Refinement Types for Logical Frameworks

William Lovas

CMU-CS-10-138
September 2010

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

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

This research was sponsored by the National Science Foundation under grant CCR-0121633, by the Fundação para a Ciência e Tecnologia (FCT), Portugal, under a grant from the Information and Communications Technology Institute (ICTI) at Carnegie Mellon University, and by generous support from Microsoft Corporation. The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

Keywords: refinement types, logical frameworks, subtyping, intersection types, dependent types

For laughter, for tears, for madness, for fears.

Abstract

The logical framework LF and its metalogic Twelf can be used to encode and reason about a wide variety of logics, languages, and other deductive systems in a formal, machine-checkable way.

Recent studies have shown that ML-like languages can profitably be extended with a notion of subtyping called *refinement types*. A refinement type discipline uses an extra layer of term classification above the usual type system to more accurately capture certain properties of terms.

I propose that adding refinement types to LF is both useful and practical. To support the claim, I exhibit an extension of LF with refinement types called LFR, work out important details of its metatheory, delineate a practical algorithm for refinement type reconstruction, and present several case studies that highlight the utility of refinement types for formalized mathematics. In the end I find that refinement types and LF are a match made in heaven: refinements enable many rich new modes of expression, and the simplicity of LF ensures that they come at a modest cost.

Acknowledgments

Thanks to everybody.

Perhaps that bears further refinement. But everybody comes first, since I can't possibly fit everyone worth thanking onto one page, and even if I could, I would surely miss a few despite my best efforts to the contrary.

What goes into a thesis? Having given the matter some thought, I think it comes down to the 4 S's: support, space, sustenance, and a spark.

First, support. Thanks to my advisor, Frank Pfenning, for constantly believing I had what it takes, even when I wasn't sure. Thanks to my committee—Karl Crary, Bob Harper, Adriana Compagnoni, and Carsten Schürmann—for sharing their valuable and limited time and attention, not to mention for the formative impact their guidance has had on my research. Thanks to Jake Donham for encouraging me to stick it out, even when he moved on to other endeavors himself. Thanks to Dan Licata for his assistance reconstructing all of the "known" but unpublished results about canonical forms. Thanks to Jason Reed for his uncanny ability to help me out of a research rut. Thanks to Chris Martens for teaching me how to wake up in the morning. Thanks to so many of my colleagues for so many inspiring conversations about so many mathematical curiosities over so many tasty beverages. Thanks to Tom Murphy VII, officemate and role model. Thanks to Rob Simmons, purveyor of boundless enthusiasm. Thanks to Dan Spoonhower for apt advice on writing a thesis. Thanks to Donna Malayeri for apt advice on writing the acknowledgments. Thanks to Benjamin Pierce for first showing me that subtyping was cool. Thanks to my parents for always being there for me, constantly providing the kind of encouraging praise that can only come from parents without sounding insincere.

Next, space. One occasionally requires a change of scenery, and I owe a debt of gratitude to a number of NYC cafes that served as my "office away from the office" on occasion, including Society Coffee in Harlem, the Roasting Plant in the West Village, and Kaffe 1668 in TriBeCa. Honorable mentions go to Dominican Joe in Austin, where I produced most of the original material that became the first three chapters, and Jitters right here in Pittsburgh for a chill environment close to home. Major thanks are in order as well to the New York Public Library, Battery Park City Branch, which provided an excellent and reliable workspace for a critical week in which in which much of the content of the later chapters was written. Go libraries!

Sustenance. I would like to personally thank all of the following foods and food groups for their invaluable contributions to my health and well-being during the most crucial binge period of thesis-writing: fruit, nuts, cheese, peanut butter, hummus, edamame, coffee, and tea. Oh, and tomatoes! Eaten as if they were apples!

But back to the coffee and tea for a moment, which brings me to my last point: a spark. A spark of inspiration, a spark of creativity... a spark of liquid sleep after a night without the real thing. And thus I thank coffee, possibly the greatest beverage known to man. Particular thanks are in order for Tazza D'Oro's CMU/GHC branch for providing some of the tastiest cups I've ever had, served by the friendliest and most uplifting staff I could imagine—many a work day was rescued by their verve.

Mostly, though, thanks to everybody.

Contents

List of Figures

Chapter 1

Introduction

Thesis. *Refinement types are a useful and practical extension to the LF logical framework.*

The logical framework LF [HHP93] and its metalogic Twelf [PS99] can be used to encode and reason about a wide variety of logics, languages, and other deductive systems in a formal, machine-checkable way. Deductive systems are represented using the *judgments as types* principle: the deductive system's judgments are represented as LF types, and derivations of evident judgments are represented as LF terms of those types.

Recent studies have shown that ML-like languages can profitably be extended with a notion of subtyping called *refinement types* [Fre94, Dav05, Dun07]. A refinement type discipline uses an extra layer of term classification above the usual type system to more accurately capture certain properties of terms. Refinements, also known as *sorts*, are usually only checked after ordinary typechecking succeeds, resulting in a system that admits no more terms as well-typed but one that classifies more precisely the terms that *do* typecheck—including the possibility of ruling some out as *ill-sorted*.

In this work, I show that such a refinement type system can also profitably be added to LF. Under LF's "judgments as types" representation methodology, refinement types represent more precise judgments than those represented by ordinary types, with subtyping between refinements acting as a kind of judgmental inclusion. Things previously tricky to encode in LF, like syntactic subset relations between terms of an object language, become trivial to encode with the availability of refinement types. Furthermore, refinement types come at a modest cost: they do not overly complicate the metatheory of LF, and practical algorithms exist for things like sort checking and reconstruction.

To demonstrate that adding refinement types to LF is both useful and practical, I exhibit an extension of LF with refinement types called LFR, work out important details of its metatheory, and present several example signatures that highlight the utility of refinement types for formalized mathematics. In the remainder of this chapter, I give some further background, describe my approach in more detail, and outline a roadmap for the rest of the dissertation.

1.1 Background

LF is a dependent type theory that was created as a framework for defining logics [HHP93]. Early examples of logics whose syntax and proof theory could elegantly be represented included first-order and higher-order arithmetic. LF was also recognized early on as an excellent representation language for programming language calculi [MP91], largely due to its support for higher-order abstract syntax, a technique inspired by Church [Chu40] in which object-language variable binding is encoded using the framework's variable binding.

It was later realized that the same framework was also suitable for specifying the metatheoretic properties of encoded deductive systems and proofs of those properties [Pfe00, Pfe92] (see Pfenning's *Handbook of Automated Reasoning* article [Pfe01b] for an overview), and furthermore that these proofs could be mechanically checked by verifying that they represented total relations. These ideas culminated in the implementation of the Twelf metalogical framework [PS99], which has been used in recent years to mechanize the metatheory of programming languages that are prohibitively complex to reason about on paper [Cra03, LCH07].

The key principle in LF's methodology of representation is the *judgments as types* principle: judgments about syntactic entities are represented as dependent type families, and derivations of evident judgments are simply inhabitants of those type families. The syntax of a deductive system can be thought of as a series of trivial, nullary judgments (i.e., "there is a natural number", "an expression exists"), and thus represented as a collection of simple types populated by constructors representing the various syntactic productions.

For example, suppose we wanted to model the natural numbers in LF. In ordinary mathematical discourse, we might define them using a grammar:

$$n ::= \mathsf{Z} \mid \mathsf{S}[n] \qquad\qquad \text{natural numbers}$$

In LF, we introduce a type for the natural numbers and populate it with constructors for zero and successor:

$nat : \mathsf{type}.$
$z : nat.$
$s : nat \rightarrow nat.$

Suppose we wanted to define a judgment representing addition, $n_1 + n_2 = n_3$. In LF, we would model it with a three-place type family:

$plus : nat \rightarrow nat \rightarrow nat \rightarrow \mathsf{type}.$

We would then give rules inhabiting $plus\ N_1\ N_2\ N_3$ for various values of N_1, N_2, and N_3.

Since judgments are represented by type families in the LF type theory, it is natural to consider what other forms of judgments one might be able to naturally encode with a richer type structure. For example, the linear logical framework LLF [CP96, Cer96, CP02] provides elegant representations of ephemeral, stateful judgments as linear hypotheses, and the concurrent logical framework CLF [WCPW02, WCPW04] introduces a monad to represent concurrent computations. In this work, we consider the implications of augmenting LF with a notion of subtyping and intersection types known as *refinement types*.

Refinement type systems are so called because they introduce an extra layer of classification above the normal "type" layer, a "refinement" layer. Each refinement type, or *sort*, refines an ordinary type, and we only consider the question of whether or not a term has a particular sort S if we have already established that it has the type that S refines. This *refinement restriction* lets us populate the refinement layer with a variety of interesting type-theoretic machinery, like subtyping and intersection types, without their complexity infecting the metatheory of the system as a whole: the type layer remains untouched. Furthermore, the refinement restriction ensures that even the sort layer is amenable to analysis by imposing on it the structure of the type layer: it is impossible, for example, to form pathological intersections like $P \wedge (P \rightarrow P)$ for some base type P.

This is not the first work to consider the idea of augmenting a dependently-typed logical framework with a form of subtyping. Even as early as deBruijn's Automath [dB94b], the historical progenitor to LF, people considered the idea of adding "inheritance" to a proof assistant to avoid repetition and redundancy in the formalization of mathematical ideas: Knuth had one such idea [personal communication], and deBruijn's MV [dB94a] was based around a notion of subtyping. More recent works have focused more closely on the interactions that arise when subtyping is added to a dependent type theory.

Aspinall and Compagnoni [AC01] studied a type theory $\lambda\mathrm{P}_{\leq}$ with both dependent types and subtyping, with much the same motivation as mine, but they treated subtyping directly rather than introducing a refinement layer. Their chief difficulty was breaking the cycle that arises between subtyping, kinding, and typing in order to show decidability. Aspinall [Asp00] has also studied an unconventional system of subtyping with dependent types using "power types", a type-theoretic analogue of power sets. Aspinall's system λ_{Power} has uniform "subtyping" at all levels since power "types" can in fact classify type families; although the system remains predicative, this generalization complicates the system's metatheory. Both $\lambda\mathrm{P}_{\leq}$ and λ_{Power} treat subtyping directly rather than introducing a refinement layer, but neither treats intersection types at all.

More directly related, Pfenning [Pfe93] attempted to extend LF with refinement types in a manner superficially similar to what I consider in this work. The essential novelty of this work comes from its use of the modern canonical forms-based approach to LF, which enables considerably further development than the earlier work. The early stages of this work can be seen as a reconstruction and reformulation of Pfenning's ideas, with a focus on canonical forms, decidability, and good proof-theoretic properties. The later stages are entirely novel, and were in large part made feasible due to the early simplifications that came from using the modern technology of canonical forms and hereditary substitutions.

The modern canonical forms-based approach to the development of logical frameworks is due to Watkins, who first used it in the development of CLF [WCPW02]. Canonical forms are β-normal and η-long, and in a canonical forms-based system, they are the only terms that are admitted as well-formed. Since standard substitution might introduce redexes even when substituting a normal term into a normal term, it is replaced with a notion of *hereditary substitution* that contracts redexes along the way, yielding another normal term. Since only canonical forms are admitted, type equality is just α-equivalence, and typechecking is manifestly decidable.

Canonical forms are exactly the terms one cares about when adequately encoding a language in a logical framework, so this approach loses no expressivity. Since all terms are normal, there is no notion of reduction, and thus the metatheory need not directly treat properties related to reduction, such as subject reduction, Church-Rosser, or strong normalization. All of the metatheoretic arguments can be carried out as elementary structural inductions.

The canonical forms methodology is similar to the method of specifying a type theory via a *typed operational semantics* [Gog94, Gog95], a methodology which has been used to tame the metatheory of several type theories, including a calculus of higher-order subtyping [CG03]. Both methodologies are based around the idea of capturing normalization via an induction over types, an idea central Pfenning's structural proofs of cut elimination [Pfe00] and to Joachimski and Matthes's "short " normalization proofs [JM03].

1.2 Properties as Sorts

If the methodology of LF is "judgments as types", what is the methodology of LFR? To draw out the answer, we continue with our example from above, the natural numbers. An interesting property of the natural numbers is that they have a notion of parity: some natural numbers are even and some are odd. We can capture parity succinctly with a grammar:

$e ::= \mathsf{Z} \mid \mathsf{S}[o]$	even numbers
$o ::= \mathsf{S}[e]$	odd numbers

To formally model the property of a number being even or odd, we can introduce two *refinements* of the natural numbers, one for evens and one for odds, and we can give the constructors z and s sorts to describe their properties.

$$\begin{array}{l} \mathit{even} \sqsubset \mathit{nat}. \\ \mathit{odd} \sqsubset \mathit{nat}. \\ z :: \mathit{even}. \\ s :: \mathit{even} \to \mathit{odd} \ \wedge\ \mathit{odd} \to \mathit{even}. \end{array}$$

Thus the methodology of LFR is *properties as sorts*: interesting properties of formal objects can be represented intrinsically as refinements of the types of those objects.

Recall that the judgment of addition is represented in LF by a three-place type family, *plus* $N_1\ N_2\ N_3$. Using refinements, we can also express the parity properties of addition: two even numbers add up to an even number, an even number and an odd number add up to an odd number, and so on. We can use a notation like the following to capture these properties:

$$
\begin{aligned}
&\mathit{plus} :: \mathit{even} \to \mathit{even} \to \mathit{even} \to \mathsf{type}\\
&\quad \wedge\ \mathit{even} \to \mathit{odd} \to \mathit{odd} \to \mathsf{type}\\
&\quad \wedge\ \mathit{odd} \to \mathit{even} \to \mathit{odd} \to \mathsf{type}\\
&\quad \wedge\ \mathit{odd} \to \mathit{odd} \to \mathit{even} \to \mathsf{type}.
\end{aligned}
$$

We can also capture properties of derivations using refinements, like for instance the property of not using a particular rule or set of rules. In this way we can isolate cut-free derivations in a sequent calculus, or model a logic whose rules are a subset of those of another logic. In fact, since derivations are just LF terms of certain types, we can capture complex properties involving being constructed in a particular way, just like we did above with the natural numbers, but now at the level of derivations.

