoretic aspects of it. To this end, Figure 1.1 presents the typing and kinding rules of LF. The $\equiv$ relation is a definitional equality given by β- and η-conversions. Signature Σ and context Γ well-formedness definitions are omitted, but absolutely standard. Note that in the presence of dependent types, all of these judgements are mutually dependent.

$$\frac{x{:}A \in \Gamma \quad \vdash_\Sigma \Gamma\ \mathsf{ctx}}{\Gamma \vdash_\Sigma x : A} \qquad \frac{c{:}A \in \Sigma \quad \vdash_\Sigma \Gamma\ \mathsf{ctx}}{\Gamma \vdash_\Sigma c : A}$$

$$\frac{\Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x : A.M : \Pi x{:}A.B} \qquad \frac{\Gamma \vdash_\Sigma M : \Pi x{:}A.B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma MN : [N/x]B}$$

$$\frac{\Gamma \vdash_\Sigma M : A \quad \Gamma \vdash_\Sigma A' : \mathsf{type} \quad \Gamma \vdash_\Sigma A \equiv A'}{\Gamma \vdash_\Sigma M : A'}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{ctx} \quad a : K \in \Sigma}{\Gamma \vdash_\Sigma a : K} \qquad \frac{\Gamma, x : A \vdash_\Sigma B : \mathsf{type}}{\Gamma \vdash_\Sigma \Pi x{:}A.B : \mathsf{type}}$$

$$\frac{\Gamma, x : A \vdash_\Sigma B : K}{\Gamma \vdash_\Sigma \lambda x : A.B : \Pi x{:}A.K} \qquad \frac{\Gamma \vdash_\Sigma A : \Pi x{:}B.K \quad \Gamma \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma AM : [M/x]K}$$

$$\frac{\Gamma \vdash_\Sigma A : K \quad \Gamma \vdash_\Sigma K' : \mathtt{kind} \quad \Gamma \vdash_\Sigma K \equiv K'}{\Gamma \vdash_\Sigma A : K'}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{ctx}}{\Gamma \vdash_\Sigma \mathsf{type} : \mathtt{kind}} \qquad \frac{\Gamma, x : A \vdash_\Sigma K : \mathtt{kind}}{\Gamma \vdash_\Sigma \Pi x{:}A.K : \mathtt{kind}}$$

Figure 1.1: LF Types and Kinds

LF Notation Many of the examples presented in this and the subsequent chapters of this thesis use a notation based on a TWELF implementation of LF.

– identifiers standing for constants from the signature, both of type and term level, are starting with small letters; e.g. `nat`, `succ` or `plus`

– identifiers for meta-variables standing for LF terms start with a capital letters; e.g. `N` in `succ N`

– usage of curly brackets is an alternative syntax for a Π-type; e.g. `{N`$_1$`: nat}{N`$_2$`: nat}{M: nat} plus N`$_1$` N`$_2$` M` stands for `Π (N`$_1$`: nat) . Π (N`$_2$`: nat). Π {M: nat} . plus N`$_1$` N`$_2$` M`

– by convention, if we skip type declarations of meta-variables used as indices, they are universally quantified in the order of appearance;

e.g. `plus` N_1 N_2 `M` is a shorthand for `{`N_1`: nat}{`N_2`: nat}{M: nat}` `plus` N_1 N_2 `M`

1.1.1 Example: Binary Trees

When modelling object systems using LF, the framework relies on encodings made using explicit constant declarations (both kind and type-level declarations of this form are allowed).

As an example of a simple type in LF, we will consider binary trees. First, we have to declare a constant `my_tree : type`. This creates an unindexed type family. We continue by providing constants that inhabit this type, in this case leaves and branches: `my_leaf : my_tree` and `my_branch : my_tree → my_tree → my_tree`. Note that the `my_leaf` constant does not contain any data. If we had another type available, say, that of natural numbers `nat`, we could of course declare data-containing leaves as `my_leaf : nat → my_tree`.

```
nat : type.
z   : nat.
s   : nat → nat.

my_tree   : type.
my_leaf   : nat → my_tree.
my_branch : my_tree → my_tree → my_tree.
```

There is not a lot to be said about this level of encoding, apart from its resemblance to a typical data type declaration in a programming language. We will move one step up from here, and discuss a language of *properties of binary trees*.

Let us now consider tree height. Coming from a (functional) programming background, it seems natural to expect that, having access to a type `nat` of natural numbers, we can express the tree height as a function `tree_height : my_tree → nat`. In LF this expectation is in fact incorrect, as such a construct would actually provide an additional constructor for the type `nat`. Instead, we will use a type family encoding a relation between a `my_tree` object and a natural number representing its height. The definition of the `max M N P` relation ("`P` is a maximum of `M` and `N`") is skipped here, but behaves as expected.

```
tree_height : my_tree → nat → type.
tree_height/my_leaf : {N : nat} tree_height (my_leaf N) z.
tree_height/my_branch :
   {T1 : my_tree}{T2 : my_tree}{N1 : nat}{N2 : nat}{N : nat}
   max N1 N2 N →
   tree_height T1 N1 →
   tree_height T2 N2 →
   tree_height (my_branch T1 T2) (s N).
```

Polymorphism It is worth pointing out that polymorphism is not supported in LF, making it impossible to construct a binary tree parametrized by the type of a leaf. That would require a type family something like `poly_tree : Π A : type . type`, which cannot be expressed using the LF grammar given above. We do allow dependent *types*, but dependent *kinds* are not supported. LF's strength crucially depends on the existence of canonical (i.e. long $\beta\eta$-normal) forms, which become problematic in the presence of polymorphism – in particular, η-expansion cannot be directly performed on a polymorphic variable. Interestingly, [Reed, 2008] observes that, in principle, *base-type* polymorphism in LF is possible without η-expansion being sacrificed – however, at the time of writing this thesis and to the best of our knowledge, no follow-up work based on this observation has been made.

In general, it should be said that LF is strictly less expressive than, for instance, the Calculus of Constructions by [Coquand and Huet, 1988], used as a foundation of the Coq theorem prover. This lack of expressive power is the price to pay for how easy it is in LF to work with binders.

1.2 Higher-Order Abstract Syntax

Logical frameworks would not have been a topic of active research for almost thirty years if their expressive power ended with the properties of binary trees. Indeed, the true power of the LF approach comes from its good support for higher-order abstract syntax (HOAS). In simple terms, HOAS allows using the function space of the meta-logic to model variable bindings in the object logic. This comes with a number of advantages, such as the ease with which we can implement binders with built-in support for α-conversion (meta-level variables are α-convertible) or, as we will see later in this chapter, substitution theorems for the object logic coming for free with no need to even define substitution itself. We start this overview by looking at some encodings using HOAS, and continue by discussing the technical requirements for this technique to be practicable.

1.2.1 Example: Lambda Calculus

The canonical example of using higher-order abstract syntax is a mechanization of lambda calculus. We consider a simple variant of it, allowing for anonymous functions and pairs, together with application and projections. We start by encoding the grammar of lambda terms:

```
tm    : type.
lam   : (tm → tm) → tm.
app   : tm → tm → tm.
pair  : tm → tm → tm.
proj₁ : tm → tm.
proj₂ : tm → tm.
```

Two things are worth noting about the code above. First, typically a grammar for `tm`s would include the variable case. Second, the `lam` constructor takes a function from `tm` to `tm` as its argument, whereas λ-abstraction is normally constructed by exposing the variable name x and providing a term which may have free occurrences of x. These two features are precisely where the higher-order abstract syntax is in play. Since in LF we express object logic variables as meta-logic variables, it is meaningful to consider a variable `x : tm` as long as `tm` is a correct type. We exploit this fact further to describe a lambda abstraction of the object level system, by explicitly using the meta-level variable in the argument. As a result, we obtain a meta-level function from `tm` to `tm`.

To introduce typing rules for simply-typed lambda calculus, we first need to declare the possible types. We will consider a type class `tp`, containing arrow type, conjunction and some (further unspecified) base type `o`. With `tp` available, we can then express the typing derivation as a relation between a term and its appropriate type.

```
tp   : type.
o    : tp.
conj : tp → tp → tp.
arr  : tp → tp → tp.
```

```
of_type : tm → tp → type.
of_type/proj₁ :
  of_type M (conj A B) → of_type (proj₁ M) A.
of_type/proj₂ :
  of_type M (conj A B) → of_type (proj₂ M) B.
  of_type/pair :
    of_type M A → of_type N B →
    of_type (pair M N) (conj A B).
of_type/app :
  of_type M (arr A B) → of_type N A →
  of_type (app M N) B.
  of_type/lam :
    ({x : tm} of_type x A → of_type (M x) B) →
    of_type (lam (λx. M x)) (arr A B).
```

Most of the cases defining the `of_type` relation are self-explanatory. The interesting one is `of_type/lam`, which again relies on higher-order abstract syntax to encode a familiar rule for typing a λ-abstraction. The argument taken by this constant is an LF function which, given `x` and `of_type x A`, returns an instance of `of_type (M x) B` – corresponding directly to the premise $\Gamma, x : A \vdash M : B$ of the typing rule. The conclusion of `of_type/lam` is an instance of `of_type (lam (λx. M x)) (arr A B)`, which encodes the judgement $\Gamma \vdash \lambda x.M : A \to B$.

Notably, the `of_type/lam` case exemplifies that HOAS provides α-conversion for free. When writing `of_type (M x) B` we (implicitly) make the typing assumption `M : tm → tm`, since we do apply `x : tm` to `M`. But this means the control over names of variables used can be expressed as a λ-wrapping: `λy.M y` will use `y` as the free variable in `M`.

The next step in defining a simply-typed lambda calculus is to provide semantics for it. In LF these will be represented using a relation `step` between two objects of type `tm`, denoting single-step reduction. The property `value` describes values of the language.

```
value      : tm → type.
value/lam  : value (lam (λx. M x)).
value/pair : value (pair M N).

step            : tm → tm → type.
step/app/beta   : value N → step (app (lam (λx. M x)) N) (M N).
step/proj₁/beta : step (proj₁ (pair M N)) M.
step/proj₂/beta : step (proj₂ (pair M N)) N.

step/app₁  : step M M’ → step (app M N) (app M’ N).
```

```
step/app₂  : value M → step N N' → step (app M N) (app M N').
step/proj₁ : step M M' → step (proj₁ M) (proj₁ M').
step/proj₂ : step M M' → step (proj₂ M) (proj₂ M').
```

The constant `step/app/beta` is an interesting case. Typically, a β-reduction is given as $(\lambda x.M)N \mapsto [N/x]M$, requiring that a substitution function be formally defined. As we have already mentioned, in LF variable substitution is precisely application, which greatly simplifies mechanization of any development relying on substitution. This becomes even more important when considering properties like type preservation under evaluation, which we will look at in Section 1.4.1.

1.2.2 Canonical Forms

The crucial property of a logical framework, which gives higher-order abstract syntax so many advantages, is the existence of canonical forms for all well-typed terms in the meta-language.

In the case of LF, these canonical forms are usually chosen to be β-normal and η-long. This allows pattern matching on functions: a canonical form for a term `foo` of type `nat → nat` has to start with a lambda abstraction: `foo = λx:nat. foo' x` for some `foo'`, as it is η-long. Moreover, as we require that canonical terms are β-normal, the application of an argument `bar: nat` to function `foo` can be expressed as a substitution: `foo bar = (λx:nat. foo' x) bar = [bar / x] foo'`, and vice-versa. It follows that if in the meta-logic, substitution preserves typing, so does the substitution in the object logic.

Of course, it is only when the meta-logic and object logic behave the same with respect to variables that we can exploit HOAS to the fullest. In standard LF, for example, we allow weakening and contraction in contexts, so representing single use (linear or affine) variables using LF function space is not possible directly. We will come back to this point in Chapter 3.

1.3 Adequacy of Encodings

The adequacy property establishes that the encoding we came up with corresponds precisely to the mathematical objects we were formalizing in the first place. After all, when formalizing a logical system, we want the properties obtained in the mechanized version to relate back to the original system of interest. This requires that we provide a compositional isomorphism between the original system and its encoding.

Definition 1.1 (Adequacy)**.** *Given a logical system $\mathcal{D}$, its encoding $\ulcorner D \urcorner$ is considered adequate when there exists a compositional bijection between objects in $\mathcal{D}$ and canonical forms describing $\ulcorner D \urcorner$ in* LF*.*

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \wedge B}$$

$$\frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \pi_1\ t : A} \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \pi_2\ t : B}$$

$$\frac{u \text{ value}}{(\lambda x.t)u \mapsto [u/x]t} \qquad \frac{}{\pi_1\ \langle t, u \rangle \mapsto t} \qquad \frac{}{\pi_1\ \langle t, u \rangle \mapsto u}$$

$$\frac{}{(\lambda x.t) \text{ value}} \qquad \frac{t \mapsto t'}{t\ u \mapsto t'\ u} \qquad \frac{u \mapsto u'}{t\ u \mapsto t\ u'}$$

$$\frac{}{\langle t, u \rangle \text{ value}} \qquad \frac{t \mapsto t'}{\pi_1\ t \mapsto \pi_1\ t'} \qquad \frac{t \mapsto t'}{\pi_2\ t \mapsto \pi_2\ t'}$$

Figure 1.2: λ-calculus: Typing and Reductions

We want to argue for the adequacy of the encoding of simply-typed λ-calculus (STLC) from Section 1.2.1, using induction on derivations. We aimed for it to be an encoding of the logical system given in Figure 1.2.

Establishing adequacy for the `step` and `value` relations requires noticing, as we have before, that the substitution present in the rule $(\lambda x.t)u \mapsto [u/x]t$ can be expressed in LF as a meta-level application, like in `step (app (lam (`λ`x.` $\ulcorner t \urcorner$ `x))` $\ulcorner u \urcorner$`) (`$\ulcorner t \urcorner$ $\ulcorner u \urcorner$`)`, since the canonical forms of $\ulcorner t \urcorner\ \ulcorner u \urcorner$ and $[\ulcorner u \urcorner / x]\ulcorner t \urcorner$ expressions must necessarily be the same. Of course, this argument also relies on the assumption that terms of STLC are encoded adequately.

In the case of `of_type` being an adequate representation of the typing judgement, things get more interesting, as the latter makes explicit use of the context Γ, which we hide in the encoding. We want to argue that $\Gamma \vdash t : A$ holds if and only if the type `of_type` $\ulcorner t \urcorner$ $\ulcorner A \urcorner$ is inhabited when the meta-context contains the encoding of Γ. The encoding of the context has not been given explicitly, but looking at the type of `of_type/lam` constant, we require it to be:

$$\begin{aligned}\ulcorner . \urcorner &= \cdot \\ \ulcorner \Gamma, x : A \urcorner &= \ulcorner \Gamma \urcorner, (\texttt{x : tm, o : of\_type x } \ulcorner A \urcorner)\end{aligned}$$

As the encoding function must be invertible, let us also provide the inverse; notice that the LF context necessarily contains pairs (*blocks*, in the LF terminology) of assumptions, `(x: tm, o: of_type x A)`.

$$\begin{aligned}\ulcorner . \urcorner^{-1} &= \cdot \\ \ulcorner \Gamma, (\texttt{x : tm, o : of\_type x A}) \urcorner^{-1} &= \ulcorner \Gamma \urcorner^{-1}, x : \ulcorner \mathrm{A} \urcorner^{-1}\end{aligned}$$

The interesting cases in the adequacy argument are, unsurprisingly, ones using HOAS. In each of the proof sketches below, assume that the meta-context of LF is populated by an encoding $\ulcorner \Gamma \urcorner$ of context Γ, unless explicitly stated otherwise.

Case $\dfrac{x : A \in \Gamma}{\Gamma \vdash x : A}$ if and only if `of_type x` $\ulcorner A \urcorner$ is inhabited.

$'\Longrightarrow'$

Knowing $x : A \in \Gamma$ and given the definition of $\ulcorner \Gamma \urcorner$, we conclude that `(x : tm, o : of_type x` $\ulcorner A \urcorner$`)` is in the LF meta-context. Therefore `of_type x` $\ulcorner A \urcorner$ is inhabited, with `o` being the inhabitant.

$'\Longleftarrow'$

Knowing `of_type x` $\ulcorner A \urcorner$ is inhabited, take `o` to be the inhabitant. Then necessarily `(x : tm, o : of_type x` $\ulcorner A \urcorner$`)` is in $\ulcorner \Gamma \urcorner$. Using the decoding function, we conclude that $\ulcorner \ulcorner \Gamma \urcorner \urcorner^{-1} = \Gamma$ contains $x : \ulcorner \ulcorner A \urcorner \urcorner^{-1} = x : A$. Therefore $\Gamma \vdash x : A$ using the variable typing rule.

Case $\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$ if and only if

`of_type (lam (`λ`x.` $\ulcorner t \urcorner$ `x)) (arr` $\ulcorner A \urcorner$ $\ulcorner B \urcorner$`)` is inhabited.

$'\Longrightarrow'$

By using the induction hypothesis for $\Gamma, x : A \vdash t : B$, we can conclude that the type `of_type` $\ulcorner t \urcorner$ $\ulcorner B \urcorner$ must be inhabited in the LF meta-context $\ulcorner \Gamma \urcorner$`, (x : tm, o : of_type x` $\ulcorner A \urcorner$`)`. Let `OT` stand for the inhabitant of this type. Then, using LF typing rules, we are able to construct a

term λx.λo.OT of type {x:tm} of_type x $\ulcorner A \urcorner \to$ of_type $\ulcorner t \urcorner$ $\ulcorner B \urcorner$. Therefore, of_type (lam (λx. $\ulcorner t \urcorner$ x)) (arr $\ulcorner A \urcorner$ $\ulcorner B \urcorner$) is inhabited, with of_type/lam (λx.λo.OT) being the inhabitant.
'$\Longleftarrow$'

Knowing of_type (lam (λx. $\ulcorner t \urcorner$ x)) (arr $\ulcorner A \urcorner$ $\ulcorner B \urcorner$) is inhabited, take OT to be the inhabitant. Then, by inversion on the typing of OT, we obtain OT = of_type/lam (λx.λo.OT'). It follows that OT' is an LF object of type of_type $\ulcorner t \urcorner$ $\ulcorner B \urcorner$ in the LF meta-context $\ulcorner \Gamma \urcorner$, (x:tm, o:of_type x $\ulcorner A \urcorner$). Next, using the induction hypothesis on $\ulcorner \Gamma \urcorner$, (x:tm, o:of_type x $\ulcorner A \urcorner$) $\vdash$ OT' : of_type $\ulcorner t \urcorner$ $\ulcorner B \urcorner$, we obtain $\Gamma, x : A \vdash t : B$. We then use the appropriate typing rule to conclude $\Gamma \vdash \lambda x.t : A \to B$.

Proofs of adequacy are not difficult, but can quickly become very technical. However, it is absolutely essential that adequacy is established – especially in encodings using higher-order abstract syntax, where part of the encoding specifying context representation is hidden.

1.4 Reasoning About Specifications

Throughout this chapter we have looked at examples of encodings for a variety of objects:

- my_tree, as well as tm, tp and nat are simple type families, coresponding to data types in a typical programming language;
- tree_height and of_type encode (partial) functions;
- value and step stand for, respectively, a unary and a binary relation.

What we have seen so far is therefore specifying only an object logic. Having mechanised a system specification, we are typically also interested in reasoning about it.

1.4.1 Example: Type Preservation

The first property we are interested in formally establishing is that of type preservation under evaluation of the STLC presented in Figure 1.2.

Property 1 (Preservation).
If $\Gamma \vdash M : A$ and $M \mapsto M'$ then $\Gamma \vdash M : A$.

Knowing from the previous discussion that `step` and `of_type` are both adequate encodings of the relevant relations in simply-typed λ-calculus, we can also express it using the provided encoding:

Property 1 (Preservation).
For any `M:tm`, `M':tm`, `A:tp`, if `of_type M A` and `step M M'` then also `of_type M' A`.

Formalising the theorem statement in LF is fairly straightforward using a dependent type family. Note that again meta-variables which are not declared (in this case `M`, `M'` and `A`) are implicitly universally quantified.

```
preservation : step M M' → of_type M A → of_type M' A → type.
```

It was already remarked upon that, in the LF approach, we replace satisfiability of a formula with inhabitation of a type. Indeed, Property 1 is satisfied if the `preservation` type family is inhabited. The code snippet below shows inhabitant construction cases for the implication fragment.

```
preserv/app/beta:
  preservation
    (step/app/beta (V : value N))
    (of_type/app (of_type/lam OFT) (OT₂ : of_type N B))
    (OFT N OT₂).
preserv/app₁:
  preservation ST OT₁ (OT'₁ : of_type M' (arr A B)) →
  preservation
    (step/app₁ ST) (of_type/app OT₁ OT₂) (of_type/app OT'₁ OT₂).
preserv/app₂:
  preservation ST OT₂ (OT'₂ : of_type N' A) →
  preservation
    (step/app₂ V ST) (of_type/app OT₁ OT₂) (of_type/app OT₁ OT'₂).
```

In both `preserv/app`$_1$ and `preserv/app`$_2$, we base our reasoning on establishing the property for smaller terms first. On the other hand, `preserv/app/beta` provides an inhabitant of the `preservation` type for specific arguments.

Are these enough to argue Property 1? Clearly not, since we have not said anything about the `step/proj`$_i$ cases. But what if we limit our attention to the implication fragment of the language, skipping `pair`, `proj`$_1$, `proj`$_2$? Is the preservation property established for such a fragment? And, more importantly, what is the systematic (if not automated) way of establishing this?

We need to ensure that the cases provided above define a total relation: for all possible inhabitants `ST` of `step M M'` and `OT` of `of_type M A`, we are able to construct an inhabitant `OT'` of `of_type M' A` such that `preservation ST OT OT'` is inhabited. We can observe that the provided code performs an inductive analysis of all relevant cases of the `step` relation, constructing an inhabitant of `preservation` in each case. Moreover, the size of terms decreases in the inductive calls. Therefore, for the implication fragment of STLC we have indeed considered all cases, and so the `preservation` type is inhabited and Property 1 is thus established.

1.4.2 Example: Progress

Another "sanity check" property of STLC that we want to ensure is that of evaluation progress: each well-typed term is either a value, or can be rewriten in one step to another term.

Property 2 (Progress)**.**
If $\Gamma \vdash M : A$ *then either* M *is a* value *or there exists* M' *such that* $M \mapsto M'$.

Using the encoding provided in the previous section, we can reformulate it slightly:

Property 2 (Progress)**.**
If `of_type M A` then either `value M` or `step M M'` for some `M'`.

Notice that it is not possible to express progress as an LF type family directly, since the disjunction type is not supported. We can however simulate it using a type family `val_or_step`, as below:

```
val_or_step      : tm → type.
val_or_step/val  : value M → val_or_step M.
val_or_step/step : step M M' → val_or_step M.

progress : of_type M A → val_or_step M → type.
```

A pencil-and-paper proof of this property will proceed by induction on the typing derivation. Cases where the term is already a value are easy to express:

```
progress/lam :
  progress (of_type/lam OT) (val_or_step/val value/lam).
progress/pair :
  progress (of_type/pair OT1 OT2)
           (val_or_step/val value/pair).
```

However, in the case of the `of_type/app OT`$_1$ `OT`$_2$, we cannot directly give one rule that covers all possible reductions which can be used in the application case[1]: `step/app/beta`, `step/app`$_1$ or `step/app`$_2$. Case analysis would again require the use of disjunction, which we simulate as before, using a type family.

```
progress_app_lem :
  val_or_step M1 → val_or_step M2 →
  of_type M1 (arr A B) → val_or_step (app M1 M2) → type.

progress_app_lem/app1 :
  progress_app_lem (val_or_step/step S) VS2 OT1
                   (val_or_step/step (step/app1 S)).
progress_app_lem/app2 :
  progress_app_lem (val_or_step/val V) (val_or_step/step S)  OT1
                   (val_or_step/step (step/app2 V S)).
progress_app_lem/beta :
  progress_app_lem (val_or_step/val value/lam) (val_or_step/val V2)
                   (of_type/lam OT)
                   (val_or_step/step (step/app/beta V2)).
progress/app :
  progress_app_lem E1 E2 OT1 S →
  progress OT1 E1 → progress OT2 E2 →
  progress (of_type/app OT1 OT2) S.
```

Much like when we argued that `preservation` is inhabited, when checking the inhabitation of the `progress` type, we need to ensure that all possible cases are covered, and that induction is called only on smaller terms. Property 2 is indeed established for the implication fragment, however noticing that requires a few more steps of indirection. In particular, we rely on `progress_app_lem` to be inhabited, which conceptually corresponds to using a lemma.

1.4.3 Automating Reasoning

There are a few parts to this process of verifying properties like the ones from the examples above. Firstly, the defined constants need to

[1]The same happens with `of_type/proj`$_i$

typecheck; secondly, when constructing an inhabitant recursively, we need to ensure that we cover all possible cases; and thirdly, we have to verify that the recursive calls are smaller.

The first of these tasks is fairly easy to satisfy, as we do expect logical frameworks to have decidable typing rules. There are also numerous results on coverage checking in LF-based systems, for instance by [Schürmann and Pfenning, 2003] and more recently by [Pientka and Abel, 2015].

Sadly, the strength of LF that comes from using HOAS does have a price: the definitions we write are no longer inductive in the usual sense. Indeed, recursion over an object using higher-order abstract syntax requires going under a lambda abstraction, effectively requiring that we work on open derivations.

Twelf

One of the most well-established systems for automating reasoning about LF specifications is Twelf by [Schürmann, 2000]. It uses a specially designed meta-logic which supports inductive reasoning with LF specifications. In particular, Twelf allows automatic coverage and termination checks of the provided definitions.

Using the Twelf-like approach, whenever we need a new logical framework with support for automated reasoning, we need to also develop a meta-logic for it. Unfortunately, a general meta-logic that covers any arbitrary logical framework has so far escaped our grasp. For some variations of LF, such as CLF, the meta-logic is not yet known, making automated meta-reasoning impossible. To the best of our knowledge, no general results exist on designing such meta-logics.

Beluga

Another approach to reasoning about LF specifications is presented in the Beluga system by [Pientka and Dunfield, 2010]. In it, the reasoning layer is essentially a first order logic, and we raise LF objects to that level by assigning to them postponed substitutions. Compared to Twelf, it promises more flexibility, as the meta-logic is not as tightly bound to the specification logic. We exploit this in Part III of this thesis, where we step towards changing the specification language for Beluga without altering the reasoning layer.

Chapter 2

Linear Logic

Linear logic, introduced by [Girard, 1987], is a resource-aware, substructural logic with modalities enabling duplication ("of course", !) or discarding ("why not", ?) assumptions, which are otherwise precisely single-use.

From the proof theory perspective, the main difference between intuitionistic logic and intuitionistic linear logic (or classical logic and classical linear logic) lies in the presence of certain structural rules. Weakening and contraction are absent in linear variants, making the variables behave in a resource-like fashion: we require that they are all used precisely once. When giving sequent calculus or natural deduction inference rules for a non-linear logic, we typically build weakening and contraction into the rules for connectives. Interestingly, with weakening and contraction not present, the *additive* and *multiplicative* variants of conjunction and disjunction can no longer be considered equivalent. In order to recover the full expressive power of non-linear (intuitionistic or classical) logic, we need to add operators for which weakening and contraction do again hold: these are the modalities "of course" and "why not", jointly referred to as *exponentials*.

Other *substructural logics* can also be considered. *Affine logic* admits weakening, but not contraction – each assumption can be used at most once. Systems with *subexponentials* [Nigam and Miller, 2009] exploit the non-canonical nature of exponentials, allowing us to examine a linear logic equipped with a whole class of labelled modal operators, instead of just a single pair.

Finally, before moving onto the presentation of the logic itself, it should be remarked that only a fragment of it is relevant to the core of this thesis – the linear implication ($\multimap$) connective being of primary interest. Still, given the directions of future work sketched in the subsequent chapters, as well as a large body of applications using more of the connectives from linear logic (with session types being the most prominent example), we give a more thorough presentation.

2.1 Classical Linear Logic

Connectives of classical linear logic can be split into categories, depending on their behaviour regarding the context of assumptions.

$$\begin{array}{rcl} A & ::= & a \mid 1 \mid 0 \mid \top \mid \bot \mid A \otimes B \mid A \invamp B \\ & & \mid A \,\&\, B \mid A \oplus B \mid !A \mid ?A \mid A^{\perp} \end{array}$$

- multiplicatives: conjunction ($\otimes$, "tensor") with its unit (1, "one"), disjunction ($\invamp$, "par") with its unit ($\bot$, "bottom"), which do not allow re-usage of context;
- additives: disjunction ($\oplus$, "plus") with its unit (0, "zero"), conjunction ($\&$, "with") with its unit ($\top$, "top"), which do allow using the same context in two different premises;
- exponentials: of course (!, "bang"), why not (?), which re-introduce weakening and contraction, allowing the context to be used as in classical logic;
- linear negation ($-^{\perp}$, "dual").

Notice that linear implication ($\multimap$, "lolli") is missing in this presentation. This is because, just like in classical logic, it is encodable: $A \multimap B := A^{\perp} \invamp B$.

Linear negation introduces duality between pairs of operators, as presented in Figure 2.1. Note that a stands for a propositional variable, and $a^{\perp}$ for its dual. This duality allows the presentation of the full system of classical linear logic as a single-sided sequent calculus, as can be seen in Figure 2.2. An alternative presentation using a double-sided sequent calculus is also possible, but would be unnecessarily verbose, since we effectively encode every rule twice – a shortcoming avoided here by exploiting the dual nature of linear connectives.

The axiom rule requires the context to only contain an assumption and its dual, enforcing that all assumptions are, in the end, used. Another

$$\begin{array}{llll}
(a)^\perp & = a^\perp & (a^\perp)^\perp & = a \\
1^\perp & = 0 & 0^\perp & = 1 \\
\top^\perp & = \bot & \bot^\perp & = \top \\
(A \otimes B)^\perp & = A^\perp \oplus B^\perp & (A \oplus B)^\perp & = A^\perp \otimes B^\perp \\
(A \mathbin{\&} B)^\perp & = A^\perp \parr B^\perp & (A \parr B)^\perp & = A^\perp \mathbin{\&} B^\perp \\
(!A)^\perp & = ?A^\perp & (?A)^\perp & = !A^\perp
\end{array}$$

Figure 2.1: Duality in classical linear logic

$$\frac{}{\vdash A, A^\perp}\,ax \qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}\,cut$$

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\,\otimes \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\,\parr$$

$$\frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \mathbin{\&} B}\,\& \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B}\,\oplus_1 \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}\,\oplus_2$$

$$\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}\,! \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}\,?$$

$$\frac{\vdash \Gamma}{\vdash \Gamma, ?A}\,weaken \qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\,contr$$

$$\frac{}{\vdash 1}\,1 \qquad \text{(No rule for 0)} \qquad \frac{}{\vdash \Gamma, \top}\,\top \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \bot}\,\bot$$

Figure 2.2: Classical Linear Logic

interesting feature is the behaviour of modal operators: $?\Gamma$ in the ! rule requires that all the assumptions in the context Γ begin with ? modality. This can, of course, be achieved using the ? rule. Further, such "why not" assumptions can be weakened and contracted at will.

Choice of formalism The only systems for linear logic considered in this thesis will be presented using either natural deduction or sequent calculus formalisms. It should, however, be noted that other possibilities exist. In particular, when introducing linear logic, [Girard, 1987] suggested a use of proof nets. Promising results also arise from the linear nested sequent [Lellmann, 2015] view of linear logic, for instance in [Lellmann et al., 2017]. However, because the LF methodology is typically described using natural deduction, for our purposes this formalism is more familiar and easier to work with.

2.2 Intuitionistic Linear Logic

The presentation of intuitionistic linear logic (ILL), like that of intuitionistic logic, is given as a double-sided sequent calculus with a single conclusion. Looking at the rules from Figure 2.2, it is fairly easy to see that the multiplicative disjunction $\parr$ operator is not compatible with the single-conclusion style of intuitionistic logic. It is therefore replaced with a linear implication $\multimap$ ("lolli"). We do not consider the "why not" ? operator, and we also abandon the negation $(-)^{\perp}$ operator. Importantly, the spirit of splitting classical negation into an exponential and a linear negation (duality) is preserved in the splitting of intuitionistic implication: $A \rightarrow B := !A \multimap B$.