For instance, given an encoding of the implicational fragment of natural deduction using a "true" judgment,

$$
\begin{aligned}
&\mathit{true} : o \to \mathsf{type}.\\
&{\supset}I\ : (\mathit{true}\ A \to \mathit{true}\ B) \to \mathit{true}\ (A \supset B).\\
&{\supset}E : \mathit{true}\ (A \supset B) \to \mathit{true}\ A \to \mathit{true}\ B.
\end{aligned}
$$

we can pick out normal natural deductions as a refinement using a well-known technique of pushing all of the introduction rules to the root of the derivation and all of the eliminations towards the leaves with the help of the auxiliary notion of a *neutral* natural deduction. The fact that neutral deductions are included in normal deductions is modeled by a *subsorting declaration*, $\mathit{neutral} \leq \mathit{normal}$.

$$
\begin{aligned}
&\mathit{normal} \sqsubset \mathit{true}.\\
&\mathit{neutral} \sqsubset \mathit{true}.\\
&\\
&\mathit{neutral} \leq \mathit{normal}.\\
&\\
&{\supset}I\ :: (\mathit{neutral}\ A \to \mathit{normal}\ B) \to \mathit{normal}\ (A \supset B).\\
&{\supset}E :: \mathit{neutral}\ (A \supset B) \to \mathit{normal}\ A \to \mathit{neutral}\ B.
\end{aligned}
$$

We will return to both of these examples in more depth below in Chapter 2. As this short preview demonstrates, refinements can succinctly model a variety of interesting properties of syntactic elements, judgment forms, and derivations of evident judgments.

1.3 Contributions

To demonstrate the thesis that refinement types for LF are both useful and practical, I make a number of concrete contributions.

First, I present the type theoretic core of LF with refinement types in Chapter 2. The presentation is guided by several small motivating examples, and along the way, I refine and explore the LFR representation methodology of *properties as sorts*. While the examples begin to suggest the utility of refinement types, they primarily serve to motivate the design of the type theory, which has its foundations in canonical forms, hereditary substitution, and bidirectional typing. An interesting feature which emerges from the canonical forms-based presentation is that the subsorting judgment is only needed at base sorts.

The system I present has all of the important metatheoretic properties we demand of a logical framework, like identity and substitution principles witnessing the reflexivity and transitivity of entailment, which I prove in Chapter 3. Using these properties, we can show that although subsorting is only defined at base sorts, the usual expected principles of subsorting at higher sorts are derived as inclusions of canonical forms, thereby yielding the happy situation of being permitted to use higher-sorted reasoning principles without having to account for them directly in the metatheory. All of the metatheoretic arguments are elementary inductions, suggesting that refinement types are a quite practical addition to LF.

After exploring the metatheory of LFR, I take a slight diversion in Chapter 4 to explore its meaning by showing how to soundly and completely translate it into LF with proof irrelevance, a translation known as the *subset interpretation*: sorts turn into predicates, and sorting derivations turn into proofs of those predicates. Although sound and complete, the translation turns out to be quite complicated, further demonstrating the practicality of refinement types through the balance between their expressive power and their complexity.

No one could seriously argue for the practicality of a logical framework without showing that it admits some form of type reconstruction. Encoding deductive systems entirely in the core type theory of LF would involve too much burdensome redundancy to offer the practitioner any real benefit, but type reconstruction alleviates much of the burden, making LF a practical tool for representing one's ideas. Similarly, encoding deductive systems entirely in the core type theory of LFR is too burdensome to offer any benefit; in fact, the problem is further exacerbated by the apparently greater expressivity offered by refinements: a user may be forced to write down many redundant copies of rules if a judgment can take on many forms.

So to really argue for the practicality of LFR, I outline in Chapter 5 an algorithm for *sort reconstruction*: given a signature in a concrete syntax similar to that of Twelf, my algorithm computes a most general signature in the fully explicit core type theory of LFR. Although this problem turns out to be decidable in principle, it is in general computationally infeasible, so the main thrust of the work is delineating an algorithm that is incomplete by design, but that works well on the kinds of examples we expect to arise in practice.

Then, to bolster the claim of utility, I embark on a number of larger case studies in Chapter 6, showing how the framework, methodology, and algorithms I envision would work in practice. For each case study, I compare the LFR code to the LF code one would have to write to replicate its functionality. In all cases, I show that the LFR encoding is significantly simpler than the corresponding LF encoding, demonstrating that the extensions offered by LFR are actually useful in practice. The case studies represent a realistic sampling of representational challenges that have come up either in my own work or in the work of my colleagues.

I hope to convince the reader by the end of this dissertation that my extensions to LF are both useful and practical, demonstrating my thesis. But as with any large undertaking, the work does not end here, so in concluding in Chapter 7, I suggest a number of directions for future work, including a stronger focus on porting the metalogical features of Twelf to the refinement technology of LFR. Someday in the near future, I hope to realize these ideas as a practical extension of Twelf, one that allows more concise and expressive representations of deductive systems and proofs of their properties.

Chapter 2

System and Examples

We present our system of LF with Refinements, LFR, through several examples. In what follows, R refers to atomic terms and N to normal terms. Our atomic and normal terms are exactly the terms from canonical presentations of LF.

$$\begin{array}{rll} R & ::= c \mid x \mid R\,N & \text{atomic terms} \\ N, M & ::= R \mid \lambda x.\, N & \text{normal terms} \end{array}$$

In this style of presentation, typing is defined bidirectionally by two judgments: $R \Rightarrow A$, which says atomic term R *synthesizes* type A, and $N \Leftarrow A$, which says normal term N *checks* against type A. Since λ-abstractions are always checked against a given type, they need not be decorated with their domain types.

Types are similarly stratified into atomic and normal types.

$$\begin{array}{rll} P & ::= a \mid P\,N & \text{atomic type families} \\ A, B & ::= P \mid \Pi x{:}A.\, B & \text{normal type families} \end{array}$$

The operation of hereditary substitution, written $[N/x]_A$, is a partial function which computes the normal form of the standard capture-avoiding substitution of N for x. It is indexed by the putative type of x, A, to ensure termination, but neither the variable x nor the substituted term N are required to bear any relation to this type index for the operation to be defined. Later, in Chapter 3, we will show that when N and x *do* have type A, hereditary substitution is a total function on well-formed terms.

As a philosophical aside, we note that restricting our attention to normal terms in this way is similar to the idea of restricting one's attention to cut-free proofs in a sequent calculus [Pfe00]. Showing that hereditary substitution can always compute a canonical form is analogous to showing the cut rule admissible. And just as cut admissibility may be used to prove a *cut elimination* theorem, hereditary substitution may be used to prove a *normalization* theorem relating the canonical approach to traditional formulations. We will not explore the relationship any further in the present work: the canonical terms are the only ones we care about when formalizing deductive systems in a logical framework, so we simply take the canonical presentation as primary.

Our layer of refinements uses metavariables Q for atomic sorts and S for normal sorts. These mirror the definition of types above, except for the addition of intersection and "top" sorts.

$$\begin{array}{rcll} Q & ::= & s \mid Q\,N & \text{atomic sort families} \\ S, T & ::= & Q \mid \Pi x{::}S{\sqsubset}A.\,T \mid \top \mid S_1 \wedge S_2 & \text{normal sort families} \end{array}$$

Sorts are related to types by a refinement relation, $S \sqsubset A$ ("S refines A"), discussed below. We only sort-check well-typed terms, and a term of type A can be assigned a sort S only when $S \sqsubset A$. These constraints are collectively referred to as the "refinement restriction". We occasionally omit the "$\sqsubset A$" from function sorts when it is clear from context.

We will make reference to the refinement restriction frequently, as it is a cornerstone of our approach and often enables dramatic simplifications. It is useful to keep in mind that one reason the refinement restriction works is that our language of terms (normal and atomic) remains exactly the same as that of "unrefined" LF. This reuse is made possible by the decision not to label abstractions $\lambda x.\,N$, neither by types nor by sorts, a simplification which is in turn enabled by the bidirectional typing discipline that comes with the canonical forms approach.

Recall that deductive systems are encoded in LF using the *judgments as types* principle [HHP93, HL07]: syntactic categories are represented by simple types, and judgments over syntax are represented by dependent type families. Derivations of judgments are inhabitants of those type families, and well-formed derivations correspond to well-typed LF terms. An LF signature is a collection of kinding declarations $a : K$ and typing declarations $c : A$ that establishes a set of syntactic categories, a set of judgments, and inhabitants of both. In LFR, we can represent syntactic subsets or sets of derivations that have certain *properties* using sorts. Thus we say that the methodology of LFR is *properties as sorts*.

2.1 Example: Natural Numbers

For the first running example we will return to the natural numbers in unary notation. Recall that in LF, they would be specified as follows:

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

These declarations establish a syntactic category of natural numbers populated by two constructors, a constant constructor representing zero and a unary constructor representing the successor function.

Suppose we would like to distinguish the odd and the even numbers as refinements of the type of all numbers.

$even \sqsubset nat.$
$odd \sqsubset nat.$

The form of the declaration is $s \sqsubset a$ where a is a type family already declared and s is a new sort family. Sorts headed by s are declared in this way to refine types headed by a. The relation $S \sqsubset A$ is extended through the whole sort hierarchy in a compositional way.

Next we declare the sorts of the constructors. For zero, this is easy:

$z :: even.$

The general form of this declaration is $c :: S$, where c is a constant already declared in the form $c : A$, and where $S \sqsubset A$. The declaration for the successor is slightly more interesting, because the successor of an even number is odd and the successor of an odd number is even. In order to capture both properties simultaneously we need to use an *intersection sort*, written as $S_1 \wedge S_2$.[1]

$s :: even \to odd \ \wedge \ odd \to even.$

In order for an intersection to be well-formed, both components must refine the same type. The nullary intersection $\top$ can refine any type, and represents the maximal refinement of that type. The following are suggestive rules, but we will give more precise ones below.[2]

$$\frac{s \sqsubset a \in \Sigma}{s\, N_1 \ldots N_k \sqsubset a\, N_1 \ldots N_k} \qquad \frac{S \sqsubset A \qquad T \sqsubset B}{\Pi x{::}S.\, T \sqsubset \Pi x{:}A.\, B} \qquad \frac{S_1 \sqsubset A \qquad S_2 \sqsubset A}{S_1 \wedge S_2 \sqsubset A} \qquad \frac{}{\top \sqsubset A}$$

To show that the declaration for s is well-formed, we establish that $even \to odd \wedge odd \to even \ \sqsubset \ nat \to nat$.

The *refinement relation* $S \sqsubset A$ should not be confused with the usual *subtyping relation*. Although each is a kind of subset relation,[3] they are quite different: Subtyping relates two types, is contravariant in the domains of function types, and is transitive, while refinement relates a sort to a type, so it does not make sense to consider its variance or whether it is transitive. We will discuss subtyping below and in Section 3.4.

Now suppose that we also wish to distinguish the strictly positive natural numbers. We can do this by introducing a sort pos refining nat and declaring that the successor function yields a pos when applied to anything, using the maximal sort.

$pos \sqsubset nat.$
$s :: \ldots \wedge \top \to pos.$

[1]Intersection has lower precedence than arrow.

[2]As usual, we write $A \to B$ as shorthand for the dependent type $\Pi x{:}A.\, B$ when x does not occur in B.

[3]It may help to recall the interpretation of $S \sqsubset A$: for a term to be judged to have sort S, it must already have been judged to have type A for some A such that $S \sqsubset A$. Thus, the refinement relation represents an inclusion "by fiat": every term with sort S is also a term of type A, by invariant. By contrast, subsorting $S_1 \leq S_2$ is a more standard sort of inclusion: every term with sort S_1 is also a term of sort S_2, by subsumption (see Section 3.4).

Since we only sort-check well-typed programs and s is declared to have type $nat \to nat$, the sort $\top$ here acts as a sort-level reflection of the entire nat type.

We can specify that all odd numbers are positive by declaring odd to be a subsort of pos.

$$odd \leq pos.$$

As it happens in this example, we can see that any closed instance of odd is pos even without the declaration, but if there were other constructors that returned sort odd, their applications would now also have sort pos.[4]

Putting it all together, we have the following:

$$\begin{array}{l} even \sqsubset nat. \\ odd \sqsubset nat. \\ pos \sqsubset nat. \\ odd \leq pos. \\ z :: even. \\ s :: even \to odd \;\wedge\; odd \to even \;\wedge\; \top \to pos. \end{array}$$

Now we should be able to verify that, for example, $s\ (s\ z) \Leftarrow even$. To explain how, we analogize with pure canonical LF. Recall that atomic types have the form $a\ N_1 \ldots N_k$ for a type family a and are denoted by P. Arbitrary types A are either atomic (P) or (dependent) function types ($\Pi x{:}A.\, B$). Canonical terms are then characterized by the rules shown in the left column of Figure 2.1.

There are two typing judgments, $N \Leftarrow A$ which means that N checks against A (both given) and $R \Rightarrow A$ which means that R synthesizes type A (R given as input, A produced as output). Both take place in a context Γ assigning types to variables and an implicit ambient signature Σ containing declarations for constants. To force terms to be η-long, the rule for checking an atomic term R only checks it at an atomic type P. It does so by synthesizing a type P' and comparing it to the given type P. In canonical LF, all types are already canonical, so this comparison is just α-equality.

On the right-hand side we have shown the corresponding rules for sorts. First, note that the format of the context Γ is slightly different, because it declares sorts for variables, not just types. The rules for functions and applications are straightforward analogues to the rules in ordinary LF. The rule **switch** for checking atomic terms R at atomic sorts Q replaces the equality check with a subsorting check and is the only place where we appeal to subsorting (defined below). For applications, we use the type A that refines the type S as the index parameter of the hereditary substitution.

Subsorting is exceedingly simple: it only needs to be defined on atomic sorts, and is just the reflexive and transitive closure of the declared subsorting relationship extended through applications to identical arguments.

$$\frac{s_1 {\leq} s_2 \in \Sigma}{s_1 \leq s_2} \qquad \frac{Q_1 \leq Q_2}{Q_1\ N \leq Q_2\ N} \qquad \frac{}{Q \leq Q} \qquad \frac{Q_1 \leq Q' \qquad Q' \leq Q_2}{Q_1 \leq Q_2}$$

[4]The declaration would also be necessary to establish that variables of sort odd are pos, if we were to consider "open" natural numbers.

Canonical LF	LF with Refinements
$\dfrac{\Gamma, x{:}A \vdash N \Leftarrow B}{\Gamma \vdash \lambda x.\, N \Leftarrow \Pi x{:}A.\, B}$	$\dfrac{\Gamma, x{::}S{\sqsubset}A \vdash N \Leftarrow T}{\Gamma \vdash \lambda x.\, N \Leftarrow \Pi x{::}S{\sqsubset}A.\, T}\ (\Pi\textbf{-I})$
$\dfrac{\Gamma \vdash R \Rightarrow P' \qquad P' = P}{\Gamma \vdash R \Leftarrow P}$	$\dfrac{\Gamma \vdash R \Rightarrow Q' \qquad Q' \leq Q}{\Gamma \vdash R \Leftarrow Q}\ (\textbf{switch})$
$\dfrac{x{:}A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \dfrac{c{:}A \in \Sigma}{\Gamma \vdash c \Rightarrow A}$	$\dfrac{x{::}S{\sqsubset}A \in \Gamma}{\Gamma \vdash x \Rightarrow S}\ (\textbf{var}) \qquad \dfrac{c :: S \in \Sigma}{\Gamma \vdash c \Rightarrow S}\ (\textbf{const})$
$\dfrac{\Gamma \vdash R \Rightarrow \Pi x{:}A.\, B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash R\, N \Rightarrow [N/x]_A\, B}$	$\dfrac{\Gamma \vdash R \Rightarrow \Pi x{::}S{\sqsubset}A.\, T \qquad \Gamma \vdash N \Leftarrow S}{\Gamma \vdash R\, N \Rightarrow [N/x]_A\, T}\ (\Pi\textbf{-E})$

Figure 2.1: Typing in canonical LF, and sorting in LFR.

The sorting rules we've given thus far do not yet treat intersections. In line with the general bidirectional nature of the system, the introduction rules are part of the *checking* judgment, and the elimination rules are part of the *synthesis* judgment. Binary intersection $S_1 \wedge S_2$ has one introduction and two eliminations, while nullary intersection $\top$ has just one introduction.

$$\frac{\Gamma \vdash N \Leftarrow S_1 \qquad \Gamma \vdash N \Leftarrow S_2}{\Gamma \vdash N \Leftarrow S_1 \wedge S_2}\ (\wedge\textbf{-I}) \qquad\qquad \frac{}{\Gamma \vdash N \Leftarrow \top}\ (\top\textbf{-I})$$

$$\frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_1}\ (\wedge\textbf{-E}_1) \qquad\qquad \frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_2}\ (\wedge\textbf{-E}_2)$$

Note that although (canonical forms-style) LF type synthesis is unique, LFR sort synthesis is not, due to the intersection elimination rules. The checking rule $\top$**-I** may seem alarming at first glance, since it seems to assign a sort to *any* term, well-typed or not, but recall that under the refinement restriction, these rules can only be applied to terms that have already been type-checked. In this way, the refinement restriction justifies the intuition that $\top$ represents a "maximal refinement"—the sort of a term that satisfies no special properties beyond being well-typed.

Now we can see how these rules generate a deduction of $s\ (s\ z) \Leftarrow \mathit{even}$. The context is always empty and therefore omitted. To save space, we abbreviate *even* as e, *odd* as o, and *pos* as p, and we omit reflexive uses of subsorting.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash s \Rightarrow e \to o \wedge (o \to e \wedge \top \to p)}\;\textbf{(const)}}{\vdash s \Rightarrow o \to e \wedge \top \to p}\;(\wedge\textbf{-E}_2)}{\vdash s \Rightarrow o \to e}\;(\wedge\textbf{-E}_1) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash s \Rightarrow e \to o \wedge (\ldots)}\;\textbf{(const)}}{\vdash s \Rightarrow e \to o}\;(\wedge\textbf{-E}_1) \quad \dfrac{\dfrac{}{\vdash z \Rightarrow e}\;\textbf{(const)}}{\vdash z \Leftarrow e}\;\textbf{(switch)}}{\vdash s\, z \Rightarrow o}\;(\Pi\textbf{-E})}{\vdash s\, z \Leftarrow o}\;\textbf{(switch)}}{\dfrac{\vdash s\,(s\,z) \Rightarrow e}{\vdash s\,(s\,z) \Leftarrow e}\;\textbf{(switch)}}\;(\Pi\textbf{-E})$$

Using the $\wedge$**-I** rule, we can check that $s\;z$ is both odd and positive:

$$\dfrac{\begin{matrix}\vdots \\ \vdash s\,z \Leftarrow o\end{matrix} \quad \begin{matrix}\vdots \\ \vdash s\,z \Leftarrow p\end{matrix}}{\vdash s\,z \Leftarrow o \wedge p}\;(\wedge\textbf{-I})$$

Each remaining subgoal now proceeds similarly to the above example.

To illustrate the use of sorts with non-trivial dependent type *families*, consider the definition of the *double* relation in LF. We declare a type family representing the doubling judgment and populate it with two proof rules.

> $double : nat \to nat \to \mathsf{type}.$
> $dbl/z : double\; z\; z.$
> $dbl/s : \Pi X{:}nat.\, \Pi Y{:}nat.\, double\; X\; Y \to double\; (s\; X)\; (s\; (s\; Y)).$

With sorts, we can now directly express the property that the second argument to *double* must be even. But to do so, we require a notion analogous to *kinds* that may contain sort information. We call these *classes* and denote them by L.

$$\begin{array}{ll} K ::= \mathsf{type} \mid \Pi x{:}A.\, K & \text{kinds} \\ L ::= \mathsf{sort} \mid \Pi x{::}S{\sqsubset}A.\, L \mid \top \mid L_1 \wedge L_2 & \text{classes} \end{array}$$

Classes L mirror kinds K, and they have a refinement relation $L \sqsubset K$ similar to $S \sqsubset A$. (We elide the rules here, but they are included in Appendix A.) Now, the general form of the $s \sqsubset a$ declaration is $s \sqsubset a :: L$, where $a : K$ and $L \sqsubset K$; this declares sort constant s to refine type constant a and to have class L.

For now, we reuse the type name *double* as a sort, as no ambiguity can result. As before, we use $\top$ to represent a *nat* with no additional restrictions.

> $double \sqsubset double :: \top \to even \to \mathsf{sort}.$
> $dbl/z :: double\; z\; z.$
> $dbl/s :: \Pi X{::}\top.\, \Pi Y{::}even.\, double\; X\; Y \to double\; (s\; X)\; (s\; (s\; Y)).$

Since the sort family *double* has a non-trivial class, we see now that to check these declarations, we must sort check arguments to *double*, which means that the refinement

$$\boxed{\Gamma \vdash S \sqsubset A}$$

$$\frac{\Gamma \vdash Q \sqsubset P' \Rightarrow L \qquad P' = P \qquad L = \mathsf{sort}}{\Gamma \vdash Q \sqsubset P}\ (Q\text{-}\mathbf{F}) \qquad \frac{\Gamma \vdash S \sqsubset A \qquad \Gamma, x{::}S{\sqsubset}A \vdash S' \sqsubset A'}{\Gamma \vdash \Pi x{::}S{\sqsubset}A.\, S' \sqsubset \Pi x{:}A.\, A'}\ (\Pi\text{-}\mathbf{F})$$

$$\frac{}{\Gamma \vdash \top \sqsubset A}\ (\top\text{-}\mathbf{F}) \qquad \frac{\Gamma \vdash S_1 \sqsubset A \qquad \Gamma \vdash S_2 \sqsubset A}{\Gamma \vdash S_1 \wedge S_2 \sqsubset A}\ (\wedge\text{-}\mathbf{F})$$

$$\boxed{\Gamma \vdash Q \sqsubset P \Rightarrow L}$$

$$\frac{s{\sqsubset}a{::}L \in \Sigma}{\Gamma \vdash s \sqsubset a \Rightarrow L} \qquad \frac{\Gamma \vdash Q \sqsubset P \Rightarrow \Pi x{::}S{\sqsubset}A.\, L \qquad \Gamma \vdash N \Leftarrow S}{\Gamma \vdash Q\, N \sqsubset P\, N \Rightarrow [N/x]_A\, L}$$

$$\frac{\Gamma \vdash Q \sqsubset P \Rightarrow L_1 \wedge L_2}{\Gamma \vdash Q \sqsubset P \Rightarrow L_1} \qquad \frac{\Gamma \vdash Q \sqsubset P \Rightarrow L_1 \wedge L_2}{\Gamma \vdash Q \sqsubset P \Rightarrow L_2}$$

Figure 2.2: The refinement relation, well-formedness for sort families.

relation must be relative to a context, $\Gamma \vdash S \sqsubset A$. The new rules are shown in Figure 2.2, and have a bidirectional flavor similar to the sort checking rules we saw earlier.

After these declarations, it would be a static *sort error* to pose a query such as "?- *double* X (s (s (s z)))." before any search is ever attempted. In LF, queries like this could fail after a long search or even not terminate, depending on the search strategy. One of the important motivations for considering sorts for LF is to avoid uncontrolled search in favor of decidable static properties whenever possible.

The tradeoff for such precision is that now sort checking itself is non-deterministic and has to perform search because of the choice between the two intersection elimination rules. As Reynolds has shown, this non-determinism causes intersection type checking to be PSPACE-hard [Rey96], even for normal terms as we have here [Rey89]. Our experience though shows that many sort-checking problems that arise in the context of formalizing deductive systems can be solved efficiently based on a straightforward backtracking interpretation of the rules shown here.

2.2 Adequacy

One of the greatest advantages of the LF approach to formalized mathematics is that it was designed from the outset to allow for a notion of *adequacy* of an encoding. Generally speaking, an adequacy theorem establishes an isomorphism between the "informal" mathematial objects we write on paper and their corresponding formal entities in the logical framework. In the LF logical framework in particular, adequacy typically takes the form of a compositional bijection between informal objects and canonical forms of certain type families [HHP93].

For our natural numbers example, we have the following encoding function $\epsilon(-)$ along with its inverse $\delta(-)$:[5]

$$\begin{aligned} \epsilon(\mathsf{Z}) &= z & \qquad \delta(z) &= \mathsf{Z} \\ \epsilon(\mathsf{S}[n]) &= s\ \epsilon(n) & \qquad \delta(s\ N) &= \mathsf{S}[\delta(N)] \end{aligned}$$

Theorem 2.1 (LF Adequacy, natural numbers). *The encoding $\epsilon(-)$ is a bijection between natural numbers n and canonical forms of type nat in the empty context and the signature of natural numbers, and $\delta(-)$ is its inverse.*

Proof. By induction on n, and by induction on canonical forms of type *nat*. To recognize that $\delta(-)$ is a total function, we use inversion to show that in the signature of the natural numbers and the empty context, the only ways to form canonical objects of type *nat* are using z and s. □

Note that since our encoding of the natural numbers does not allow variables, compositionality need not be shown.

A happy consequence of the refinement approach is that such LF adequacy results can be taken off-the-shelf to serve as the basis of an LFR adequacy result. Assuming our refinement signature is based on an LF signature already known to be adequate with respect to some informal presentation, we immediately get for free an encoding function, its inverse decoding function, and the knowledge that the encoding composes with substitution, yielding a true isomorphism between informal objects and canonical forms of certain type families. To extend this to an adequacy result for the refinement portion of a signature, we follow our slogan, *properties as sorts*: we must show that properties of informal objects correspond to sorting derivations, and vice versa. In other words, we must show that the encoding preserves the properties of interest.

Recall that earlier we gave an informal specification of even and odd numbers with a grammar. The properties of being even or odd can equivalently be captured by a pair of mutually inductive judgments on natural numbers, n **even** and n **odd**:

$$\frac{}{\mathsf{Z}\ \textbf{even}} \qquad\qquad \frac{n\ \textbf{odd}}{\mathsf{S}[n]\ \textbf{even}} \qquad\qquad \frac{n\ \textbf{even}}{\mathsf{S}[n]\ \textbf{odd}}$$

Our encoding is an adequate representation of parity because there is a correspondence between the informal parity judgments and their formal characterization as sorts:

Theorem 2.2 (LFR Adequacy, even and odd numbers). *The encoding $\epsilon(-)$ preserves the properties of being even and odd:*

[5]Recent approaches based on canonical forms-only presentations of LF employ a clever technique where the bijection is given as a single relation which is shown to be functional in both directions [HL07], but for our purposes, it will be simpler to work directly with encoding and decoding functions.

1. *n **even** if and only if $\cdot \vdash \epsilon(n) \Leftarrow even$*

2. *n **odd** if and only if $\cdot \vdash \epsilon(n) \Leftarrow odd$*

Proof. ($\Longrightarrow$) By induction on the given property derivation. It is always straightforward to build the required sorting derivation.

($\Longleftarrow$) By induction on the given sorting derivation. We can use inversion principles specialized to the LFR signature of even and odd numbers to enumerate the ways we can derive $\cdot \vdash N \Leftarrow even$ or $\cdot \vdash N \Leftarrow odd$, and then show for each one that $\delta(N)$ **even** or $\delta(N)$ **odd**, respectively. □

Let us digress for a moment to compare our approach with the pure LF approach. The extrinsic characterization of parity through the judgments n **even** and n **odd** can easily be represented in LF without the use of refinements:

$even : nat \to \mathsf{type}.$
$odd : nat \to \mathsf{type}.$

$even/z : even\ z.$
$even/s : \Pi N{:}nat.\ odd\ N \to even\ (s\ N).$
$odd/s : \Pi N{:}nat.\ even\ N \to odd\ (s\ N).$

The code above is typical of the usual strategy for representing properties in LF: first, the properties are recast as judgments, and then those judgments are represented in LF using the "judgments as types" methodology. In fact, any property that can be captured using sorts can be captured using judgments and "judgments as types" (and a small measure of proof irrelevance, as we show in Chapter 4). The disadvantage of this extrinsic formulation is that now, whenever another judgment requires that a natural number be even, one must either pass in evidence of this fact or one must prove that such evidence always exists. Both approaches are cumbersome and awkward: the programmer must pay a high cost to use such rich representations.