The complete syntax of formulas in intuitionistic linear logic is therefore the following:

$$A \quad ::= \quad a \mid 1 \mid 0 \mid \top \mid A \otimes B \mid A \multimap B \mid A \,\&\, B \mid A \oplus B \mid !A$$

Figure 2.3 presents sequent calculus inference rules for ILL. Just as lambda calculus terms can be used to annotate derivations of intuitionistic logic, it is possible to annotate linear logic derivations with linear lambda calculus terms. We do not give these annotations here, keeping the discussion purely logical; we will, however, use them in subsequent chapters.

The rules for the "of course" (!) operator are interesting. First, the $!_R$ rule requires that all assumptions in the context begin with the ! modality. Second, there are three different left rules, corresponding to the structural rules of contraction and weakening, which we want the ! modality – and no other connective in the system – to have. The rule $!_L derel$ ("dereliction"), allows a linear assumption to be *demoted* to a persistent one. The $!_R$ rule, on the other hand, *promotes* the conclusion of the judgement to be permanent.

$$\frac{}{A \vdash A}\,ax \qquad \frac{\Gamma \vdash A \qquad \Delta, A \vdash C}{\Gamma, \Delta \vdash C}\,cut$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\,\otimes_R \qquad \frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C}\,\otimes_L$$

$$\frac{\Delta, A \vdash B}{\Delta \vdash A \multimap B}\,\multimap_R \qquad \frac{\Delta \vdash A \quad \Gamma, B \vdash C}{\Delta, \Gamma, A \multimap B \vdash C}\,\multimap_L$$

$$\frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \,\&\, B}\,\&_R \qquad \frac{\Delta, A \vdash C}{\Delta, A \,\&\, B \vdash C}\,\&_{L_1} \qquad \frac{\Delta, B \vdash C}{\Delta, A \,\&\, B \vdash C}\,\&_{L_2}$$

$$\frac{\Delta \vdash A}{\Delta \vdash A \oplus B}\,\oplus_{R_1} \qquad \frac{\Delta \vdash B}{\Delta \vdash A \oplus B}\,\oplus_{R_2} \qquad \frac{\Delta, A \vdash C \quad \Delta, B \vdash C}{\Delta, A \oplus B \vdash C}\,\oplus_L$$

$$\frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\,!_R \qquad \frac{\Gamma, A \vdash C}{\Gamma, !A \vdash C}\,!_L derel$$

$$\frac{\Gamma, !A, !A \vdash C}{\Gamma, !A \vdash C}\,!_L contr \qquad \frac{\Gamma \vdash C}{\Gamma, !A \vdash C}\,!_L weak$$

$$\frac{}{\vdash 1}\,1_R \qquad \frac{\Delta \vdash C}{\Delta, 1 \vdash C}\,1_L \qquad \frac{}{\Gamma, 0 \vdash C}\,0_L \qquad \frac{}{\Gamma \vdash \top}\,\top_R$$

$$\text{(No rule for } 0_R\text{)} \qquad \text{(No rule for } \top_L\text{)}$$

Figure 2.3: Intuitionistic Linear Logic

2.2.1 Proof Theory

Linear logic satisfies the usual proof theoretic metrics that allow us to decide whether a logic is considered well-behaved. These include initiality, associated with η-expansion, which allows restricting the axiom rule to work only on atomic propositions $\frac{}{a \vdash a}$, and cut elimination, which lets us consider only proofs which are cut-free. The formulations of these properties for ILL are given as:

Theorem 2.1 (Initiality). *The general axiom rule $\frac{}{A \vdash A}$ is derivable if we allow using its atomic variant $\frac{}{a \vdash a}$.*

Theorem 2.2 (Cut elimination)**.** *For any derivation $\mathcal{D} :: \Gamma \vdash A$, there exists a cut-free variant of it.*

One of the most important consequences of cut elimination is the subformula property, which in turn allows for algorithmic search for proofs of a given formula. Moreover, thanks to the more structured behaviour of its connectives, linear logic – especially its non-exponential fragment – allows for more intricate proof search procedures. This led, for instance, to the development (by [Andreoli, 1992]) of focusing as a proof technique, which was later also adapted to intuitionistic and classical logics.

Observing the distinction between additive and multiplicative connectives led to the establishment of a Curry-Howard relation between linear logic and session types [Caires and Pfenning, 2010, Wadler, 2015]. Seen through this lens, multiplicative connectives correspond to sending and receiving actions, whereas additives stand for internal and external choice operators. Finally, the exponential can be seen as a replication.

2.3 Resource-Aware Systems

Linear logic is often described as a "resource-aware logic" or a "logic of resources". Indeed, the lack of weakening and contraction in linear logic seems to be a perfect fit for modelling resource-based systems. No contraction means that no resource can be used twice, and no weakening forces us to use every resource we have. (We should note that affine systems are also a good match, as we can sometimes have leftover resources.)

2.3.1 Example: Restaurant Menu

Lunch Menu : 200 DKK

- ⋆ *A soup (tomato or onion, depending on availability)*
- ⋆ *Vegetarian or fish meal*
- ⋆ *Water ad libitum*
- ⋆ *Choice of cake or cheese board: 50 DKK extra*

One of the most commonly seen examples explaining usage of linear logic connectives for resource modelling is that of a restaurant menu.

What does 200 Danish kroner get us? A lunch consists of a seasonal soup course, a vegetarian or fish main meal, and unlimited water. If we pay another 50 kroner we can also get a cake or some cheese. Notice that some choices are *external* (whether the soup is tomato or onion depends on what is available) and some are *internal* (whether we get a vegetarian meal or fish is something we decide on). This is precisely the behaviour of additive operators: $\oplus$ represents a disjunction, an external choice (of whoever is offering to feed us), whereas $\&$ is a conjunction, an internal choice (where we decide if we want fish). *Water ad libitum* can mean any amount of water, so water in this example is a persistent resource. We mark that it is available in any quantity using the "of course" operator. Finally, we have an option of ordering a desert (cake or cheese), but only if we are willing to pay extra. We can also choose not to have desert, which is represented using type 1. The menu can be described in linear logic in the following manner:

```
lunch = 200 DKK ⊸ soup ⊗ main ⊗ ! water ⊗ maybe_dessert
soup = tomato ⊕ onion
main = vegetarian & fish
maybe_desert = desert & 1
dessert = (50 DKK ⊸ cake & cheese)
```

When trying to mechanise an example like this in LF, we quickly encounter a number of problems: it is rather difficult to express that the money we have can be used only once, and you can have as many dinners and deserts as you want. While this is obviously excellent news (especially if the cake is good), it raises issues of adequacy.

LF was originally developed using intuitionistic logic as its basis, so has no native support for linear implication $\multimap$, and everything we have access to can be used arbitrarily often. To model linear systems using an LF-like approach, a variant of LF designed with linearity in mind is needed.

One can see this as a test of the methodology. Given that the gains from using LF can be best appreciated when the higher order abstract syntax allows for "free" variable binders and substitution theorems, how hard is it to develop a variant of LF that allows the bound variables to behave precisely as we want them to?

Chapter 3

Encoding Linear Systems

The establishment of a correspondence between session types and linear logic led to a rise of interest in formal reasoning frameworks using the latter. The hope is that, if such a framework is expressive enough to capture session types, formal reasoning about protocols and abstract concurrency models will become less cumbersome.

However, the interest in using linear logic as a basis for a LF-like system is not new. The first effort in this direction was the Linear Logical Framework (LLF) by [Cervesato and Pfenning, 1996], a variant of LF designed specifically to work with resource-aware systems. As it is a spiritual predecessor to both of the lines of enquiry pursued in this thesis, its design is worth studying in more detail.

The presentation that we give here does not precisely follow that in the original paper, which required terms to be η-long, but not necessarily β-normal. Rather, it adheres to the modern methods of building such frameworks, designed to ensure that only canonical forms are typeable. This greatly simplifies the design of the meta-theory and makes adequacy arguments easier to express: since the only well-typed terms are canonical, the compositional bijection we are looking for in the adequacy arguments is between elements of the object logic and well-typed terms in (a variant of) LF, rather than canonical objects of the framework. To this end, the presentation we give here will use hereditary substitution, as described by [Harper and Licata, 2007] in the Canonical LF paper and originally introduced for the Concurrent Logical Framework CLF by [Watkins et al., 2004]. We actually go one step further and use a spine presentation from [Cervesato and Pfenning, 2003], moving the system closer to a sequent calculus presentation while at the same time making termination of hereditary substitution easier to argue. Moreover, as the subsequent work focuses on the fragment of LLF using only linear implication, we restrict our attention to this fragment – skipping the other two connectives present in the original presentation: $\&$ and $\top$.

3.1 Linear Logical Framework

When mechanising theorems about the encodings, like those in Examples 1.4.1 and 1.4.2, we rely on the presence of dependent types to give the theorem statement. It is not clear what a dependent linear type would be: should assumptions serving as indices to a type count as used? Whatever the answer, the behaviour of such linear dependent types is far from ideal. At the same time, the meta-theoretical properties of linear systems are themselves not always proven using linear proofs. These make a clear argument for not making LLF purely linear, but rather extending LF with linear operators. This way, everything expressed in LF can also be encoded in LLF, making it a *conservative* extension.

Much like LF, LLF distinguishes three levels of objects: kinds, types and terms. However, since we are only interested in the typability of canonical terms, we need to distinguish between two syntactic categories of types: canonical types are used to give types to canonical terms, whereas atomic types are used to give types to other terms. As we aim for a system in which only β-normal terms are typeable, we have chosen a spine presentation for the atomic terms. This way, the potential future redex in the form of a head variable, is exposed, rather than hidden in a chain of applications, which are replaced here with a spine – a list of arguments applicable to the head. This way, when the substitution for a head variable occurs, we can immediately reduce the newly constructed redex. The complete syntax of LLF can be given as:

Kinds	K	$::=$	$\mathsf{type} \mid \Pi x{:}A.K$
Atomic Types	P, Q	$::=$	$a \cdot S$
Canonical Types	A, B	$::=$	$P \mid \Pi x{:}A.B \mid A \multimap B$
Heads	H	$::=$	$x \mid c$
Spines	S	$::=$	$\epsilon \mid M\,;S \mid M\,\hat{;}\,S$

Atomic Terms	R	$::=$	$H \cdot S$
Canonical Terms	M, N	$::=$	$R \mid \lambda x.M \mid \widehat{\lambda} x.M \mid$
Unrestricted Contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A$
Linear Contexts	Δ	$::=$	$\cdot \mid \Delta, x\hat{:}A$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A$

In addition to the kinds, terms and types already present in LF, we extend the types syntax with linear implication $\multimap$. On the term level, this requires allowing for linear abstraction $\widehat{\lambda} x.M$. Choosing the spine presentation replaces application and linear application with extending the spine: $M\,;S$ mimics an intuitionistic application and $M\,\hat{;}\,S$ a linear one.

Before looking more closely at the meta-theory of LLF, let us briefly discuss some example encodings making use of the new connectives of the framework.

3.1.1 Example: Blocks World

The first example we will look at is a classic planning problem. Given a set of wooden blocks on a table, the goal is to move them to some pre-defined end position, for instance a single stack. Only one block can be moved at a time and some blocks are initially on top of others, so only top blocks from each stack can be moved. In past AI research, this problem was typically presented by giving start and goal configurations and then looking for a list of legal moves from the start to the goal.

What makes linear logic the right tool to encode the rules of this system is the inherent notion of *state*, changing as we move a block from one stack to the next. A block is not lost during a move, but its old state is. We will therefore use block state as a linear resource when describing legal moves.

We distinguish between two states of a block: either it is on the table, or on another block. This is expressed via `on_table` and `on_top_of`. A block can also be a top block in its stack, `is_top` in the presented code snippet.

```
#          : type.
block      : type.

on_table   : block → type.
on_top_of  : block → block → type.
is_top     : block → type.
```

There are three possible moves, which can be described as changes of state: moving a block from the table to the top of another stack, moving it from the top of one stack to the top of another, and putting the top block of a stack down on the table[1].

```
   move/from_table_to_top :
     (on_table B₁ ⊸ is_top B₁ ⊸ is_top B₂ ⊸ #) ⊸
     (on_top_of B₁ B₂ ⊸ is_top B₁ ⊸ #).

   move/from_top_to_table :
     (is_top B₁ ⊸ on_top_of B₁ B₂ ⊸ #) ⊸
     (is_top B₁ ⊸ on_table B₁ ⊸ is_top B₂ ⊸ #).

   move/from_top_to_top :
     (is_top B₁ ⊸ on_top_of B₁ B₂ ⊸ is_top B₃ ⊸ #) ⊸
     (is_top B₁ ⊸ is_top B₂ ⊸ on_top_of B₁ B₃ ⊸ #).
```

These rules can be seen as *context rewriting rules*. Controlling what is in the context is achieved by explicitly listing these elements. (This is, in fact, typical of resource-based systems: most of the work is done via context manipulation.) We use `# :` **type** as an end marker, which can be constructed using `move/...` rules; in this way, the only possibility of getting an inhabitant of # type is by manipulating blocks in the context.

Finding a sequence of moves between the start and goal configurations can be expressed as an inhabitation problem: can we rewrite the context describing the starting configuration into the context containing the goal one?

```
b₁ : block.
b₂ : block.
b₃ : block.

% Start state: 3 blocks, all of them on the table
% Goal state: 3 blocks on top of one another; b₃ on the table, b₁ on top

start : type =
 on_table b₁ ⊸ is_top b₁ ⊸
 on_table b₂ ⊸ is_top b₂ ⊸
 on_table b₃ ⊸ is_top b₃ ⊸ #.

goal : type =
 on_table b₃ ⊸ on_top_of b₂ b₃ ⊸
 on_top_of b₁ b₂ ⊸ is_top b₁ ⊸ #.
```

[1]We skip moving from the table back to the table, as this does not actually change the state of the system.

```
move :
  start ⊸ goal =
  λf. λot3. λoto23. λoto12. λit1.
    (move/from_table_to_top
      (λot. λit. λit'.
        (move/from_table_to_top
          (λot2. λit2. λit2'. f ot it ot2 it2 ot3 it2')
        oto23 it'))
    oto12 it1).
```

3.1.2 Example: Linear Lambda Calculus

Our second example encodes a linear lambda calculus. We are interested in the higher-order abstract syntax in the linear case. The code below looks quite similar to that in Example 1.2.1; however, it uses linear implication instead of intuitionistic one. We use $\widehat{\lambda}$x.M to denote a linear abstraction and `M ^ N` to stand for linear application.

```
  tm    : type.
  lam   : (tm ⊸ tm) ⊸ tm.
  app   : tm ⊸ tm ⊸ tm.
  pair  : tm ⊸ tm ⊸ tm.
  proj₁ : tm ⊸ tm.
  proj₂ : tm ⊸ tm.
```

Notice that the `lam` case makes explicit use of the linear function space. This forces other constants to also use linearity, as they could not otherwise use variables introduced under a lambda. For instance, with the app variant of `app : tm → tm → tm`, it would not be possible to typecheck the term `lam (`$\widehat{\lambda}$`x. lam (`$\widehat{\lambda}$`y . app x y))`, as it tries to use linear variables `x` and `y` as if they were intuitionistic.

With such a presentation of syntax, the typing rules `of_type/...` can remain intuitionistic.

```
  tp   : type.
  o    : tp.
  arr  : tp → tp → tp.
  conj : tp → tp → tp.

  of_type : tm → tp → type.
  of_type/lam :
    ({x : tm} of_type x A → of_type (M ^ x) B) →
    of_type (lam ^ (λ̂x . M ^ x)) (arr A B).
```

```
of_type/app :
  of_type M (arr A B) → of_type N A →
  of_type (app ^ M ^ N) B.
of_type/pair :
  of_type M A → of_type N B →
  of_type (pair ^ M ^ N) (conj A B).
of_type/proj₁ :
  of_type M (conj A B) → of_type (proj₁ ^ M) A.
of_type/proj₂ :
  of_type M (conj A B) → of_type (proj₂ ^ M) B.
```

Importantly, the typing rules above use the intuitionistic arrow, and the terms themselves (`M, N, …`) have to be used intuitionistically, since they are used as indices to `of_type`. However, the constructors for `tm` all rely on a linear function space, so we cannot write an object of type `tm` which re-uses variables.

We should mention that this is not the only way to encode linear lambda calculus in LLF. It is also possible to leave the syntax precisely as in Example 1.2.1 and to enforce linearity through the typing rules. In that case, we can write non-linear terms of type `tm`; however, we cannot assign types to them.

3.2 Meta-theory of LLF

A typical modern presentation of an LF-based framework usually omits some of the judgements (or even theorems!) needed to make a complete argument of the system's well-behavedness in the interests of brevity. Indeed, these judgements and theorems are mostly standard – it is simply a matter of "fine-tuning" them to the concrete framework at hand. A good overview of the full meta-theory of logical frameworks is given by [Harper and Licata, 2007]. In this case, however, we find it important to present the LLF system in full detail, as adaptations of it were used while developing the systems presented in the main parts of this thesis.

Typing judgements in LLF are parametrized by a signature Σ, containing term- and type-level constants. The full forms of these judgements therefore have the shape $\Gamma; \Delta \vdash_\Sigma J$ with J being type checking for canonical terms ($M \Leftarrow A$), type synthesis for heads ($H \Rightarrow A$) or spine checking ($S > A \Rightarrow P$). Typically we omit the signature Σ as it is fixed. The inference rules for typing LLF terms are presented in Figure 3.1 and rely on hereditary substitution $[M/x]A$ which we define in Figure 3.4

$$\frac{\Gamma, x{:}A; \Delta \vdash M \Leftarrow B}{\Gamma; \Delta \vdash \lambda x.M \Leftarrow \Pi x{:}A.B} \qquad \frac{\Gamma \vdash A \ \mathsf{type} \quad \Gamma; \Delta, x \hat{:} A \vdash M \Leftarrow B}{\Gamma; \Delta \vdash \widehat{\lambda} x.M \Leftarrow A \multimap B}$$

$$\frac{\Gamma; \Delta_1 \vdash H \Rightarrow A \quad \Gamma; \Delta_2 \vdash S > A \Rightarrow P \quad \Gamma \vdash P = Q}{\Gamma; \Delta_1, \Delta_2 \vdash H \cdot S \Leftarrow Q}$$

$$\frac{c{:}A \in \Sigma \quad \vdash \Gamma \ \mathsf{ctx}}{\Gamma; \cdot \vdash c \Rightarrow A} \qquad \frac{x{:}A \in \Gamma \quad \vdash \Gamma \ \mathsf{ctx}}{\Gamma; \cdot \vdash x \Rightarrow A} \qquad \frac{\Gamma \vdash A \ \mathsf{type}}{\Gamma; x \hat{:} A \vdash x \Rightarrow A}$$

$$\frac{\Gamma \vdash P \Rightarrow \ \mathsf{type}}{\Gamma; \cdot \vdash \epsilon > P \Rightarrow P} \qquad \frac{\Gamma; \cdot \vdash M \Leftarrow A \quad \Gamma; \Delta \vdash S > [M/x]_A B \Rightarrow P}{\Gamma; \Delta \vdash M\,; S > \Pi x{:}A.B \Rightarrow P}$$

$$\frac{\Gamma; \Delta_1 \vdash M \Leftarrow A \quad \Gamma; \Delta_2 \vdash S > B \Rightarrow P}{\Gamma; \Delta_1, \Delta_2 \vdash M \hat{;} S > A \multimap B \Rightarrow P}$$

Figure 3.1: Typing LLF Terms

and discuss later in this chapter. The typing judgement depends on two contexts: intuitionistic Γ and linear Δ. Importantly, the typing rule for the $H \cdot S$ term changes the mode of operation from type checking ($H \cdot S \Leftarrow Q$) to type synthesis ($H \Rightarrow A$, $S > A \Rightarrow P$), relying also on type equivalence of atomic types. Notice that when typechecking the linear abstraction case $\widehat{\lambda} x.M$, we need to verify that the type of the variable, A, is in fact well-formed given the current intuitionistic context. This step is necessary to avoid a situation where a linear, dependently typed variable depends on an intuitionistic variable which gets introduced to the context later, e.g. we want to reject terms such as $\widehat{\lambda} y : bx.\lambda x : a.y$ where y is of type bx (a term $\lambda x.\widehat{\lambda} y.y$ would be perfectly acceptable).

The typing rules of LLF depend on the validity of the signature ($\vdash \Sigma$ sig), and of the both unrestricted and linear contexts (denoted $\vdash_\Sigma \Gamma$ ctx and $\Gamma \vdash_\Sigma \Delta$ linctx, respectively). These validity rules are presented in Figure 3.2. Fixed Σ means that dependent types in Σ must have their dependencies satisfied within it. All constants in Σ are reusable, and therefore not linear.

The validity of contexts and signatures depends, in turn, on well-formed types and kinds, given in Figure 3.3. Notice that no form of dependency on linear assumptions is allowed.

When extending a spine with an unrestricted term, we rely on a substitution operation in both the typing and kinding rules: $[M/x]_A B$

$$\frac{}{\vdash \cdot \ \mathsf{sig}} \qquad \frac{\vdash \Sigma\ \mathsf{sig} \quad \cdot \vdash_\Sigma K\ \mathsf{kind}}{\vdash \Sigma, a : K\ \mathsf{sig}} \qquad \frac{\vdash \Sigma\ \mathsf{sig} \quad \cdot \vdash_\Sigma A\ \mathsf{type}}{\vdash \Sigma, c : A\ \mathsf{sig}}$$

$$\frac{\vdash \Sigma\ \mathsf{sig}}{\vdash_\Sigma \cdot\ \mathsf{ctx}} \qquad \frac{\vdash_\Sigma \Gamma\ \mathsf{ctx} \quad \Gamma \vdash_\Sigma A\ \mathsf{type}}{\vdash_\Sigma \Gamma, x{:}A\ \mathsf{ctx}}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathsf{ctx}}{\Gamma \vdash_\Sigma \cdot\ \mathsf{linctx}} \qquad \frac{\Gamma \vdash_\Sigma \Delta\ \mathsf{linctx} \quad \Gamma \vdash_\Sigma A\ \mathsf{type}}{\Gamma \vdash_\Sigma \Delta, x\hat{:}A\ \mathsf{linctx}}$$

Figure 3.2: Signature and Context Validity

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

$$\frac{\vdash \Gamma\ \mathsf{ctx}}{\Gamma \vdash \epsilon > \ \mathsf{type} \Rightarrow\ \mathsf{type}} \qquad \frac{\Gamma \vdash M \Leftarrow A \quad \Gamma \vdash S > [M/x]_A K \Rightarrow\ \mathsf{type}}{\Gamma \vdash S\,; M > \Pi x{:}A.K \Rightarrow\ \mathsf{type}}$$

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

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

Figure 3.3: Kind and Type Formation

and $[M/x]_A K$, respectively. Crucially, using a standard substitution might result in creating a redex, which would violate the "only canonical forms have types" principle of canonical LF. Therefore, instead of a simple inductive definition of substitution, we use hereditary substitution, presented in Figures 3.4 and 3.5. Hereditary substitution is in fact a family of relations, $[M/x]^j_\alpha$, $j \in \{k, t, s, c\}$ (for kinds, types, spines and canonical terms, respectively), annotated by a simple type α of the substituted variable. This simple type can be obtained from a normal type using the *type erasure* operation, which replaces dependent types with their non-dependent variants.

$$\begin{aligned} (a \cdot S)^- &= a \\ (\Pi x{:}A.B)^- &= A^- \to B^- \\ (A \multimap B)^- &= A^- \multimap B^- \end{aligned}$$

$$\frac{}{[M/x]^k_\alpha\ \mathsf{type} = \ \mathsf{type}} \qquad \frac{[M/x]^k_\alpha K = K' \quad [M/x]^t_\alpha A = A'}{[M/x]^k_\alpha(\Pi y{:}A.K) = \Pi y{:}A'.K'}$$

$$\frac{[M/x]^t_\alpha A = A' \quad [M/x]^t_\alpha B = B'}{[M/x]^t_\alpha(\Pi y{:}A.B) = \Pi y{:}A'.B'} \qquad \frac{[M/x]^t_\alpha A = A' \quad [M/x]^t_\alpha B = B'}{[M/x]^t_\alpha(A \multimap B) = A' \multimap B'}$$

$$\frac{[M/x]^s_\alpha S = S'}{[M/x]^t_\alpha(a \cdot S) = a \cdot S'}$$

Figure 3.4: Hereditary Substitution in LLF: Types and Kinds

$$\frac{[M/x]^s_\alpha S = S' \quad [M/x]^c_\alpha N = N'}{[M/x]^s_\alpha(N\,;S) = N\,;'\,S'} \qquad \frac{[M/x]^s_\alpha S = S' \quad [M/x]^c_\alpha N = N'}{[M/x]^s_\alpha(N\,\hat{;}\,S) = N\,\hat{;}'S'}$$

$$\frac{}{[M/x]^s_\alpha \epsilon = \epsilon} \qquad \frac{[M/x]^s_\alpha S = S' \quad \mathsf{reduce}(M : \alpha, S') = H \cdot S''}{[M/x]^c_\alpha(x \cdot S) = H \cdot S''}$$

$$\frac{[M/x]^s_\alpha S = S' \quad x \neq y}{[M/x]^c_\alpha(y \cdot S) = y \cdot S'} \qquad \frac{[M/x]^s_\alpha S = S'}{[M/x]^c_\alpha(c \cdot S) = c \cdot S'}$$

$$\frac{[M/x]^c_\alpha N = N'}{[M/x]^c_\alpha(\lambda y.N) = \lambda y.N'} \qquad \frac{[M/x]^c_\alpha N = N'}{[M/x]^c_\alpha(\widehat{\lambda} y.N) = \widehat{\lambda} y.N'}$$

Figure 3.5: Hereditary Substitution in LLF: Terms

Both the type erasure and the marker indicating the substitution relation variant are often omitted; for instance, $[M/x]_A B$ is in fact $[M/x]^t_{A^-} B$.

The single instance where a hereditary substitution does not behave like an inductively defined one is the $[M/x]^c_\alpha(x \cdot S)$ case. Thanks to our use of the spine variant of the calculus, upon substitution we immediately have access to the new head, M, and the spine S' – the result of substituting $[M/x]^s_\alpha S$. We can continue reducing any newly created redex using a recursive $\mathsf{reduce}(M : \alpha, S)$ function, defined below. Notice that the simple type of the head term decreases in each recursive call, guaranteeing termination.

$$\begin{aligned}
&\mathsf{reduce}(\lambda x.M : \alpha \to \beta, (N\,;S)) &&= \mathsf{reduce}([N/x]^c_\alpha M : \beta, S)\\
&\mathsf{reduce}(\widehat{\lambda} x.M : \alpha \multimap \beta, (N\,\hat{;}\,S)) &&= \mathsf{reduce}([N/x]^c_\alpha M : \beta, S)\\
&\mathsf{reduce}(H \cdot S : a, \epsilon) &&= H \cdot S\\
&\mathsf{reduce}(M : \alpha, S) &&= \bot
\end{aligned}$$

3.2.1 Hereditary Substitution Properties

Writing $[M/x]^t_{A^-}B$ in the typing rules suggests that herediary substitution can be treated as a partial function. This is indeed the case.

Lemma 3.1 (Uniqueness of hereditary substitution)**.**

(i) If $\mathsf{reduce}(M : \alpha, S) = H \cdot S$ *and* $\mathsf{reduce}(M : \alpha, S) = H' \cdot S'$*, then* $H = H'$ *and* $S = S'$*;*

(ii) If $[M/x]^e_\alpha E = E'$ *and* $[M/x]^e_\alpha E = E''$*, then* $E' = E''$ *(for* $e \in \{k, t, s, c\}$*,* $E \in \{K, A, S, M\}$*).*

When substitution is defined as a relation, in addition to the usual requirement of it being type preserving, we must also argue that it is actually a well-defined operation in all relevant cases.

Theorem 3.1 (Hereditary substitution I)**.** *Hereditary substitution on well-typed terms is a terminating total function.*

Theorem 3.2 (Hereditary substitution II)**.** *Hereditary substitution on well-typed terms preserves typing.*

3.2.2 Decidability of Type Checking

As already hinted at in Chapter 1, a logical system must have a number of meta-theoretic properties for it to be useful as a logical framework. Most notably, as we heavily rely on the *judgements as proofs* paradigm, we require typing to be decidable.

Lemma 3.2 (Decidability of hereditary substitution)**.**

(i) For any $e \in \{k, t, s, c\}$*,* $E \in \{K, A, S, M\}$*, given* M, α, x*, either there exists* E' *such that* $[M/x]^e_\alpha E = E'$ *or* $[M/x]^e_\alpha E = \bot$*;*

(ii) *Given M, α and S, either there exist H and S' such that* $\mathsf{reduce}(M : \alpha, S) = H \cdot S'$ *or* $\mathsf{reduce}(M : \alpha, S) = \bot$

Lemma 3.3 (Decidability of formation)**.**

(i) *For all Σ, it is decidable whether $\vdash \Sigma$ sig holds;*

Assume Σ such that $\vdash \Sigma$ sig

(ii) *For all Γ, it is decidable whether $\vdash_\Sigma \Gamma$ ctx holds;*

Assume Γ such that $\vdash_\Sigma \Gamma$ ctx

(iii) *For all Δ, it is decidable whether $\Gamma \vdash_\Sigma \Delta$ linctx holds;*

(iv) *For all K, it is decidable whether $\Gamma \vdash K$ kind holds;*

(v) *For all A, it is decidable whether $\Gamma \vdash A$ type holds;*

(vi) *For all P, it is decidable whether there exists K such that $\Gamma \vdash P > K \Rightarrow \mathsf{type}$;*

Assume Δ such that $\Gamma \vdash_\Sigma \Delta$ linctx

(vii) *For all M, A, it is decidable whether $\Gamma; \Delta \vdash M \Leftarrow A$ holds;*

(viii) *For all H, it is decidable whether there exists A such that $\Gamma; \Delta \vdash H \Rightarrow A$;*

(ix) *For all S, A and P, it is decidable whether $\Gamma; \Delta \vdash S > A \Rightarrow P$ holds.*

It is easy to see that the *(vii)* case of the lemma above proves:

Theorem 3.3 (Decidability of typing LLF derivation)**.** *Typing is decidable.*

3.3 Beyond LLF

A number of implementations of the Linear Logical Framework exist. Work by [McCreight and Schürmann, 2004] directly introduced a metalogic for LLF. An approach by [Reed, 2009] was to construct a variant of the Twelf system optimized to allow for linear abstraction encodings.

As we have mentioned at the beginning of this chapter, LLF (in its original presentation) only allows for a small subset of intuitionistic linear logic connectives to be used: $\multimap$, $\&$, and $\top$. The motivation for such a choice may not be immediately obvious – but in fact, this comes from a limitation of linear logic itself: these three operators form a maximal subset of linear logic with canonical forms. On the other hand, to use LLF for encodings of concurrent systems, we need both linear implication and tensor to be present. To see where the problem with canonicity occurs, consider the η-expansion of a variable of $\otimes$ type:

Example 3.1. *Let $W = (a \otimes b) \otimes (c \otimes d)$. The two derivations below, when annotated with proof terms, yield two different η-expansion proofs for a variable of type $(a \otimes b) \otimes (c \otimes d)$.*

$$\dfrac{\dfrac{}{W \vdash W} \qquad \dfrac{\dfrac{}{a \otimes b \vdash a \otimes b} \qquad \dfrac{\dfrac{}{c \otimes d \vdash c \otimes d} \qquad \dfrac{\cdots}{a, b, c, d \vdash W}}{c \otimes d, a, b \vdash W}}{a \otimes b, c \otimes d \vdash W}}{(a \otimes b) \otimes (c \otimes d) \vdash (a \otimes b) \otimes (c \otimes d)}$$

$$\dfrac{\dfrac{}{W \vdash W} \qquad \dfrac{\dfrac{}{c \otimes d \vdash c \otimes d} \qquad \dfrac{\dfrac{}{a \otimes b \vdash a \otimes b} \qquad \dfrac{\cdots}{a, b, c, d \vdash W}}{a \otimes b, c, d \vdash W}}{a \otimes b, c \otimes d \vdash W}}{(a \otimes b) \otimes (c \otimes d) \vdash (a \otimes b) \otimes (c \otimes d)}$$

CLF, a system by [Watkins et al., 2002], is a proposed solution based on concurrent computations encapsulated in a monad, to address the lack of canonicity. Unfortunately, and despite the recent re-design of its meta-theory by [Schack-Nielsen, 2011], CLF does not yet have a reasoning meta-logic. It nevertheless remains a very promising and expressive framework. An implementation of CLF, Celf, has been proposed by [Schack-Nielsen and Schürmann, 2008].