The LFR methodology of "properties as sorts" avoids the indirection through judgments and the caveats that come along with it. Instead of the programmer bearing the burden of conjuring up evidence or proving its existence, the sort checker verifies automatically that representations are used in a manner consistent with their properties. Refinements enable direct, succinct, and intrinsic representations of many interesting properties.

We now return to the adequacy of our encoding. Having established the adequacy of the encodings of even and odd, we proceed similarly for the part of the signature describing positive natural numbers. Since that part of the signature involves a subsorting declaration, the adequacy argument is slightly more interesting. First, we give a judgment representing positivity:

$$\frac{}{\mathsf{S}[n]\ \textbf{pos}}$$

We could consider an additional rule stating that any odd number is positive, but such a rule is admissible.

Lemma 2.3. *If n* **odd***, then n* **pos***.*

Proof. By inversion, n must have the form $\mathsf{S}[n']$, and by rule, $\mathsf{S}[n']$ **pos**. □

Since we are considering a representation of natural numbers without any free variables (i.e., we make a "closed world" asumption), a similar metatheorem holds of our encoding, making the declaration that $odd \leq pos$ redundant. We proceed assuming the declaration anyway to illustrate how subsorting interacts with adequacy.

Theorem 2.4 (LFR Adequacy, positive numbers). *The encoding $\epsilon(-)$ preserves the property of being positive: n* **pos** *if and only if $\cdot \vdash \epsilon(n) \Leftarrow pos$.*

Proof. ($\Longrightarrow$) By inversion, n has the form $\mathsf{S}[n']$, and thus $\epsilon(n) = s\ \epsilon(n')$. We can easily build a derivation of $\cdot \vdash s\ \epsilon(n') \Leftarrow pos$ by noting that $\cdot \vdash \epsilon(n') \Leftarrow \top$.

($\Longleftarrow$) Suppose $\cdot \vdash N \Leftarrow pos$, and show that $\delta(N)$ **pos**. By inversion, $N = R$ such that either $\cdot \vdash R \Rightarrow odd$ or $\cdot \vdash R \Rightarrow pos$, since the only subsorts of pos are odd and pos itself. If $\cdot \vdash R \Rightarrow odd$, then $\cdot \vdash R \Leftarrow odd$, so by Theorem 2.2, $\delta(R)$ **odd** and by Lemma 2.3, $\delta(R)$ **pos**. If $\cdot \vdash R \Rightarrow pos$, then by inversion, $R = s\ N'$, so $\delta(R) = \mathsf{S}[\delta(N')]$ and by rule $\delta(R)$ **pos**. □

2.3 Example: the Call-By-Value λ-Calculus

As a second example, we represent the untyped call-by-value λ-calculus. To avoid confusion with the framework abstraction and application, we will write "\" for the object-language abstraction and infix "@" for object-language application.

$$e ::= x \mid \backslash x.\, e \mid e_1 \mathbin{@} e_2$$

The λ-calculus can be adequately represented by the following LF signature using *higher-order abstract syntax*:

exp : type.

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

The encoding $\epsilon_X(e)$ is given relative to the set of free variables X of the expression e, and its inverse is defined by induction on the canonical forms of type exp.

$$\begin{aligned} \epsilon_{X,x}(x) &= x & \delta_{X,x}(x) &= x \\ \epsilon_X(\backslash x.\, e) &= lam\ (\lambda x.\, \epsilon_{X,x}(e)) & \delta_X(lam\ \lambda x.\, N) &= \backslash x.\, \delta_{X,x}(N) \\ \epsilon_X(e_1 \mathbin{@} e_2) &= app\ \epsilon_X(e_1)\ \epsilon_X(e_2) & \delta_X(app\ N_1\ N_2) &= \delta_X(N_1) \mathbin{@} \delta_X(N_2) \end{aligned}$$

Here, since the encoding involves variables and binding, we must be careful to specify the LF contexts that are appropriate for the encoding. If $X = x_1, \ldots, x_n$, let Γ_X represent the context $x_1{:}exp, \ldots, x_n{:}exp$.

Theorem 2.5 (LF Adequacy, expressions). *The encoding $\epsilon_X(-)$ is a bijection between λ-calculus expressions e with free variables among X and canonical forms of type exp in the context Γ_X and the LF signature containing declarations for lam and app. Furthermore, this bijection is compositional in the sense that it commutes with substitution.*

Proof. By induction on the structure λ-calculus expressions, and by induction on canonical forms of type *exp*, using the appropriate inversion principles. □

Informally, we can distinguish the set of *values* from the set of arbitrary *computations* using a pair of grammars:

$$v ::= x \mid \backslash x.\, c$$
$$c ::= v \mid c_1 \,@\, c_2$$

We can use sorts to make the distinction formal. While this can be encoded in LF in a variety of ways, they are all significantly more cumbersome than the LFR encoding.

$cmp \sqsubset exp.$	% the sort of computations
$val \sqsubset exp.$	% the sort of values
$val \leq cmp.$	% every value is a (trivial) computation

$lam :: (val \to cmp) \to val.$
$app :: cmp \to cmp \to cmp.$

The most interesting declaration is the one for the constant *lam*. The argument type ($val \to cmp$) indicates that *lam* binds a variable which stands for a value and the body is an arbitrary computation. The result type *val* indicates that any λ-abstraction is a value. Now we have, for example: $\cdot \vdash lam\ \lambda x.\, lam\ \lambda y.\, x \Leftarrow val$.

To argue that our encoding is adequate, we must show that it preserves the properites of being a value and being a computation. As before, we recast the grammars for values and computations as judgments *e* **val** and *e* **cmp**:

$$\frac{}{x\ \mathbf{val}} \qquad \frac{e\ \mathbf{cmp}}{\backslash x.\, e\ \mathbf{val}} \qquad \frac{e\ \mathbf{val}}{e\ \mathbf{cmp}} \qquad \frac{e_1\ \mathbf{cmp} \qquad e_2\ \mathbf{cmp}}{e_1 \,@\, e_2\ \mathbf{cmp}}$$

Additionally, just as we had to specify the LF contexts that were appropriate for the representation of expressions, we must take care to specify the set of LFR contexts appropriate for the representation of values and computations. Since our intention is that variables stand for values, we represent them with variables of sort *val*: if $X = x_1, \ldots, x_n$, then let $\overline{\Gamma}_X$ denote the LFR context $x_1{::}val{\sqsubset}exp, \ldots, x_n{::}val{\sqsubset}exp$. The LFR adequacy theorem is then very similar to that for natural numbers:

Theorem 2.6 (LFR Adequacy, values and computations). *Suppose the free variables of e are contained in X. Then the encoding $\epsilon_X(-)$ preserves the properties of being a value and being a computation:*

$$\boxed{e_1 \Downarrow e_2}$$

$$\frac{}{\backslash x.\, e \Downarrow \backslash x.\, e}\ \textit{ev-lam} \qquad \frac{e_1 \Downarrow \backslash x.\, e_1' \quad e_2 \Downarrow e_2' \quad [e_2'/x]\, e_1' \Downarrow e}{e_1 @ e_2 \Downarrow e}\ \textit{ev-app}$$

$$\boxed{c \pmb{\Downarrow} v}$$

$$\frac{}{\backslash x.\, c \pmb{\Downarrow} \backslash x.\, c}\ \textit{ev-lam} \qquad \frac{c_1 \pmb{\Downarrow} \backslash x.\, c_1' \quad c_2 \pmb{\Downarrow} v_2 \quad [v_2/x]\, c_1' \pmb{\Downarrow} v}{c_1 @ c_2 \pmb{\Downarrow} v}\ \textit{ev-app}$$

Figure 2.3: Two ways of specifying the evaluation judgment.

1. *e **val** if and only if $\overline{\Gamma}_X \vdash \epsilon_X(e) \Leftarrow val$*

2. *e **cmp** if and only if $\overline{\Gamma}_X \vdash \epsilon_X(e) \Leftarrow cmp$*

Proof. Similar to Theorem 2.2.

($\Longrightarrow$) By induction on the derivation of e **val**.

($\Longleftarrow$) By induction on the given derivation using the inversion principles appropriate to the LFR signature of values and computations and the context $\overline{\Gamma}_X$. □

Of course, what makes the calculus "call-by-value" is its evaluation strategy. The evaluation judgment for the call-by-value λ-calculus is given in Figure 2.3, in two different ways: both ways have the exact same rules, but the second way has a more precise judgment form in that it specifies that the left-hand argument to the evaluation relation is an arbitrary computation while the right-hand argument is necessarily a value. Just as visual inspection can informally verify this property of the rules, LFR sort-checking can verify the property formally.

The evaluation judgment $e_1 \Downarrow e_2$ is represented as a two-place relation in LF. Since the declarations below are intended to represent a logic program, we follow the logic programming convention of reversing the arrows in the declaration of *ev-app*:

$eval : exp \to exp \to \mathsf{type}.$

$\textit{ev-lam} : eval\ (lam\ \lambda x.E\ x)\ (lam\ \lambda x.E\ x).$

$$\begin{aligned}\textit{ev-app} : {} & eval\ (app\ E_1\ E_2)\ E\\ & \leftarrow eval\ E_1\ (lam\ \lambda x.E_1'\ x)\\ & \leftarrow eval\ E_2\ E_2'\\ & \leftarrow eval\ (E_1'\ E_2')\ E.\end{aligned}$$

The more precise form of the evaluation judgment $c \pmb{\Downarrow} v$ can be represented in LFR as a refinement *eval′* of the *eval* type family with a more precise class:

$eval' \sqsubset eval :: cmp \rightarrow val \rightarrow \mathsf{sort}.$

$ev\text{-}lam :: eval'\ (lam\ \lambda x.C\ x)\ (lam\ \lambda x.C\ x).$

$$\begin{aligned} ev\text{-}app :: {} & eval'\ (app\ C_1\ C_2)\ V \\ & \leftarrow eval'\ C_1\ (lam\ \lambda x.C_1'\ x) \\ & \leftarrow eval'\ C_2\ V_2 \\ & \leftarrow eval'\ (C_1'\ V_2)\ V. \end{aligned}$$

Sort checking the above declarations demonstrates that when evaluation returns at all, it returns a syntactic value. Moreover, if sort reconstruction gives C_1' the "most general" sort $val \rightarrow cmp$, the declarations also ensure that the language is indeed call-by-value: it would be a sort error to ever substitute a computation for a *lam*-bound variable, for example, by evaluating $(C_1'\ C_2)$ instead of $(C_1'\ V_2)$ in the *ev-app* rule. We explore this example further in Chapter 5 when we outline an algorithm for computing most general sorts.

Adequacy for the judgment $e_1 \Downarrow e_2$ establishes a bijection between derivations $\mathcal{E} :: e_1 \Downarrow e_2$ and canonical forms of type $eval\ \epsilon(e_1)\ \epsilon(e_2)$. However, since the derivations of $e_1 \Downarrow e_2$ are isomorphic to the derivations of $c \Downarrow v$, there is little interesting to say about the LFR adequacy of this encoding, so we postpone a discussion of adequacy for judgments until the next section. We will note, though, that such an adequacy argument depends on a basic well-formedness criterion: for any expressions e_1 and e_2, we have $\cdot \vdash eval\ \epsilon(e_1)\ \epsilon(e_2) \Leftarrow \mathsf{type}$, a criterion follows from the adequacy of the encoding of expressions. There is an analogous well-formedness criterion for the LFR encoding: for a computation c and a value v, we have $\cdot \vdash eval'\ \epsilon(c)\ \epsilon(v) \sqsubset eval\ \epsilon(c)\ \epsilon(v)$. Since the encoding of computations and values is adequate, the refinement corresponding to the judgment form is well-formed.

2.4 Example: Normal Natural Deductions

For our final example, we consider natural deduction encoded in LF, with normal natural deductions as a refinement. Figure 2.4 gives the informal presentation of natural deduction, in which the metavariables ϕ and ψ range over a set of propositions including implication $\phi \supset \psi$. Formally, natural deductions are represented by encoding the judgment ϕ **true** as a type family and the inference rules as its constructors. The hypothetical judgment is represented using the LF function space.

$o : \mathsf{type}.$
$\supset : o \rightarrow o \rightarrow o.$ % implication, used as an infix operator

$true : o \rightarrow \mathsf{type}.$
$\supset\! I\ : (true\ Phi \rightarrow true\ Psi) \rightarrow true\ (Phi \supset Psi).$
$\supset\! E : true\ (Phi \supset Psi) \rightarrow true\ Phi \rightarrow true\ Psi.$

$$\boxed{\phi\ \mathbf{true}}$$

$$\frac{\begin{array}{c}\overline{\phi\ \mathbf{true}}\ u\\ \vdots\\ \psi\ \mathbf{true}\end{array}}{\phi\supset\psi\ \mathbf{true}}\supset I^u \qquad\qquad \frac{\phi\supset\psi\ \mathbf{true}\quad \phi\ \mathbf{true}}{\psi\ \mathbf{true}}\supset E$$

$$\boxed{\phi\ \mathbf{normal}} \qquad\qquad \boxed{\phi\ \mathbf{neutral}}$$

$$\frac{\begin{array}{c}\overline{\phi\ \mathbf{neutral}}\ u\\ \vdots\\ \psi\ \mathbf{normal}\end{array}}{\phi\supset\psi\ \mathbf{normal}}\supset I^u \qquad \frac{\phi\ \mathbf{neutral}}{\phi\ \mathbf{normal}}\ * \qquad\qquad \frac{\phi\supset\psi\ \mathbf{neutral}\quad \phi\ \mathbf{normal}}{\psi\ \mathbf{neutral}}\supset E$$

Figure 2.4: Natural deductions and normal/neutral natural deductions.

The adequacy argument for this representation gives an encoding $\epsilon_\gamma(-)$ relative to a collection of labeled hypotheses $\gamma = u_1{:}\phi_1, \ldots, u_n{:}\phi_n$.

$$\epsilon_{\gamma,u:\phi}\left(\overline{\phi\ \mathbf{true}}\ u\right) = u$$

$$\epsilon_\gamma\left(\frac{\begin{array}{c}\overline{\phi\ \mathbf{true}}\ u\\ \mathcal{D}\\ \psi\ \mathbf{true}\end{array}}{\phi\supset\psi\ \mathbf{true}}\supset I^u\right) = \supset\! I\ (\lambda u.\, \epsilon_{\gamma,u:\phi}(\mathcal{D}))$$

$$\epsilon_\gamma\left(\frac{\begin{array}{c}\mathcal{D}_1\\ \phi\supset\psi\ \mathbf{true}\end{array}\quad \begin{array}{c}\mathcal{D}_2\\ \phi\ \mathbf{true}\end{array}}{\psi\ \mathbf{true}}\supset E\right) = \supset\! E\ \epsilon_\gamma(\mathcal{D}_1)\ \epsilon_\gamma(\mathcal{D}_2)$$

We assume an adequate encoding of propositions $\epsilon(\phi)$ as objects of type o. If $\gamma = u_1{:}\phi_1, \ldots, u_n{:}\phi_n$, then let $\Gamma_\gamma = u_1{:}\mathit{true}\ \epsilon(\phi_1), \ldots, u_n{:}\mathit{true}\ \epsilon(\phi_n)$.

Theorem 2.7 (LF Adequacy, natural deduction). *The encoding $\epsilon_\gamma(-)$ is a bijection between natural deductions $\mathcal{D} :: \phi$ **true** with hypotheses among γ and canonical forms of type true $\epsilon(\phi)$ in the context Γ_γ and the LF signature for natural deduction. Furthermore, this bijection is compositional in the sense that it commutes with substitution.*

As shown in Figure 2.4, we can isolate the normal natural deductions by defining a pair of mutually inductive judgments ϕ **normal** and ϕ **neutral**. The rule labeled with an asterisk "$*$" represents a judgmental inclusion; if we imagine this rule being suppressed in derivations, it is evident that derivations of ϕ **normal** and ϕ **neutral** are just special forms of derivations of ϕ **true**. We can make this intuition precise by defining them as LFR sort families refining *true*, using subsorting to encode the judgmental inclusion.[6]

```
normal ⊏ true.
neutral ⊏ true.

neutral ≤ normal.        % judgmental inclusion

⊃I  :: (neutral Phi → normal Psi) → normal (Phi ⊃ Psi).
⊃E :: neutral (Phi ⊃ Psi) → normal Phi → neutral Psi.
```

Adequacy for the judgments ϕ **normal** and ϕ **neutral** is a statement about particular forms of derivations of ϕ **true**. In a way, it is quite similar to the adequacy of the encoding of even and odd natural numbers and the adequacy of the encoding of values and computations. The similarity follows from the uniformity of the LF approach: since syntax and deductions are both represented as canonical LF terms, refinements can uniformly represent subsets of either.

Note that in both the statement and the proof of this theorem, we consider a derivation $\mathcal{D}$ of ϕ **normal** or ϕ **neutral** to be identical to the derivation of ϕ **true** with the same structure. In particular, we consider the judgmental inclusion rule "$*$" to be a no-op: it simply allows us to treat a derivation of ϕ **neutral** as if it were a derivation of ϕ **normal** without actually constructing a new derivation. Equating derivations in this way may seem peculiar, but it is in keeping with the conventions we have been assuming for syntax: for instance a value v is still considered to be an expression—just one with a particular form.

Given a collection of labeled natural deduction hypotheses $\gamma = u_1{:}\phi_1, \ldots, u_n{:}\phi_n$, let $\overline{\Gamma}_\gamma = u_1{::}\mathit{neutral}\ \epsilon(\phi_1), \ldots, u_n{::}\mathit{neutral}\ \epsilon(\phi_n)$.

Theorem 2.8 (LFR Adequacy, normal and neutral natural deductions). *Let $\mathcal{D}$ be a derivation of ϕ* **true** *whose hypotheses come from γ. The encoding $\epsilon_\gamma(-)$ of natural deductions preserves the properties of being normal and being neutral.*

1. $\mathcal{D} :: \phi$ **normal** *if and only if* $\Gamma_\gamma \vdash \epsilon_\gamma(\mathcal{D}) \Leftarrow \mathit{normal}\ \epsilon(\phi)$

2. $\mathcal{D} :: \phi$ **neutral** *if and only if* $\Gamma_\gamma \vdash \epsilon_\gamma(\mathcal{D}) \Leftarrow \mathit{neutral}\ \epsilon(\phi)$

Proof. Straightforward induction in both directions. □

[6]In this presentation of normal natural deductions, there are no restrictions on the applicability of the judgmental inclusion: any deduction of ϕ **neutral** is as good as a deduction of ϕ **normal** regardless of the form of ϕ. Later in Section 6.2.2, we will return to this example to explore the question of encoding restricted variations on the judgmental inclusion.

2.5 *All* Properties as Sorts?

Now that we have seen a few examples of LFR sorts in action, we might ask the question of how far it can go. We have already mentioned the fact that any property that can be captured with sorts can be captured with judgments as well. Does the converse also hold?

Certainly not, for judgments are just arbitrary logical specifications of recursively enumerable properties. It is not difficult for example to craft rules specifying an undecidable judgment e **normalizing** that holds of all and only those untyped λ-calculus expressions that have a normal form, and it is equally easy to represent this judgment as a predicate $\mathit{normalizing} : \mathit{exp} \to \mathsf{type}$ in LF. But as we will see in Chapter 3, the sort checking problem for LFR is decidable, so clearly arbitrary judgments are capable of capturing more properties than refinement types. Even if we restrict our attention to judgments capturing decidable properties, there are plenty of judgments we would not expect to be definable using sorts. Consider for example the decidable judgment n **prime** which holds of all prime natural numbers: such a rich property will not be representable using sorts.

The power of sorts is something like the power of regular tree grammars, extended to higher-order binding trees. Regular tree grammars and the equivalent tree automata have long played a role in the development of type systems for logic programs in the form of regular tree types [DZ92] and regular unary logic (RUL) programs [YS91], and work on refinement types has long drawn inspiration from them [Fre94, Dav05]. All of the examples we've seen so far—even and odd natural numbers, values of the λ-calculus, normal natural deductions—take the form of higher-order regular tree grammars, and the fact they could also all be written as unary judgments of a certain form is a reflection of the equivalence between regular tree grammars and tree automata. We take a moment to sketch the correspondence intuitively; for a more formal development of tree automata, we refer the reader to *Tree Automata Techniques and Applications* [CDG+07].

Tree automata are somewhat like finite automata, except instead of tracking a state through a string, tree automata track states through trees. A (bottom-up, non-deterministic) tree automata contains transition rules of the form $f(q_1, \ldots, q_n) \longrightarrow q$, where f is a first-order function symbol and q_1 through q_n are the states of its children; the transition rule indicates that the automaton can apply the rewrite anywhere in its input to transition the node labelled by f into state q. If the root of the input ever transitions to a final state, it is accepted.

For example, a tree automata for recognizing even and odd numbers would consist of the states *even* and *odd*, with function symbols s and z and transition rules:

$$\begin{aligned} z &\longrightarrow \mathit{even} \\ s(\mathit{even}) &\longrightarrow \mathit{odd} \\ s(\mathit{odd}) &\longrightarrow \mathit{even} \end{aligned}$$

To verify that the number 3 is odd, we start with the term $s(s(s(z)))$ and attempt to find an execution path that leads it to a final state of *odd*. We can reach *odd* through the following series of transitions. In each transition, the rewritten subterm is underlined.

$$\begin{aligned} s(s(s(\underline{z}))) &\longrightarrow s(s(\underline{s(even)})) \\ &\longrightarrow s(\underline{s(odd)}) \\ &\longrightarrow \underline{s(even)} \\ &\longrightarrow odd \end{aligned}$$

Intuitively, this tree automaton corresponds to our signature of even and odd numbers:

$z :: even.$
$s :: even \rightarrow odd$
$\quad \wedge\ odd \rightarrow even.$

Generally, states q correspond to our sorts, and function symbols f correspond to our constructors. The transitions rules of an automaton for a particular root symbol correspond to sort declarations for the corresponding constructor. Subsorting declarations like $odd \leq pos$ can be represented as "ε-rules" of the form $q_1 \longrightarrow q_2$, which do not consume any root symbol, but rather just change the state of the automaton. We might write

$$odd \longrightarrow pos$$

to indicate that any number that has reached state *odd* can move to state *pos*.

Non-deterministic tree automata can also be defined in a "top-down" style, working from the root of the tree to the leaves, so the rules have the form $q(f(x_1, \ldots, x_n)) \longrightarrow f(q_1(x_1), \ldots, q_n(x_n))$.

$$\begin{aligned} even(z) &\longrightarrow z \\ odd(s(x)) &\longrightarrow s(even(x)) \\ even(s(x)) &\longrightarrow s(odd(x)) \end{aligned}$$

Presented in this style, a term t is accepted as satisfying q if, starting from state $q(t)$, there is a sequence of transitions resulting in just the term t by itself. To see that 3 is odd in the top-down style, we perform the following execution:

$$\begin{aligned} \underline{odd(s(s(s(z))))} &\longrightarrow s(\underline{even(s(s(z)))}) \\ &\longrightarrow s(s(\underline{odd(s(z))})) \\ &\longrightarrow s(s(s(\underline{even(z)}))) \\ &\longrightarrow s(s(s(z))) \end{aligned}$$

The top-down presentation of tree automata emphasizes the connection to logic programming and treating sorts as predicates, a connection which we explore further in the subset interpretation of Chapter 4. Every tree automaton can be simulated by a unary-predicate logic program obeying certain syntactic constraints called a *regular unary logic* *program*, or RUL program [YS91]. Our sorts can be simulated by similarly constrained logic programs, but ones that may include binding of parameters and higher-order subgoals as in Elf and Twelf.

In either presentation, the "non-determinism" in the automaton is reflected in the fact that there are multiple rules for the successor constructor. This is the same as the non-determinism that arises from intersection sorts. Compare the automaton executions above to the sorting derivation for $s\ (s\ (s\ z)) \Leftarrow odd$, where the omitted subderivations correspond to the non-deterministic choices made in the execution of the automaton:

$$
\dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash s \Rightarrow even \to odd\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash s \Rightarrow odd \to even\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash s \Rightarrow even \to odd\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \vdash z \Rightarrow even\end{array}}{\vdash z \Leftarrow even}\,(\textbf{switch})}{\vdash s\ z \Rightarrow odd}\,(\Pi\textbf{-E})}{\vdash s\ z \Leftarrow odd}\,(\textbf{switch})}{\vdash s\ (s\ z) \Rightarrow even}\,(\Pi\textbf{-E})}{\vdash s\ (s\ z) \Leftarrow even}\,(\textbf{switch})}{\vdash s\ (s\ (s\ z)) \Rightarrow odd}\,(\Pi\textbf{-E})}{\vdash s\ (s\ (s\ z)) \Leftarrow odd}\,(\textbf{switch})
$$

We shall not endeavor to formalize the correspondence any further, but perhaps the above informal discussion will help the reader to bear in mind the kinds of properties we mean when we say *properties as sorts*.

2.6 Summary

Through three examples spanning a variety of deductive systems, we have explored the *properties as sorts* representation methodology of LFR. We saw how the adequacy of an LF encoding serves as the foundation for demonstrating adequacy of an LFR encoding: given an adequate LF encoding, one proves that it *preserves the properties* represented by the sorts in an LFR encoding extending it. This methodology and its notion of adequacy are uniform between the level of syntax and the level of judgments. Sometimes, though, as with the example of evaluation in the λ-calculus, a property of a judgment isn't a subset, but rather a more stringent well-formedness condition on the form of the judgment itself.

Now that we have an idea how LFR can be used to represent properties of deductive systems, we move on to proving important metatheoretic results about the framework itself.

Chapter 3

Metatheory

In this chapter, we present some metatheoretic results about our framework. After a preliminary discussion of hereditary substitution, we demonstrate that sort checking in our framework is decidable. Then we prove two foundationally important results for a logical framework, the identity and substitution principles that witness the reflexivity and transitivity of entailment. The proofs of these results follow a similar pattern to previous work using hereditary substitutions [WCPW02, NPP07, HL07]. We give sketches of all proofs. Technically tricky proofs can be found in Appendix B.

Finally, we present some surprising results about subsorting: despite our framework only defining subsorting at base sorts, it nonetheless admits the usual higher-sort subsorting principles as a derived notion of inclusion of canonical forms. We present a variety of equivalent formulations of this derived notion and show that they all admit the usual rules for deriving subsorting judgments, including the rules for distributing intersection sorts over function sorts. In fact, not only do we admit the usual rules, but the converse also holds: any inclusion of canonical forms between two sorts is reflected by some derivation using the usual rules. Thus the usual rules represent a sound and complete axiomatization of inclusion between the canonical forms of sorts in LFR.

3.1 Hereditary Substitution

Recall that we replace ordinary capture-avoiding substitution with *hereditary substitution*, $[N/x]_A$, an operation which substitutes a normal term into a canonical form yielding another canonical form, contracting redexes "in-line". The operation is indexed by the putative type of N and x to facilitate a proof of termination. In fact, the type index on hereditary substitution need only be a simple type to ensure termination. To that end, we denote simple types by α and inductively define an erasure of normal types to simple types, written $(A)^-$.

$$\alpha ::= a \mid \alpha_1 \to \alpha_2 \qquad (a\ N_1 \ldots N_k)^- = a \qquad (\Pi x{:}A.\, B)^- = (A)^- \to (B)^-$$

For clarity, we also index hereditary substitutions by the syntactic category on which they operate, so for example we have $[N/x]^{\mathrm{n}}_A\, M = M'$ and $[N/x]^{\mathrm{s}}_A\, S = S'$. Figure 3.1 lists

Judgment:	**Substitution into:**
$[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R = R'$	Atomic terms (yielding atomic)
$[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R = (N', \alpha')$	Atomic terms (yielding normal)
$[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N = N'$	Normal terms
$[N_0/x_0]^{\mathrm{p}}_{\alpha_0}\, P = P'$	Atomic types
$[N_0/x_0]^{\mathrm{a}}_{\alpha_0}\, A = A'$	Normal types
$[N_0/x_0]^{\mathrm{q}}_{\alpha_0}\, Q = Q'$	Atomic sorts
$[N_0/x_0]^{\mathrm{s}}_{\alpha_0}\, S = S'$	Normal sorts
$[N_0/x_0]^{\mathrm{k}}_{\alpha_0}\, K = K'$	Kinds
$[N_0/x_0]^{\mathrm{l}}_{\alpha_0}\, L = L'$	Classes
$[N_0/x_0]^{\gamma}_{\alpha_0}\, \Gamma = \Gamma'$	Contexts

Figure 3.1: Judgments defining hereditary substitution.

all of the judgments defining substitution. We write $[N/x]^{\mathrm{n}}_{A}\, M = M'$ as short-hand for $[N/x]^{\mathrm{n}}_{(A)^-}\, M = M'$.