Any extension of LLF using more of the connectives taken from linear logic must address the canonicity problem. One way to extend the language expressivity is to follow [Reed, 2009] in adapting a more expressive logic, in which we can then embed a (fragment of) ILL. This idea serves as the base of the work presented in Part II of this thesis. Another approach is to separate representation from reasoning, as in the Beluga system by [Pientka and Dunfield, 2010]: we would then have more freedom in designing the representation fragment without affecting the reasoning capabilities. We attempt this solution in Part III.

Part II

HyLF

Chapter 4

Hybridising Logical Framework

This part of the thesis is an attempt at building an expressive logical framework, able to capture systems that use non-standard bindings or require support for some form of proof irrelevance. This work is, in part, motivated by the difficulties encountered when trying to build a general framework for working with the family of intuitionistic modal logics described in [Simpson, 1994].

The main idea is to generalise the work of the Hybrid Logical Framework HLF[1] by [Reed, 2009], a logical framework using techniques known from hybrid logic to encode linearity.

We ask if a more in-depth investigation into hybrid logic can expand the expressiveness of a single, general logical framework. The resulting HyLF framework does not directly extend HLF and it connects to hybrid logic much more tightly than its predecessor; it is still worth sketching the idea behind HLF, if only to see precisely how far we can take some of its core concepts.

4.1 Resource Counting

To understand how a hybrid system can encode linearity, it is helpful to first look at an alternative presentation of linear logic using resource counting, based on [Cervesato et al., 2000]. $\Delta \vdash M : A[\mathrm{U}]$ is a linear judgement, where $\mathrm{U} : \Delta \to \mathbb{N}$ counts how many times each variable from Δ has been used in M. A typing derivation is then considered

[1] We distinguish Reed's Hybrid Logical Framework and ours throughout this thesis by using different short forms, HLF and HyLF, respectively.

linear (or valid) if U is a constant function 1 ($\mathbb{1}$). (As a convention, if we omit mappings in the annotation, they are equal to 0.) A fragment of the system is presented in Figure 4.1.

$$\frac{x : A \in \Delta}{\Delta \vdash x : A[x \mapsto 1]}\ x \qquad \frac{}{\Delta \vdash () : \top[\mathbb{1}]}\ \top_I$$

$$\frac{\Delta, x : A \vdash M : B[\mathrm{U}, x \mapsto 1]}{\Delta \vdash \widehat{\lambda} x.M : A \multimap B[\mathrm{U}]}\ \multimap_I$$

$$\frac{\Delta \vdash M : A \multimap B[\mathrm{U}] \quad \Delta \vdash N : A[\mathrm{V}]}{\Delta \vdash M\,\hat{}\,N : B[\mathrm{U} + \mathrm{V}]}\ \multimap_E$$

$$\frac{\Delta \vdash M : A[\mathrm{U}] \quad \Delta \vdash N : B[\mathrm{U}]}{\Delta \vdash \langle M, N \rangle : A \& B[\mathrm{U}]}\ \&_I$$

$$\frac{\Delta \vdash M : A \& B[\mathrm{U}]}{\Delta \vdash \pi_1 M : A[\mathrm{U}]}\ \&_{E_1} \qquad \frac{\Delta \vdash M : A \& B[\mathrm{U}]}{\Delta \vdash \pi_2 M : B[\mathrm{U}]}\ \&_{E_2}$$

Figure 4.1: Resource Counting Logic

The variable case simply marks the variable as used. To enforce linear usage of a freshly introduced variable x in the $\multimap_I$ case, we require that the function maps x to 1. Linear application requires that we merge the resources used to produce the function and its argument, which is done via the $\mathrm{U} + \mathrm{V}$ construction: $(\mathrm{U} + \mathrm{V})(x) = \mathrm{U}(x) + \mathrm{V}(x)$. For the additive product, the two branches should use the same resource counting function. The additive unit simply marks every resource as used.

4.2 Hybrid LF

Like LLF, Hybrid LF is a conservative extension of LF, which adds a new construct of *worlds*, along with some hybrid operations. Reed proposes that these worlds have a specific, fixed structure to them, in order to encode linear resource usage: to be more precise, worlds form a monoid with $*$ being a binary operation and ε – its neutral element. This way, the U function described in the previous section can be encoded by assigning each linear variable a world resource. These resources, joined together by the $*$ operator, are then used to annotate the typing judgement with a compound world consisting of markers for all the used variables. It serves our purpose to present only a fragment of the HLF system: we skip the $\&$ and $\top$ operators in this presentation, as we are interested in the hybrid aspect of this work.

It should be noted that it is not precisely a hybrid logic in the sense of [Prior, 1967], despite similarities in naming hybrid connectives and the name of the framework itself: neither the assumptions nor types are annotated with worlds, but rather the judgement itself.

Kinds	K	$::=$	$\mathsf{type} \mid \Pi x{:}A.K \mid \forall\alpha.K$
Types	A, B	$::=$	$a \cdot S \mid \Pi x{:}A.B \mid \forall\alpha.A \mid \Pi\alpha.A \mid A@p \mid {\downarrow}\alpha.A$
Heads	H	$::=$	$x \mid c$
Spines	S	$::=$	$\epsilon \mid M\,;S \mid p\,;S$
Terms	M, N	$::=$	$H \cdot S \mid \lambda x.M \mid \lambda\alpha.M$
Worlds	p, q	$::=$	$\alpha \mid p * q \mid \varepsilon$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A \mid \Gamma, \alpha{:}\omega$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A$

The kinds of HLF are extended, compared to LF, to include world abstractions. Similarly, types now include four new hybrid operators: $\forall\alpha.A$ stands for quantification over worlds, $\Pi\alpha.A$ is a function dependent on a world, $A@p$ can be read as "type A at world p", and the "here" operator ${\downarrow}\alpha.A$ binds the current world into variable α.

The main judgement of the system is $\Gamma \vdash M \Leftarrow A[p]$, read as "term M has type A at world p in context Γ". Unlike typical hybrid logics, HLF does not use worlds to annotate variables in Γ. Worlds cannot be declared in the signature, as they only serve a purpose when annotating the judgement itself. Moreover, worlds come equipped with a congruence relation $\equiv$, and equality in HLF is defined modulo it. The congruence on worlds is defined as:

$$\frac{}{(p * q) * r \equiv p * (q * r)} \qquad \frac{}{p * q \equiv q * p} \qquad \frac{}{p * \varepsilon \equiv p}$$

$$\frac{}{p \equiv p} \qquad \frac{q \equiv p}{p \equiv q} \qquad \frac{p \equiv q \quad q \equiv r}{p \equiv r} \qquad \frac{p \equiv p' \quad q \equiv q'}{p * q \equiv p' * q'}$$

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

$$\frac{\Gamma, \alpha : \omega \vdash M \Leftarrow A[p]}{\Gamma \vdash M \Leftarrow \forall \alpha.A[p]} \qquad \frac{\Gamma \vdash M \Leftarrow A[q]}{\Gamma \vdash M \Leftarrow A@q[p]}$$

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

$$\frac{\Gamma \vdash H \Rightarrow A \quad \Gamma \vdash S > A[\epsilon] \Rightarrow a \cdot S[p] \quad p \equiv q}{\Gamma \vdash H \cdot S \Leftarrow a \cdot S[q]}$$

$$\frac{}{\Gamma \vdash \epsilon > A[p] \Rightarrow A[p]} \qquad \frac{\Gamma \vdash p \quad \Gamma \vdash S > ([p/\alpha]A)[q] \Rightarrow P[r]}{\Gamma \vdash p\,; S > \Pi \alpha.A[q] \Rightarrow P[r]}$$

$$\frac{\Gamma \vdash M \Leftarrow A[\epsilon] \quad \Gamma \vdash S > ([M/x]_A B)[p] \Rightarrow C[q]}{\Gamma \vdash M\,; S > \Pi x{:}A.B[p] \Rightarrow C[q]}$$

$$\frac{\Gamma \vdash r \quad \Gamma \vdash S > ([r/\alpha]A)[p] \Rightarrow C[q]}{\Gamma \vdash S > \forall \alpha.A[p] \Rightarrow C[q]} \qquad \frac{\Gamma \vdash S > ([p/\alpha]A)[p] \Rightarrow C[q]}{\Gamma \vdash S > \downarrow\alpha.A[p] \Rightarrow C[q]}$$

$$\frac{\Gamma \vdash S > A[p] \Rightarrow C[r]}{\Gamma \vdash S > A@p[q] \Rightarrow C[r]} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \frac{c : A \in \Sigma}{\Gamma \vdash c \Rightarrow A}$$

$$\frac{\alpha : \omega \in \Gamma}{\Gamma \vdash \alpha} \qquad \frac{}{\Gamma \vdash \varepsilon} \qquad \frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p * q}$$

Figure 4.2: Type Checking in HLF

The typing rules for the system are given in Figure 4.2, with the rules for new constructs highlighted. Notice that types $\forall\alpha.A$, $A@p$ and $\downarrow\alpha.A$ are actually *refinement types* – they do not change the term when introduced, or the spine when eliminated.

HLF relies on substitutions for world and term variables. For the latter, it uses hereditary substitution much like the one presented for LLF. Since substituting for a world can never trigger a redex, world substitution can be given via a typical inductive definition.

Crucially, a linear implication can now be encoded on the meta level:

$$A \multimap B := \forall \alpha \,.\, {\downarrow}\beta \,.\, (A@\alpha \rightarrow B@(\beta * \alpha))$$

Here, α represents the fresh world associated with the assumption A. To ensure that this newly added assumption is indeed used in B, we require that α is part of the judgement annotation when constructing an object of type B – it therefore acts as a resource marker for using this particular assumption of type A.

4.3 From HLF to HyLF

Although HLF is interesting in its own right as an implementation of LLF, the key idea that we find worthy of further exploration is that of using hybrid connectives to allow more intricate encodings.

The world equivalence relation that HLF uses is a simple but powerful tool allowing for the encoding of some form of proof irrelevance: the parts of the object system that are encoded using worlds can, by exploiting the world structure, be exchanged to equivalent variants without leaving a trace. This is appealing, for instance, when encoding modal logics, where the world reachability relation is typically described using a Kripke structure or as a direct relation on the worlds (in IS5 logic, as an example, the relation in question is an equivalence), which should remain proof irrelevant.

At the same time, seeing that HLF is a well-behaved system using (some) hybrid connectives gives us hope that a more general framework based on hybrid logic is achievable. Combining support for arbitrary structure on worlds with the expressive power of hybrid operators should, in principle, allow for the encoding of whole classes of logics. The meta-level encoding of the $\multimap$ operator from linear logic is but one example of a possible application of such a generalised framework.

Of course, with great power comes great responsibility, and such a general system would likely require some work on the part of the user when they define their own instance of it. A completely arbitrary algebra on worlds is not guaranteed to be well-behaved, and might endanger decidability. It is the user's responsibility to ensure that their world algebra behaves as required by the framework. Still, this approach promises more flexibility and ease of use for encodings that use a less "mainstream" meta-logic and context behaviour, while still providing them with the full power of the logical framework methodology, including higher-order abstract syntax.

Chapter 5

Hybrid Extensions in a Logical Framework

Originally published in: Logical Frameworks and Meta Languages: Theory and Practice, 2014

Joint work with: Taus Brock-Nannestad, Nicolas Guenot, Carsten Schürmann

Abstract

We discuss the extension of the LF logical framework with operators for manipulating *worlds*, as found in hybrid logics or in the HLF framework. To overcome the restrictions of HLF, we present a more general approach to worlds in LF, where the structure of worlds can be described in an explicit way. We give a canonical presentation for this system and discuss the encoding of logical systems, beyond the limited scope of linear logic that formed the main goal of HLF.

5.1 Hybrid Logics and LF

The LF logical framework [Harper et al., 1993] has been successfully used to represent adequately many logics and systems, and it greatly simplifies encodings by providing a representation language with an object-level based on the λ-calculus. This offers the possibility to use *higher-order abstract syntax*, as well as hypothetical judgements, where the usual notions of abstraction and substitution are primitives.

There are however some systems that cannot be encoded adequately in LF without heavy manipulation of structures that must be dealt with manually both when defining their encoding and when reasoning about the system. One such example can be obtained by extending a standard logic, such as intuitionistic logic, by *hybrid* operations as suggested by [Prior, 1967] and introduced later in standard proof theory — some theory for hybrid logics can be found for example in [Tzakova, 1999], [Areces et al., 2001] and [Galmiche and Salhi, 2011]. The idea of hybrid logics is simply to make explicit the Kripke semantics usually given to logics, in particular modal logics, by allowing inference rules to manipulate the *worlds* of the semantics. This yields elegant proof systems for logics with connectives performing complex operations on these worlds. For example, one can define a natural deduction system for the intuitionistic form of modal logics [Simpson, 1994] where rules for $\Box$ are:

$$\Box_I \frac{\Gamma, xRy \vdash A[y]}{\Gamma \vdash \Box A[x]} \qquad \Box_E \frac{\Gamma \vdash \Box A[x] \quad (xRy)}{\Gamma \vdash A[y]}$$

where $A[x]$ indicates that A is provable at a particular world x, while an assumption of the shape xRy in the context is a witness of the condition that for this rule to hold, y must be reachable from x in the relation R of the associated Kripke semantics. The properties of such a modal logic then depend on the axioms on the relation used in the semantics, and for example a reflexive and transitive relation yields IS4.

The problem with the encoding of such a system in LF is that worlds and assumptions of shape xRy must be encoded and manipulated manually, so that each time a property of R needs to be used, the same procedure is applied. What is lacking in LF is support for manipulating structures inside the syntax to, for example, automatically handle reflexivity, transitivity or other properties. Such an infrastructure has been developed for the specific purpose of encoding linearity in the HLF framework [Reed, 2009] that extends LF with some support for hybrid operations. In HLF, the types and terms can use worlds that are not always variables but can also be *compounds* built from a binary $*$ operator with unit ε. This structure of worlds was used to encode linear implication in HLF at the level of the representation language: reasoning *linearly* is possible in HLF in the sense that $\multimap$ is available as a type — a macro using primitive operations on worlds.

If we consider the naive encoding of a modal logic in LF, we need to explicitly manipulate the worlds and define the constants corresponding to the rules of the congruence:

```
o : type.
ω : type.
pf : o → ω → type.
rc : ω → ω → type.
```

```
⊃ : o → o → o.
□ : o → o.
◊ : o → o.

refl : {α : ω} rc α α.
trans : {α, γ, σ : ω} rc α γ → rc γ σ → rc α σ.

□_I : {A : o}{α : ω} ({γ : ω} rc α γ → pf A γ) → pf (□ A) α.
□_E : {A : o} {α, γ : ω} pf (□ A) α → rc α γ → pf A γ.
(…)
```

but this encoding is not adequate in LF because of the many ways of making two worlds equivalent. The situation could not really be improved in HLF, since the relation on worlds that is needed here does not fit the syntax of $*$ and ε under AC-unification. As a consequence, encoding modal logics in LF or HLF is problematic from the viewpoint of adequacy, since there can be several proofs of reachability that are all identified in the proof on paper. In a similar way, the work we present stems from an attempt to represent and reason about a calculus based on hypersequents for some variant of linear logic [Montesi, 2013], which fails in HLF because of the structure of worlds: it is used to ensure linearity and cannot be used to also represent the connections between sequents in a hypersequent. This is essentially the same problem as encountered when encoding modal logics in HLF, which is uneasy since the relation on worlds is incompatible with the notion of equality on worlds in this framework.

The goal of the work presented here is therefore to define a more general extension of LF that allows a more extensive use of the expressivity of hybrid operations. To do this, we follow the standard presentation of LF in its canonical form [Harper and Licata, 2007] and add ingredients from HLF, and more, to support the encoding of advanced hybrid systems. The key to do this is the generalisation of the structure of worlds, from a fixed set of operations $\{*, \varepsilon\}$ to an abstract notion combining any number of operators and an equivalence relation on worlds. The resulting framework is then parametric in the definition given for worlds.

We start in Section 5.2 by describing our hybrid framework, called HyLF, and discuss its reduction, normal forms and notions of substitution in this setting. Then, in Section 5.3, we illustrate the use of the system by considering an encoding of modal logics exploiting an advanced structure of worlds.

5.2 Extending Hybrid LF

Instead of starting from HLF and enriching the system, we go back to the standard framework of LF [Harper et al., 1993], in its form relying on canonical typing derivations [Harper and Licata, 2007]. In particular, we use the standard λ-calculus as our base, without *spines* [Cervesato and Pfenning, 2003], to keep the theory as simple as possible. The language of terms and types of our HyLF framework is an extension of canonical LF that supports various user-defined operators on worlds.

In the following, we denote by letters such as x, y and z *term variables*, by M, N and V *canonical terms*, by R and S *atomic terms*, by c *term constants*, by A and B *canonical types*, by F and G *atomic types*, by a *type constants* and by K or L *kinds* of HyLF. Moreover, we use Greek letters such as α or γ for *world variables* and p or q for *worlds* in general. Terms, types and kinds are defined by the following grammar:

$$\begin{array}{rcl} K, L & ::= & \mathtt{type} \mid \Pi x : A.K \mid \forall\alpha.K \\ A, B & ::= & F \mid \Pi x : A.B \mid \forall\alpha.A \mid A@p \mid {\downarrow}\alpha.A \\ F, G & ::= & a \mid F\ M \mid F\ \{p\} \\ M, N & ::= & R \mid \lambda x.M \mid \lambda\{\alpha\}.M \mid M\ \mathtt{at}\ p \mid \mathtt{here}\ \alpha.M \\ R, S & ::= & x \mid c \mid R\ M \mid R\ \{p\} \mid R\ \mathtt{to}\ p \mid \mathtt{ccw}\ R \end{array} \tag{5.1}$$

This system is similar to HLF [Reed, 2009], with primitives at the level of terms reflecting the elimination rules of the world operators in the object language. For the sake of simplicity, no cartesian product is used in HyLF, and we collapse the dependent product and the universal quantification when it comes to worlds.

The generalisation of HyLF with respect to HLF lies in the way worlds can be defined: instead of defining one fixed structure of worlds with the operator $*$ and its unit ε, we will make the whole framework parametric in the definition of worlds. The first step in the instantiation of the framework is to define the language of worlds, always of the shape:

$$p, q \ ::= \ \alpha \mid o(\vec{p}) \qquad \text{where } o : |\vec{p}| \in \mathcal{O}$$

where o is an operator defined in the *operators signature* $\mathcal{O}$ that contains entries of the form $o : k$ indicating that o is an operator of arity k, and where $\vec{p}$ is a sequence of worlds of length k. The second step of the instantiation is to define the *equivalence* relation $\equiv$ over worlds, which must form a congruence for the operators in $\mathcal{O}$, by specifying additional equations.

Definition 5.1. *An* instance HYLF$(\mathcal{O}, \equiv)$ *of the* HYLF *parametric framework is defined by providing some operators signature $\mathcal{O}$ and a congruence $\equiv$ over worlds.*

In the following, we will write HYLF when discussing the properties of any particular instance, and specify the exact operators signature and congruence only if necessary. The typing rules for the canonical term level of HYLF are shown in Figure 5.1. Binding a new world can be done in this system through a λ-abstraction, but also with $\texttt{here}\,\alpha.M$, a construct binding the *current* world.

Example 5.1. *The structure of worlds described in* HLF *is obtained in* HYLF *by an instantiation where the language of worlds is defined through the signature* $\{* : 2, \varepsilon : 0\}$, *which corresponds to the grammar:*

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

and where the congruence over worlds is defined by the rules:

$$\frac{}{(p * q) * p' \equiv p * (q * p')} \qquad \frac{}{p * q \equiv q * p} \qquad \frac{}{p * \varepsilon \equiv p}$$

$$\frac{}{p \equiv p} \qquad \frac{q \equiv p}{p \equiv q} \qquad \frac{p \equiv q \quad q \equiv p'}{p \equiv p'} \qquad \frac{p \equiv p' \quad q \equiv q'}{p * q \equiv p' * q'}$$

which are implementing AC-unification. ■

In the rules from Figure 5.1, we use two kinds of judgements to indicate whether the type is *synthesized* from the term or is *checked* against the term, always at some world p, which will be denoted by $\Omega; \Gamma \vdash R \Rightarrow A[p]$ and by $\Omega; \Gamma \vdash M \Leftarrow A[p]$, respectively. In both cases, Ω is a set of world variables and Γ is a list of assumptions of the shape $x : A[p]$. This means that assumptions are assigned some world, as is often done in sequent calculi for modal logics [Galmiche and Salhi, 2011], but not in HLF. The two judgement forms are meant to enforce a separation between canonical and atomic terms, so that all terms typed are canonical. Moreover, in these rules:

- the condition $p \in \mathcal{W}$ ensures that p appears in the set $\mathcal{W}$ of worlds *well-formed* according to the signature $\mathcal{O}$,
- the list Σ is some *constants signature* implicitly associated to the judgement — we could write $\vdash_\Sigma$ but omit this for the sake of readability,

$$\Pi i \, \frac{\Omega; \Gamma, x : A[p] \vdash M \Leftarrow B[p]}{\Omega; \Gamma \vdash \lambda x.M \Leftarrow \Pi x : A.B[p]}$$

$$\downarrow i \, \frac{\Omega; \Gamma \vdash M\{p/\alpha\} \Leftarrow A\{p/\alpha\}[p]}{\Omega; \Gamma \vdash \mathtt{here}\, \alpha.M \Leftarrow \downarrow\alpha.A[p]}$$

$$@i \, \frac{\Omega; \Gamma \vdash M \Leftarrow A[q] \quad p \in \mathcal{W}}{\Omega; \Gamma \vdash M \,\mathtt{at}\, q \Leftarrow A@q[p]} \qquad \forall i \, \frac{\Omega, \alpha; \Gamma \vdash M \Leftarrow A[p] \quad \alpha \notin \Omega}{\Omega; \Gamma \vdash \lambda\{\alpha\}.M \Leftarrow \forall\alpha.A[p]}$$

. . . .

$$s \, \frac{\Omega; \Gamma \vdash R \Rightarrow F[q] \quad p \equiv q \quad p \in \mathcal{W}}{\Omega; \Gamma \vdash R \Leftarrow F[p]}$$

. . . .

$$\Pi e \, \frac{\Omega; \Gamma \vdash R \Rightarrow \Pi x : A.B[p] \qquad \Omega; \Gamma \vdash M \Leftarrow A[p]}{\Omega; \Gamma \vdash R\ M \Rightarrow B[M/x]_p[p]}$$

$$\downarrow e \, \frac{\Omega; \Gamma \vdash R \Rightarrow \downarrow\alpha.A[p]}{\Omega; \Gamma \vdash \mathtt{ccw}\, R \Rightarrow A\{p/\alpha\}[p]}$$

$$x \, \frac{x : A[p] \in \Gamma \quad p \in \mathcal{W}}{\Omega; \Gamma \vdash x \Rightarrow A[p]} \qquad @e \, \frac{\Omega; \Gamma \vdash R \Rightarrow A@p[q] \quad p \in \mathcal{W}}{\Omega; \Gamma \vdash R \,\mathtt{to}\, p \Rightarrow A[p]}$$

$$c \, \frac{c : A \in \Sigma \quad p \in \mathcal{W}}{\Omega; \Gamma \vdash c \Rightarrow A[p]} \qquad \forall e \, \frac{\Omega; \Gamma \vdash R \Rightarrow \forall\alpha.A[q] \quad p \in \mathcal{W}}{\Omega; \Gamma \vdash R\ \{p\} \Rightarrow A\{p/\alpha\}[q]}$$

Figure 5.1: Typing rules for HyLF terms

- we can go from one kind of judgement to the other only in the s rule, which *swaps* from synthesis to checking, and this is also the only rule relying on the congruence,
- in the axioms x and c, the context Γ and signature Σ should be checked for well-formation, following rules that we omit here but are straightforward,
- the notation $\{p/\alpha\}$ corresponds to the standard notion of capture-avoiding substitution in a term, of a world for a world variable,
- the notation $[M/x]_p$ corresponds to the notion of hereditary substitution, which we define below.

$$
{\scriptstyle a}\,\frac{a : K \in \Sigma}{\Omega;\Gamma \vdash a \Rightarrow K}
\qquad
{\scriptstyle \forall f}\,\frac{\Omega;\Gamma \vdash F \Rightarrow \forall\alpha.K \quad p \in \mathcal{W}}{\Omega;\Gamma \vdash F\ \{p\} \Rightarrow K\{p/\alpha\}}
$$

$$
{\scriptstyle \Pi f}\,\frac{\Omega;\Gamma \vdash F \Rightarrow \Pi x : A.K \quad \Omega;\Gamma \vdash M \Leftarrow A[p]}{\Omega;\Gamma \vdash F\ M \Rightarrow K[M/x]_p}
$$

$$
{\scriptstyle f}\,\frac{\Omega;\Gamma \vdash F \Rightarrow \texttt{type}}{\Omega;\Gamma \vdash F :: \texttt{type}}
\qquad
{\scriptstyle \forall}\,\frac{\Omega,\alpha;\Gamma \vdash A :: \texttt{type} \quad \alpha \notin \Omega}{\Omega;\Gamma \vdash \forall\alpha.A :: \texttt{type}}
$$

$$
{\scriptstyle \Pi}\,\frac{\Omega;\Gamma \vdash A :: \texttt{type} \quad \Omega;\Gamma, x : A[p] \vdash B :: \texttt{type}}{\Omega;\Gamma \vdash \Pi x : A.B :: \texttt{type}}
$$

$$
{\scriptstyle \downarrow}\,\frac{\Omega,\alpha;\Gamma \vdash A :: \texttt{type}}{\Omega;\Gamma \vdash \downarrow\alpha.A :: \texttt{type}}
\qquad
{\scriptstyle @}\,\frac{\Omega;\Gamma \vdash A :: \texttt{type} \quad p \in \mathcal{W}}{\Omega;\Gamma \vdash A@p :: \texttt{type}}
$$

$$
{\scriptstyle t}\,\frac{}{\Omega;\Gamma \vdash \texttt{type} :: \texttt{kind}}
\qquad
{\scriptstyle \forall k}\,\frac{\Omega,\alpha;\Gamma \vdash K :: \texttt{kind} \quad \alpha \notin \Omega}{\Omega;\Gamma \vdash \forall\alpha.K :: \texttt{kind}}
$$

$$
{\scriptstyle \Pi k}\,\frac{\Omega;\Gamma \vdash A :: \texttt{type} \quad \Omega;\Gamma, x : A[p] \vdash K :: \texttt{kind}}{\Omega;\Gamma \vdash \Pi x : A.K :: \texttt{kind}}
$$

Figure 5.2: Kinding rules for HyLF

Finally, we show in Figure 5.2 rules for kinding type families in HyLF, which are again an extension of the standard rules for LF, where abstraction can be performed over worlds and atomic types can also be applied to worlds. Notice that the same conditions are used as in Figure 5.1, so that contexts and signatures should be checked at axiom rules ***a*** and ***t***. There are three new judgements in these rules, $\Omega;\Gamma \vdash F \Rightarrow K$, $\Omega;\Gamma \vdash A :: \texttt{type}$, and $\Omega;\Gamma \vdash K :: \texttt{kind}$ which represent having a certain kind K, the validity of a type A, and the validity of a kind K, respectively.

The term level of this system reflects the extension of the type level by offering primitives to manipulate worlds. The meaning of the constructs can be intuitively understood as:

- the universal quantification over worlds $\forall\alpha.A$ yields a simple mechanism using abstraction and application, distinguished from the standard λ-calculus constraints by the $\{p\}$ syntax used in abstraction and application, so that $\lambda\{\alpha\}.M$ is related only to $R\ \{p\}$ and not standard application,

- the world localisation operation $A@p$ yields the two operations $M\,\mathtt{at}\,p$ and $R\,\mathtt{to}\,p$ which indicate that some term M must be evaluated at a world p, and that R has been moved to the world p, respectively: this plays a role in the semantics of computation in this setting, where reduction happens at a certain world to reflect the constraints imposed by typing judgements,
- the current world (binding) operation $\downarrow\alpha.A$ is similar to the world quantification but it yields a mechanism for binding the world where a term M will be evaluated using the operation $\mathtt{here}\,\alpha.M$, and associating this name to the world where some term R is currently evaluated, with the operation of *call-current-world* written $\mathtt{ccw}\,R$.

Canonical forms. In a logical framework such as LF, it is important to be able to isolate *canonical forms*, so that adequacy can later be proven, to correctly relate structures being encoded and their actual LF encodings. This is why the typing rules for HYLF are *bidirectional* and restrict the formation of terms to the grammar given in (5.1). However, we need to have the notion of *reduction* to offer some way to represent the dynamics of systems we encode — for example, reductions for cut elimination in a given logic presented in a sequent calculus. This cannot be achieved inside a canonical system, as reductions correspond to elimination of *detours*, where an introduction rule appears immediately above the corresponding elimination rule in a typing derivation.

In order to recover a system where reductions are possible, we need to bypass the restrictions imposed by the use of $\Rightarrow$ and $\Leftarrow$ annotations. Moving from one kind of judgement to the other is already possible using the swap rule $\boldsymbol{s}$. All we need is therefore a rule $\boldsymbol{s}^{-1}$ opposite to this rule:

$$\boldsymbol{s}^{-1}\ \frac{\Omega; \Gamma \vdash M \Leftarrow A[q] \quad p \equiv q \quad p \in \mathcal{W}}{\Omega; \Gamma \vdash M \Rightarrow A[p]}$$

to be able to type non-canonical forms. Notice that this rule applies at any type and not just on atomic types, yielding the ability to type sequences of introduction and elimination rules. Here, we chose the canonical presentation where this rule does not appear, and we kept reduction as an *"external"* device. In the following, we will call *well-typed* a term M such that for a given type A there exists a typing derivation for $\Omega; \cdot \vdash M \Leftarrow A[p]$ in HyLF — this implies that M is canonical, since the $\boldsymbol{s}^{-1}$ rule is not used and it is the only rule that can break canonicity.

Reduction. Allowing non-canonical forms allows us to type more terms, but we want to reason under some equivalence relation such that any non-canonical term is associated to a canonical term. This relies on the notion of reduction over non-canonical terms from a grammar where M appears in the category R as shown in (5.1) — this can be obtained with the $\boldsymbol{s}^{-1}$ rule shown above. Since in the hybrid setting all terms are reduced *at a certain world*, the reduction relation $\longrightarrow_p$ must be parametrized by some p where evaluation happens. The main reduction rules are:

$$\begin{aligned}(\lambda x.M)\ N &\longrightarrow_p M\{N/x\}\\ \mathtt{ccw}\,(\mathtt{here}\,\alpha.M) &\longrightarrow_p M\{p/\alpha\}\\ (M\,\mathtt{at}\,p)\,\mathtt{to}\,p &\longrightarrow_p M\\ (\lambda\{\alpha\}.M)\ \{q\} &\longrightarrow_p M\{q/\alpha\}\end{aligned} \tag{5.2}$$

where the first is simply β-reduction and the others represent the elimination of other detours in HyLF typing derivations. But there are more rules needed here, to allow reduction under any construct. These rules are standard in most cases, and we have for example:

$$\begin{aligned}\lambda x.M &\longrightarrow_p \lambda x.N && \textit{if } M \longrightarrow_p N\\ M\ N &\longrightarrow_p M'\ N' && \textit{if } M \longrightarrow_p M' \textit{ and } N \longrightarrow_p N'\\ \mathtt{ccw}\,M &\longrightarrow_p \mathtt{ccw}\,N && \textit{if } M \longrightarrow_p N\\ \mathtt{here}\,\alpha.M &\longrightarrow_p \mathtt{here}\,\alpha.N && \textit{if } M \longrightarrow_p N\end{aligned}$$

but the reduction rules involving the `at` and `to` operators have a specific effect on the world where evaluation happens:

$$\begin{aligned}M\,\mathtt{at}\,q &\longrightarrow_p N\,\mathtt{at}\,q && \textit{if } M \longrightarrow_q N\\ M\,\mathtt{to}\,p &\longrightarrow_p N\,\mathtt{to}\,p && \textit{if } M \longrightarrow_q N \quad \textit{for some } q \in \mathcal{W}\end{aligned}$$

corresponding to the meaning of these operations. Indeed, even when $M\,\mathtt{at}\,q$ is evaluated at p, the evaluation of M is performed at world q, and if M is evaluated into N at q, then $M\,\mathtt{to}\,N$ transfers the result of the evaluation to world p — this can be related to the `fetch` and `get` operations affecting the current world of evaluation in a modal λ-calculus as the one presented in [Murphy VII et al., 2004].

Apart from this use of worlds in the evaluation of terms, the computational semantics of HyLF is based on standard notions. In particular, the key element during reduction is *substitution*. There are two kinds of substitution used in (5.2): the substitution $\{M/x\}$ of some term u for a term variable x, capture-avoiding and using α-conversion for λ-abstractions, and the substitution $\{p/\alpha\}$ of a complex world p for a world variable α. This second form of substitution is defined in a standard way, relying on the α-conversion of world names in binding operations of world abstraction and of current world abstraction. Intuitively, this is a simultaneous replacement of all the free occurrences of α by the world p, in any term. We will not discuss here the properties of the $\longrightarrow_p$ reduction or of its reflexive, transitive closure $\twoheadrightarrow_p$.

Substitution. The dynamics of non-canonical terms is based on the notion of substitution. In the canonical HyLF system, we cannot define the usual notion of substitution because it does not necessarily yield a canonical form. Such a notion can be defined here, and thus preserve canonical forms, only if it is parametrized to an *hereditary* form of substitution, where the redexes created by substitution are reduced immediately [Watkins et al., 2004].

Definition 5.2. *For well-typed terms M, N, some variable x and a world p, the* hereditary substitution $M[N/x]_p$ *of N for x in M at world p is defined by the relation presented in Figure 5.3.*

Notice that only the case of crossing an application can create new redexes in LF, but here there are three more non-trivial cases corresponding to other reduction rules in HyLF. However, none of these new cases trigger a term substitution, and substitution of worlds never creates new redexes, so that it does not need to be defined hereditarily to stay in the canonical fragment. Indeed, all redexes in (5.2) rely on the shape of terms rather than on worlds, except of $(M\,\mathtt{at}\,p)\,\mathtt{to}\,q$, but it is well-typed only if $p = q$. We can now state that hereditary substitution is actually a particular implementation of the reductions shown in (5.2). Proof for some representative cases is provided in the Appendix.

Theorem 5.1 (Hereditary substitution).
For any terms M and N, if for a given world p there exists V such that $M[N/x]_p = V$, then we have the reduction $M\{N/x\} \twoheadrightarrow_p V$.

Notice that the correspondence of hereditary substitution and reduction of non-canonical HyLF terms is established only for well-typed

$$\frac{}{x[N/x]_p = N} \qquad \frac{}{y[N/x]_p = y} \qquad \frac{R[N/x]_p = R' \quad M[N/x]_p = M'}{(R\ M)[N/x]_p = R'\ M'}$$

$$\frac{}{c[N/x]_p = c} \qquad \frac{R[N/x]_p = \lambda y.V' \quad M[N/x]_p = M' \quad V'[M'/y]_p = V}{(R\ M)[N/x]_p = V}$$

$$\frac{M[N/x]_p = V}{(\lambda y.M)[N/x]_p = \lambda y.V} \qquad \frac{M[N/x]_p = V}{(\lambda\{\alpha\}.M)[N/x]_p = \lambda\{\alpha\}.V}$$

$$\frac{M[N/x]_q = V}{(M\,\texttt{at}\,q)[N/x]_p = V\,\texttt{at}\,q} \qquad \frac{M[N/x]_p = V}{(\texttt{here}\,\alpha.M)[N/x]_p = \texttt{here}\,\alpha.V}$$

$$\frac{R[N/x]_q = S}{(R\,\texttt{to}\,p)[N/x]_p = S\,\texttt{to}\,p} \qquad \frac{R[N/x]_q = M\,\texttt{at}\,p}{(R\,\texttt{to}\,p)[N/x]_p = M}$$

$$\frac{R[N/x]_p = S}{(\texttt{ccw}\,R)[N/x]_p = \texttt{ccw}\,S} \qquad \frac{R[N/x]_p = \texttt{here}\,\alpha.M}{(\texttt{ccw}\,R)[N/x]_p = M\{p/\alpha\}}$$

$$\frac{R[N/x]_p = S}{(R\ \{q\})[N/x]_p = S\ \{q\}} \qquad \frac{R[N/x]_p = \lambda\{\alpha\}.M}{(R\ \{q\})[N/x]_p = M\{q/\alpha\}}$$

Figure 5.3: Hereditary substitution

terms, which simplifies the situation as it prevents the creation of redexes through world substitutions making world substitution non-hereditary.

A critical property of the notion of hereditary substitution is that it preserves typeability in the canonical system, along with the fact that given two well-typed terms M and N, we can perform the substitution of N for a variable inside M. Proving this requires an induction, made more complex by the case where a β-redex is created, since it involves the type of the substituted N. This is however standard, and we only need to consider a simple approximation of the type of N.

Definition 5.3 (Type erasure). *For a term N of type A, the* simple type

$\tau(N)$ of N is defined as $[\![A]\!]_\tau$, where:

$$\begin{array}{ccc} & [\![\Pi x : A.B]\!]_\tau = [\![A]\!]_\tau \to [\![B]\!]_\tau & [\![\forall\alpha.A]\!]_\tau = [\![A]\!]_\tau \\ [\![a]\!]_\tau = a & [\![F\ M]\!]_\tau = [\![F]\!]_\tau & [\![A@p]\!]_\tau = [\![A]\!]_\tau \\ & [\![F\ \{p\}]\!]_\tau = [\![F]\!]_\tau & [\![\downarrow\alpha.A]\!]_\tau = [\![A]\!]_\tau \end{array}$$

We can now state the main theorem allowing us to use the notion of substitution in the canonical presentation of HyLF. More details on this result in standard LF but also in HLF can be found in the literature [Harper and Licata, 2007, Reed, 2009]. A generalized formulation of this theorem, together with representative cases of its proof, can be found in the Appendix.

Theorem 5.2 (Substitution)**.**
For any terms M and N such that $\Omega;\Gamma, x : A[q], \Delta \vdash M \Leftarrow B[p]$ and $\Omega;\Gamma \vdash N \Leftarrow A[q]$, it follows that $\Omega;\Gamma, \Delta[N/x] \vdash M[N/x] \Leftarrow B[N/x][p]$.

There is another theorem that allows us to perform the same operation on worlds, corresponding to the observation that if a world variable is used in a typing derivation, this derivation is parametric in that variable. Consistently replacing the variable with any given world always therefore yields a valid typing derivation. The proof of representative cases can again be found in the Appendix.

Theorem 5.3 (World substitution)**.**
For a world q and a term M with $\Omega, \alpha;\Gamma \vdash M \Leftarrow B[p]$, there is a derivation of the judgement $\Omega;\Gamma\{q/\alpha\} \vdash M\{q/\alpha\} \Leftarrow B\{q/\alpha\}[p\{q/\alpha\}]$.

We will not go into further details about the properties of terms and derivations forming the meta-theory of the HyLF framework, but rather present examples of how extending LF with hybrid constructs allows us to elegantly represent logics that are defined by a hybrid system.

5.3 Encoding Logics in HyLF

We consider here two ways of using the hybrid operations to encode logics and systems. The first one is the standard encoding idea of LF, where the given system is defined with rules represented by typed constants added to a signature, so that adequacy can be proven between the system as seen *"on paper"* and its LF representation. The second approach follows the HLF example of encoding certain logical connectives into the type level of LF, and arguing that the typing rules from LF correspond to the rules intended for this connective, so that the form of reasoning embodied by this connective is made available to encode further systems, using standard encodings.

Notice that this first approach, in the HʏLF framework, requires not only the definition of typed constants representing the rules of the system, but also the definition of operators and a congruence on worlds to instantiate the framework, as it is now parametric. However, there is a trade-off, where the added specification of the level of worlds subsequently makes the representation of the rules simpler.

Intuitionistic modal logics. The most natural system that can be encoded using the hybrid operations from HʏLF is a natural deduction calculus for intuitionistic modal logics defined by Simpson [Simpson, 1994], using rules shown in Figure 5.4. In this system, the relation on worlds defining the particular flavour of modal logic used is mentioned explicitly, so that the same rules properly represent different modal logics, for example IK, IS4 or IS5. This system is well-suited for a presentation in HʏLF, as we will be able to define inference rules as constants and simply change the definition of the congruence on worlds to switch between different logics — by specifying exactly the axioms defining the Kripke semantics of these logics.

The presentation of the IK system is made slightly more precise than the one given by Simpson, on the syntactic level, as we use the sequent notation and distinguish between three parts of a context, denoted by Ω, Σ and Γ, to hold available world names, assumptions on R and logical assumptions, respectively. In this system, worlds are always just names such as x, y or z. A sequent is written $\Omega; \Sigma; \Gamma \vdash A[x]$ for provability of A at a world x under these three contexts. The formulas are defined by the standard grammar:

$$A, B \ ::= \ a \ | \ A \supset B \ | \ \Box A \ | \ \Diamond A$$

where one can observe that the IK system is *modal* but not hybrid in the sense that worlds are used in sequents but not mentioned in formulas. The presentation we have given is equivalent to the original one [Simpson, 1994], and the distinctions made inside contexts are meant to make adequacy as obvious as possible for the given encoding of the system in HʏLF.

The first step of the encoding is to define the structure of the worlds, and the congruence relation $\equiv$. In all modal logics that can be rep-

$$ax\ \frac{A[x] \in \Gamma}{\Omega;\Sigma;\Gamma \vdash A[x]} \qquad {\supset} i\ \frac{\Omega;\Sigma;\Gamma, A[x] \vdash B[x]}{\Omega;\Sigma;\Gamma \vdash A \supset B[x]}$$

$${\supset} e\ \frac{\Omega;\Sigma;\Gamma \vdash A[x] \qquad \Omega;\Sigma;\Gamma \vdash A \supset B[x]}{\Omega;\Sigma;\Gamma \vdash B[x]}$$

$$\Box i\ \frac{\Omega, y;\Sigma, xRy;\Gamma \vdash A[y]}{\Omega;\Sigma;\Gamma \vdash \Box A[x]} \qquad \Box e\ \frac{\Omega;\Sigma, xRy;\Gamma \vdash \Box A[x]}{\Omega;\Sigma, xRy;\Gamma \vdash A[y]}$$

$$\Diamond i\ \frac{\Omega;\Sigma, xRy;\Gamma \vdash A[y]}{\Omega;\Sigma, xRy;\Gamma \vdash \Diamond A[x]}$$

$$\Diamond e\ \frac{\Omega;\Sigma;\Gamma \vdash \Diamond A[x] \qquad \Omega;\Sigma, xRy;\Gamma, A[y] \vdash B[z]}{\Omega;\Sigma;\Gamma \vdash B[z]}$$

Figure 5.4: Inference rules for the basic logic IK

resented using the rules of IK shown above, the grammar of worlds is:

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

where R is of arity 2 and is meant to represent reachability as described by the relation of the Kripke semantics, and $*$ and ε of arities 2 and 0 respectively are used to encode sets of worlds.

Remark 5.1. *In the natural deduction presentation of basic modal logic* IK *as well as in its extensions, the assumptions of the shape pRq involve only world names, so that it should be xRy, but our grammar does not enforce such a restriction. Indeed, the current definition of* HyLF *only allows the operators to be specified with some arity, and through a complete grammar.*

This is however not a problem, since the encoding of rules preserves the invariant that in any world of the shape pRq, both p and q are variables: the worlds inside the assumptions are never accessed and decomposed by the rules, but simply compared, so that replacing a variable by a compound world does not break the encoding.

The precise meaning of operators on worlds is partly given through the congruence. There will be a part of this relation common to all the systems based on the rules given for IK, which will actually be the congruence for the logic IK itself. Then, extending $\equiv$ with other axioms concerning R yields other, richer logics. This basic part of the congruence $\equiv$ is defined by the equations:

$$\begin{array}{rclrcl} p * q & \equiv & q * p & \quad p * \varepsilon & \equiv & p \\ p * (q * o) & \equiv & (p * q) * o & \quad p * p & \equiv & p \end{array}$$

We will now define the constants meant to represent the inference rules of the system. These terms are given types representing the structure of formulas and sequents in IK, following the usual approach of LF, where $\to$ stands for a non-dependent product:

```
o : type.
pf : o → ∀ α. type.
⊃ : o → o → o.
□ : o → o.
◇ : o → o.
```

Then, the purely implicational part of IK is described by rules for $\supset$ which do not use the reachability relation:

```
⊃_I : (pf A {x} → pf B {x}) → pf (A ⊃ B) {x}
⊃_E : pf (A ⊃ B){x} → pf A {x} → pf B {x}
```

where we omit all the outer bindings on A, B and x, that are necessary only to obtain a fully closed term and can be easily reconstructed.

We now consider the modal part of the system, encoding the rules for $\Box$ and $\Diamond$, which actually affect the worlds:

```
□_I : (∀ α . pf A {α} @ (s * x R α)) → pf (□ A){x} @ s.
□_E : pf (□ A) {x} @ (s * x R y) → pf A {y} @ (s * x R y).
◇_I : pf A {x} @ (s * x R y) → pf (◇ A) {y} @ (s * y R x).
◇_E : pf (◇ A) {x} @ s →
      (∀ α . pf A {α} @ s → pf B {y} @ (s * x R α)) →
      pf B {y} @ s.
```

where we omit bindings on A, B, s, x and y. This encoding of the IK rules can be proven adequate in a straightforward way, since all types used rely on the following correspondence between a sequent and its encoding:

$$\Omega; \Sigma; \Gamma \vdash A[x] \quad \leftrightarrow \quad \texttt{pf A \{x\} @ s}$$

where s is some world representing Σ by turning recursively Σ', xRy into `s' * x R y`, where `s'` represents Σ', and ε is the empty set. Then, Ω and

Γ are handled implicitly as usual in LF using the binders on world and term variables from the representation language. The same applies to logics such as for example IS4, where the Kripke semantics contains the reflexivity and transitivity axioms for R. Representing IS4 is achieved in our encoding by extending the congruence with the equations:

$$xRx \equiv \varepsilon \qquad xRy * yRz \equiv xRz$$

without changing the rules from Figure 5.4. The effect of this extension is to modify the set of formulas validated by the logic, so that, in particular we can prove the axioms $\Box A \supset A$ and $A \supset \Diamond A$ by using reflexivity, as well as $\Box A \supset \Box\Box A$ and $\Diamond\Diamond A \supset \Diamond A$ by using transitivity. These axioms illustrate how the use of the congruence can control precisely the modal logic being represented, just as the axioms of some Kripke semantics. In order to obtain IS5, we just add the following axiom:

$$xRy * xRz \equiv yRz$$

to the ones used before to define IS4. Various other axioms from the standard proof theory of modal logics can be added in a similar way.

Linear reasoning. Another use of hybrid operations in HyLF consists in extending the representation language of types with an encoding of linear implication $\multimap$. This allows to subsequently represent other systems using a type $A \multimap B$, and in particular this is the way a sequent calculus for linear logic can be adequately represented, as done in HLF [Reed, 2009] where it was the goal of the introduction of hybrid operators. We can use in HyLF the exact same encoding as in HLF, provided that we use:

$$A \multimap B \triangleq \forall\alpha.{\downarrow}\gamma.\,(A@\alpha \to B@(\alpha * \gamma))$$

because the two operations: $\forall\alpha$ and $@p$ behave the same in both frameworks. This encoding yields a direct encoding of the rule of introduction for $\multimap$, in HyLF:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Omega, \alpha; \Gamma, A@\alpha[p] \vdash B[\alpha * p]}{\Omega, \alpha; \Gamma, A@\alpha[p] \vdash B@(\alpha * p)[p]}\,@i}{\Omega, \alpha; \Gamma \vdash A@\alpha \to B@(\alpha * p)[p]}\,\Pi i}{\Omega, \alpha; \Gamma \vdash {\downarrow}\gamma.(A@\alpha \to B@(\alpha * \gamma))[p]}\,{\downarrow}i}{\Omega; \Gamma \vdash \forall\alpha.{\downarrow}\gamma.(A@\alpha \to B@(\alpha * \gamma))[p]}\,\forall i$$

and similarly a direct encoding of the elimination rule based on the elimination rules for each of the components used in the encoding of $\multimap$ in HʏLF:

$$\dfrac{\Pi e\;\dfrac{{\downarrow}e\;\dfrac{\forall e\;\dfrac{\Omega;\Gamma \vdash \forall\alpha.{\downarrow}\gamma.(A@\alpha \to B@(\alpha * \gamma))[p]}{\Omega;\Gamma \vdash {\downarrow}\gamma.(A@q \to B@(q * \gamma))[p]}}{\Omega;\Gamma \vdash A@q \to B@(q * p)[p]} \qquad \Omega;\Gamma \vdash A@q[p]}{@e\;\dfrac{\Omega;\Gamma \vdash B@(q * p)[p]}{\Omega;\Gamma \vdash B[p]}}}{}$$

where we omit terms and simplify notations by omitting also the side conditions. The constraints on worlds induced by the use of @ restrict the set of valid proofs of $A \multimap B$ to those proofs of $A \to B$ that are actually linear ones. Details on this encoding and the use of $\multimap$ to represent other systems in a logical framework can be found in the literature [Reed, 2009].

Towards modal reasoning. Just as linearity could be encoded at the representation level of HLF and HʏLF, it is conceivable to extend this language further, by defining modalities such as the $\Box$ and $\Diamond$ of modal logics within the syntax of types in HʏLF. This would allow us to represent systems for which an adequate encoding relies on the ability to control separate worlds within a relation as in Kripke semantics. Using the ideas found in the encoding of linearity from HLF, one encoding of $\Box$ could be:

$$\Box A \;\triangleq\; \forall\alpha.{\downarrow}\gamma.\, A@(\alpha \lhd \alpha R \gamma)$$

with $\lhd$ standing for an operator on worlds designed to allow the distinction, in the current world of a sequent, between an actual world and a constraint on the relation R that must be validated. In the structure of worlds, we would then require enough operators to keep a structured form of information about the relation. Notice that with an encoding as shown above, the distributivity axiom **K** is provable without any requirement on the structure of worlds, and the axioms **T** and **4** yield the standard conditions on worlds, reflexivity and transitivity. Note that in the variant of the framework presented in this paper, enforcing that $\lhd$ take precisely a single world variable as its first argument is not possible. A modification allowing a compound world to contain a fixed set of zones, built using different operators, would therefore be required. We leave this for future work.

Furthermore, encoding the $\Diamond$ connective this way is more complicated, as it requires to use existential quantification over worlds. This is not inconceivable, but the current HyLF framework would need to be extended beyond the definition given here in terms of syntax and typing rules.

5.4 Conclusion and Future Work

We have presented here an extension of the standard LF framework that allows hybrid operators to be explicitly used for encodings, and discussed its properties, as well as the representation of systems for modal logics in this setting. This opens a number of questions for future work:

- we still need to fully develop the meta-theory of HyLF, and in particular the underlying notion of reduction as well as the expressive power of the framework — from a practical viewpoint, we would need to impose some restrictions on the structure given to worlds, since for example we could think of defining any non-decidable congruence, that would lead to problems in attempts at performing unification,

- we can now try to encode other logics in HyLF than the few modal systems we mentioned, for example any temporal, spatial or epistemic logic, but also just other presentations of logic, for example based on the notion of hypersequents, that could be encoded using worlds,

- we need to investigate further the question of encoding the operators necessary for modal reasoning, starting with $\Box$ and $\Diamond$, but more generally we could attempt to identify other forms of reasoning that can be reified by some modality, and encoded into the world structure available in HyLF,

- on the side of implementation, the extended features of the worlds in HyLF yield the question of feasibility for any reasonable implementation of the algorithms of unification or coverage, but we hope that restrictions imposed on the congruence can reduce the complexity of the problem,

- we could also consider further extensions to the syntax of types and terms in HʏLF, in particular to allow existential quantification over worlds, and over terms, or introduce a cartesian product as done in HLF,
- the expressive power of this hybrid framework might allow to encode complex systems that combine several aspects requiring a particular world structure, such as the hybrid linear logic presented in [Chaudhuri and Despeyroux, 2014], and the freedom offered in the definition of operators on worlds could even be enough to define some general means of combining the encodings, so that for example the two levels of structure on worlds needed for the linear treatment of context on one side, and the access to the worlds of modal logics on the other, could merge.

5.A Appendix

We present here partial proofs and generalized reformulations of theorems mentioned in Section 5.2 of this paper.

Theorem 5.1 (Hereditary substitution)**.**
For any terms M and N, if for a given world p there exists V such that $M[N/x]_p = V$, then we have the reduction $M\{N/x\} \twoheadrightarrow_p V$.

Proof. By structural induction on the derivation of $M[N/x]_p = V$. The following are representative cases:

Case: $$\frac{M[N/x]_p = V}{(\lambda y.M)[N/x]_p = \lambda y.V}$$

By the induction hypothesis, we obtain $M\{N/x\} \twoheadrightarrow_p V$. Using the definition of capture-avoiding substitution and the properties of $\twoheadrightarrow$, we conclude that indeed $(\lambda y.M)\{N/x\} = \lambda y.M\{N/x\} \twoheadrightarrow_p \lambda y.V$

Case: $$\frac{}{x[N/x]_p = N}$$

By the definition of capture-avoiding substitution, $x\{N/x\} = N$. Therefore $x\{N/x\}$ reduces in zero steps to itself: $N \twoheadrightarrow_p N$.

Case: $\dfrac{R[N/x]_p = R' \quad M[N/x]_p = M'}{(R\ M)[N/x]_p = R'\ M'}$

By the induction hypothesis, we obtain $R\{N/x\} \longrightarrow\!\!\!\!\rightarrow_p R'$ and $M\{N/x\} \longrightarrow\!\!\!\!\rightarrow_p M'$. Using the definition of capture-avoiding substitution and the standard distributivity properties of $\longrightarrow\!\!\!\!\rightarrow$, we conclude that indeed $(R\ M)\{N/x\} = (R\{N/x\})\ (M\{N/x\}) \longrightarrow\!\!\!\!\rightarrow_p R'\,M'$.

Case: $\dfrac{R[N/x]_p = \lambda y.V' \quad M[N/x]_p = M' \quad V'[M'/y]_p = V}{(R\ M)[N/x]_p = V}$

By the induction hypothesis, we obtain $R\{N/x\} \longrightarrow\!\!\!\!\rightarrow_p \lambda y.V'$, as well as $M\{N/x\} \longrightarrow\!\!\!\!\rightarrow_p M'$ and $V'\{M'/y\} \longrightarrow\!\!\!\!\rightarrow_p V$. Therefore, using the definitions of capture-avoiding substitution and relations $\longrightarrow\!\!\!\!\rightarrow$ and $\longrightarrow$, as well as the distributivity properties of $\longrightarrow\!\!\!\!\rightarrow$, we are able to conclude that $(R\ M)\{N/x\} = (R\{N/x\})\ (M\{N/x\}) \longrightarrow\!\!\!\!\rightarrow_p (\lambda y.V')\,M' \longrightarrow_p V'\{M'/y\} \longrightarrow\!\!\!\!\rightarrow_p V$.

Case: $\dfrac{R[N/x]_q = M\,\mathtt{at}\,p}{(R\,\mathtt{to}\,p)[N/x]_p = M}$

By the induction hypothesis, we obtain $R\{N/x\} \longrightarrow\!\!\!\!\rightarrow_p M\,\mathtt{at}\,p$. Therefore, using the definitions of capture-avoiding substitution and relations $\longrightarrow\!\!\!\!\rightarrow$ and $\longrightarrow$, as well as the properties of $\longrightarrow\!\!\!\!\rightarrow$, we conclude that $(R\,\mathtt{to}\,p)\{N/x\} = R\{N/x\}\,\mathtt{to}\,p \longrightarrow\!\!\!\!\rightarrow_p (M\,\mathtt{at}\,p)\,\mathtt{to}\,p \longrightarrow_p M$.

$\square$

Theorem 5.2 (Substitution)**.**
Given any term N such that $\Omega;\Gamma \vdash N \Leftarrow A[q]$, and:

- *given any term M such that $\mathcal{D} :: \Omega;\Gamma, x : A[q], \Delta \vdash M \Leftarrow B[p]$, it follows that $\Omega;\Gamma,\Delta[N/x] \vdash M[N/x] \Leftarrow B[N/x][p]$;*
- *given any term R such that $\mathcal{D} :: \Omega;\Gamma, x : A[q], \Delta \vdash R \Rightarrow B[p]$, it follows that either $\Omega;\Gamma,\Delta[N/x] \vdash R[N/x] \Rightarrow B[N/x][p]$ or $\Omega;\Gamma,\Delta[N/x] \vdash R[N/x] \Leftarrow B[N/x][p]$.*

Proof. We prove both parts by mutual lexicographic induction over first the simple type of N, $\tau(N)$, and second – the derivation of $\mathcal{D}$. The following are representative cases:

Case: $$\frac{\Omega;\Gamma,x:A[q],\Delta,y:B_0[p]\vdash R \Leftarrow B_1[p]}{\Omega;\Gamma,x:A[q],\Delta\vdash \lambda y.R \Leftarrow \Pi y{:}B_0.B_1[p]}$$
By the induction hypothesis and substitution properties, we obtain that $\Omega;\Gamma,\Delta[N/x],y:B_0[N/x][p]\vdash R[N/x] \Leftarrow B_1[N/x][p]$. Therefore, using the typing rule for lambda abstraction, we can also conclude $\Omega;\Gamma,\Delta[N/x]\vdash \lambda y.R[N/x] \Leftarrow \Pi y{:}B_0[N/x].B_1[N/x][p]$ – which, using the definition of capture-avoiding substitution, can be equivalently written as $\Omega;\Gamma,\Delta[N/x]\vdash(\lambda y.R)[N/x] \Leftarrow (\Pi y{:}B_0.B_1)[N/x][p]$

Case: $$\frac{\Omega;\Gamma,x:A[q],\Delta\vdash R\Rightarrow F[p'] \qquad p\equiv p' \qquad p\in\mathcal{W}}{\Omega;\Gamma,x:A[q],\Delta\vdash R \Leftarrow F[p]}$$
By the induction hypothesis, we may obtain one of two possible results:

- $\Omega;\Gamma,\Delta[N/x]\vdash R[N/x]\Rightarrow F[N/x][p']$ – in which case we may use the swap rule again, to move between synthesizing and checking, obtaining: $\Omega;\Gamma,\Delta[N/x]\vdash R[N/x] \Leftarrow F[N/x][p]$.

- $R[N/x]=M'$ and $\Omega;\Gamma,\Delta[N/x]\vdash M' \Leftarrow F[N/x][p']$ – in which case, we have to first invert the last typing rule used. Since M' is still of an atomic type $F[N/x]$, we can conclude that the last inference rule used was in fact the phase changing one above, therefore $M'=R[N/x]$ is in fact a maximally applied atomic term:
$$\frac{\Omega;\Gamma,x:A[q],\Delta[N/x]\vdash R[N/x]\Rightarrow F[N/x][p''] \quad p'\equiv p'' \quad p'\in\mathcal{W}}{\Omega;\Gamma,\Delta[N/x]\vdash R[N/x] \Leftarrow F[N/x][p']}$$
– but in this case, using the fact that the structure on worlds is a congruence, we can also rewrite the last step to be
$$\frac{\Omega;\Gamma,x:A[q],\Delta[N/x]\vdash R[N/x]\Rightarrow F[N/x][p''] \quad p\equiv p'' \quad p\in\mathcal{W}}{\Omega;\Gamma,\Delta[N/x]\vdash R[N/x] \Leftarrow F[N/x][p]}$$
which is precisely, what we needed to show.

Case: $$\frac{}{\Omega;\Gamma,x:A[p],\Delta\vdash x\Rightarrow A[p]}$$
Since $\Omega;\Gamma\vdash N \Leftarrow A[p]$, using weakening we are able to obtain $\Omega;\Gamma,\Delta[N/x]\vdash N \Leftarrow A[p]$.

Case: $\dfrac{\Omega;\Gamma,x{:}A[p],\Delta \vdash R \Rightarrow \Pi y{:}B_0.B_1[p] \quad \Omega;\Gamma,x{:}A[p],\Delta \vdash M \Leftarrow B_1[p]}{\Omega;\Gamma,x{:}A[p],\Delta \vdash R\,M \Rightarrow B_1[M/y][p]}$

By the induction hypothesis on the typing derivation for M, we obtain $\Omega;\Gamma,\Delta[N/x] \vdash M' \Leftarrow B_0[N/x][p]$ for $M' = M[N/x]$. Further, applying the induction hypothesis to the typing derivation of R yields two possible outcomes:

- $\Omega;\Gamma,\Delta[N/x] \vdash R[N/x] \Rightarrow \Pi y{:}B_0[N/x].B_1[N/x][p']$ – in which case we simply use the same typing rule again to obtain: $\Omega;\Gamma,\Delta[N/x] \vdash R[N/x]\,M[N/x] \Leftarrow B_1[N/x][M'/y][p]$. Since $B_1[N/x][M'/y] = B_1[M/y][N/x]$, this, in turn, is equivalent to $\Omega;\Gamma,\Delta[N/x] \vdash (R\,M)[N/x] \Leftarrow (B_1[t/y])[N/x][p]$

- $R[N/x] = M_0$ and $\Omega;\Gamma,\Delta[N/x] \vdash M_0 \Leftarrow \Pi y{:}B_0[N/x].B_1[N/x][p']$ – in which case, we have to first invert the last typing rule used, concluding that M_0 must in fact be $\lambda y.M''$ for some M'':

  $$\dfrac{\Omega;\Gamma,\Delta[N/x],y : B_0[N/x] \vdash M'' \Leftarrow B_1[N/x][p']}{\Omega;\Gamma,\Delta[N/x] \vdash \lambda y.M'' \Leftarrow \Pi y{:}B_0[N/x].B_1[N/x][p']}$$

  This is the case where hereditary substitution acts differently to an inductively defined one. Recall that the appropriate rule in this case requires us to continue substituting, as $(\lambda y.M'')\ M'$ contains a redex:

  $$\dfrac{R[N/x] = \lambda y.M'' \quad M[N/x] = M' \quad M''[M'/y] = M'''}{(R\ M)[N/x] = M'''}$$

  We can now use the inductive hypothesis again, this time on derivations $\Omega;\Gamma,\Delta[N/x] \vdash M' \Leftarrow B_0[N/x][p]$ as well as $\Omega;\Gamma,\Delta[N/x],y : B_0[N/x] \vdash M'' \Leftarrow B_1[N/x][p']$, to obtain $\Omega;\Gamma,\Delta[N/x] \vdash M''[M'/y] \Leftarrow B_1[N/x][M'/y][p]$, or equivalently $\Omega;\Gamma,\Delta[N/x] \vdash M''[M'/y] \Leftarrow (B_1[M/y])[N/x][p]$

□

Theorem 5.3 (World substitution)**.**
Given a world $w \in \mathcal{W}$, and

- *given a term M such that $\mathcal{D} :: \Omega,\alpha;\Gamma \vdash M \Leftarrow B[p]$, it follows that $\Omega;\Gamma\{q/\alpha\} \vdash M\{q/\alpha\} \Leftarrow B\{q/\alpha\}[p\{q/\alpha\}]$;*

- *given a term R such that $\mathcal{D} :: \Omega,\alpha;\Gamma \vdash R \Rightarrow B[p]$, it follows that $\Omega;\Gamma\{q/\alpha\} \vdash R\{q/\alpha\} \Rightarrow B\{q/\alpha\}[p\{q/\alpha\}]$.*

Proof. By induction on the typing derivation $\mathcal{D}$. The following are representative cases:

Case: $\dfrac{\Omega, \alpha; \Gamma, x : A[p] \vdash M \Leftarrow B[p]}{\Omega, \alpha; \Gamma \vdash \lambda x.M \Leftarrow \Pi x : A.B[p]}$

Using the induction hypothesis we obtain:
$\Omega; \Gamma\{q/\alpha\}, x : A\{q/\alpha\}[p\{q/\alpha\}] \vdash M\{q/\alpha\} \Leftarrow B\{q/\alpha\}[p\{q/\alpha\}]$.
We continue by using the λ abstraction typing rule again, which yields $\Omega; \Gamma\{q/\alpha\} \vdash \lambda x.M\{q/\alpha\} \Leftarrow \Pi x : A\{q/\alpha\}.B\{q/\alpha\}[p\{q/\alpha\}]$, which is precisely $\Omega; \Gamma\{q/\alpha\} \vdash (\lambda x.M)\{q/\alpha\} \Leftarrow (\Pi x : A.B)\{q/\alpha\}[p\{q/\alpha\}]$.