Our formulation of hereditary substitution is defined judgmentally by inference rules; all of the rules for substitution into terms are included in Appendix A.2 for reference, but we discuss a few of them here. Observe that the only place β-redexes might be introduced is when substituting a normal term N into an atomic term R: N might be a λ-abstraction, and the variable being substituted for may occur at the head of R. Therefore, the judgments defining substitution into atomic terms are the most interesting ones.

We denote substitution into atomic terms by two judgments: $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R = R'$, for when the head of R is *not* x_0, and $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R = (N', \alpha')$, for when the head of R *is* x_0, where α' is the simple type of the output N'. The former is just defined compositionally; the latter is defined by two rules:

$$\frac{}{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, x_0 = (N_0, \alpha_0)}\ (\textbf{subst-rn-var})$$

$$\frac{\begin{array}{c}[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_1 = (\lambda x.\, N_1, \alpha_2 \to \alpha_1) \\ [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_2 = N_2' \qquad [N_2'/x]^{\mathrm{n}}_{\alpha_2}\, N_1 = N_1'\end{array}}{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_1\; N_2 = (N_1', \alpha_1)}\ (\textbf{subst-rn-}\boldsymbol{\beta})$$

The rule **subst-rn-var** just returns the substitutend N_0 and its putative type index α_0. The rule **subst-rn-β** applies when the result of substituting into the head of an application is a λ-abstraction; it avoids creating a redex by hereditarily substituting into the body of the abstraction, and in this way it corresponds quite closely to β-reduction in a call-by-value typed operational semantics [Gog95].

A simple lemma establishes that these two judgments are mutually exclusive by examining the head of the input atomic term.

$$\text{head}(x) = x \qquad \text{head}(c) = c \qquad \text{head}(R\ N) = \text{head}(R)$$

Lemma 3.1.

1. *If* $[N_0/x_0]^{\text{rr}}_{\alpha_0}\, R = R'$, *then* $\text{head}(R) \neq x_0$.
2. *If* $[N_0/x_0]^{\text{rn}}_{\alpha_0}\, R = (N', \alpha')$, *then* $\text{head}(R) = x_0$.

Proof. By induction on the given derivation. □

Substitution into normal terms has two rules for atomic terms R, one which calls the "rr" judgment and one which calls the "rn" judgment.

$$\frac{[N_0/x_0]^{\text{rr}}_{\alpha_0}\, R = R'}{[N_0/x_0]^{\text{n}}_{\alpha_0}\, R = R'}\ (\textbf{subst-n-atom}) \qquad \frac{[N_0/x_0]^{\text{rn}}_{\alpha_0}\, R = (R', a')}{[N_0/x_0]^{\text{n}}_{\alpha_0}\, R = R'}\ (\textbf{subst-n-atom-norm})$$

Note that the latter rule requires both the term and the type returned by the "rn" judgment to be atomic.

Every other syntactic category's substitution judgment is defined compositionally, tacitly renaming bound variables to avoid capture. For example, the remaining rule defining substitution into normal terms, the rule for substituting into a λ-abstraction, just recurses on the body of the abstraction.

$$\frac{[N_0/x_0]^{\text{n}}_{\alpha_0}\, N = N'}{[N_0/x_0]^{\text{n}}_{\alpha_0}\, \lambda x.\, N = \lambda x.\, N'}$$

Although we have only defined hereditary substitution relationally, it is easy to show that it is in fact a partial function by proving that there only ever exists one "output" for a given set of "inputs".

Theorem 3.2 (Functionality of Substitution). *Hereditary substitution is a functional relation. In particular:*

1. *If* $[N_0/x_0]^{\text{rr}}_{\alpha_0}\, R = R_1$ *and* $[N_0/x_0]^{\text{rr}}_{\alpha_0}\, R = R_2$, *then* $R_1 = R_2$,
2. *If* $[N_0/x_0]^{\text{rn}}_{\alpha_0}\, R = (N_1, \alpha_1)$ *and* $[N_0/x_0]^{\text{rn}}_{\alpha_0}\, R = (N_2, \alpha_2)$, *then* $N_1 = N_2$ *and* $\alpha_1 = \alpha_2$,
3. *If* $[N_0/x_0]^{\text{n}}_{\alpha_0}\, N = N_1$ *and* $[N_0/x_0]^{\text{n}}_{\alpha_0}\, N = N_2$, *then* $N_1 = N_2$,

and similarly for other syntactic categories.

Proof. Straightforward induction on the first derivation, applying inversion to the second derivation. The cases for rules **subst-n-atom** and **subst-n-atom-norm** require Lemma 3.1 to show that the second derivation ends with the same rule as the first one. □

Additionally, it is worth noting that hereditary substitution behaves just like "ordinary" substitution on terms that do not contain the distinguished free variable.

Theorem 3.3 (Trivial Substitution). *Hereditary substitution for a non-occurring variable has no effect.*

1. *If* $x_0 \notin \mathrm{FV}(R)$*, then* $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R$*,*

2. *If* $x_0 \notin \mathrm{FV}(N)$*, then* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N$*,*

and similarly for other syntactic categories.

Proof. Straightforward induction on term structure. □

3.2 Decidability

A hallmark of the canonical forms/hereditary substitution approach is that it allows a decidability proof to be carried out comparatively early, before proving anything about the behavior of substitution, and without dealing with any complications introduced by β/η-conversions inside types. Ordinarily in a dependently typed calculus, one must first prove a substitution theorem before proving typechecking decidable, since typechecking relies on type equality, type equality relies on β/η-conversion, and β/η-conversions rely on substitution preserving well-formedness. (See for example [HP05] for a typical non-canonical forms-style account of LF definitional equality.)

In contrast, if only canonical forms are permitted, then type equality is just α-convertibility, so one only needs to show *decidability* of substitution in order to show decidability of typechecking. Since LF encodings represent judgments as type families and proof-checking as typechecking, it is comforting to have a decidability proof that relies on so few assumptions.

Lemma 3.4. *If* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$*, then* α' *is a subterm of* α_0*.*

Proof. By induction on the derivation of $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$. In rule **subst-rn-var**, α' is the same as α_0. In rule **subst-rn-**β, our inductive hypothesis tells us that $\alpha_2 \to \alpha_1$ is a subterm of α_0, so α_1 is as well. □

By working in a constructive metalogic, we are able to prove decidability of a judgment by proving an instance of the law of the excluded middle; the computational content of the proof then represents a decision procedure.

Theorem 3.5 (Decidability of Substitution). *Hereditary substitution is decidable. In particular:*

1. *Given* N_0*,* x_0*,* α_0*, and* R*, either* $\exists R'. [N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'$*, or* $\nexists R'. [N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'$*,*

2. *Given* N_0*,* x_0*,* α_0*, and* R*, either* $\exists (N', \alpha'). [N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$*, or* $\nexists (N', \alpha'). [N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$*,*

3. *Given* N_0*,* x_0*,* α_0*, and* N*, either* $\exists N'. [N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N'$*, or* $\nexists N'. [N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N'$*,*

and similarly for other syntactic categories.

Proof. By lexicographic induction on the type subscript α_0, the main subject of the substitution judgment, and the clause number. For each applicable rule defining hereditary substitution, the premises are at a smaller type subscript, or if the same type subscript, then a smaller term, or if the same term, then an earlier clause. The case for rule **subst-rn-**β relies on Lemma 3.4 to know that α_2 is a strict subterm of α_0. $\square$

Theorem 3.6 (Decidability of Subsorting). *Given Q_1 and Q_2, either $Q_1 \leq Q_2$ or $Q_1 \nleq Q_2$.*

Proof. Since the subsorting relation $Q_1 \leq Q_2$ is just the reflexive, transitive closure of the declared subsorting relation $s_1 \leq s_2$, it suffices to compute this closure, check that the heads of Q_1 and Q_2 are related by it, and ensure that all of the arguments of Q_1 and Q_2 are equal. $\square$

We prove decidability of typing by exhibiting a deterministic algorithmic system that is equivalent to the original. Instead of synthesizing a single sort for an atomic term, the algorithmic system synthesizes an intersection-free list of sorts, Δ.

$$\Delta ::= \cdot \mid \Delta, Q \mid \Delta, \Pi x{::}S{\sqsubset}A.\, T$$

(As usual, we freely overload comma to mean list concatenation, as no ambiguity can result.) One can think of Δ as the intersection of all its elements. Instead of applying intersection eliminations, the algorithmic system eagerly breaks down top-level intersections using a "split" operator, leading to a deterministic "minimal-synthesis" system.

$$\begin{array}{rcl} \mathrm{split}(Q) = Q & \qquad & \mathrm{split}(S_1 \wedge S_2) = \mathrm{split}(S_1), \mathrm{split}(S_2) \\ \mathrm{split}(\Pi x{::}S{\sqsubset}A.\, T) = \Pi x{::}S{\sqsubset}A.\, T & & \mathrm{split}(\top) = \cdot \end{array}$$

$$\frac{c{::}S \in \Sigma}{\Gamma \vdash c \Rrightarrow \mathrm{split}(S)} \qquad \frac{x{::}S{\sqsubset}A \in \Gamma}{\Gamma \vdash x \Rrightarrow \mathrm{split}(S)} \qquad \frac{\Gamma \vdash R \Rrightarrow \Delta \qquad \Gamma \vdash \Delta \mathbin{@} N = \Delta'}{\Gamma \vdash R\, N \Rrightarrow \Delta'}$$

The rule for applications uses an auxiliary judgment $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$ which computes the possible types of $R\, N$ given that R synthesizes to all the sorts in Δ. It has two key rules:

$$\frac{}{\Gamma \vdash \cdot \mathbin{@} N = \cdot} \qquad \frac{\Gamma \vdash \Delta \mathbin{@} N = \Delta' \qquad \Gamma \vdash N \Lleftarrow S \qquad [N/x]^{\mathrm{s}}_{A}\, T = T'}{\Gamma \vdash (\Delta, \Pi x{::}S{\sqsubset}A.\, T) \mathbin{@} N = \Delta', \mathrm{split}(T')}$$

The other rules force the judgment to be defined when neither of the above two rules apply.

$$\frac{\Gamma \vdash \Delta \mathbin{@} N = \Delta' \qquad \Gamma \nvdash N \Lleftarrow S}{\Gamma \vdash (\Delta, \Pi x{::}S{\sqsubset}A.\, T) \mathbin{@} N = \Delta'} \qquad \frac{\Gamma \vdash \Delta \mathbin{@} N = \Delta' \qquad \nexists T'.\, [N/x]^{\mathrm{s}}_{A}\, T = T'}{\Gamma \vdash (\Delta, \Pi x{::}S{\sqsubset}A.\, T) \mathbin{@} N = \Delta'}$$

$$\frac{\Gamma \vdash \Delta \mathbin{@} N = \Delta'}{\Gamma \vdash (\Delta, Q) \mathbin{@} N = \Delta'}$$

Finally, to tie everything together, we define a new checking judgment $\Gamma \vdash N \Leftarrow S$ that makes use of the algorithmic synthesis judgment; it looks just like $\Gamma \vdash N \Leftarrow S$ except for the rule for atomic terms, which must search through the synthesized list for a suitable atomic sort.

$$\frac{\Gamma \vdash R \Rrightarrow \Delta \qquad Q' \in \Delta \qquad Q' \leq Q}{\Gamma \vdash R \Lleftarrow Q} \qquad \frac{\Gamma, x{::}S{\sqsubset}A \vdash N \Lleftarrow T}{\Gamma \vdash \lambda x.\, N \Lleftarrow \Pi x{::}S{\sqsubset}A.\, T}$$

$$\frac{}{\Gamma \vdash N \Lleftarrow \top} \qquad \frac{\Gamma \vdash N \Lleftarrow S_1 \qquad \Gamma \vdash N \Lleftarrow S_2}{\Gamma \vdash N \Lleftarrow S_1 \wedge S_2}$$

This new algorithmic system is manifestly decidable: despite the negative conditions in some of the premises, the definitions of the judgments are well-founded by the ordering used in the following proof. (If we wished, we could also explicitly synthesize a definition of $\Gamma \nvdash N \Lleftarrow S$, but it would not illuminate the algorithm any further.)

Theorem 3.7. *Algorithmic sort checking is decidable. In particular:*

1. *Given Γ and R, either $\exists \Delta.\, \Gamma \vdash R \Rrightarrow \Delta$ or $\nexists \Delta.\, \Gamma \vdash R \Rrightarrow \Delta$.*
2. *Given Γ, N, and S, either $\Gamma \vdash N \Lleftarrow S$ or $\Gamma \nvdash N \Lleftarrow S$.*
3. *Given Γ, Δ, and N, $\exists \Delta'.\, \Gamma \vdash \Delta \mathbin{@} N = \Delta'$.*

Proof. By lexicographic induction on the term R or N, the clause number, and the sort S or the list of sorts Δ. For each applicable rule, the premises are either known to be decidable, or at a smaller term, or if the same term, then an earlier clause, or if the same clause, then either a smaller S or a smaller Δ. For clause 3, we must use our inductive hypothesis to argue that the rules cover all possibilities, and so a derivation always exists. □

Note that the algorithmic synthesis system sometimes outputs an empty Δ even when the given term is ill-typed, since the $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$ judgment is always defined.

It is straightforward to show that the algorithm is sound and complete with respect to the original bidirectional system.