Case: $\dfrac{\Omega, \alpha; \Gamma \vdash R \Rightarrow F[r] \quad p \equiv r \quad p \in \mathcal{W}}{\Omega, \alpha; \Gamma \vdash R \Leftarrow F[p]}$

By the induction hypothesis, $\Omega; \Gamma\{q/\alpha\} \vdash R\{q/\alpha\} \Rightarrow F\{q/\alpha\}[r\{q/\alpha\}]$. Further, as $\equiv$ is a congruence rule, $p\{q/\alpha\} \equiv r\{q/\alpha\}$. Therefore, using the same typing rule yields $\Omega; \Gamma\{q/\alpha\} \vdash R\{q/\alpha\} \Leftarrow F\{q/\alpha\}[p\{q/\alpha\}]$.

Case: $\dfrac{x : A[p] \in \Gamma \quad p \in \mathcal{W}}{\Omega, \alpha; \Gamma \vdash x \Rightarrow A[p]}$

If $x : A[p] \in \Gamma$, then also $x : A\{q/\alpha\}[p\{q/\alpha\}] \in \Gamma\{q/\alpha\}$ – and therefore $\Omega; \Gamma\{q/\alpha\} \vdash x \Rightarrow A\{q/\alpha\}[p\{q/\alpha\}]$.

Case: $\dfrac{\Omega, \alpha; \Gamma \vdash R \Rightarrow \Pi x : A.B[p] \qquad \Omega, \alpha; \Gamma \vdash M \Leftarrow A[p]}{\Omega, \alpha; \Gamma \vdash R\ M \Rightarrow B[M/x]_p[p]}$

Using the induction hypothesis, we obtain:
$\Omega; \Gamma\{q/\alpha\} \vdash R\{q/\alpha\} \Rightarrow \Pi x : A\{q/\alpha\}.B\{q/\alpha\}[p\{q/\alpha\}]$ and $\Omega; \Gamma\{q/\alpha\} \vdash M\{q/\alpha\} \Leftarrow A\{q/\alpha\}[p\{q/\alpha\}]$. Using the typing rule for application yields:
$\Omega; \Gamma\{q/\alpha\} \vdash R\{q/\alpha\}\ M\{q/\alpha\} \Rightarrow B\{q/\alpha\}[M\{q/\alpha\}/x]_{p\{q/\alpha\}}[p\{q/\alpha\}]$, or equivalently $\Omega; \Gamma\{q/\alpha\} \vdash (R\ M)\{q/\alpha\} \Rightarrow (B[M/x]_p)\{q/\alpha\}[p\{q/\alpha\}]$.

□

Part III

Lincx

Chapter 6

Programming with Higher-Order Abstract Syntax

In this part of the thesis we will focus on a different approach to building logical frameworks – one which allows for reasoning about specifications without the need to re-design the reasoning language with each new instance. The BELUGA system by [Pientka and Dunfield, 2010] provides a clean separation between the level of encodings (data) and that of proofs (computation). This separation makes it possible to replace one of the layers without significant changes to the other, as long as we keep the same interface between the two. The next chapter presents the first attempt at such a replacement; for now, however, let us focus on the overall design of the framework we intend to re-use.

6.1 Contexts in LF

The main source of difficulty when reasoning about LF specifications comes from one of its biggest strengths: using higher-order abstract syntax in the encodings inherently forces programming with (or reasoning about) open terms of the meta-level. Moreover, we cannot manipulate, or in fact even directly access, the bound variables present in the encodings.

6.1.1 Example: Variable Occurrence Counting

A very simple example of reaching the limits of the pure LF approach is counting the occurrences of a specific bound variable. We will re-consider the arrow fragment of the STLC system from Example 1.2.1.

```
tm  : type.
lam : (tm → tm) → tm.
app : tm → tm → tm.
```

How would one go about writing a variable counter for a calculus like this? In a functional programming language not using HOAS, with a recursive function `var_occ_count: var → tm → nat`, with `var` being the abstraction for variables used in terms `tm`. In LF, though, we cannot express even the type of this function, as a bound variable is, in fact, declared in the LF context, which we do not have direct access to.

The remedy proposed by [Pientka, 2008] is to consider programming with *contextual modal objects*. Instead of defining functions and properties of encodings directly in LF, we use Beluga, a functional programming language which includes open LF objects together with their contexts as one of the available data types. This introduces two significant changes: first, we switch from the logical to the functional paradigm; second and more important, the LF encodings we are used to working with can now be lifted to another abstraction level, where they can be considered with their respective contexts. In this approach, a contextual type $[\Psi \vdash A]$ behaves as type A when its dependencies, in the form of context Ψ, are satisfied.

A contextual term is, then, one having such a contextual type. This way, we can explicitly mention the variable whose occurrences we want to count as part of the context. The counting function can then be declared as taking a contextual object depending on, among others, a variable x, and returning a natural number – the number of occurrences of x in the object.

```
schema ctx = tm.

rec var_occ_count: (γ : ctx) [γ, x : tm ⊢ tm] → nat =
fn e ⇒ case e of
| [γ, x : tm ⊢ x] ⇒ 1
| [γ, x : tm ⊢ lam (λy. M)] ⇒
    var_occ_count [γ, y:tm, x:tm ⊢ M]
| [γ, x : tm ⊢ app M N] ⇒
    var_occ_count [γ, x: tm ⊢ M] + var_occ_count [γ, x : tm ⊢ N]
| [γ, x : tm ⊢ _] ⇒ 0
;
```

Notice that, while `var_occ_count` is declared to perform *computation* on an open LF object, it is not itself an LF expression. The first argument of `var_occ_count`, γ `: ctx`, describes a context of terms `tm`, in which (when extended with `x : tm`) the second argument, a contextual object of type `[`γ`, x : tm` $\vdash$ `tm]`, resides. The counting function can then be declared using pattern matching on contextual terms.

The interesting cases are, as usual, those for variables and binders. The variable case is split: either the variable used is the same one we are counting (in which case, return `1`), or it is not (return `0`). The latter is expressed here as a blank `_`, which matches all cases not previously considered. We may sometimes have to be a bit more precise, which is why Beluga introduces *parameter variables* to talk about arbitrary variable cases. In the `lam (`λ`y.M)` case, we instantiate the *context variable* γ in the recursive call to be γ`, y:tm`.

Even though meta-variables and context variables come from the computation level, they can, of course, appear in the LF level as well – after all, for γ`, x:tm` $\vdash$ `M` to be a correct LF object, the context variable γ needs to be a correct context. This dependency forms the previously mentioned interface between the data and computation layers.

6.2 Contextual Modal Logic

We will now take a look at the design of Contextual LF, an LF variant compatible with programming with open terms as described in the previous section. The type theory behind it is an adaptation of Contextual Modal Type Theory by [Nanevski et al., 2008], exploring the idea that, in between a modal $\Box A$, which ensures *universal validity*, and A, which guarantees only *local truth*, there is a whole spectrum of *relative truths*, depending on some assumptions, which can be expressed in the form of a context Ψ. This relativity can be captured in a $[\Psi \vdash A]$[1] type construct.

Types	A, B	$::=$	$a \mid A \to B \mid [\Psi \vdash A]$
Contexts	Γ, Ψ	$::=$	$\cdot \mid \Gamma, x : A$
Modal Contexts	Δ	$::=$	$\cdot \mid \Delta, u : [\Psi \vdash A]$

[1] $[\Psi]A$ in some presentations

$$\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash A}\ ax \qquad \frac{u : [\Psi \vdash A] \in \Delta \quad \Delta; \Gamma \vdash \Psi}{\Delta; \Gamma \vdash A}\ cax$$

$$\frac{\Delta; \Gamma, x : A \vdash B}{\Delta; \Gamma \vdash A \to B} \to_I \qquad \frac{\Delta; \Gamma \vdash A \to B \quad \Delta; \Gamma \vdash A}{\Delta; \Gamma \vdash B} \to_E$$

$$\frac{\Delta; \Psi \vdash A}{\Delta; \Gamma \vdash [\Psi \vdash A]}\ \Box_I \qquad \frac{\Delta; \Gamma \vdash [\Psi \vdash A] \quad \Delta, u : [\Psi \vdash A]; \Gamma \vdash C}{\Delta; \Gamma \vdash C}\ \Box_E$$

$$\frac{\Delta; \Gamma \vdash B_1 \ldots \Delta; \Gamma \vdash B_n}{\Delta; \Gamma \vdash \{y_i : B_i\}_n^{i=1}}\ ctx$$

Figure 6.1: Contextual Modal Logic

Figure 6.1 presents the natural deduction system for a non-dependent intuitionistic contextual modal logic. Notice that the variable rule *cax*, used when taking a hypothesis $u : [\Psi \vdash A]$ from Δ, ensures the existence of *context transformation* (substitution) from the existing context, Γ, to one that the assumption u requires, Ψ.

6.3 Contextual LF

Contextual LF is built on the basis of the contextual modal type theory, by extending it to include dependent types and providing term assignment for formulas. To make the presentation simpler, we skip the former in this section, as it is re-introduced in the system presented in the subsequent chapter. We also omit the separation of atomic and canonical terms, as our primary interest is in the new constructs required to make it possible to write functions on open LF terms – not the mechanics of making the type theory for the system beautiful.

Types	A, B	$::=$	$a \mid A \to B$
Terms	M, N	$::=$	$c \mid x \mid u[\sigma] \mid \lambda x.M \mid M\,N$
Substitutions	σ, τ	$::=$	$\cdot \mid \mathsf{id}_\psi \mid \sigma, M$
Contexts	Γ, Ψ	$::=$	$\cdot \mid \psi \mid \Gamma, x : A$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, a : \mathsf{type} \mid \Sigma, c : A$

Meta Contexts	Δ	$::=$	$\cdot \mid \Delta, \psi : G \mid \Delta, u : [\Psi \vdash A]$
Context Schemata	G	$::=$	$A \mid G + A$

The distinction between the contextual modal context (meta-context) Δ and the normal context Γ is preserved; however, we do not consider $[\Psi \vdash A]$ to be an LF-level type. In particular, we do not allow assumptions of this form in Γ, and, as a consequence, we cannot λ-abstract over them in Contextual LF. Instead, we consider $[\Psi \vdash A]$ a meta-type, belonging only to a meta-context and typing meta-variables. Additional constructs such as context variables[2] γ are added in order to capture some recurring concepts in mechanizations, as was already suggested in the previous section. Context variables are annotated with schemata, describing types of LF level variables allowed in a concrete context instantiating such context variable. For instance **schema** `ctx = tm` describes a type of context containing only `tm` assumptions, and γ `: ctx` stands for any context of such `tm` assumptions. In the simply typed case a schema is just a union of possible types – therefore **schema** `ctx' = nat + bool` denotes a type of a context which may contain boolean and natural number assumptions.

In order to move between the object level and the meta-level, the notion of simultaneous substitutions is introduced. It provides term assignment to the context transformation rule *ctx* seen in Figure 6.1. As the correct context can now include a context variable ψ, there must also be a substitution term for it, id_ψ. New constructs in Contextual LF also include $u[\sigma]$, a term standing for a closure for meta-variable.

Meta Variables	X	$::=$	$u \mid \psi$
Meta Objects	C	$::=$	$\widetilde{\Psi}.M \mid \Psi$
Meta Types	U	$::=$	$[\Psi \vdash A] \mid G$
Meta Substitutions	Θ	$::=$	$\cdot \mid \Theta, [C/X]$

Meta contexts do not get extended directly in the LF layer. Instead, they are created in the computational layer. Certain refinements can be given to meta-variables and context variables, leading to simultaneous meta substitution Θ.

To sketch the idea of how the pieces of Contextual LF fit together, Figures 6.2 and 6.3 present its typing rules. As we are not using dependent

[2]In the full variant of Contextual LF, also parameter variables and substitution variables.

$$\frac{c : A \in \Sigma}{\Delta; \Gamma \vdash c : A} \qquad \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}$$

$$\frac{u : [\Psi \vdash A] \in \Delta \quad \Delta; \Gamma \vdash \sigma : \Psi}{\Delta; \Gamma \vdash u[\sigma] : A}$$

$$\frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Delta; \Gamma \vdash M : A \to B \quad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M\,N : B}$$

$$\frac{}{\Delta; \Gamma \vdash \cdot : \cdot} \qquad \frac{}{\Delta; \psi, \Gamma \vdash \mathsf{id}_\psi : \psi} \qquad \frac{\Delta; \Gamma \vdash \sigma : \Psi \quad \Delta; \Gamma \vdash M : A}{\Delta; \Gamma \vdash (\sigma, M) : \Psi, x : A}$$

Figure 6.2: Contextual LF, data level

$$\frac{\psi : G \in \Delta}{\Delta \vdash \psi : G} \qquad \frac{}{\Delta \vdash \cdot : G} \qquad \frac{\Delta \vdash \Gamma : G \quad A \in G}{\Delta \vdash (\Gamma, x : A) : G}$$

$$\frac{\Delta; \Psi \vdash M : A}{\Delta \vdash \widetilde{\Psi}.M : [\Psi \vdash A]}$$

$$\frac{}{\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta \vdash \Theta : \Delta' \quad \Delta \vdash C : U}{\Delta \vdash \Theta, C/X : \Delta', X : U}$$

Figure 6.3: Contextual LF, computation level

types in this short introduction, many of them are greatly simplified, but the extension to dependent types does not introduce any interesting new challenges. The syntactic restriction on what constitutes a type allows us to preserve most of the typing rules from Figure 6.1 intact, the changes in the substitution σ typing rules being purely aesthetic.

A significant difference is that, on the data level, we no longer allow a contextual object to appear on its own – it has to be accompanied by a closure, in the form of a simultaneous substitution σ. The idea here is that, once we know how to instantiate the meta-variable, we can continue substituting σ.

The simultaneous substitution σ can be defined inductively, as in Figure 6.4, although typically hereditary presentations are chosen. Meta-substitution interacts with simultaneous substitution when substituting

$$
\begin{array}{lll}
\boxed{\sigma_\Psi(x)} & \text{Variable lookup} & \\
(\sigma, M)_{\Psi, x:A}(x) = M : A & & \\
(\sigma, M)_{\Psi, y:A}(x) = \sigma_\Psi(x) & \text{where } y \neq x & \\
\sigma_\Psi(x) & = \bot & \\
\boxed{[\sigma]_\Psi M} & & \\
[\sigma]_\Psi(c) & = c & \\
[\sigma]_\Psi(x) & = M & \text{where } \sigma_\Psi(x) = M : A \\
[\sigma]_\Psi(\lambda y.N) & = \lambda y.N' & \text{where } [\sigma]_\Psi N = N', \\
 & & \text{choosing } y \notin \Psi, y \notin \mathsf{FV}(\sigma) \\
[\sigma]_\Psi(u[\tau]) & = u[\tau'] & \text{where } [\sigma]_\Psi \tau = \tau' \\
[\sigma]_\Psi(M\,N) & = M'\,N' & \text{where } [\sigma]_\Psi M = M' \text{ and } [\sigma]_\Psi N = N' \\
\boxed{[\sigma]_\Psi \tau} & & \\
[\sigma]_\Psi(\cdot) & = \cdot & \\
[\sigma]_\Psi(\mathsf{id}_\psi) & = \mathsf{id}_\psi & \\
[\sigma]_\Psi(\tau, M) & = (\tau', M') & \text{where } [\sigma]_\Psi \tau = \tau' \text{ and } [\sigma]_\Psi M = M'
\end{array}
$$

Figure 6.4: Simultaneous substitution

for meta-objects, as we require the postponed substitution present in the closure to be executed. The relevant meta-substitution rules are presented in Figure 6.5. $\mathsf{id}(\Psi)$ is simply an identity substitution x/x for all elements of Ψ. Note that these rules are oversimplified – reasoning about Contextual LF presented this way would be extremely cumbersome, and we would typically use something more like the spine presentation with hereditary substitution introduced in Chapter 3.

Both simultaneous substitution σ and meta-substitution Θ are type-preserving, and typing is decidable.

Connecting Data and Computation The general idea of the interface between data and computation used in Beluga is this: in the computational layer, we allow general pattern matching on contextual objects, using them as an index language in Beluga. Whenever a pattern is matched, the concretization of meta-variables that it uses spawns a meta-substitution which is injected to the Contextual LF layer. The technical details of the computational layer are not relevant for the work that follows,

$\boxed{\Theta_\Delta(X)}$ Contextual variable lookup

$$
\begin{array}{lll}
(\Theta, C/X)_{\Delta,X:U}(X) = C : U & & \\
(\Theta, C/Y)_{\Delta,Y:\_}(X) & = \Theta_\Delta(X) & \text{where } Y \neq X \\
\Theta_\Delta(X) & = \bot &
\end{array}
$$

$\boxed{[\![\Theta]\!]_\Delta(M) = M'}$

$$
\begin{array}{lll}
[\![\Theta]\!]_\Delta(x) & = x & \\
[\![\Theta]\!]_\Delta(c) & = c & \\
[\![\Theta]\!]_\Delta(u[\sigma]) & = M' & \text{where } \Theta_\Delta(u) = \widetilde{\Psi}.M : (\Psi \vdash A) \\
& & \text{and } [\![\Theta]\!]_\Delta\, \sigma = \sigma' \\
& & \text{and } [\sigma']_\Psi M = M' \\
[\![\Theta]\!]_\Delta(\lambda x.M) & = \lambda x.M' & \text{where } [\![\Theta]\!]_\Delta M = M' \\
[\![\Theta]\!]_\Delta(M\,N) & = M'\,N' & \text{where } [\![\Theta_\Delta]\!]\, M = M' \\
& & \text{and } [\![\Theta]\!]_\Delta\, N = N' \\
\\
[\![\Theta]\!]_\Delta(\cdot) & = \cdot & \\
[\![\Theta]\!]_\Delta(\mathsf{id}_\psi) & = \mathsf{id}(\Psi) & \text{where } \Theta_\Delta(\psi) = \Psi \\
[\![\Theta]\!]_\Delta(\sigma, M) & = \sigma', M' & \text{where } [\![\Theta]\!]_\Delta\, \sigma = \sigma' \text{ and } [\![\Theta]\!]_\Delta\, M = M'
\end{array}
$$

$$
\begin{array}{lll}
(\Theta, C/X)_{\Delta,X:U}(X) = C : U & & \\
(\Theta, C/Y)_{\Delta,Y:\_}(X) & = \Theta_\Delta(X) & \text{where } Y \neq X \\
\Theta_\Delta(X) & = \bot &
\end{array}
$$

Figure 6.5: Meta-substitution

and therefore are beyond the scope of this brief introduction. They can be found, for instance, in [Pientka, 2008] and [Cave and Pientka, 2012]. The important point here is that, as long as the index language we want to use behaves *similarly enough* to Contextual LF, the computation layer will not require changes. In the next chapter we will therefore propose Lincx, a linear contextual LF framework whose behaviour is compatible with Beluga's expectations for an index type.

Chapter 7

Lincx: A Linear Logical Framework with First-class Contexts

Originally published in: 26th European Symposium on Programming, 2017

Joint work with: Aina Linn Georges, Shawn Otis, Brigitte Pientka

Abstract

Linear logic provides an elegant framework for modelling stateful, imperative and concurrent systems by viewing a context of assumptions as a set of resources. However, mechanizing the meta-theory of such systems remains a challenge, as we need to manage and reason about mixed contexts of linear and intuitionistic assumptions.

We present Lincx, a contextual linear logical framework with first-class mixed contexts. Lincx allows us to model (linear) abstract syntax trees as syntactic structures that may depend on intuitionistic and linear assumptions. It can also serve as a foundation for reasoning about such structures. Lincx extends the linear logical framework LLF with first-class (linear) contexts and an equational theory of context joins that can otherwise be very tedious and intricate to develop. This work may be also viewed as a generalization of contextual LF that supports both intuitionistic and linear variables, functions, and assumptions.

We describe a decidable type-theoretic foundation for Lincx that only characterizes canonical forms and show that our equational theory of context joins is associative and commutative. Finally, we outline how Lincx may serve as a practical foundation for mechanizing the meta-theory of stateful systems.

7.1 Introduction

Logical frameworks make it easier to mechanize formal systems and proofs about them by providing a single meta-language with abstractions and primitives for common and recurring concepts, like variables and assumptions in proofs. This can have a major impact on the effort and cost of mechanization. By factoring out and abstracting over low-level details, it reduces the time it takes to mechanize formal systems, avoids errors in manipulating low-level operations, and makes the mechanizations themselves easier to maintain. It can also make an enormous difference when it comes to proof checking and constructing meta-theoretic proofs, as we focus on the essential aspect of a proof without getting bogged down in the quagmire of bureaucratic details.

The contextual logical framework [Nanevski et al., 2008, Pientka, 2008], an extension of the logical framework LF [Harper et al., 1993], is designed to support a broad range of common features that are needed for mechanizations of formal systems. To model variables, assumptions and derivations, programmers can take advantage of higher-order abstract syntax (HOAS) trees; a context of assumptions together with properties about uniqueness of assumptions can be represented abstractly using first-class contexts and context variables [Pientka, 2008]; single and simultaneous substitutions together with their equational theory are supported via first-class substitutions [Cave and Pientka, 2013, Cave and Pientka, 2015]; finally, derivation trees that depend on a context of assumption can be precisely described via contextual objects [Nanevski et al., 2008]. This last aspect is particularly important. By encapsulating and representing derivation trees together with their surrounding context of assumptions, we can analyse and manipulate these rich syntactic structures via pattern matching, and can construct (co)inductive proofs by writing recursive programs about them [Pientka and Dunfield, 2010, Cave and Pientka, 2012]. This leads to a modular and robust design where we cleanly separate the representation of formal systems and derivations from the (co)inductive reasoning about them.

Substructural frameworks such as the linear logical framework LLF [Cervesato and Pfenning, 1996] provide additional abstractions to elegantly model the behaviour of imperative operations such as updating and deallocating memory [Walker and Watkins, 2001, Fluet et al., 2006] and concurrent computation (see for example session types [Caires and Pfenning, 2010]). However, it has been very challenging to mechanize proofs about LLF specifications as we must manage mixed contexts of unrestricted and linear assumptions. When constructing a derivation tree, we must often split the linear resources and distribute them to the premises relying on a *context join* operation, written as $\Psi = \Psi_1 \bowtie \Psi_2$. This operation should be commutative and associative. Unrestricted assumptions present in Ψ should be preserved in both contexts Ψ_1 and Ψ_2. The mix of unrestricted and restricted assumptions leads to an intricate equational theory of contexts that often stands in the way of mechanizing linear or separation logics in proof assistants and has spurred the development of specialized tactics [McCreight, 2009, Bengtson et al., 2012].

Our main contribution is the design of Lincx (read: "lynx"), a contextual linear logical framework with first-class contexts that may contain both intuitionistic and linear assumptions. On the one hand our work extends the linear logical framework LLF with support for first-class linear contexts together with an equational theory of context joins, contextual objects and contextual types; on the other we can view Lincx as a generalization of contextual LF to model not only unrestricted but also linear assumptions. Lincx hence allows us to abstractly represent syntax trees that depend on a mixed context of linear and unrestricted assumptions, and can serve as a foundation for mechanizing the meta-theory of stateful systems where we implement (co)inductive proofs about linear contextual objects by pattern matching following the methodology outlined by [Cave and Pientka, 2012] and [Thibodeau et al., 2016]. Our main technical contributions are:

(i) *A bi-directional decidable type system that only characterizes canonical forms of our linear* LF *objects.* Consequently, exotic terms that do not represent legal objects from our object language are prevented. It is an inherent property of our design that bound variables cannot escape their scope, and no separate reasoning about scope is required. To achieve this we rely on hereditary substitution to guarantee normal forms are preserved. Equality of two contextual linear LF objects reduces then to syntactic equality (modulo α-renaming).

(ii) *Definition of first-class (linear) contexts together with an equational theory of context joins.* A context in Lincx may contain both unrestricted and linear assumptions. This not only allows for a uniform representation of contexts, but also leads to a uniform representation of simultaneous substitutions. Context variables are indexed and their indices are freely built from elements of an infinite, countable set through a context join operation ($\bowtie$) that is associative, commutative and has a neutral element. This allows a canonical representation of contexts and context joins. In particular, we can consider contexts equivalent modulo associativity and commutativity. This substantially simplifies the meta-theory of Lincx and also directly gives rise to a clean implementation of context joins which we exploit in our mechanization of the meta-theoretic properties of Lincx.

(iii) *Mechanization of* Lincx *together with its meta-theory in the proof assistant* Beluga *[Pientka and Cave, 2015].* Our development takes advantage of higher-order abstract syntax to model binding structures compactly. We only model linearity constraints separately. We have mechanized our bi-directional type-theoretic foundation together with our equational theory of contexts. In particular, we mechanized all the key properties of our equational theory of context joins and the substitution properties our theory satisfies.

We believe that Lincx is a significant step towards modelling (linear) derivation trees as well-scoped syntactic structures that we can analyse and manipulate via case-analysis and implementing (co)inductive proofs as (co)recursive programs. As it treats contexts, where both unrestricted and linear assumptions live, abstractly and factors out the equational theory of context joins, it eliminates the need for users to explicitly state basic mathematical definitions and lemmas and build up the basic necessary infrastructure. This makes the task easier and lowers the costs and effort required to mechanize properties about imperative and concurrent computations.

7.2 Motivating Examples

To illustrate how we envision using (linear) contextual objects and (linear) contexts, we implement two program transformations on object languages that exploit linearity. We first represent our object languages in Lincx and then write recursive programs that analyse the syntactic structure of these objects by pattern matching. This highlights the role that contexts and context joins play.

7.2.1 Example: Code Simplification

To illustrate the challenges that contexts pose in the linear setting, we implement a program that translates linear Mini-ML expressions that feature let-expression into a linear core lambda calculus. We define the linear Mini-ML using the linear type `ml` and our linear core lambda calculus using the linear type `lin` as our target language. We introduce a linear LF type together with its constructors using the keyword **Linear LF**.

```
Linear LF ml : type =
| lam  : (ml ⊸ ml) ⊸ ml
| app  : ml ⊸ ml ⊸ ml
| letv : ml ⊸ (ml ⊸ ml) ⊸ ml;
```

```
Linear LF lin : type =
| llam  : (lin ⊸ lin) ⊸ lin
| lapp  : lin ⊸ lin ⊸ lin;
```

We use the linear implication ⊸ to describe the linear function space and we model variable bindings that arise in abstractions and let-expressions using higher-order abstract syntax, as is common in logical frameworks. This encoding technique exploits the function space provided by LF to model variables. In linear LF it also ensures that bound variables are used only once.

Our goal is to implement a simple translation of Mini-ML expressions to the core linear lambda calculus by eliminating all let-expressions and transforming them into function applications. We thus need to traverse Mini-ML expressions recursively. As we go under an abstraction or a let-expression, our sub-expression will not, however, remain closed. We therefore model a Mini-ML expression together with its surrounding context in which it is meaningful. Our function `trans` takes a Mini-ML expression in a context γ, whose type is written as [γ ⊢ `ml`], and returns a corresponding expression in the linear lambda calculus in a context δ, an object of type [δ ⊢ `lin`]. More precisely, there exists such a corresponding context δ. Due to linearity, the context of the result of translating a Mini-ML term has the same length as the original context. This invariant is however not explicitly tracked.

We first define the structure of such contexts using context schema declarations. The tag `l` ensures that any declaration of type `ml` in a context of schema `ml_ctx` must be linear. Similarly, any declaration of type `lin` in a context of schema `core_ctx` must be linear.

```
schema ml_ctx   = l (ml);
schema core_ctx = l (lin);
```

```
rec trans : ($\gamma$:ml_ctx)[$\gamma$ ⊢ ml] → Result =
fn e ⇒ case e of
| [x$\hat{:}$ml ⊢ x] ⇒ Return [x$\hat{:}$lin ⊢ x]
| [$\gamma$ ⊢ lam ^ ($\widehat{\lambda}$x. M)] ⇒
  let Return [$\delta$, x$\hat{:}$lin ⊢ M’] = trans [$\gamma$, x$\hat{:}$ml ⊢ M] in
  Return [$\delta$ ⊢ llam ^ ($\widehat{\lambda}$x. M’)]
| [$\gamma_{(1\bowtie 2)}$ ⊢ app ^ M ^ N]
  where M:[$\gamma_1$ ⊢ ml] and
        N:[$\gamma_2$ ⊢ ml] and
        $\gamma_{(1\bowtie 2)} = \gamma_1 \bowtie \gamma_2$⇒
  let Return [$\delta_1$ ⊢ M’] = trans [$\gamma_1$ ⊢ M] in
  let Return [$\delta_2$ ⊢ N’] = trans [$\gamma_2$ ⊢ N] in
  Return [$\delta_{(1\bowtie 2)}$ ⊢ lapp ^ M’ ^ N’]
    where $\delta_{(1\bowtie 2)}$ = $\delta_1$ ⋈ $\delta_2$
| [$\gamma_{(1\bowtie 2)}$ ⊢ let ^ M ^ ($\widehat{\lambda}$x. N)]
  where M:[$\gamma_1$⊢ ml] and
        N:[$\gamma_2$, x$\hat{:}$ml ⊢ ml] and
        $\gamma_{(1\bowtie 2)} = \gamma_1 \bowtie \gamma_2$ ⇒
  let Return [$\delta_1$ ⊢ M’] = trans [$\gamma_1$ ⊢ M] in
  let Return [$\delta_2$, x$\hat{:}$lin ⊢ N’]  = trans [$\gamma_2$, x$\hat{:}$ml ⊢ N] in
  Return [$\delta_{(1\bowtie 2)}$ ⊢ lapp ^ (llam ^ ($\widehat{\lambda}$x. N’)) ^ M’]
    where $\delta_{(1\bowtie 2)}$ = $\delta_1$ ⋈ $\delta_2$;
```

Figure 7.1: Translation of linear ML-expressions to a linear core language

To characterize the result of this translation, we define a recursive type:

```
inductive Result : type = Return : ($\delta$:core_ctx) [$\delta$ ⊢ lin] → Result;
```

By writing round parenthesis in (δ:core_ctx) we indicate that we do not pass δ explicitly to the constructor Return, but it can always be reconstructed. It is merely an annotation declaring the schema of δ.

We now define a recursive function trans using the keyword **rec** (see Figure 7.1). First, let us highlight some high level principles and concepts that we use. We write [Ψ ⊢ N] to describe an expression N that is meaningful in the context Ψ. For example, [γ ⊢ lam ^ ($\widehat{\lambda}$x. M)] denotes a term of type ml in the context γ where γ is a context variable that describes contexts abstractly. We call M a meta-variable. It stands for an ml term that may depend on the context γ,x$\hat{:}$ml. In general, all meta-variables are associated with a stuck substitution, written N[σ] or M[σ]. We usually omit the substitution σ, if it is the identity substitution. One substitution that frequently arises in practice is the empty substitution that is written as `[]` and maps from the empty context to an unrestricted context Ψ. It hence acts as a weakening substitution.

Our code simplification is implemented using pattern matching on `[`γ $\vdash$ `ml]` objects and specifying constraints on contexts. In the variable case, since we have a linear context, we require that `x` be the only variable in the context[1]. In the lambda case `[`γ $\vdash$ `lam ^ (`$\widehat{\lambda}$`x.M)]` we write `^` for linear application and linear abstraction. We expect the type of `M` to be inferred as `[`γ`,x`$\hat{:}$`ml` $\vdash$ `ml]`, since we interpret every pattern variable to depend on all its surrounding context unless otherwise specified. We now recursively translate `M` in the extended context γ`, x`$\hat{:}$`ml`, unpack the result and rebuild the equivalent linear term. Note that we pattern match on the result translating `M` by writing `Result [`δ`, x`$\hat{:}$`lin` $\vdash$ `M']`. However, we do not necessarily know that the output `core_ctx` context is of the same length as the input `ml_ctx` context and hence necessarily has the shape `[`δ`, x`$\hat{:}$`lin]`, as we do not track this invariant explicitly. To write a covering program we would need to return an error, if we would encounter `Return [` $\vdash$ `M']`, i.e. a closed term where δ is empty. We omit this case here.

The third and fourth cases are the most interesting ones, as we must split the context. When analysing `[`$\gamma_{(1\bowtie 2)}$ $\vdash$ `app ^ M ^ N]`, term `M` has some type `[`γ_1 $\vdash$ `ml]` and `N` has some type `[`γ_2 $\vdash$ `ml]` where $\gamma_{(1\bowtie 2)} = \gamma_1 \bowtie \gamma_2$. We specify these type annotations and context constraints explicitly. Note that we overload the $\bowtie$ symbol in this example: when it occurs as a subscript it is part of the name, while when we write $\gamma_1 \bowtie \gamma_2$ it refers to the operation on contexts. Then we can simply recursively translate `M` and `N` and rebuild the final result where we explicitly state $\delta_{(1\bowtie 2)}$ `=` $\delta_1 \bowtie \delta_2$. We proceed similarly to translate recursively every let-expression into a function application.

Type checking verifies that a given object is well-typed modulo context joins. This is non-trivial. As an example, consider a contextual `lin` object `[`$\delta_{(1\bowtie 2)}$ $\vdash$`lapp ^ (llam ^ (`$\widehat{\lambda}$`x. N')) ^ M']` where $\delta_{(1\bowtie 2)}$ `=` $\delta_1 \bowtie \delta_2$. Clearly, we should be able to type check it also if the user instead wrote δ `=` $\delta_2 \bowtie \delta_1$. Hence we want our underlying type theory to reason about context constraints modulo associativity and commutativity.

As the astute reader will have noticed, we only allow (at most) one context variable in every context, making it illegal to write objects like

[1]In case we have a mixed context, we could specify instead that the rest of the context is unrestricted, using the keywords **where** and unr.

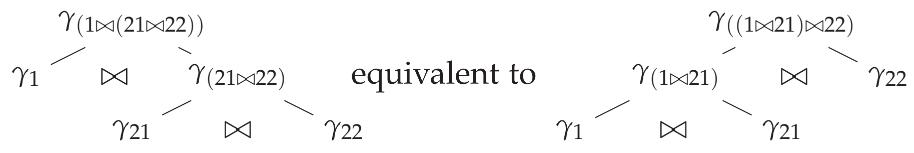
$\gamma_{(1\bowtie(21\bowtie22))}$
γ_1 $\bowtie$ $\gamma_{(21\bowtie22)}$
γ_{21} $\bowtie$ γ_{22}
equivalent to
$\gamma_{((1\bowtie21)\bowtie22)}$
$\gamma_{(1\bowtie21)}$ $\bowtie$ γ_{22}
γ_1 $\bowtie$ γ_{21}

Figure 7.2: Context Joins

[δ_1,$\delta_2 \vdash$ `lapp ^ (llam ^ (`$\widehat{\lambda}$`x. N')) ^ M'`]. Furthermore, we have deliberately chosen the subscripts for our context variables to emphasize their encoding in our underlying theory. Note that all context variables that belong to the same tree of context splits have the same name, but differ in their subscripts. The context variables γ_1 and γ_2 are called *leaf-level context variables*. The context variable $\gamma_{(1\bowtie2)}$ is their direct parent and sits at the root of this tree. One can think of the tree of context joins as an abstraction of the typing derivation.

To emphasize this idea, consider the deeply nested pattern: [$\gamma_{((11\bowtie12)\bowtie2)} \vdash$ `lapp ^ (lapp ^ (llam ^ (`$\widehat{\lambda}$`x. M)) ^ N') ^ K`] in which `M : [`γ_{11}`, x`$\hat{:}$` ml` $\vdash$ `ml]`, `N : [`$\gamma_{12} \vdash$ `ml]`, and `K : [`$\gamma_2 \vdash$ `ml]`, and where we again encode the splitting of γ in its subscript. Our underlying equational theory of context joins treats $\gamma_{(11\bowtie(12\bowtie2))}$ as equivalent to $\gamma_{((11\bowtie12)\bowtie2)}$ or $\gamma_{((12\bowtie11)\bowtie2)}$ as it takes into account commutativity and associativity. However, it may require us to generate a new intermediate node $\gamma_{(1\bowtie21)}$ and eliminate intermediate nodes (such as $\gamma_{21\bowtie22)}$).

Our encoding of context variables is hence crucial to allow the rearrangement of context constraints, but also to define what it means to instantiate a given context variable such as γ_{21} with a concrete context Ψ. If Ψ contains also unrestricted assumptions then instantiating γ_{21} will have a global effect, as unrestricted assumptions are shared among all nodes in this tree of context joins. This latter complication could possibly be avoided if we separate the context of intuitionistic assumptions and the context of linear assumptions. However, this kind of separation between intuitionistic and linear assumptions is not trivial in the dependently typed setting, because types of linear assumptions may depend on intuitionistic assumptions.

This design of context variables and capturing their dependency is essential to Lincx and to the smooth extension of contextual types to the linear setting. As the leaf-level context variables uniquely describe a context characterized by a tree of context joins, we only track the leaf-level context variables as assumptions while type checking an object, but justify the validity of context variables that occur as interior nodes through the leaf-level variables. We want to emphasize that this kind of encoding of context variables does not need to be exposed to programmers.

7.2.2 Example: CPS translation

As a second example, we implement the translation of programs into continuation passing style following [Danvy and Filinski, 1992]. Concretely, we follow closely the existing implementation of type-preserving CPS translation in Beluga from [Belanger et al., 2013], but enforce that the continuations are used linearly, an idea from [Berdine et al., 2002]. Although context splits do not arise in this example, as we only have one linear variable (standing for the continuation) in our context, we include it, to showcase the mix and interplay of intuitionistic and linear function spaces in encoding program transformations.

Our source language is a simple language consisting of natural numbers, functions, applications and let-expressions. We only model well-typed expressions by defining a type `source` that is indexed by types `tp`.

```
Linear LF tp : type =
| nat   : tp
| arr   : tp → tp → tp;
```

```
Linear LF source : tp → type =
| app   : source (arr S T) → source S
          → source T
| lam   : (source S → source T)
          → source (arr S T)
| z     : source nat
| s     : source nat → source nat;
```

In our target language we distinguish between expressions, characterized by the type exp and values, defined by the type `value`. Continuations take values as their argument and return an exp. We ensure that each continuation itself is used exactly once by abstracting exp over the linear function space.

```
Linear LF exp : type =
| kapp     : value (arr S T) → value S → (value T → exp) ⊸ exp
| halt     : value S → exp
and value : tp → type =
| klam     : (value S → (value T → exp) ⊸ exp) → value (arr S T)
| kz       : value nat
| ksuc     : value nat → value nat ;
```

We can now define our `source` and `value` contexts as unrestricted contexts by marking the schema element with the tag u.

```
schema sctx = u (source T);
schema vctx = u (value T);
```

To guarantee that the resulting expression is well-typed, we define a context relation Ctx_Rel to relate the source context to the value context (see Figure 7.3). Notice that we explicitly state that the type S of a source and target expression is closed; it does not depend on γ or δ. To distinguish between objects that depend on their surrounding context and objects that do not, we associate every index and pattern variable with a substitution (the identity substitution by default); if we want to state that a given variable is closed, we associate it with the empty substitution [].

We can now define the translation itself (see Figure 7.3). The function cpse takes in a context relation Ctx_Rel [γ] [δ] and a source term of type source S[] that depends on context γ. It then returns the corresponding expression of type exp, depending on context δ extended by a continuation from value S to exp. The fact that the continuation is used only once in exp is enforced by declaring it linear in the context. The translation proceeds by pattern matching on the source term. We concentrate here on the interesting cases.

Parameter Variable If we encounter a variable from the context γ, written as #p, we look up the corresponding variable #q in the target context δ by using the context relation and we pass it to the continuation k. We omit here the definition of the lookup function which is straightforward. We use _ where we believe that the omitted object can reasonably be inferred. Finally, we note that the expression k #q is well-typed in the context δ, k$\hat{:}$value _ $\to$ exp, as k is well-typed in the context that only contains the declaration k$\hat{:}$value _ $\to$ exp and #q is well-typed in the context δ.

Constant z We first retrieve the target context δ to build the final expression by pattern matching on the context relation r. Then we pass kz to the continuation k in the context δ,k$\hat{:}$value nat $\to$ exp. Note that an application k kz is well-typed in δ,k$\hat{:}$value nat $\to$ exp, as kz is well-typed in δ, i.e. its unrestricted part.

Lambda Abstraction To convert functions, we extend the context γ and the context relation r and convert the term M recursively in the extended

```
data Ctx_Rel: {γ:sctx}{δ:vctx} type =
Nil  : Ctx_Rel [] []
Cons : Ctx_Rel [γ] [δ] → Ctx_Rel [γ, x:source S[]] [δ, v:value S[]];

rec cpse:(γ:sctx)(δ:vctx)(S:[ ⊢ tp])
         Ctx_Rel [γ] [δ] →
         [γ ⊢ source S[]] →
         [δ, k^:value S[] → exp ⊢ exp] =
fn r, e ⇒ case e of
| [γ ⊢ #p] ⇒
  let [δ⊢ #q] = lookup r [γ ⊢ #p] in
  [δ, k^:value _ → exp ⊢ k #q]

| [γ ⊢ z] ⇒ let (r : Ctx_Rel [γ] [δ]) = r in
  [δ,k^:value nat → exp ⊢ k kz]

| [γ ⊢ suc N] ⇒
  let [δ,k^:value nat → exp ⊢ P] = cpse r [γ ⊢ N] in
  [δ,k^:value nat → exp ⊢  P[λp. k (ksuc p)] ]

| [γ ⊢ lam λx. M] ⇒
  let [δ, v:value S[], k^:value T[] → exp ⊢ P] = cpse [Cons r] [γ, x:
    source _ ⊢ M] in
  [δ, k^:value (arr S[] T[]) → exp ⊢ k (klam (λx.λ̂c. P))]

| [γ ⊢ app M N] ⇒
  let [δ, k1^:value (arr S[] T[]) → exp ⊢ P] = cpse r [γ ⊢ M] in
  let [δ, k2^:value S[] → exp ⊢ Q] = cpse r [γ ⊢ N] in
  [δ,k^:value T[] → exp ⊢ P[λf. Q[λx. kapp f x ^ k]]];
```

Figure 7.3: CPS Translation

context to obtain the target expression P. We then pass to the continuation k the value klam λx.$\widehat{\lambda}$c.P.

Application Finally, let us consider the source term app M N. We translate both M and N recursively to produce the target terms P and Q respectively. We then substitute for the continuation variable k2 in Q a continuation consuming the local argument of an application. A continuation is then built from this, expecting the function to which the local argument is applied and substituted for k1 in P producing a well-typed expression, if a continuation for the resulting type S is provided.

We take advantage of our built-in substitution here to reduce any administrative redexes. The term (λx. kapp f x ^ k) that we substitute for references to k2 in Q will be β-reduced wherever that k2 appears in a function call position, such as the function calls introduced in the variable case. We hence reduce administrative redexes using the built-in (linear) LF application.

7.3 Lincx: A Linear Logical Framework with First-Class Contexts

Throughout this section we gradually introduce Lincx, a contextual linear logical framework with first-class contexts (i.e. context variables) that generalizes the linear logical framework LLF [Cervesato and Pfenning, 1996] and contextual LF [Cave and Pientka, 2012]. Figure 7.4 presents both contextual linear LF (see Section 7.3.1) and its meta-language (see Section 7.3.6).

7.3.1 Syntax of Contextual Linear LF

Lincx allows for linear types, written $A \multimap B$, and dependent types $\Pi x{:}A.B$ where x may be unrestricted in B. We follow recent presentations where we only describe canonical LF objects using hereditary substitution.

As usual, our framework supports constants, (linear) functions, and (linear) applications. We only consider objects in η-long β-normal form, as these are the only meaningful terms in a logical framework. While the grammar characterizes objects in β-normal form, the bi-directional typing rules will also ensure that objects are η-long. Normal canonical terms are either intuitionistic lambda abstractions, linear lambda abstractions, or neutral atomic terms. We define (linear) applications as neutral atomic terms using a spine representation [Cervesato and Pfenning, 2003], as it makes the termination of hereditary substitution easier to establish. For example, instead of $x\ M_1 \ldots M_n$, we write $x \cdot M_1;\ \ldots;\ M_n;\ \epsilon$. The three possible variants of a spine head are: a variable x, a constant c or a parameter variable closure $p[\sigma]$.

Our framework contains *ordinary bound variables* x which may refer to a variable declaration in a context Ψ or may be bound by either the unrestricted or linear lambda-abstraction, or by the dependent type $\Pi x{:}A.B$. Similarly to contextual LF, Lincx also allows two kinds of *contextual variables* as terms. First, the meta-variable u of type $(\Psi \vdash P)$

Contextual Linear LF

Kinds	K	$::=$	$\mathsf{type} \mid \Pi x{:}A.K$
Types	A, B	$::=$	$P \mid \Pi x{:}A.B \mid A \multimap B$
Atomic Types	P, Q	$::=$	$a \cdot S$
Heads	H	$::=$	$x \mid c \mid p[\sigma]$
Spines	S	$::=$	$\epsilon \mid M\,;S \mid M\,\hat{;}\,S$
Atomic Terms	R	$::=$	$H \cdot S \mid u[\sigma]$
Canonical Terms	M, N	$::=$	$R \mid \lambda x.M \mid \widehat{\lambda} x.M$
Variable Declarations	D	$::=$	$x{:}A \mid x\hat{:}A \mid x\check{:}A$
Contexts	Ψ, Φ	$::=$	$\cdot \mid \psi_m \mid \Psi, D$
Substitutions	σ, τ	$::=$	$\cdot \mid \mathsf{id}_\psi \mid \sigma, M$

Meta-Language

Meta-Variables	X	$::=$	$u \mid p \mid \psi_i$
Meta-Objects	C	$::=$	$\widetilde{\Psi}.R \mid \widetilde{\Psi}.H \mid \Psi$
Context Schema Elem.	E	$::=$	$\lambda(\overrightarrow{x_i{:}A_i}).A \mid \lambda(\overrightarrow{x_i{:}A_i}).\widehat{A}$
Context Schemata	G	$::=$	$E \mid G + E$
Context Var. Indices	m	$::=$	$\epsilon \mid i \mid m \bowtie n$
Meta Types	U	$::=$	$\Psi \vdash P \mid \Psi \vdash \#A \mid G$
Meta-Contexts	Δ	$::=$	$\cdot \mid \Delta, X : U$
Meta-Substitutions	Θ	$::=$	$\cdot \mid \Theta, C/X$

Figure 7.4: Contextual Linear LF with first-class contexts

stands for a general LF object of atomic type P and uses the variables declared in Ψ. Second, the parameter variable p of type $(\Psi \vdash \#A)$ stands for a variable object of type A from the context Ψ. These contextual variables are associated with a postponed substitution σ representing a closure. The intention is to apply σ as soon as we know what u (or p resp.) stands for.

The system has one mixed context Ψ containing both intuitionistic and linear assumptions: $x{:}A$ is an intuitionistic assumption in the context (also called *unrestricted assumption*), $x\hat{:}A$ represents a linear assumption and $x\check{:}A$ stands for its dual, an unavailable assumption. It is worth noting that we use $\widehat{\ }$ throughout the system description to indicate a linear object – be it term, variable, name etc. Similarly, $\check{\ }$ always denotes an unavailable resource.

In the simultaneous substitution σ, we do not make the domain explicit. Rather, we think of a substitution together with its domain Ψ; the i-th element in σ corresponds to the i-th declaration in Ψ. The expression id_ψ denotes the identity substitution with domain ψ_m for some index m; we write $\cdot$ for the empty substitution. We build substitutions using normal terms M. We must however be careful: note that a variable x is only a normal term if it is of base type. As we push a substitution σ through a λ-abstraction $\lambda x.M$, we need to extend σ with x. The resulting substitution σ, x might not be well-formed, since x might not be of base type and, in fact, we do not know its type. This is taken care of in our definition of substitution, based on contextual LF [Cave and Pientka, 2013]. As we substitute and replace a context variable with a concrete context, we unfold and generate an (η-expanded) identity substitution for a given context Ψ.

7.3.2 Contexts and Context Joins

Since linearity introduces context splitting, context maintenance is crucial in any linear system. When we allow for first-class contexts, as we do in Lincx, it becomes much harder: we now need to ensure that, upon instantiation of the context variables, we do not accidentally join two contexts sharing a linear variable. To enforce this in Lincx, we allow for at most one (indexed) context variable per context and use indices to abstractly describe splitting. This lets us generalize the standard equational theory for contexts based on context joins to include context variables.

As mentioned above, contexts in Lincx are mixed. Besides linear and intuitionistic assumptions, we allow for unavailable assumptions following the approach of [Schack-Nielsen and Schürmann, 2010], in order to maintain symmetry when splitting a context: if $\Psi = \Psi_1 \bowtie \Psi_2$, then Ψ_1 and Ψ_2 both contain all the variables declared in Ψ; however, if Ψ_1 contains a linear assumption $x \mathrel{\hat{:}} A$, Ψ_2 will contain its unavailable counterpart $x \mathrel{\check{:}} A$ (and vice-versa).

To account for context splitting in the presence of context variables, we index the latter. The indices are freely built from elements of an infinite, countable set $\mathcal{I}$, through a join operation ($\bowtie$). It is associative and commutative, with ϵ as its neutral element. In other words, $(\mathcal{I}^*, \bowtie, \epsilon)$ is a (partial) free commutative monoid over $\mathcal{I}$. For our presentation it is important that no element of the monoid is invertible, that is if $m \bowtie n = \epsilon$ then $m = n = \epsilon$. In the process of joining contexts, it is crucial to ensure that each linear variable is used only once: we do not allow a join of $\Psi, x \hat{:} A$ with $\Phi, x \hat{:} A$. To express the fact that indices m and n share no elements of $\mathcal{I}$ and hence the join of ψ_m with ψ_n is meaningful, we use the notation $m \perp n$. In fact we will overload $\bowtie$, changing it into a partial operation $m \bowtie n$ that fails when $m \not\perp n$. This is because we want the result of joining two context variables to continue being a correct context upon instantiation. We will come back to this point in Section 7.3.6, when discussing meta-substitution for context variables.

To give more intuition, the implementation of the indices in our formalization of the system is using binary numbers, where $\mathcal{I}$ contains powers of 2, $\bowtie$ is defined as a binary *OR* and $\epsilon = 0$. $m \perp n$ holds when m and n use different powers of 2 in their binary representation. We can also simply think of indices m as sets of elements from $\mathcal{I}$ with $\bowtie$ being $\cup$ for sets not sharing any elements.

The only context variables tracked in the meta-context Δ are the *leaf-level* context variables ψ_i. We require that these use elements of the carrier set $i \in \mathcal{I}$ as indices. To construct context variables for use in contexts, we combine leaf-level context variables using $\bowtie$ on indices. Consider again the tree describing the context joins (see Figure 7.2). In this example, we have the leaf-level context variables γ_1, γ_{21}, and γ_{22}. These are the only context variables we track in the meta-context Δ. Using a binary encoding we would use the subscripts 100, 010 and 001 instead of 1, 21, and 22.

Rules of constructing a well-formed context (Figure 7.5) describe four possible initial cases of context construction. First, the empty context, written simply as $\cdot$, is well-formed. Next, there are two possibilities why a context denoted by a context variable ψ_i is well-formed. If the context variable ψ_i is declared in the meta-context Δ, then it is well-formed and describes a leaf-variable. To guarantee that also context variables that describe intermediate nodes in our context tree are well-formed, we have a composition rule that allows joining two well-formed context variables using $\bowtie$ operation on indices; the restriction we make on $\bowtie$ ensures that they do not share any leaf-level variables. ψ_ϵ forms a well-formed context as long as for some index i, there is a context variable ψ_i declared in Δ. ψ_ϵ then stands for the intuitionistic part of the context we are

$\boxed{\Delta \vdash \Psi\ \mathsf{ctx}}$ $\quad$ Ψ is a valid context under meta-context Δ

$$\frac{}{\Delta \vdash \cdot\ \mathsf{ctx}} \qquad \frac{\psi_i \in \mathsf{dom}(\Delta)}{\Delta \vdash \psi_\epsilon\ \mathsf{ctx}} \qquad \frac{\psi_i \in \mathsf{dom}(\Delta)}{\Delta \vdash \psi_i\ \mathsf{ctx}}$$

$$\frac{\Delta \vdash \psi_k\ \mathsf{ctx} \quad \Delta \vdash \psi_l\ \mathsf{ctx} \quad m = k \bowtie l}{\Delta \vdash \psi_m\ \mathsf{ctx}}$$

$$\frac{\Delta \vdash \Psi\ \mathsf{ctx} \quad \Delta; \overline{\Psi} \vdash A\ \mathsf{type} \quad D \in \{x{:}A, x\hat{:}A, x\check{:}A\}}{\Delta \vdash \Psi, D\ \mathsf{ctx}}$$

Figure 7.5: Well-formed contexts

abstracting over using ψ_i. Finally, the last case for context extensions is straightforward.

In general we write Γ for contexts that do not start with a context variable and Ψ, Γ for the extension of context Ψ by the variable declarations of Γ.

When defining our inference rules, we will often need to access the *intuitionistic part* of a context. Much like in linear LF [Cervesato and Pfenning, 1996], we introduce the function $\overline{\Psi}$ which is defined as follows:

$$\begin{array}{lcl}
\boxed{\overline{\Psi}} & & \text{Intuitionistic part of } \Psi \\
\overline{\cdot} & = & \cdot \\
\overline{\psi_m} & = & \psi_\epsilon \\
\overline{\Psi, x{:}A} & = & \overline{\Psi}, x{:}A \\
\overline{\Psi, x\hat{:}A} & = & \overline{\Psi}, x\check{:}A \\
\overline{\Psi, x\check{:}A} & = & \overline{\Psi}, x\check{:}A
\end{array}$$

Note that this function does not remove any variable declarations from Ψ, it simply makes them unavailable. Further, when applying this function to a context variable, it drops all the indices, indicating access to only the shared part of the context variable. After we instantiate ψ_m with a concrete context, we will apply the operation. Extracting the intuitionistic part of a context is hence simply postponed.

Further, we define notation $\mathsf{unr}(\Psi)$ to denote an unrestricted context, i.e. a context that only contains unrestricted assumptions; while $\overline{\Psi}$

$\boxed{\Psi = \Psi_1 \bowtie \Psi_2}$ Context Ψ is a join of Ψ_1 and Ψ_2

$$\frac{}{\cdot = \cdot \bowtie \cdot} \qquad \frac{m = k \bowtie l}{\psi_m = \psi_k \bowtie \psi_l}$$

$$\frac{\Psi = \Psi_1 \bowtie \Psi_2}{\Psi, x{:}A = \Psi_1, x{:}A \bowtie \Psi_2, x{:}A} \qquad \frac{\Psi = \Psi_1 \bowtie \Psi_2}{\Psi, x\check{:}A = \Psi_1, x\check{:}A \bowtie \Psi_2, x\check{:}A}$$

$$\frac{\Psi = \Psi_1 \bowtie \Psi_2}{\Psi, x\hat{:}A = \Psi_1, x\hat{:}A \bowtie \Psi_2, x\check{:}A} \qquad \frac{\Psi = \Psi_1 \bowtie \Psi_2}{\Psi, x\hat{:}A = \Psi_1, x\check{:}A \bowtie \Psi_2, x\hat{:}A}$$

Figure 7.6: Joining contexts

drops all linear assumptions, $\mathsf{unr}(\Psi)$ simply verifies that Ψ is a purely intuitionistic context. In other words, $\mathsf{unr}(\Psi)$ holds if and only if $\overline{\Psi} = \Psi$. We omit here its (straightforward) judgmental definition.

The rules for joining contexts (see Figure 7.6) follow the approach presented by Schack-Nielsen in his PhD dissertation [Schack-Nielsen, 2011], but are generalized to take into account context variables. Because of the monoid structure of context variable indices, the description can be quite concise while still preserving the desired properties of this operation. For instance the expected property $\Psi = \Psi \bowtie \overline{\Psi}$ follows, on the context variable level, from ϵ being the neutral element of $\bowtie$. Indeed, for any ψ_m, we have that $\psi_m = \psi_m \bowtie \psi_\epsilon$.

It is also important to note that, thanks to the determinism of $\bowtie$, context joins are unique. In other words, if both $\Psi = \Psi_1 \bowtie \Psi_2$ and $\Phi = \Psi_1 \bowtie \Psi_2$, it follows that $\Psi = \Phi$. On the other hand, context splitting is non-deterministic: given a context Ψ we have numerous options of splitting it into Ψ_1 and Ψ_2, since each linear variable can go to either of the components.

We finish this section by describing the equational theory of context joins. We expect joining contexts to be a commutative and associative operation, and the unrestricted parts of contexts in the join should be equal. Further, it is always possible to extend a valid join with a ground unrestricted context, and $\overline{\Psi}$ can always be joined with Ψ without changing the result.

Lemma 7.1 (Theory of context joins).

(i) (Commutativity) If $\Psi = \Psi_1 \bowtie \Psi_2$ *then* $\Psi = \Psi_2 \bowtie \Psi_1$.

(ii) (Associativity$_1$) If $\Psi = \Psi_1 \bowtie \Psi_2$ and $\Psi_1 = \Psi_{11} \bowtie \Psi_{12}$ then there exists a context Ψ_0 s.t. $\Psi = \Psi_{11} \bowtie \Psi_0$ and $\Psi_0 = \Psi_{12} \bowtie \Psi_2$.

(iii) (Associativity$_2$) If $\Psi = \Psi_1 \bowtie \Psi_2$ and $\Psi_2 = \Psi_{21} \bowtie \Psi_{22}$ then there exists a context Ψ_0 s.t. $\Psi_0 = \Psi_1 \bowtie \Psi_{21}$ and $\Psi = \Psi_0 \bowtie \Psi_{22}$.

(iv) If $\Psi = \Psi_1 \bowtie \Psi_2$ then $\overline{\Psi} = \overline{\Psi_1} = \overline{\Psi_2}$.

(v) If $\mathsf{unr}(\Gamma)$ and $\Psi = \Psi_1 \bowtie \Psi_2$ then $\Psi, \Gamma = \Psi_1, \Gamma \bowtie \Psi_2, \Gamma$.

(vi) For any Ψ, $\Psi = \Psi \bowtie \overline{\Psi}$.

We will need these properties to prove lemmas about typing and substitution, specifically for the cases that call for specific context joins.

7.3.3 Typing for Terms and Substitutions

We now describe the bi-directional typing rules of Lincx terms (see Figure 7.7). All typing judgements have access to the meta-context Δ, context Ψ, and to a fixed well-typed signature Σ where we store constants c together with their types and kinds. Lincx objects may depend on variables declared in the context Ψ and a fixed meta-context Δ which contains contextual variables such as meta-variables u, parameter variables p, and context variables. Although the rules are bi-directional, they do not give a direct algorithm, as we need to split a context Ψ into contexts Ψ_1 and Ψ_2 such that $\Psi = \Psi_1 \bowtie \Psi_2$ (see for example the rule for checking $H \cdot S$ against a base type P). This operation is in itself non-deterministic, however since our system is linear there is only one split that makes the components (for example H and S in $H \cdot S$) typecheck.

Typing rules presented in Figure 7.7 are, perhaps unsurprisingly, a fusion between contextual LF and linear LF. As in contextual LF, the typing for meta-variable closures and parameter variable closures is straightforward. A meta-variable $u : (\Psi \vdash P)$ represents an open LF object (a "hole" in a term). As mentioned earlier it has, associated with it, a postponed substitution σ, applied as soon as u is made concrete. Similarly, a parameter variable $p : (\Psi \vdash \#A)$ represents an LF variable – either an unrestricted or linear one.

As in linear LF, we have two lambda abstraction rules (one introducing intuitionistic, the other linear assumptions) and two corresponding variable cases. Moreover, we ensure that types only depend on the unrestricted part of a context when checking that two types are equal. As

$\boxed{\Delta;\Psi \vdash M \Leftarrow A}$ Term M checks against type A

$$\frac{\Delta;\Psi, x{:}A \vdash M \Leftarrow B}{\Delta;\Psi \vdash \lambda x.M \Leftarrow \Pi x{:}A.B} \qquad \frac{\Delta;\Psi, x\hat{:}A \vdash M \Leftarrow B}{\Delta;\Psi \vdash \widehat{\lambda} x.M \Leftarrow A \multimap B}$$

$$\frac{u : (\Phi \vdash P) \in \Delta \quad \Delta;\Psi \vdash \sigma \Leftarrow \Phi \quad \Delta;\overline{\Psi} \vdash [\sigma]_{\overline{\Phi}} P = Q}{\Delta;\Psi \vdash u[\sigma] \Leftarrow Q}$$

$$\frac{\Delta;\Psi_1 \vdash H \Rightarrow A \quad \Delta;\Psi_2 \vdash S > A \Rightarrow P \quad \Delta;\overline{\Psi} \vdash P = Q \quad \Psi = \Psi_1 \bowtie \Psi_2}{\Delta;\Psi \vdash H \cdot S \Leftarrow Q}$$

$\boxed{\Delta;\Psi \vdash H \Rightarrow A}$ Head H synthesizes a type A

$$\frac{c{:}A \in \Sigma \quad \mathsf{unr}(\Psi)}{\Delta;\Psi \vdash c \Rightarrow A} \qquad \frac{p : (\Phi \vdash \#A) \in \Delta \quad \Delta;\Psi \vdash \sigma \Leftarrow \Phi}{\Delta;\Psi \vdash p[\sigma] \Rightarrow [\sigma]_{\overline{\Phi}} A}$$

$$\frac{\mathsf{unr}(\Psi) \quad x{:}A \in \Psi}{\Delta;\Psi \vdash x \Rightarrow A} \qquad \frac{\mathsf{unr}(\Psi_1) \quad \mathsf{unr}(\Psi_2)}{\Delta;\Psi_1, x\hat{:}A, \Psi_2 \vdash x \Rightarrow A}$$

$\boxed{\Delta;\Psi \vdash S > A \Rightarrow P}$ Spine S synthesizes type P

$$\frac{\mathsf{unr}(\Psi)}{\Delta;\Psi \vdash \epsilon > P \Rightarrow P} \qquad \frac{\Delta;\overline{\Psi} \vdash M \Leftarrow A \quad \Delta;\Psi \vdash S > [M/x]_A B \Rightarrow P}{\Delta;\Psi \vdash M\,;S > \Pi x{:}A.B \Rightarrow P}$$

$$\frac{\Delta;\Psi_1 \vdash M \Leftarrow A \quad \Delta;\Psi_2 \vdash S > B \Rightarrow P \quad \Psi = \Psi_1 \bowtie \Psi_2}{\Delta;\Psi \vdash M\,\hat{;}\,S > A \multimap B \Rightarrow P}$$

Figure 7.7: Typing rules for terms

we rely on hereditary substitutions, this equality check ends up being syntactic equality. Similarly, when we consider a spine $M\,;S$ and check it against the dependent type $\Pi x{:}A.B$, we make sure that M has type A in the unrestricted context before continuing to check the spine S against $[M/x]_A B$. When we encounter a spine $M\,\hat{;}\,S$ and check it against the linear type $A \multimap B$ in the context Ψ, we must show that there exists a split s.t. $\Psi = \Psi_1 \bowtie \Psi_2$ and then check that the term M has type A in the context Ψ_1 and the remaining spine S is checked against B to synthesize a type P.