Lemma 3.8. *If $\Gamma \vdash R \Rightarrow S$, then for all $S' \in$ split(S), $\Gamma \vdash R \Rightarrow S'$.*

Proof. By induction on S, making use of the $\wedge$-**E**$_1$ and $\wedge$-**E**$_2$ rules. □

Theorem 3.9 (Soundness of Algorithmic Typing).

1. *If $\Gamma \vdash R \Rrightarrow \Delta$, then for all $S \in \Delta$, $\Gamma \vdash R \Rightarrow S$.*
2. *If $\Gamma \vdash N \Lleftarrow S$, then $\Gamma \vdash N \Leftarrow S$.*
3. *If $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$, and for all $S \in \Delta$, $\Gamma \vdash R \Rightarrow S$, then for all $S' \in \Delta'$, $\Gamma \vdash R\, N \Rightarrow S'$.*

Proof. By induction on the given derivation, using Lemma 3.8. □

For completeness, we use the notation $\Delta \subseteq \Delta'$ to mean that Δ is a sublist of Δ'.

Lemma 3.10. *If* $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$ *and* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\Pi x{::}S{\sqsubset}A.\, T \in \Delta$ *and* $\Gamma \vdash N \Lleftarrow S$ *and* $[N/x]^{\mathsf{s}}_{A}\, T = T'$, *then* $\mathsf{split}(T') \subseteq \Delta'$.

Proof. By straightforward induction on the derivation of $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$. □

Theorem 3.11 (Completeness for Algorithmic Typing).

1. *If* $\Gamma \vdash R \Rightarrow S$, *then* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\mathsf{split}(S) \subseteq \Delta$.

2. *If* $\Gamma \vdash N \Leftarrow S$, *then* $\Gamma \vdash N \Lleftarrow S$.

Proof. By straightforward induction on the given derivation. In the application case, we make use of the fact that $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$ is always defined and apply Lemma 3.10. □

Soundness, completeness, and decidability of the algorithmic system gives us a decision procedure for the judgment $\Gamma \vdash N \Leftarrow S$. First, decidability tells us that either $\Gamma \vdash N \Lleftarrow S$ or $\Gamma \nvdash N \Lleftarrow S$. Then soundness tells us that if $\Gamma \vdash N \Lleftarrow S$ then $\Gamma \vdash N \Leftarrow S$, while completeness tells us that if $\Gamma \nvdash N \Lleftarrow S$ then $\Gamma \nvdash N \Leftarrow S$.

Decidability theorems and proofs for other syntactic categories' formation judgments proceed similarly. When all is said and done, we have enough to show that the problem of sort checking an LFR signature is decidable.

Theorem 3.12 (Decidability of Sort Checking). *Sort checking is decidable. In particular:*

1. *Given* Γ, N, *and* S, *either* $\Gamma \vdash N \Leftarrow S$ *or* $\Gamma \nvdash N \Leftarrow S$,

2. *Given* Γ, S, *and* A, *either* $\Gamma \vdash S \sqsubset A$ *or* $\Gamma \nvdash S \sqsubset A$, *and*

3. *Given* Σ, *either* $\vdash \Sigma\ \mathsf{sig}$ *or* $\nvdash \Sigma\ \mathsf{sig}$.

Proof. Corollary of Theorems 3.7, 3.9, and 3.11, and the analogous results regarding the refinement and signature formation judgments. □

Now that we have established decidability for our original rules, we return to proving other metatheoretic properties they enjoy. We will not have anything further to say about the algorithmic formulation of sort checking until Section 3.4 below, where we develop a similarly algorithmic system for subsorting.

3.3 Identity and Substitution Principles

Since well-typed terms in our framework must be canonical, that is β-normal and η-long, it is non-trivial to prove $S \to S$ for non-atomic S, or to compose proofs of $S_1 \to S_2$ and $S_2 \to S_3$. The Identity and Substitution principles ensure that our type theory makes logical sense by demonstrating the reflexivity and transitivity of entailment. Reflexivity is witnessed by η-expansion, while transitivity is witnessed by hereditary substitution.

The Identity principle effectively says that synthesizing (atomic) objects can be made to serve as checking (normal) objects. The Substitution principle dually says that checking objects may stand in for synthesizing assumptions, that is, variables.

3.3.1 Substitution

The goal of this section is to give a careful proof of the following substitution theorem.

> Suppose $\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0$. Then:
>
> 1. If
>   - $\vdash \Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}}$ ctx , and
>   - $\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash S \sqsubset A$, and
>   - $\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$,
>
>   then
>   - $[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_{\mathrm{R}} = \Gamma'_{\mathrm{R}}$ and $\vdash \Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}}$ ctx , and
>   - $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S'$ and $[N_0/x_0]^{\mathrm{a}}_{A_0}\, A = A'$ and $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash S' \sqsubset A'$, and
>   - $[N_0/x_0]^{\mathrm{n}}_{A_0}\, N = N'$ and $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow S'$,
> 2. If
>   - $\vdash \Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}}$ ctx and
>   - $\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash R \Rightarrow S$,
>
>   then
>   - $[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_{\mathrm{R}} = \Gamma'_{\mathrm{R}}$ and $\vdash \Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}}$ ctx , and $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S'$, and either
>   - $[N_0/x_0]^{\mathrm{rr}}_{A_0}\, R = R'$ and $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash R' \Rightarrow S'$, or
>   - $[N_0/x_0]^{\mathrm{rn}}_{A_0}\, R = (N', \alpha')$ and $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow S'$,
>
> and similarly for other syntactic categories.
> (**Theorem 3.19** below.)

To prove the substitution theorem, we require a lemma about how substitutions compose. The corresponding property for a ordinary non-hereditary substitution says that $[N_0/x_0]\,[N_2/x_2]\,N = [[N_0/x_0]\,N_2/x_2]\,[N_0/x_0]\,N$. For hereditary substitutions, the situation is analogous, but we must be clear about which substitution instances we must assume to be defined and which we may conclude to be defined: If the three "inner" substitutions are defined, then the two "outer" ones are also defined, and equal. Note that the composition lemma is something like a diamond property; the notation below is meant to suggest this connection.

Lemma 3.13 (Composition of Substitutions). *Suppose $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_2 = N_2^{\backprime}$ and $x_2 \notin \mathrm{FV}(N_0)$. Then:*

1. *If $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N = N^{\backprime}$ and $[N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, N = N'$, then for some $N^{\backprime\prime}$,*
   $[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N^{\backprime} = N^{\backprime\prime}$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N' = N^{\backprime\prime}$,

2. *If $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R = R^{\backprime}$ and $[N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R = R'$, then for some $R^{\backprime\prime}$,*
   $[N_2^{\backprime}/x_2]^{\mathrm{rr}}_{\alpha_2}\, R^{\backprime} = R^{\backprime\prime}$ and $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R' = R^{\backprime\prime}$,

3. *If* $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R = R^{\backprime}$ *and* $[N_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, R = (N', \beta)$*, then for some* $N^{\backprime\prime}$,
   $[N_2^{\backprime}/x_2]^{\mathrm{rn}}_{\alpha_2}\, R^{\backprime} = (N^{\backprime\prime}, \beta)$ *and* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N' = N^{\backprime\prime}$,

4. *If* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R = (N^{\backprime}, \beta)$ *and* $[N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R = R'$*, then for some* $N^{\backprime\prime}$,
   $[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N^{\backprime} = N^{\backprime\prime}$ *and* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R' = (N^{\backprime\prime}, \beta)$,

and similarly for other syntactic categories.

Proof (sketch). By lexicographic induction, first on the unordered pair $\{\alpha_0, \alpha_2\}$, and second on the first substitution derivation in each clause. Unordered pairs are ordered by

$$\{\alpha, \beta\} < \{\alpha', \beta'\} \quad \text{if } \alpha < \alpha' \text{ and } \beta = \beta',\ \text{or } \alpha = \alpha' \text{ and } \beta < \beta'$$

where $\{\alpha, \beta\} = \{\beta, \alpha\}$. The cases for rule **subst-rn-**β in clauses 3 and 4 appeal to the induction hypothesis at a smaller type using Lemma 3.4. The case in clause 4 swaps the roles of α_0 and α_2, necessitating the unordered induction metric. (The full proof may be found in Appendix B.1.) □

We also require a simple lemma about substitution into subsorting derivations:

Lemma 3.14 (Substitution into Subsorting). *If* $Q_1 \leq Q_2$ *and* $[N_0/x_0]^{\mathrm{q}}_{\alpha_0}\, Q_1 = Q_1'$ *and* $[N_0/x_0]^{\mathrm{q}}_{\alpha_0}\, Q_2 = Q_2'$*, then* $Q_1' \leq Q_2'$.

Proof. Straightforward induction using Theorem 3.2 (Functionality of Substitution), since the subsorting rules depend only on term equalities, and not on well-formedness. □

Next, we must state the substitution theorem in a form general enough to admit an inductive proof. Following previous work on canonical forms-based LF [WCPW02, HL07], we strengthen its statement to one that does not presuppose the well-formedness of the context or the classifying sorts, but instead merely presupposes that hereditary substitution is defined on them.[1] We call this strengthened theorem "proto-substitution" and prove it in several parts. In order to capture the convention that we only sort-check well-typed terms, proto-substitution includes hypotheses about well-typedness of terms (set in gray); these hypotheses use an erasure Γ^* that transforms an LFR context into an LF context.

$$\cdot^* = \cdot \qquad\qquad\qquad\qquad (\Gamma, x{::}S{\sqsubset}A)^* = \Gamma^*, x{:}A$$

The structure of the proof under this convention requires that we interleave the proof of the core LF proto-substitution theorem. Generally, reasoning related to core LF presuppositions is analogous to refinement-related reasoning and can be dealt with mostly orthogonally, but the presuppositions are necessary in certain cases.

[1]Taking care to stage the theorem to require only minimal assumptions simplifies the proof dramatically. If we were to presuppose well-formedness of the context and the classifying sort, then we would have to ensure the well-formedness of the sort output by the synthesis judgment in order to be able to apply the inductive hypothesis for the checking premise of the Π-**E** rule. To ensure the well-formedness of the sort synthesized by the Π-**E** rule, though, we would need the substitution theorem on sorts. As a consequence, at the very least we would have to include the substitution theorem for sorts in our mutual induction, but it's not even clear that such a strategy is tenable: it is difficult if not impossible to find an suitable induction metric to justify the inductive call to substitution on sorts at the point where it is needed.

Theorem 3.15 (Proto-Substitution, terms).

1. *If*
   - $\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0$ *(and* $\Gamma^*_{\mathrm{L}} \vdash N_0 \Leftarrow A_0$*)*, *and*
   - $\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$ *(and* $\Gamma^*_{\mathrm{L}}, x_0{:}A_0, \Gamma^*_{\mathrm{R}} \vdash N \Leftarrow A$*)*, *and*
   - $[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_{\mathrm{R}} = \Gamma'_{\mathrm{R}}$, *and*
   - $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S'$ *(and* $[N_0/x_0]^{\mathrm{a}}_{A_0}\, A = A'$*)*,

   then

   - $[N_0/x_0]^{\mathrm{n}}_{A_0}\, N = N'$, *and*
   - $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow S'$ *(and* $\Gamma^*_{\mathrm{L}}, (\Gamma'_{\mathrm{R}})^* \vdash N' \Leftarrow A'$*)*.

2. *If*
   - $\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0$ *(and* $\Gamma^*_{\mathrm{L}} \vdash N_0 \Leftarrow A_0$*)*, *and*
   - $\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash R \Rightarrow S$ *(and* $\Gamma^*_{\mathrm{L}}, x_0{:}A_0, \Gamma^*_{\mathrm{R}} \vdash R \Rightarrow A$*)*, *and*
   - $[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_{\mathrm{R}} = \Gamma'_{\mathrm{R}}$,

   then

   - $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S'$ *(and* $[N_0/x_0]^{\mathrm{a}}_{A_0}\, A = A'$ *)*, *and*
   - *either*
     - $[N_0/x_0]^{\mathrm{rr}}_{A_0}\, R = R'$ *and*
     - $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash R' \Rightarrow S'$ *(and* $\Gamma^*_{\mathrm{L}}, (\Gamma'_{\mathrm{R}})^* \vdash R' \Rightarrow A'$*)*,

     or
     - $[N_0/x_0]^{\mathrm{rn}}_{A_0}\, R = (N', (A')^-)$ *and*
     - $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow S'$ *(and* $\Gamma^*_{\mathrm{L}}, (\Gamma'_{\mathrm{R}})^* \vdash N' \Leftarrow A'$*)*.

Note: We tacitly assume the implicit signature Σ is well-formed. We do *not* tacitly assume that any of the contexts, sorts, or types are well-formed. We *do* tacitly assume that contexts respect the usual variable conventions in that bound variables are always fresh, both with respect to other variables bound in the same context and with respect to other free variables in terms outside the scope of the binding.

Proof (sketch). By lexicographic induction on $(A_0)^-$ and the derivation $\mathcal{D}$ hypothesizing $x_0{::}S_0{\sqsubset}A_0$.

The most involved case is that for application $R_1\, N_2$. When $\mathrm{head}(R_1) = x_0$ hereditary substitution carries out a β-reduction, and the proof invokes the induction hypothesis at a smaller type but not a subderivation. This case also requires Lemma 3.13 (Composition): since function sorts are dependent, the typing rule for application carries out a substitution, and we need to compose this substitution with the $[N_0/x_0]^{\mathrm{s}}_{\alpha_0}$ substitution.

In the case where we check a term at sort $\top$, we require the core LF assumptions in order to invoke the core LF proto-substitution theorem. (The full proof may be found in Appendix B.2.) □

Next, we can prove analogous proto-substitution theorems for sorts/types and for classes/kinds.

Theorem 3.16 (Proto-Substitution, sorts and types).