Finally, we consider the typing rules for substitutions, presented in

$\boxed{\Delta;\Psi \vdash \sigma \Leftarrow \Phi}$ Substitution σ maps variables from Φ to Ψ

$$\frac{\mathsf{unr}(\Psi)}{\Delta;\Psi \vdash \cdot \Leftarrow \cdot} \qquad \frac{\mathsf{unr}(\Gamma)}{\Delta;\psi_m,\Gamma \vdash \mathsf{id}_\psi \Leftarrow \psi_m}$$

$$\frac{\Delta;\Psi \vdash \sigma \Leftarrow \Phi \quad \Delta;\overline{\Psi} \vdash M \Leftarrow [\sigma]_{\overline{\Phi}}A}{\Delta;\Psi \vdash \sigma, M \Leftarrow \Phi, x{:}A}$$

$$\frac{\Delta;\Psi_1 \vdash \sigma \Leftarrow \Phi \quad \Delta;\Psi_2 \vdash M \Leftarrow [\sigma]_{\overline{\Phi}}A \quad \Psi = \Psi_1 \bowtie \Psi_2}{\Delta;\Psi \vdash \sigma, M \Leftarrow \Phi, x\hat{:}A}$$

$$\frac{\Delta;\Psi \vdash \sigma \Leftarrow \Phi \quad \overline{\Psi} = \overline{\Psi'} \quad \Delta;\Psi' \vdash M \Leftarrow [\sigma]_{\overline{\Phi}}A}{\Delta;\Psi \vdash \sigma, M \Leftarrow \Phi, x\check{:}A}$$

Figure 7.8: Typing rules for substitutions

Figure 7.8. We exercise care in making sure the range context in the base cases, i.e. where the substitution is empty or the identity, is unrestricted. This guarantees weakening and contraction for unrestricted contexts.

The substitution σ, M is well-typed with domain $\Phi, x{:}A$ and range Ψ, if σ is a substitution from Φ to the context Ψ and in addition M has type $[\sigma]_\Phi A$ in the unrestricted context $\overline{\Psi}$. The substitution σ, M is well-typed with domain $\Phi, x\hat{:}A$ and range Ψ, if there exists a context split $\Psi = \Psi_1 \bowtie \Psi_2$ s.t. σ is a substitution with domain Φ and range Ψ_1 and M is a well-typed term in the context Ψ_2. The substitution σ, M is well-typed with domain $\Phi, x\check{:}A$ and range Ψ, if σ is a substitution from Φ to Ψ and for some context Ψ', $\overline{\Psi} = \overline{\Psi'}$, M is a well-typed term in the context Ψ'. This last rule, extending the substitution domain by an unavailable variable, is perhaps a little surprising. Intuitively we may want to skip the unavailable variable of a substitution. This would however mean that we have to perform not only context splitting, but also substitution splitting when defining the operation of simultaneous substitution. An alternative is to use an arbitrary term M to be substituted for this unavailable variable, as the typing rules ensure it will never actually occur in the term in which we substitute. When establishing termination of type-checking, it is then important that M type checks in a context that can be generated from the one we already have. We ensure this with a side condition $\overline{\Psi} = \overline{\Psi'}$. By enforcing that the unrestricted parts of Ψ and Ψ'

are equal we limit the choices that we have for Ψ' deciding which linear variables to take (linear) and which to drop (unavailable), and deciding on the index of context variable.

When considering an identity substitution id_ψ, we allow for some ambiguity: we can use any ψ_m for both the domain and range of id_ψ. Upon meta-substitution, all instantiations of ψ_m will have the same names and types of variables; the only thing differentiating them will be their status (intuitionistic, linear or unavailable). Since substitutions do not store information about the status of variables they substitute for (this information is available only in the domain and range), the constructed identity substitution will be the same regardless of the initial choice of ψ_m – it will however have a different type.

The observation above has a more general consequence, allowing us to avoid substitution splits when defining the operation of hereditary substitution: if a substitution in Lincx transforms context Φ to context Ψ, it does so also for their unrestricted fragments.

Lemma 7.2. *If* $\Delta; \Psi \vdash \sigma \Leftarrow \Phi$ *then* $\Delta; \overline{\Psi} \vdash \sigma \Leftarrow \overline{\Phi}$.

7.3.4 Hereditary Substitution

Next we will characterise the operation of hereditary substitution, which allows us to consider only normal forms in our grammar and typing rules, making the decidability of type-checking easy to establish.

As usual, we annotate hereditary substitutions with an approximation of the type of the term we substitute for to guarantee termination.

$$\text{Type approximations} \quad \alpha, \beta \quad ::= \quad a \mid \alpha \to \beta \mid \alpha \multimap \beta$$

We then define the dependency erasure operator $(-)^-$ as follows:

$$\boxed{A^- = \alpha} \qquad \alpha \text{ is a type approximation of } A$$

$$\begin{aligned}
(a \cdot S)^- &= a \\
(\Pi x{:}A.B)^- &= A^- \to B^- \\
(A \multimap B)^- &= A^- \multimap B^-
\end{aligned}$$

We will sometimes tacitly apply the dependency erasure operator $(-)^-$ in the following definitions. Hereditary single substitution for Lincx is standard and closely follows [Cave and Pientka, 2013], since linearity does not induce any complications. When executing the current substitution would create redexes, we proceed by hereditarily performing another substitution, using a reduction operation $\mathsf{reduce}(M, S)$.

$\boxed{\mathsf{reduce}(M : \alpha, S) = N}$ Result of reducing M applied to spine S is N

$$\begin{aligned}
&\mathsf{reduce}(\lambda x.M : \alpha \rightarrow \beta, (N\,;S)) &&= \mathsf{reduce}([N/x]_\alpha M : \beta, S)\\
&\mathsf{reduce}(\widehat{\lambda} x.M : \alpha \multimap \beta, (N\,\hat{;}S)) &&= \mathsf{reduce}([N/x]_\alpha M : \beta, S)\\
&\mathsf{reduce}(R : a, \epsilon) &&= R\\
&\mathsf{reduce}(M : \alpha, S) &&= \bot
\end{aligned}$$

For the sake of completeness, the full rules for hereditary single substitution can be found in the Appendix 7.A.1 with rules presented in Figure 7.12.

Termination can be readily established:

Theorem 7.1 (Termination of hereditary single substitution)**.** *The hereditary substitutions $[M/x]_\alpha(N)$ and $\mathsf{reduce}(M : \alpha, S)$ terminate, either by failing or successfully producing a result.*

The following theorem provides typing for the hereditary substitution. We use J to stand for any of the forms of judgements defined above.

Theorem 7.2 (Hereditary single substitution property)**.**

(i) *If $\Delta; \overline{\Psi} \vdash M \Leftarrow A$ and $\Delta; \Psi, x{:}A \vdash J$ then $\Delta; \Psi \vdash [M/x]_A J$.*

(ii) *If $\Delta; \Psi_1 \vdash M \Leftarrow A$, $\Delta; \Psi_2, x\hat{:}A \vdash J$ and $\Psi = \Psi_1 \bowtie \Psi_2$ then also $\Delta; \Psi \vdash [M/x]_A J$*

(iii) *If $\Delta; \Psi_1 \vdash M \Leftarrow A$, $\Delta; \Psi_2 \vdash S > A \Rightarrow B$, $\Psi = \Psi_1 \bowtie \Psi_2$ and $\mathsf{reduce}(M : A^-, S) = M'$ then also $\Delta; \Psi \vdash M' \Leftarrow B$*

We can easily generalize hereditary substitution to simultaneous substitution. We focus here on the simultaneous substitution in a canonical terms (see Figure 7.9). Hereditary simultaneous substitution relies on a lookup function that is defined below. Note that $(\sigma, M)_{\Psi, x\check{:}A}(x) = \bot$,

$\boxed{[\sigma]^{\widetilde{\Phi}}_{\Psi} M}$ Substitution of the variables of Ψ in a canonical term (leaving elements of $\widetilde{\Phi}$ unchanged)

$$
\begin{array}{lll}
[\sigma]^{\widetilde{\Phi}}_{\Psi}(\lambda y.N) & = \lambda y.N' & \text{where } [\sigma]^{\widetilde{\Phi},y}_{\Psi} N = N', \\
 & & \text{choosing } y \notin \Psi, y \notin \mathsf{FV}(\sigma) \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(\widehat{\lambda} y.N) & = \widehat{\lambda} y.N' & \text{where } [\sigma]^{\widetilde{\Phi},\hat{y}}_{\Psi} N = N', \\
 & & \text{choosing } y \notin \Psi, y \notin \mathsf{FV}(\sigma) \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(u[\tau]) & = u[\tau'] & \text{where } [\sigma]^{\widetilde{\Phi}}_{\Psi} \tau = \tau' \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(c \cdot S) & = c \cdot S' & \text{where } [\sigma]^{\widetilde{\Phi}}_{\Psi} S = S' \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(x \cdot S) & = \mathsf{reduce}(M : \alpha, S') & \text{where } \Psi = \Psi_1 \bowtie \Psi_2, \text{ and } x \notin \widetilde{\Phi}, \\
 & & \text{and } \sigma_{\Psi_1}(x) = M : \alpha, \\
 & & \text{and } [\sigma]^{\widetilde{\Phi}}_{\Psi_2} S = S' \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(y \cdot S) & = y \cdot S' & \text{where } y \in \widetilde{\Phi}, \text{ and } [\sigma]^{\widetilde{\Phi}}_{\Psi} S = S' \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(y \cdot S) & = y \cdot S' & \text{where } \widetilde{y} \in \widetilde{\Phi}, \text{ and } [\sigma]^{\widetilde{\Phi} \backslash \widetilde{y}}_{\Psi} S = S' \\
[\sigma]^{\widetilde{\Phi}}_{\Psi}(p[\tau] \cdot S) & = p[\tau'] \cdot S' & \text{where } \Psi = \Psi_1 \bowtie \Psi_2, \\
 & & \text{and } \widetilde{\Phi} = \widetilde{\Phi}_1 \bowtie \widetilde{\Phi}_2, \\
 & & \text{and } [\sigma]^{\widetilde{\Phi_1}}_{\Psi_1} \tau = \tau', \text{ and } [\sigma]^{\widetilde{\Phi_2}}_{\Psi_2} S = S'
\end{array}
$$

Figure 7.9: Simultaneous substitution

since we assume x to be unavailable in the domain of σ.

$\boxed{\sigma_{\Psi}(x)}$ Variable lookup

$$
\begin{array}{lll}
(\sigma, M)_{\Psi, x:A}(x) & = M : A^{-} & \\
(\sigma, M)_{\Psi, x \hat{:} A}(x) & = M : A^{-} & \\
(\sigma, M)_{\Psi, y:A}(x) & = \sigma_{\Psi}(x) & \text{where } y \neq x \\
(\sigma, M)_{\Psi, y \check{:} A}(x) & = \sigma_{\Psi}(x) & \text{where } y \neq x \\
\sigma_{\Psi}(x) & = \bot &
\end{array}
$$

Unlike many previous formulations of contextual LF, we do not allow substitutions to be directly extended with variables. Instead, following a more recent approach from [Cave and Pientka, 2013], we require that substitutions must be extended with η-long terms, thus guaranteeing unique normal forms for substitutions. For this reason, we maintain a list of variable names and statuses which are not to be changed, $\widetilde{\Phi}$ in $[\sigma]^{\widetilde{\Phi}}_{\Psi}$.

This list gets extended every time we pass through a lambda expression. We use it when substituting in $y \cdot S$ – if $y \in \widetilde{\Phi}$ or $\widehat{y} \in \widetilde{\Phi}$ we simply leave the head unchanged. It is important to preserve not only the name of the variable, but also its status (linear, intuitionistic or unavailable), since we sometimes have to perform a split on $\widetilde{\Phi}$. Such split works precisely like one on complete contexts, since types play no role in context splitting.

As simultaneous substitution is a transformation of contexts, it is perhaps not surprising that it becomes more complex in the presence of context splitting. Consider for instance the case where we push the substitution σ through an expression $p[\tau] \cdot S$. While σ has domain Ψ (and is ignoring variables from $\widetilde{\Phi}$) and $p[\tau] \cdot S$ is well-typed in (Ψ, Φ), the closure $p[\tau]$ is well-typed in a context (Ψ_1, Φ_1) and the spine S is well-typed in a context (Ψ_2, Φ_2) where $\Psi = \Psi_1 \bowtie \Psi_2$ and $\Phi = \Phi_1 \bowtie \Phi_2$. As a consequence, $[\sigma]^{\widetilde{\Phi}}_{\Psi}\tau$ and $[\sigma]^{\widetilde{\Phi}}_{\Psi}S$ would be ill-typed, however $[\sigma]^{\widetilde{\Phi_1}}_{\Psi_1}\tau$ and $[\sigma]^{\widetilde{\Phi_2}}_{\Psi_2}S$ will work well. Notice that it is only the domain of the substitution that we need to split, not the substitution itself.

Similarly to the case for hereditary single substitution, the theorem below provides typing for simultaneous substitution.

Theorem 7.3 (Simultaneous substitution property)**.** *If* $\Delta; \Psi \vdash J$ *and* $\Delta; \Phi \vdash \sigma \Leftarrow \Psi$ *then* $\Delta; \Phi \vdash [\sigma]_{\Psi} J$*.*

7.3.5 Decidability of Type Checking in Contextual Linear LF

In order to establish a decidability result for type checking, we observe that the typing judgements are syntax directed. Further, when a context split is necessary (e.g. when checking $\Delta, \Psi \vdash \sigma, M \Leftarrow \Phi, x \hat{:} A$), it is possible to enumerate all the possible correct splits (all Ψ_1, Ψ_2 such that $\Psi = \Psi_1 \bowtie \Psi_2$). For exactly one of them it will hold that $\Delta; \Psi_1 \vdash \sigma \Leftarrow \Phi$ and $\Delta; \Psi_2 \vdash M \Leftarrow [\sigma]_{\overline{\Phi}} A$. Finally, in the $\Delta, \Psi \vdash \sigma, M \Leftarrow \Phi, x \check{:} A$ case, thanks to explicit mention of all the variables (including unavailable ones), we can enlist all possible contexts Ψ' well-formed under Δ and such that $\overline{\Psi} = \overline{\Psi'}$.

Theorem 7.4 (Decidability of type checking)**.** *Type checking is decidable.*

7.3.6 Lincx's Meta-Language

To use contextual linear LF as an index language in Beluga, we have to be able to lift Lincx objects to meta-types and meta-objects and the definition

$\boxed{\vdash \Delta\ \mathsf{mctx}}$ $\qquad$ Δ is a valid meta-context

$$\frac{}{\vdash \cdot\ \mathsf{mctx}} \qquad \frac{\vdash \Delta\ \mathsf{mctx} \quad \Delta; \overline{\Psi} \vdash P\ \mathsf{type}}{\vdash \Delta, u : (\Psi \vdash P)\ \mathsf{mctx}}$$

$$\frac{\vdash \Delta\ \mathsf{mctx} \quad \Delta; \overline{\Psi} \vdash A\ \mathsf{type}}{\vdash \Delta, p : (\Psi \vdash \#A)\ \mathsf{mctx}} \qquad \frac{\vdash \Delta\ \mathsf{mctx} \quad i \in \mathcal{I}}{\vdash \Delta, \psi_i : G\ \mathsf{mctx}} \star$$

Figure 7.10: Well-formed meta-contexts

of the meta-substitution operation. We are basing our presentation on one for contextual LF [Cave and Pientka, 2012].

Figure 7.4 presents the meta-language of Lincx. Meta-objects are either contextual objects or contexts. The former may be instantiations to parameter variables $p : (\Psi \vdash \#A)$ or meta-variables $u : (\Psi \vdash P)$. These objects are written $\widetilde{\Psi}.R$ where $\widetilde{\Psi}$ denotes a list of variables obtained by dropping all the type information from the declaration, but retaining the information about variable status (intuitionistic, linear or unavailable).

$$\begin{array}{lcl}
\boxed{\widetilde{\Psi}} & & \text{Name and status of variables from } \Psi \\
\widetilde{\cdot} & = & \cdot \\
\widetilde{\psi_m} & = & \psi_m \\
\widetilde{\Psi, x{:}A} & = & \widetilde{\Psi}, x \\
\widetilde{\Psi, x\hat{:}A} & = & \widetilde{\Psi}, \widehat{x} \\
\widetilde{\Psi, x\check{:}A} & = & \widetilde{\Psi}, \check{x}
\end{array}$$

Contexts as meta-objects are used to instantiate context variables $\psi_i : G$. When constructing those we must exercise caution, as we need to ensure that no linear variable is used in two contexts that are, at any point, joined. At the same time, instantiations for context variables differing only in the index (ψ_i and ψ_j) have to use precisely the same variable names and their unrestricted fragments have to be equal. It is also important to ensure that the constructed context is of a correct schema G. Schemas describe possible shapes of contexts, and each schema element can be either linear ($\lambda(\overrightarrow{x_i{:}A_i}).\widehat{A}$) or intuitionistic ($\lambda(\overrightarrow{x_i{:}A_i}).A$). This can be extended to also allow combinations of linear and intuitionistic schema elements.

$\boxed{\Psi \perp_\psi \Theta}$ Context Ψ is linearly disjoint from the range of Θ for ψ_j

$$\frac{}{\Psi \perp_\psi (\cdot)} \qquad \frac{\Psi \perp_\psi \Theta \quad \Psi' = \Psi \bowtie \Psi_j}{\Psi \perp_\psi (\Theta, \Psi_j/\psi_j)} \qquad \frac{\Psi \perp_\psi \Theta \quad X \neq \psi_j}{\Psi \perp_\psi (\Theta, C/X)}$$

$\boxed{\Delta \vdash \Theta \Leftarrow \Delta'}$ Θ has domain Δ' and range Δ

$$\frac{}{\Delta \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta \vdash \Theta \Leftarrow \Delta' \quad \Delta \vdash \Psi_i \Leftarrow G \quad \Psi_i \perp_\psi \Theta}{\Delta \vdash \Theta, \Psi_i/\psi_i \Leftarrow \Delta', \psi_i : G}$$

$$\frac{\Delta \vdash \Theta \Leftarrow \Delta' \quad \Delta \vdash C \Leftarrow [\![\Theta]\!]_{\Delta'} U}{\Delta \vdash \Theta, C/X \Leftarrow \Delta', X : U}$$

Figure 7.11: Typing rules for meta-substitutions

We now give rules for a well-formed meta-context Δ (see Figure 7.10). It is defined on the structure of Δ and is mostly straightforward. As usual, we assume the names we choose are fresh. The noteworthy case arises when we extend Δ with a context variable ψ_i. Because all context variables ψ_j will describe parts of the same context, we require their schemas to be the same. This side condition ($\star$) can be formally stated as: $\forall j.\psi_j \in \mathsf{dom}(\Delta) \rightarrow \psi_j : G \in \Delta$. Moreover, to avoid manually ensuring that indices of context variables do not cross, we require that leaf context variables use elements of the carrier set $i \in \mathcal{I}$ (i.e. they are formed without using the $\bowtie$ operation).

Typing of meta-terms is straightforward and follows precisely the schema presented in previous work. Due to space limitation we move the presentation of these rules to the Appendix (see Figure 7.13 and Figure 7.14).

Because of the interdependencies when substituting for context variables, we diverge slightly from standard presentations of typing of meta-substitutions.

First, we do not at all consider single meta-substitutions, as they would be limited only to parameter and meta-variables. In the general case it is impossible to meaningfully substitute only one context variable, as this would break the invariant that all instantiations of context variables share variable names and the intuitionistic part of the context.

Second, the typing rules for the simultaneous meta-substitution (see Figure 7.11) are specialized in the case of substituting for a context variable. When extending Θ with an instantiation Ψ_i for a context variable $\psi_i : G$, we first verify that context Ψ_i has the required schema G. We also have to check that Ψ_i can be joined with *any other* instantiation Ψ_j for context variable ψ_j already present in Θ (that is, $\Psi_i \perp_\psi \Theta$). This is enough to ensure the desired properties of meta-substitution for context variables.

We can now define the simultaneous meta-substitution. The operation itself is straightforward, as linearity does not complicate things on the meta-level (see Figure 7.16 in the Appendix). What is slightly more involved is the variable lookup function.

$\boxed{\Theta_\Delta(X)}$ Contextual variable lookup

$$\begin{array}{lll}
(\Theta, \Psi/\psi_i)_{\Delta,\psi_i:G}(\psi_\epsilon) & = \overline{\Psi} & \\
(\Theta, \Psi/\psi_i)_{\Delta,\psi_i:G}(\psi_i) & = \Psi & \\
(\Theta, \Psi/\psi_i)_{\Delta,\psi_i:G}(\psi_m) & = \Phi & \text{where } \Phi = \Psi \bowtie \Psi' \\
 & & \text{and } m = i \bowtie n \\
 & & \text{and } \Theta_\Delta(\psi_n) = \Psi' \\
(\Theta, \Psi/\psi_i)_{\Delta,\psi_i:G}(\psi_m) & = \Theta_\Delta(\psi_m) & \text{where } i \perp_\psi m \\
(\Theta, C/X)_{\Delta,X:U}(X) & = C : U & \\
(\Theta, C/Y)_{\Delta,Y:\_}(X) & = \Theta_\Delta(X) & \text{where } Y \neq X \\
\Theta_\Delta(X) & = \bot &
\end{array}$$

On parameter and meta-variables it simply returns the correct meta-object, to which the simultaneous substitution from the corresponding closure is then applied. The lookup is a bit more complicated for context variables, since Θ only contains substitutions for leaf context variables ψ_i. For arbitrary ψ_m we must therefore deconstruct the index $m = i_1 \bowtie \cdots \bowtie i_k$ and return $\Theta_\Delta(\psi_{i_1}) \bowtie \cdots \bowtie \Theta_\Delta(\psi_{i_k})$. Finally, for ψ_ϵ we simply have to find any Ψ/ψ_i in Θ and return $\overline{\Psi}$ – the typing rules for Θ ensure that the choice of ψ_i is irrelevant, as the unrestricted part of the substituted context is shared.

Theorem 7.5 (Simultaneous meta-substitution property)**.** *Having both* $\Delta \vdash \Theta \Leftarrow \Delta'$ *and* $\Delta'; \Psi \vdash J$*, it follows that* $\Delta; [\![\Theta]\!]_{\Delta'}\Psi \vdash [\![\Theta]\!]_{\Delta'} J$*.*

7.3.7 Writing Programs about Lincx Objects

We sketch here why LINCX is a suitable index language for writing programs and proofs. [Thibodeau et al., 2016] describe several requirements for plugging in an index language into the (co)inductive foundation for writing programs and proofs about them. They fall into three different classes. We will briefly touch on each one.

First, it requires that the index domain satisfies meta-substitution properties that we also prove for LINCX. Second, comparing two objects should be decidable. We satisfy this criteria, since we only characterize $\beta\eta$-long canonical forms and equality reduces to syntactic equality. The third criterion is unification of index objects. While we do not describe a unification algorithm for LINCX objects, we believe it is a straightforward extension of [Schack-Nielsen and Schürmann, 2010]. Finally, we require a notion of coverage of LINCX objects which is a straightforward extension of [Pientka and Abel, 2015].

7.4 Mechanization of Lincx

We have mechanized[2] key properties of our underlying theory in the proof assistant BELUGA. In particular, we encoded the syntax, typing rules of LINCX together with single and simultaneous hereditary substitution operations in the logical framework LF relying on HOAS encodings to model binding. Our encoding is similar to [Martens and Crary, 2012] of LF in LF, but we also handle meta-variables and simultaneous substitutions. Since BELUGA only intrinsically supports intuitionistic binding structures and contexts, linearity must be enforced separately. We do this through an explicit context of variable declarations, connecting each variable to a flag and a type. To model contexts with context variable indices we use a binary encoding. The implementation of LINCX in BELUGA was crucial to arrive at our understanding of modelling context variables using commutative monoids.

As mentioned in Section 7.3.2, the context variable indices take context splitting into account by describing elements from a countably infinite set $\mathcal{I}$, along with a neutral element and a join operation that is commutative and associative. We implement these indices using binary strings, where ϵ is the empty string, and a string with a single positive bit

[2]LINCX mechanization: `https://github.com/Beluga-lang/Beluga/tree/master/examples/lincx_mechanization`

represents a leaf-level variable. In other words, through this abstraction, every context variable in Δ is a binary string with a single positive bit. [Schack-Nielsen, 2011] uses a similar encoding for managing flags for linear, unrestricted, and unavailable assumptions in concrete contexts. Our encoding lifts these ideas to modelling context variables. We then implement the $\bowtie$ operation as a binary OR operation which fails when the two strings have a common positive (for instance a join between 001 and 011 would fail). The following describes the join of M and N, forming K:

```
LF bin_or : bin → bin → bin → type =
| bin_or_nil_l : bin_or nil M M
| bin_or_nil_r : bin_or M nil M
| bin_or_l     : bin_or M N K
                → bin_or (cons one M) (cons zero N) (cons one K)
| bin_or_r     : bin_or M N K
                → bin_or (cons zero M) (cons one N) (cons one K)
| bin_or_zero  : bin_or M N K
                → bin_or (cons zero M) (cons zero N) (cons zero K);
```

We then proceed to prove commutativity, associativity and uniqueness of bin_or. Finally, we mechanized the proofs of the properties about our equational theory of context joins as total functions in Beluga. In particular, we mechanized proofs of Lemmas 7.1 and 7.2. Here we take advantage of Beluga's first-class contexts and in the base cases rely on the commutativity and associativity properties of the binary encoding of context variable indices. We note that context equality is entirely syntactic and can thus be defined simply in terms of reflection.

Although we had to model our mixed contexts of unrestricted and linear assumptions explicitly, Beluga's support for encoding formal systems using higher-order abstract syntax still significantly simplified our definitions of typing rules and hereditary substitution operation. In particular, it allowed us to elegantly model variable bindings in abstractions and Π-types.

Inductive properties about typing and substitution are implemented as recursive functions in Beluga. Many of the proofs in this paper become fairly tedious and complex on paper and mechanizing Lincx therefore helps us build trust in our foundation. Given the substantial amount of time and lines of code we devote to model contexts and context joins, our mechanization also demonstrates the value Lincx can bring to mechanizing linear systems or more generally systems that work with resources.

7.5 Related Work

The idea of using logical framework methodology to build a specification language for linear logic dates back two decades, beginning with linear logical framework LLF [Cervesato and Pfenning, 1996] providing $\multimap$, & and $\top$ operators from intuitionistic linear logic, the maximal set of connectives for which unique canonical forms exist. The idea was later expanded to the concurrent logical framework CLF [Watkins et al., 2002], which uses a monad to encapsulate less well-behaved operators. The quest to design meta-logics that allow us to reason about linear logical frameworks has been marred with difficulties in the past.

In proof theory, [McDowell, 1997, McDowell and Miller, 2002] and later [Gacek et al., 2012] propose a two-level approach to reason about formal systems where we rely on a first-order sequent calculus together with inductive definitions and induction on natural numbers as a meta-reasoning language. We encode our formal system in a specification logic that is then embedded in the first-order sequent calculus, the reasoning language. The design of the two-level approach is in principle modular and in fact [McDowell, 1997] describes a linear specification logic. However the context of assumptions is encoded as a list explicitly in this approach. As a consequence, we need to reason modulo the equational properties of context joins and we may need to prove properties about the uniqueness of assumptions. Such bureaucratic reasoning then still pollutes our main proof.

In type theory, [McCreight and Schürmann, 2004] give a tailored meta-logic $\mathcal{L}_\omega^+$ for linear LF, which is an extension of the meta-logic for LF [Schürmann, 2000]. While $\mathcal{L}_\omega^+$ also characterize partial linear derivations using contextual objects that depend on a linear context, the approach does not define an equational theory on contexts and context variables. It also does not support reasoning about contextual objects modulo such an equational theory. In addition $\mathcal{L}_\omega^+$ does not cleanly separate the meta-theoretic (co)inductive reasoning about linear derivations from specifying and modelling the linear derivations themselves. We believe the modular design of Beluga, i.e. the clean separation of representing and modelling specifications and derivations on one hand and reasoning about such derivations on the other, offers many advantages. In particular, it is more robust and also supports extensions to (co)inductive definitions [Cave and Pientka, 2012, Thibodeau et al., 2016].

The hybrid logical framework HLF [Reed, 2009] is in principle capable to support reasoning about linear specifications. In HLF, we reason about objects that are valid at a specific world, instead of objects that are valid within a context. However, contexts and worlds seem closely connected. Most recently [Bock and Schürmann, 2015] propose a contextual logical framework XLF. Similarly to Lincx, it is also based on contextual modal type theory with first-class contexts. However, context variables have a strong nominal flavor in their system. In particular, Bock and Schürmann allow multiple context variables in the context and each context variable is associated with a list of variable names (and other context variable domains) from which it must be disjoint – otherwise the system is prone to repetition of linear variables upon instantiation.

On a more fundamental level the difference between HLF and XLF on the one hand and our approach on the other is how we think about encoding meta-theoretic proofs. HLF and XLF follow the philosophy of Twelf system and encoding proofs as relations. This makes it sometimes challenging to establish that a given relation constitutes an inductive proof and hence both systems have been rarely used to establish such meta-theoretic proofs. More importantly, the proof-theoretic strength of this approach is limited. For example, it is challenging to encode formal systems and proofs that rely on (co)inductive definitions such as proofs by logical relations and bisimulation proofs within the logical framework itself. We believe the modular design of separating cleanly between Lincx as a specification framework and embedding Lincx into the proof and programming language Beluga provides a simpler foundation for representing the meta-theory of linear systems. Intuitively, meta-proofs about linear systems only rely on linearity to model the linear derivations – however the reasoning about these linear derivation trees is not linear, but remains intuitionistic.

7.6 Conclusion and Future Work

We have presented Lincx, a linear contextual modal logical framework with first-class contexts as a foundation to model linear systems and derivations. In particular, Lincx satisfies the necessary requirements to serve as a specification and index language for Beluga and hence provides a suitable foundation for implementing proofs about (linear) derivation trees as recursive functions. We have also mechanized the key equational properties of context joins in Beluga. This further increases our confidence in our development.

There is a number of research questions that naturally arise and we plan to pursue in the future. First, we plan to extend Lincx with additional linear connectives such as $\top$ and $A\&B$. These additional connectives are for example present in [Cervesato and Pfenning, 1996]. We omitted them here to concentrate on modelling context joins and their equational theory, but we believe it is straightforward to add them.

Dealing with first-class contexts in the presence of linear operators outside the set $\{\multimap, \&, \top\}$ is more challenging, as they may break canonicity. We plan to follow the approach in CLF [Watkins et al., 2002] enclosing them into a monad to control their behaviour. Having also additive operators would allow us to for example model the meta-theory of session type systems [Caires and Pfenning, 2010] and reason about concurrent computation. Further we plan to add first-class substitution variables [Cave and Pientka, 2013] to Lincx. This woud allow us to abstractly describe relations between context. This seems particularly important as we allow richer schemas definitions that model structured sequences.

Last but not least, we would like to implement Lincx as a specification language for Beluga to enable reasoning about linear specifications in practice.

7.A Appendix

We present here partial proofs and generalized reformulations of lemmas and theorems mentioned in Section 7.3 of this paper.

Lemma 7.2. *If* $\Delta; \Psi \vdash \sigma \Leftarrow \Phi$ *then* $\Delta; \overline{\Psi} \vdash \sigma \Leftarrow \overline{\Phi}$.

Proof. Proof by induction on typing derivation $\mathcal{D} :: \Delta; \Psi \vdash \sigma \Leftarrow \Phi$. We show a couple of cases below, the remaining ones are straightforward.

Case $\mathcal{D} = \dfrac{\mathsf{unr}(\Gamma)}{\Delta; \psi_m, \Gamma \vdash \mathsf{id}_{\psi_m} \Leftarrow \psi_m}$

$\mathsf{unr}(\Gamma)$ by assumption

$\Delta; \psi_\epsilon, \Gamma \vdash \mathsf{id}_{\psi_m} \Leftarrow \psi_\epsilon$ by substitution typing

Case $\mathcal{D} = \dfrac{\Delta;\Psi_1 \vdash \sigma \Leftarrow \Phi \qquad \Delta;\Psi_2 \vdash M \Leftarrow [\sigma]_{\overline{\Phi}}A \qquad \Psi = \Psi_1 \bowtie \Psi_2}{\Delta;\Psi \vdash \sigma, M \Leftarrow \Phi, x \hat{:} A}$

$\Delta;\overline{\Psi_1} \vdash \sigma \Leftarrow \overline{\Phi}$ by IH
$\overline{\Psi} = \overline{\Psi_1} = \overline{\Psi_2}$ by Lemma 7.1(iv)
$\Delta;\overline{\Psi} \vdash \sigma, M \Leftarrow \overline{\Phi}, x \check{:} A$ by substitution typing
□

7.A.1 Hereditary single substitution

Hereditary single substitution in Lincx closely follows [Cave and Pientka, 2013]. We present complete rules in Figure 7.12.

The following theorem establishes typing for single substitutions. Notice that it is more general compared to the variant presented in Section 7.3.4.

Theorem 7.2 (Hereditary single substitution property)**.**

(i) If $\Delta;\overline{\Psi} \vdash M \Leftarrow A$ *and* $\Delta;\Psi, x{:}A, \Phi \vdash J$ *then* $\Delta;\Psi, [M/x]_A\Phi \vdash [M/x]^*_A(J)$ *where* $* \in \{c, s, l\}$.

(ii) If $\Delta;\Psi_1 \vdash M \Leftarrow A$, $\Delta;\Psi_2, x \hat{:} A, \Phi \vdash J$ *and* $\Psi = \Psi_1 \bowtie \Psi_2$ *then* $\Delta;\Psi, \Phi \vdash [M/x]^*_A(J)$ *where* $* \in \{c, s, l\}$.

(iii) If $\Delta;\Psi_1 \vdash M \Leftarrow A$, $\Delta;\Psi_2 \vdash S > A \Rightarrow B$, $\Psi = \Psi_1 \bowtie \Psi_2$ *and* $\mathsf{reduce}(M : A^-, S) = M'$ *then* $\Delta;\Psi \vdash M' \Leftarrow B$

In order to obtain a similar result for simultaneous substitution, we first need to show a number of properties.

Lemma 7.3. *If* $\Phi, \Gamma = \Phi_1, \Gamma_1 \bowtie \Phi_2, \Gamma_2$ *and* $\Psi \vdash \sigma \Leftarrow \Phi$, *then* $\Phi, [\sigma]^{\cdot}_{\Phi}\Gamma = \Phi_1, [\sigma]^{\cdot}_{\Phi_1}\Gamma_1 \bowtie \Phi_2, [\sigma]^{\cdot}_{\Phi_2}\Gamma_2$

Lemma 7.4. *Suppose* $\Psi = \Psi_1 \bowtie \Psi_2$, *then* $\Delta;\Phi \vdash \sigma \Leftarrow \Psi$ *iff* $\Delta;\Phi_1 \vdash \sigma \Leftarrow \Psi_1$ *and* $\Delta;\Phi_2 \vdash \sigma \Leftarrow \Psi_2$ *for* $\Phi = \Phi_1 \bowtie \Phi_2$.

Lemma 7.5. $[\sigma]^{\widetilde{\Gamma_1}, \widetilde{\Gamma_2}}_{\Psi} J = [\sigma]^{\widetilde{\Gamma_1}, \check{x}, \widetilde{\Gamma_2}}_{\Psi} J$

Proof. By induction on the simultaneous substitution.
□

$$\boxed{[M/x]^c_\alpha N = N'}$$

$$\begin{array}{lll}
[M/x]^c_\alpha(\lambda y.N) & = \lambda y.N' & \text{where } [M/x]^c_\alpha N = N', \\
 & & \text{choosing } y \neq x, y \notin \mathsf{FV}(M) \\
[M/x]^c_\alpha(\widehat{\lambda} y.N) & = \widehat{\lambda} y.N' & \text{where } [M/x]^c_\alpha N = N', \\
 & & \text{choosing } y \neq x, y \notin \mathsf{FV}(M) \\
[M/x]^c_\alpha(u[\sigma]) & = u[\sigma'] & \text{where } [M/x]^s_\alpha \sigma = \sigma' \\
[M/x]^c_\alpha(c \cdot S) & = c \cdot S' & \text{where } [M/x]^l_\alpha S = S' \\
[M/x]^c_\alpha(x \cdot S) & = N & \text{where } [M/x]^l_\alpha S = S' \text{ ,} \\
 & & \text{and } \mathsf{reduce}(M : \alpha, S') = N \\
[M/x]^c_\alpha(y \cdot S) & = y \cdot S' & \text{where } [M/x]^l_\alpha S = S' \text{ and } x \neq y \\
[M/x]^c_\alpha(p[\sigma] \cdot S) & = p[\sigma'] \cdot S' & \text{where } [M/x]^s_\alpha \sigma = \sigma' \text{ and } [M/x]^l_\alpha S = S'
\end{array}$$

$$\boxed{[M/x]^l_\alpha S = S'}$$

$$\begin{array}{lll}
[M/x]^l_\alpha(\epsilon) & = \epsilon & \\
[M/x]^l_\alpha(N \,;\, S) & = N' \,;\, S' & \text{where } [M/x]^c_\alpha N = N' \text{ and } [M/x]^l_\alpha S = S' \\
[M/x]^l_\alpha(N \,\hat{;}\, S) & = N' \,\hat{;}\, S' & \text{where } [M/x]^c_\alpha N = N' \text{ and } [M/x]^l_\alpha S = S'
\end{array}$$

$$\boxed{[M/x]^s_\alpha \sigma = \sigma'}$$

$$\begin{array}{lll}
[M/x]^s_\alpha(\cdot) & = \cdot & \\
[M/x]^s_\alpha(\mathsf{id}_\psi) & = \mathsf{id}_\psi & \\
[M/x]^s_\alpha(\sigma, N) & = \sigma', N' & \text{where } [M/x]^s_\alpha \sigma = \sigma' \text{ and } [M/x]^c_\alpha N = N'
\end{array}$$

Figure 7.12: Hereditary single substitution

Again, the theorem is more general compared to the variant presented in Section 7.3.4.

Theorem 7.3 (Simultaneous substitution property)**.** *If* $\Delta;\Psi,\Gamma \vdash J$ *and* $\Delta;\Psi' \vdash \sigma \Leftarrow \Psi$ *then* $\Delta;\Psi',[\sigma]^{\cdot}_{\overline{\Psi}}\Gamma \vdash [\sigma]^{\widetilde{\Gamma}}_{\overline{\Psi}}(J)$.

Proof. Proof by induction on the typing derivation $\mathcal{D} :: \Delta;\Psi,\Gamma \vdash J$. We show some representative cases.

Case $\mathcal{D} = \dfrac{\Delta;\Psi,\Gamma,x\hat{:}A \vdash M \Leftarrow B}{\Delta;\Psi,\Gamma \vdash \widehat{\lambda}x.M \Leftarrow A \multimap B}$

$\Delta;\Psi,[\sigma]^{\cdot}_{\overline{\Psi}}(\Gamma,x\hat{:}A) \vdash [\sigma]^{\widetilde{\Gamma},\hat{x}}_{\overline{\Psi}}M \Leftarrow [\sigma]^{\widetilde{\overline{\overline{\Gamma}}},\check{x}}_{\overline{\Psi}}B$ by IH

$\Delta;\Psi,[\sigma]^{\cdot}_{\overline{\Psi}}\Gamma,x\hat{:}[\sigma]^{\Gamma}_{\overline{\Psi}}A) \vdash [\sigma]^{\widetilde{\Gamma},\hat{x}}_{\overline{\Psi}}M \Leftarrow [\sigma]^{\widetilde{\overline{\overline{\Gamma}}},\check{x}}_{\overline{\Psi}}B$ by definition of substitution

$\Delta;\Psi,[\sigma]^{\cdot}_{\overline{\Psi}}\Gamma[\sigma]^{\cdot}_{\overline{\Psi}}A) \vdash \widehat{\lambda}x.[\sigma]^{\widetilde{\Gamma}}_{\overline{\Psi}}M \Leftarrow [\sigma]^{\cdot}_{\overline{\Psi}}A \multimap [\sigma]^{\widetilde{\overline{\overline{\Gamma}}},\check{x}}_{\overline{\Psi}}B$ by typing rule

$\Delta;\Psi,[\sigma]^{\cdot}_{\overline{\Psi}}\Gamma[\sigma]^{\cdot}_{\overline{\Psi}}A) \vdash \widehat{\lambda}x.[\sigma]^{\widetilde{\Gamma}}_{\overline{\Psi}}M \Leftarrow [\sigma]^{\cdot}_{\overline{\Psi}}A \multimap [\sigma]^{\widetilde{\overline{\overline{\Gamma}}}}_{\overline{\Psi}}B$ By Lemma 7.5

$\Delta;\Psi,[\sigma]^{\cdot}_{\overline{\Psi}}\Gamma \vdash [\sigma]^{\widetilde{\Gamma}}_{\overline{\Psi}}\,\widehat{\lambda}\,x.M \Leftarrow [\sigma]^{\widetilde{\Gamma}}_{\overline{\Psi}}(A \multimap B)$ by definition of substitution

Case $\mathcal{D} = \dfrac{\Delta;\Psi_1 \vdash M \Leftarrow A \qquad \Delta;\Psi_2 \vdash S > B \Rightarrow P \quad \Phi,\Gamma = \Psi_1 \bowtie \Psi_2}{\Delta;\Phi,\Gamma \vdash M \mathbin{\hat{;}} S > A \multimap B \Rightarrow P}$

$\Psi_1 = \Phi_1,\Gamma_1$
$\Psi_2 = \Phi_2,\Gamma_2$
$\Phi = \Phi_1 \bowtie \Phi_2$
$\Gamma = \Gamma_1 \bowtie \Gamma_2$ by Lemma 7.1
$\Delta;\Psi_1' \vdash \sigma \Leftarrow \Psi_1$
$\Delta;\Psi_2' \vdash \sigma \Leftarrow \Psi_2$
$\Psi' = \Psi_1' \bowtie \Psi_2'$ by Lemma 7.4

$\Delta;\Phi_1,[\sigma]^{\cdot}_{\overline{\Phi_1}}\Gamma_1 \vdash [\sigma]^{\widetilde{\Gamma_1}}_{\Phi_1}M \Leftarrow [\sigma]^{\widetilde{\overline{\overline{\Gamma_1}}}}_{\overline{\Phi_1}}A$

$\Delta;\Phi_2,[\sigma]^{\cdot}_{\overline{\Phi_2}}\Gamma_2 \vdash [\sigma]^{\widetilde{\Gamma_2}}_{\Phi_2}S > [\sigma]^{\widetilde{\overline{\overline{\Gamma_2}}}}_{\overline{\Phi_2}}B \Rightarrow [\sigma]^{\widetilde{\overline{\overline{\Gamma_2}}}}_{\overline{\Phi_2}}P$ by IH

$\Phi,[\sigma]_{\overline{\Phi}}\Gamma = \Phi_1,[\sigma]_{\overline{\Phi_1}}\Gamma_1 \bowtie \Phi_2,[\sigma]_{\overline{\Phi_2}}\Gamma_2$

$\Delta;\Phi,[\sigma]_{\overline{\Phi}}\Gamma \vdash [\sigma]^{\widetilde{\Gamma_1}}_{\Phi_1}M \mathbin{\hat{;}} [\sigma]^{\widetilde{\Gamma_2}}_{\Phi_2}S > [\sigma]^{\widetilde{\overline{\overline{\Gamma_1}}}}_{\overline{\Phi_1}}A \multimap [\sigma]^{\widetilde{\overline{\overline{\Gamma_2}}}}_{\overline{\Phi_2}}B \Rightarrow [\sigma]^{\widetilde{\overline{\overline{\Gamma_2}}}}_{\overline{\Phi_2}}P$ By typing rule

$\Delta;\Phi,[\sigma]_{\overline{\Phi}}\Gamma \vdash [\sigma]^{\widetilde{\Gamma_1}}_{\Phi_1}M \mathbin{\hat{;}} [\sigma]^{\widetilde{\Gamma_2}}_{\Phi_2}S > [\sigma]^{\widetilde{\overline{\overline{\Gamma}}}}_{\overline{\Phi}}A \multimap [\sigma]^{\widetilde{\overline{\overline{\Gamma}}}}_{\overline{\Phi}}B \Rightarrow [\sigma]^{\widetilde{\overline{\overline{\Gamma}}}}_{\overline{\Phi}}P$ By Lemma 7.1

$\Delta;\Phi,[\sigma]_{\overline{\Phi}}\Gamma \vdash [\sigma]^{\widetilde{\Gamma_1}}_{\Phi_1}M \mathbin{\hat{;}} S > [\sigma]^{\widetilde{\overline{\overline{\Gamma}}}}_{\overline{\Phi}}A \multimap [\sigma]^{\widetilde{\overline{\overline{\Gamma}}}}_{\overline{\Phi}}B \Rightarrow [\sigma]^{\widetilde{\overline{\overline{\Gamma}}}}_{\overline{\Phi}}P$ By substitution

□

$\boxed{\Delta \vdash \Psi \Leftarrow G}$ Context Ψ checks against schema G

$$\frac{}{\Delta \vdash \cdot \Leftarrow G} \qquad \frac{\psi_i : G \in \Delta}{\Delta \vdash \psi_\epsilon \Leftarrow G} \qquad \frac{\psi_i : G \in \Delta}{\Delta \vdash \psi_i \Leftarrow G}$$

$$\frac{\Delta \vdash \psi_k \Leftarrow G \quad \Delta \vdash \psi_l \Leftarrow G \quad m = k \bowtie l}{\Delta \vdash \psi_m \Leftarrow G}$$

$$\frac{\Delta \vdash \Psi \Leftarrow G \quad \lambda(\overrightarrow{x_i{:}A_i}).B \in G \quad \Delta;\Psi \vdash \sigma \Leftarrow (\overrightarrow{x_i{:}A_i}) \quad \Delta;\overline{\Psi} \vdash A = [\sigma]_{(\overrightarrow{x_i{:}A_i})} B}{\Delta \vdash \Psi, x{:}A \Leftarrow G}$$

$$\frac{\Delta \vdash \Psi \Leftarrow G \quad \lambda(\overrightarrow{x_i{:}A_i}).\widehat{B} \in G \quad \Delta;\Psi \vdash \sigma \Leftarrow (\overrightarrow{x_i{:}A_i}) \quad \Delta;\overline{\Psi} \vdash A = [\sigma]_{(\overrightarrow{x_i{:}A_i})} B}{\Delta \vdash \Psi, x\hat{:}A \Leftarrow G}$$

$$\frac{\Delta \vdash \Psi \Leftarrow G \quad \lambda(\overrightarrow{x_i{:}A_i}).\widehat{B} \in G \quad \Delta;\Psi \vdash \sigma \Leftarrow (\overrightarrow{x_i{:}A_i}) \quad \Delta;\overline{\Psi} \vdash A = [\sigma]_{(\overrightarrow{x_i{:}A_i})} B}{\Delta \vdash \Psi, x\check{:}A \Leftarrow G}$$

Figure 7.13: Typing rules for contexts of a given schema

7.A.2 Typing for Meta-Terms

Rules for constructing a context of a given schema presented in Figure 7.13 describe four possible initial cases of context construction, which correspond to four cases of constructing a valid context. Note that since their instantiations should contain the same variables (albeit with some variables becoming unavailable), all the context variables should have the same schema. Next, we have three cases for context extension, i.e.: extending a context with either an unrestricted, a linear or an unavailable variable. In either case, we must ensure that the schema as an element of the proper type, and the proper status, for the context variable, which is to say, an unrestricted variable should expects an unrestricted element, while a linear or unavailable variable expects a linear schema element.

Typing of other meta-terms, as presented in Figure 7.14 is straightforward: a meta-object $\widetilde{\Psi}.R$ has type $(\Psi \vdash P)$, if R has type P in the context Ψ. The typing of variable objects, i.e. an object of type $(\Psi \vdash \#A)$, must be reconsidered carefully. A parameter type is inhabited only by variable objects, i.e. either concrete variables from Ψ or parameter variable associated with a variable substitution. The typing for parameter variables

$\boxed{\Delta \vdash C \Leftarrow U}$ Meta-level term C checks against type U

$$\frac{\Delta; \Psi \vdash R \Leftarrow P}{\Delta \vdash \widetilde{\Psi}.R \Leftarrow (\Psi \vdash P)}$$

$$\frac{x{:}A \in \Psi \quad \mathsf{unr}(\Psi)}{\Delta \vdash \widetilde{\Psi}.x \Leftarrow (\Psi \vdash \#A)} \qquad \frac{\mathsf{unr}(\Psi_1) \quad \mathsf{unr}(\Psi_2)}{\Delta \vdash (\widetilde{\Psi_1}, \widehat{x}, \widetilde{\Psi_2}).x \Leftarrow (\Psi_1, x\hat{:}A, \Psi_2 \vdash \#A)}$$

$$\frac{p : (\Phi \vdash \#A) \in \Delta \quad \Delta; \Psi \vdash \pi \Leftarrow \Phi \quad \Delta; \overline{\Psi} \vdash B = [\pi]_{\overline{\Phi}}(A)}{\Delta \vdash \widetilde{\Psi}.p[\pi] \Leftarrow (\Psi \vdash \#B)}$$

Figure 7.14: Typing rules for meta-terms

$\boxed{\mathsf{id}(\Psi)}$ Identity substitution

$$\begin{aligned}
\mathsf{id}(\cdot) &= \cdot \\
\mathsf{id}(\mathsf{id}_\psi) &= \mathsf{id}_\psi \\
\mathsf{id}(\Psi, x{:}A) &= \mathsf{id}(\Psi), \eta\text{-}\mathsf{exp}_{(A^-)}(x, \epsilon) \\
\mathsf{id}(\Psi, x\hat{:}A) &= \mathsf{id}(\Psi), \eta\text{-}\mathsf{exp}_{(A^-)}(x, \epsilon) \\
\mathsf{id}(\Psi, x\check{:}A) &= \mathsf{id}(\Psi), \eta\text{-}\mathsf{exp}_{(A^-)}(x, \epsilon)
\end{aligned}$$

$\boxed{\eta\text{-}\mathsf{exp}_A(H, S)}$ η-expansion

$$\begin{aligned}
\eta\text{-}\mathsf{exp}_a(H, S) &= H \cdot S \\
\eta\text{-}\mathsf{exp}_{\alpha \to \beta}(H, S) &= \lambda x.\eta\text{-}\mathsf{exp}_\beta(H, S @ \eta\text{-}\mathsf{exp}_\alpha(x)) \\
\eta\text{-}\mathsf{exp}_{\alpha \multimap \beta}(H, S) &= \widehat{\lambda} x.\eta\text{-}\mathsf{exp}_\beta(H, S \widehat{@} \eta\text{-}\mathsf{exp}_\alpha(x))
\end{aligned}$$

Figure 7.15: Simultaneous meta-substitution, auxiliary definitions

follows the typing for meta-objects. There are two cases to consider when we have a concrete variable x from the context: either Ψ contains only unrestricted variable declarations and $x : A$ is one of them; or x is in fact a linear variable of type A which forces Ψ to be a context with only one linear declaration $x\hat{:}A$.

$\boxed{[\![\Theta]\!]_\Delta M}$ Simultaneous meta-substitution for terms

$$\begin{array}{lll}
[\![\Theta]\!]_\Delta(\lambda x.M) & = \lambda x.M' & \text{where } [\![\Theta]\!]_\Delta M = M' \\
[\![\Theta]\!]_\Delta(\widehat{\lambda} x.M) & = \widehat{\lambda} x.M' & \text{where } [\![\Theta]\!]_\Delta M = M' \\
[\![\Theta]\!]_\Delta(u[\sigma]) & = R' & \text{where } \Theta_\Delta(u) = \widetilde{\Psi}.R : (\Psi \vdash P) \\
 & & \text{and } [\![\Theta]\!]_\Delta\, \sigma = \sigma' \\
 & & \text{and } [\sigma']_\Psi R = R' \\
[\![\Theta]\!]_\Delta(c \cdot S) & = c \cdot S' & \text{where } [\![\Theta]\!]_\Delta\, S = S' \\
[\![\Theta]\!]_\Delta(x \cdot S) & = x \cdot S' & \text{where } [\![\Theta]\!]_\Delta\, S = S' \\
[\![\Theta]\!]_\Delta(p[\sigma] \cdot S) & = M' & \text{where } \Theta_\Delta(p) = \widetilde{\Psi}.x : (\Psi \vdash \#A) \\
 & & \text{and } [\![\Theta]\!]_\Delta\, \sigma = \sigma' \\
 & & \text{and } [\![\Theta]\!]_\Delta\, S = S' \text{ and } \sigma'_\Psi(x) = M : \alpha \\
 & & \text{and } \mathsf{reduce}(M : \alpha, S') = M' \\
[\![\Theta]\!]_\Delta(p[\sigma] \cdot S) & = q[\tau'] \cdot S' & \text{where } \Theta_\Delta(p) = \widetilde{\Psi}.q[\pi] : (\Psi \vdash \#A) \\
 & & \text{and } [\![\Theta]\!]_\Delta\, \sigma = \sigma' \\
 & & \text{and } [\![\Theta]\!]_\Delta\, S = S' \text{ and } [\sigma']\pi = \tau
\end{array}$$

$\boxed{[\![\Theta]\!]_\Delta \sigma}$ Simultaneous meta-substitution for substitutions

$$\begin{array}{lll}
[\![\Theta]\!]_\Delta(\cdot) & = \cdot & \\
[\![\Theta]\!]_\Delta(\mathsf{id}_\psi) & = \mathsf{id}(\Psi) & \text{where } \Theta_\Delta(\psi_\epsilon) = \Psi \\
[\![\Theta]\!]_\Delta(\sigma, M) & = \sigma', M' & \text{where } [\![\Theta]\!]_\Delta\, \sigma = \sigma' \text{ and } [\![\Theta]\!]_\Delta\, M = M'
\end{array}$$

Figure 7.16: Simultaneous meta-substitution

Postlude

In this thesis we have looked at extending the power of LF-based systems in two different dimensions, proposing two new frameworks. The first one, HyLF, focuses on adding more expressive power when formalising encodings. The other, Lincx, expands the realm of the kinds of programs and properties we can express in an encoding.

Neither of these topics can be considered a closed research question at this point. The *Future Work* section of the Lincx paper remains relevant, as the results presented here are fairly recent. With HyLF, we are currently interested in a more systematic approach to building this framework. Concretely, we are looking into making it a more *hybrid* framework, one that adheres to the "worlds as types" principle, while also removing the – slightly unusually behaving – $\downarrow\alpha.A$ type. Adding a *reachable world* type, $\Diamond p$ for some world p, is also under investigation, although geometric theories known from modal logic are not quite as well-behaved in the context of logical frameworks as we were originally hoping for. A separate, and equally interesting, question is that of the modularity of the HyLF approach: how can we systematically combine subsystems using hybrid worlds with different algebras in a way that leads to a well-behaved, consistent end-framework?

Also interestingly, both HyLF and Lincx are based, at least to some extent, on modal logic: HyLF uses hybrid operators and modal worlds known from the Kripke semantics of modal logic, and contextual modal type theory and judgmental reconstruction of modal logic serve as a basis for Lincx and its predecessors. With that in mind, attempting to build a contextual hybrid logical framework seems like a natural next step. Although no work has yet been done on this front, it remains an idea for a HyLF implementation.

Bibliography

[Andreoli, 1992] Andreoli, J. (1992). Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347.

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

[Barendregt and Hemerik, 1990] Barendregt, H. and Hemerik, K. (1990). Types in lambda calculi and programming languages. In Jones, N. D., editor, *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, volume 432 of *Lecture Notes in Computer Science*, pages 1–35. Springer.

[Belanger et al., 2013] Belanger, O. S., Monnier, S., and Pientka, B. (2013). Programming type-safe transformations using higher-order abstract syntax. In Gonthier, G. and Norrish, M., editors, *3rd International Conference on Certified Programs and Proofs (CPP'13)*, Lecture Notes in Computer Science (LNCS 8307), pages 243–258. Springer.

[Bengtson et al., 2012] Bengtson, J., Jensen, J. B., and Birkedal, L. (2012). Charge! - A framework for higher-order separation logic in Coq. In Beringer, L. and Felty, A. P., editors, *Third International Conference on Interactive Theorem Proving (ITP'12)*, Lecture Notes in Computer Science (LNCS 7406), pages 315–331. Springer.

[Berdine et al., 2002] Berdine, J., O'Hearn, P. W., Reddy, U. S., and Thielecke, H. (2002). Linear continuation-passing. *Higher-Order and Symbolic Computation*, 15(2-3):181–208.

[Bock and Schürmann, 2015] Bock, P. B. and Schürmann, C. (2015). A contextual logical framework. In *20th International Conference on Logic* *for Programming, Artificial Intelligence and Reasoning (LPAR'15)*, Lecture Notes in Computer Science (LNCS 9450), pages 402–417. Springer.

[Caires and Pfenning, 2010] Caires, L. and Pfenning, F. (2010). Session types as intuitionistic linear propositions. In Gastin, P. and Laroussinie, F., editors, *21th International Conference on Concurrency Theory (CONCUR'10)*, Lecture Notes in Computer Science (LNCS 6269), pages 222–236. Springer.

[Cave and Pientka, 2012] Cave, A. and Pientka, B. (2012). Programming with binders and indexed data-types. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM.

[Cave and Pientka, 2013] Cave, A. and Pientka, B. (2013). First-class substitutions in contextual type theory. In *8th ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, pages 15–24. ACM.

[Cave and Pientka, 2015] Cave, A. and Pientka, B. (2015). A case study on logical relations using contextual types. In Cervesato, I. and K.Chaudhuri, editors, *10th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'15)*, pages 18–33. Electronic Proceedings in Theoretical Computer Science (EPTCS).

[Cervesato et al., 2000] Cervesato, I., Hodas, J. S., and Pfenning, F. (2000). Efficient resource management for linear logic proof search. *Theor. Comput. Sci.*, 232(1-2):133–163.

[Cervesato and Pfenning, 1996] Cervesato, I. and Pfenning, F. (1996). A linear logical framework. In Clarke, E., editor, *11th Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey. IEEE Press.

[Cervesato and Pfenning, 2003] Cervesato, I. and Pfenning, F. (2003). A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688.

[Chaudhuri and Despeyroux, 2014] Chaudhuri, K. and Despeyroux, J. (2014). A hybrid linear logic for constrained transition systems. In Matthes, R. and Schubert, A., editors, *TYPES'13 Post-proceedings*, to appear in LIPIcs.

[Coquand and Huet, 1988] Coquand, T. and Huet, G. (1988). The calculus of constructions. *Information and Computation*, 76(2):95 – 120.

[Danvy and Filinski, 1992] Danvy, O. and Filinski, A. (1992). Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391.

[De Bruijn, 1980] De Bruijn, N. G. (1980). A survey of the project automath. *To H.B. Curry : Essays on combinatory logic, lambda calculus and formalism*, pages 579–606.

[Fluet et al., 2006] Fluet, M., Morrisett, G., and Ahmed, A. J. (2006). Linear regions are all you need. In Sestoft, P., editor, *15th European Symposium on Programming (ESOP'06)*, Lecture Notes in Computer Science (LNCS 3924), pages 7–21. Springer.

[Gacek et al., 2012] Gacek, A., Miller, D., and Nadathur, G. (2012). A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning*, 49(2):241–273.

[Galmiche and Salhi, 2011] Galmiche, D. and Salhi, Y. (2011). Sequent calculi and decidability for intuitionistic hybrid logic. *Journal of Information and Computation*, 209(12):1447–1463.

[Girard, 1987] Girard, J.-Y. (1987). Linear logic. *Theor. Comput. Sci.*, 50(1):1–102.

[Harper et al., 1993] Harper, R., Honsell, F., and Plotkin, G. (1993). A framework for defining logics. *Journal of the ACM*, 40(1):143–184.

[Harper and Licata, 2007] Harper, R. and Licata, D. (2007). Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673.

[Lellmann, 2015] Lellmann, B. (2015). Linear nested sequents, 2-sequents and hypersequents. In *Proceedings of the 24th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods - Volume 9323*, TABLEAUX 2015, pages 135–150, New York, NY, USA. Springer-Verlag New York, Inc.

[Lellmann et al., 2017] Lellmann, B., Olarte, C., and Pimentel, E. (2017). A uniform framework for substructural logics with modalities. In Eiter, T. and Sands, D., editors, *LPAR-21. 21st International Conference on Logic* *for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 435–455. EasyChair.

[Martens and Crary, 2012] Martens, C. and Crary, K. (2012). LF in LF: Mechanizing the metatheories of LF in Twelf. In *7th International Workshop on Logical Frameworks and Meta-languages:Theory and Practice (LFMTP'12)*, pages 23–32. ACM.

[McCreight, 2009] McCreight, A. (2009). Practical tactics for separation logic. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, Lecture Notes in Computer Science (LNCS 5674), pages 343–358. Springer.

[McCreight and Schürmann, 2004] McCreight, A. and Schürmann, C. (2004). A meta-linear logical framework. In *4th International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*.

[McDowell, 1997] McDowell, R. (1997). *Reasoning in a Logic with Definitions and Induction*. PhD thesis, University of Pennsylvania.

[McDowell and Miller, 2002] McDowell, R. C. and Miller, D. A. (2002). Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136.

[Montesi, 2013] Montesi, F. (2013). *Choreographic Programming*. PhD thesis.

[Murphy VII et al., 2004] Murphy VII, T., Crary, K., Harper, R., and Pfenning, F. (2004). A symmetric modal lambda calculus for distributed computing. In *LICS'04*, pages 286–295.

[Nanevski et al., 2008] Nanevski, A., Pfenning, F., and Pientka, B. (2008). Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49.

[Nigam and Miller, 2009] Nigam, V. and Miller, D. (2009). Algorithmic specifications in linear logic with subexponentials. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 129–140, New York, NY, USA. ACM.

[Pientka, 2008] Pientka, B. (2008). A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM.

[Pientka and Abel, 2015] Pientka, B. and Abel, A. (2015). Structural recursion over contextual objects. In Altenkirch, T., editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, pages 273–287. Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl.

[Pientka and Cave, 2015] Pientka, B. and Cave, A. (2015). Inductive Beluga:Programming Proofs (System Description). In Felty, A. P. and Middeldorp, A., editors, *25th International Conference on Automated Deduction (CADE-25)*, Lecture Notes in Computer Science (LNCS 9195), pages 272–281. Springer.

[Pientka and Dunfield, 2010] Pientka, B. and Dunfield, J. (2010). Beluga: a framework for programming and reasoning with deductive systems (System Description). In Giesl, J. and Haehnle, R., editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer.

[Prior, 1967] Prior, A. (1967). *Past, Present and Future*. Oxford University Press.

[Reed, 2008] Reed, J. (2008). Base-type polymorphism in lf.

[Reed, 2009] Reed, J. (2009). *A hybrid logical framework*. PhD thesis, Carnegie Mellon.

[Schack-Nielsen, 2011] Schack-Nielsen, A. (2011). *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen.

[Schack-Nielsen and Schürmann, 2008] Schack-Nielsen, A. and Schürmann, C. (2008). Celf – A logical framework for deductive and concurrent systems (system description). In Armando, A., Baumgartner, P., and Dowek, G., editors, *4th International Joint Conference on Automated Reasoning (IJCAR'08)*, Lecture Notes in Computer Science (LNCS 5195), pages 320–326.

[Schack-Nielsen and Schürmann, 2010] Schack-Nielsen, A. and Schürmann, C. (2010). Pattern unification for the lambda calculus with linear and affine types. In Crary, K. and Miculan, M., editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'10)*, volume 34 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 101–116.

[Schürmann, 2000] Schürmann, C. (2000). *Automating the Meta Theory of Deductive Systems.* PhD thesis, Department of Computer Science, Carnegie Mellon University. CMU-CS-00-146.

[Schürmann and Pfenning, 2003] Schürmann, C. and Pfenning, F. (2003). A coverage checking algorithm for LF. In Basin, D. A. and Wolff, B., editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, volume 2758 of *Lecture Notes in Computer Science*, pages 120–135. Springer.

[Simpson, 1994] Simpson, A. (1994). *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis.

[Thibodeau et al., 2016] Thibodeau, D., Cave, A., and Pientka, B. (2016). Indexed codata. In Garrigue, J., Keller, G., and Sumii, E., editors, *21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*, pages 351–363. ACM.

[Tzakova, 1999] Tzakova, M. (1999). Tableau calculi for hybrid logics. In Murray, N., editor, *TABLEAUX'99*, volume 1617 of *LNCS*, pages 278–292.

[Wadler, 2015] Wadler, P. (2015). Propositions as types. *Commun. ACM*, 58(12):75–84.

[Walker and Watkins, 2001] Walker, D. and Watkins, K. (2001). On regions and linear types. In Pierce, B. C., editor, *6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 181–192. ACM.

[Watkins et al., 2002] Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. (2002). A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University.

[Watkins et al., 2004] Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. (2004). A concurrent logical framework I: The propositional fragment. In Berardi, S., Coppo, M., and Damiani, F., editors, *TYPES'03 Post-proceedings*, volume 3085 of *LNCS*, pages 355–377.