1. *If*
   - $\Gamma_L \vdash N_0 \Leftarrow S_0$ *(and $\Gamma_L^* \vdash N_0 \Leftarrow A_0$)*,
   - $\Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R \vdash S \sqsubset A$ *(and $\Gamma_L^*, x_0{:}A_0, \Gamma_R^* \vdash A \Leftarrow \mathsf{type}$)*, *and*
   - $[N_0/x_0]^\gamma_{A_0}\, \Gamma_R = \Gamma_R'$,

   then
   - $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S'$ *(and $[N_0/x_0]^{\mathrm{a}}_{A_0}\, A = A'$)*, *and*
   - $\Gamma_L, \Gamma_R' \vdash S' \sqsubset A'$, *(and $\Gamma_L^*, (\Gamma_R')^* \vdash A' \Leftarrow \mathsf{type}$)*.
2. *If*
   - $\Gamma_L \vdash N_0 \Leftarrow S_0$ *(and $\Gamma_L^* \vdash N_0 \Leftarrow A_0$)*,
   - $\Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R \vdash Q \sqsubset P \Rightarrow L$ *(and $\Gamma \vdash P \Rightarrow K$)*, *and*
   - $[N_0/x_0]^\gamma_{A_0}\, \Gamma_R = \Gamma_R'$,

   then
   - $[N_0/x_0]^{\mathrm{q}}_{A_0}\, Q = Q'$ *(and $[N_0/x_0]^{\mathrm{p}}_{A_0}\, P = P'$)*, *and*
   - $[N_0/x_0]^{\mathrm{l}}_{A_0}\, L = L'$ *(and $[N_0/x_0]^{\mathrm{k}}_{A_0}\, K = K'$)*, *and*
   - $\Gamma_L, \Gamma_R' \vdash Q' \sqsubset P' \Rightarrow L'$ *(and $\Gamma_L^*, (\Gamma_R')^* \vdash P' \Rightarrow K'$)*.

Proof. By induction on the derivation hypothesizing $x_0{::}S_0{\sqsubset}A_0$, using Theorem 3.15 (Proto-Substitution, terms). The reasoning is essentially the same as the reasoning for Theorem 3.15. □

Theorem 3.17 (Proto-Substitution, classes and kinds).
If

- $\Gamma_L \vdash N_0 \Leftarrow S_0$ *(and $\Gamma_L^* \vdash N_0 \Leftarrow A_0$)*,
- $\Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R \vdash L \sqsubset K$ *(and $\Gamma_L^*, x_0{:}A_0, \Gamma_R^* \vdash K \Leftarrow \mathsf{kind}$)*, *and*
- $[N_0/x_0]^\gamma_{A_0}\, \Gamma_R = \Gamma_R'$,

then

- $[N_0/x_0]^{\mathrm{l}}_{A_0}\ L = L^{\backprime}$ *(and* $[N_0/x_0]^{\mathrm{k}}_{A_0}\ K = K^{\backprime}$*)* , *and*
- $\Gamma_{\mathrm{L}}, \Gamma^{\backprime}_{\mathrm{R}} \vdash L^{\backprime} \sqsubset K^{\backprime}$, *(and* $\Gamma^{*}_{\mathrm{L}}, (\Gamma^{\backprime}_{\mathrm{R}})^{*} \vdash K^{\backprime} \Leftarrow \mathsf{kind}$*)* .

Proof. By induction on the derivation hypothesizing $x_0{::}S_0{\sqsubset}A_0$, using Theorem 3.16 (Proto-Substitution, sorts and types). □

Then, we can finish proto-substitution by proving a proto-substitution theorem for contexts.

Theorem 3.18 (Proto-Substitution, contexts).
If

- $\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0$ *(and* $\Gamma^{*}_{\mathrm{L}} \vdash N_0 \Leftarrow A_0$*)* , *and*
- $\vdash \Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0\ \mathsf{ctx}$ *(and* $\vdash \Gamma^{*}_{\mathrm{L}}, x_0{:}A_0, \Gamma^{*}_{\mathrm{R}}\ \mathsf{ctx}$*)* ,

then

- $[N_0/x_0]^{\gamma}_{A_0}\ \Gamma_{\mathrm{R}} = \Gamma^{\backprime}_{\mathrm{R}}$, *and*
- $\vdash \Gamma_{\mathrm{L}}, \Gamma^{\backprime}_{\mathrm{R}}\ \mathsf{ctx}$ *(and* $\vdash \Gamma^{*}_{\mathrm{L}}, (\Gamma^{\backprime}_{\mathrm{R}})^{*}\ \mathsf{ctx}$*)* .

Proof. Straightforward induction on Γ_{R}. □

Finally, we have enough to obtain a proof of the desired substitution theorem.

Theorem 3.19 (Substitution). *Suppose* $\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0$. *Then:*

1. *If*
   - $\vdash \Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}}\ \mathsf{ctx}$, *and*
   - $\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash S \sqsubset A$, *and*
   - $\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$,

   then

   - $[N_0/x_0]^{\gamma}_{A_0}\ \Gamma_{\mathrm{R}} = \Gamma'_{\mathrm{R}}$ *and* $\vdash \Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}}\ \mathsf{ctx}$, *and*
   - $[N_0/x_0]^{\mathrm{s}}_{A_0}\ S = S'$ *and* $[N_0/x_0]^{\mathrm{a}}_{A_0}\ A = A'$ *and* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash S' \sqsubset A'$, *and*
   - $[N_0/x_0]^{\mathrm{n}}_{A_0}\ N = N'$ *and* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow S'$,
2. *If*
   - $\vdash \Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}}\ \mathsf{ctx}$ *and*
   - $\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash R \Rightarrow S$,

   then

   - $[N_0/x_0]^{\gamma}_{A_0}\ \Gamma_{\mathrm{R}} = \Gamma'_{\mathrm{R}}$ *and* $\vdash \Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}}\ \mathsf{ctx}$, *and* $[N_0/x_0]^{\mathrm{s}}_{A_0}\ S = S'$, *and either*

- $[N_0/x_0]^{\mathrm{rr}}_{A_0}\,R = R'$ *and* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash R' \Rightarrow S'$ *, or*
- $[N_0/x_0]^{\mathrm{rn}}_{A_0}\,R = (N', \alpha')$ *and* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow S'$ *,*

and similarly for other syntactic categories.

Proof. Straightforward corollary of Proto-Substitution Theorems 3.15, 3.16, 3.17, and 3.18. □

Having proven substitution, we henceforth tacitly assume that all subjects of a judgment are sufficiently well-formed for the judgment to make sense. In particular, we assume that all contexts are well-formed, and whenever we assume $\Gamma \vdash N \Leftarrow S$, we assume that for some well-formed type A, we have $\Gamma \vdash S \sqsubset A$ and $\Gamma \vdash N \Leftarrow A$. These assumptions embody our refinement restriction: we only sort-check a term if it is already well-typed and even then only at sorts that refine its type.

Similarly, whenever we assume $\Gamma \vdash S \sqsubset A$, we tacitly assume that $\Gamma \vdash A \Leftarrow \mathsf{type}$, and whenever we assume $\Gamma \vdash L \sqsubset K$, we tacitly assume that $\Gamma \vdash K \Leftarrow \mathsf{kind}$.

3.3.2 Identity

Just as we needed a composition lemma to prove the substitution theorem, in order to prove the identity theorem we need a lemma about how η-expansion commutes with substitution.[2]

In stating this lemma, we require a judgment that predicts the simple type output of "rn" substitution. This judgment just computes the simple type as in "rn" substitution, but without computing anything having to do with substitution. Since it resembles a sort of "approximate typing judgment", we write it $x_0{:}\alpha_0 \vdash R : \alpha$. As with "rn" substitution, it is only defined when the head of R is x_0.

$$\frac{}{x_0{:}\alpha_0 \vdash x_0 : \alpha_0} \qquad \frac{x_0{:}\alpha_0 \vdash R : \alpha \to \beta}{x_0{:}\alpha_0 \vdash R\,N : \beta}$$

Lemma 3.20. *If* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\,R = (N', \alpha')$ *and* $x_0{:}\alpha_0 \vdash R : \alpha$*, then* $\alpha' = \alpha$*.*

Proof. Straightforward induction. □

Lemma 3.21 (Commutativity of Substitution and η-expansion). *Substitution commutes with η-expansion. In particular:*

1. (a) *If* $[\eta_\alpha(x)/x]^{\mathrm{n}}_{\alpha}\,N = N'$*, then* $N = N'$ *,*

   (b) *If* $[\eta_\alpha(x)/x]^{\mathrm{rr}}_{\alpha}\,R = R'$*, then* $R = R'$ *,*

   (c) *If* $[\eta_\alpha(x)/x]^{\mathrm{rn}}_{\alpha}\,R = (N, \beta)$*, then* $\eta_\beta(R) = N$ *,*

2. *If* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\,\eta_\alpha(R) = N'$*, then*

[2]The categorically-minded reader might think of this as the right and left unit laws for $\circ$ while thinking of the composition lemma above as the associativity of $\circ$, where $\circ$ in the category represents substitution, as usual.

(a) *if* head$(R) \neq x_0$, *then* $[N_0/x_0]^{\mathsf{rr}}_{\alpha_0} R = R'$ *and* $\eta_\alpha(R') = N'$,

(b) *if* head$(R) = x_0$ *and* $x_0{:}\alpha_0 \vdash R : \alpha$, *then* $[N_0/x_0]^{\mathsf{rn}}_{\alpha_0} R = (N', \alpha)$,

and similarly for other syntactic categories.

Proof (sketch). By lexicographic induction on α and the given substitution derivation. The proofs of clauses 1a, 1b, and 1c analyze the substitution derivation, while the proofs of clauses 2a and 2b analyze the simple type α at which R is η-expanded. (The full proof may be found in Appendix B.3.) □

Note: By considering the variable being substituted for to be a bound variable subject to α-conversion[3], we can see that our commutativity theorem is equivalent to an apparently more general one where the η-expanded variable is not the same as the substituted-for variable. For example, in the case of clause (1a), we would have that if $[\eta_\alpha(x)/y]^{\mathsf{n}}_\alpha N = N'$, then $[x/y]\, N = N'$. We will freely make use of this fact in what follows when convenient.

Theorem 3.22 (Expansion). *If* $\Gamma \vdash S \sqsubset A$ *and* $\Gamma \vdash R \Rightarrow S$, *then* $\Gamma \vdash \eta_A(R) \Leftarrow S$.

Proof (sketch). By induction on S. The $\Pi x{::}S_1{\sqsubset}A_1.\, S_2$ case relies on Theorem 3.19 (Substitution) to show that $[\eta_{A_1}(x)/x]^{\mathsf{s}}_{A_1} S_2$ is defined and on Lemma 3.21 (Commutativity) to show that it is equal to S_2. (The full proof may be found in Appendix B.4.) □

Theorem 3.23 (Identity). *If* $\Gamma \vdash S \sqsubset A$, *then* $\Gamma, x{::}S{\sqsubset}A \vdash \eta_A(x) \Leftarrow S$.

Proof. Corollary of Theorem 3.22 (Expansion). □

3.4 Subsorting at Higher Sorts

Our bidirectional typing discipline limits subsorting checks to a single rule, the **switch** rule when we switch modes from checking to synthesis. Since we insist on typing only canonical forms, this rule is limited to checking at atomic sorts Q, and consequently, subsorting need only be defined on atomic sorts. These observations naturally lead one to ask, what is the status of higher-sort subsorting in LFR? How do our intuitions about things like structural rules, variance, and distributivity—in particular, the rules shown in Figure 3.2—fit into the LFR picture?

It turns out that despite not *explicitly* including subsorting at higher sorts, LFR *implicitly* includes an intrinsic notion of higher-sort subsorting through the η-expansion associated with canonical forms. The simplest way of formulating this intrinsic notion is as a variant of the identity principle: S is taken to be a subsort of T if $\Gamma, x{::}S{\sqsubset}A \vdash \eta_A(x) \Leftarrow T$. This notion is equivalent to a number of other alternate formulations, including a subsumption-based formulation and a substitution-based formulation.

[3]In other words, by reading $[N_0/x_0]^{\mathsf{n}}_{\alpha_0} N = N'$ as something like $\mathrm{subst}^{\mathsf{n}}_{\alpha_0}(N_0,\, x_0.\, N) = N'$, where x_0 is bound in N.

$\boxed{S_1 \leq S_2}$

$$\frac{}{S \leq S}\,\textbf{(refl)} \qquad \frac{S_1 \leq S_2 \qquad S_2 \leq S_3}{S_1 \leq S_3}\,\textbf{(trans)} \qquad \frac{S_2 \leq S_1 \qquad T_1 \leq T_2}{\Pi x{::}S_1.\,T_1 \leq \Pi x{::}S_2.\,T_2}\,\textbf{(S-}\Pi\textbf{)}$$

$$\frac{}{S \leq \top}\,(\top\textbf{-R}) \qquad \frac{T \leq S_1 \qquad T \leq S_2}{T \leq S_1 \wedge S_2}\,(\wedge\textbf{-R})$$

$$\frac{S_1 \leq T}{S_1 \wedge S_2 \leq T}\,(\wedge\textbf{-L}_1) \qquad \frac{S_2 \leq T}{S_1 \wedge S_2 \leq T}\,(\wedge\textbf{-L}_2)$$

$$\frac{}{\top \leq \Pi x{::}S.\,\top}\,(\top/\Pi\textbf{-dist}) \qquad \frac{}{(\Pi x{::}S.\,T_1) \wedge (\Pi x{::}S.\,T_2) \leq \Pi x{::}S.\,(T_1 \wedge T_2)}\,(\wedge/\Pi\textbf{-dist})$$

Figure 3.2: Derived rules for subsorting at higher sorts.

Theorem 3.24 (Alternate Formulations of Subsorting). *Suppose that for some Γ_0, we have $\Gamma_0 \vdash S_1 \sqsubset A$ and $\Gamma_0 \vdash S_2 \sqsubset A$, and define:*

1. $S_1 \leq_1 S_2 \overset{def}{=}$ *for all Γ and R: if $\Gamma \vdash R \Rightarrow S_1$, then $\Gamma \vdash \eta_A(R) \Leftarrow S_2$.*

2. $S_1 \leq_2 S_2 \overset{def}{=}$ *for all Γ: $\Gamma, x{::}S_1{\sqsubset}A \vdash \eta_A(x) \Leftarrow S_2$.*

3. $S_1 \leq_3 S_2 \overset{def}{=}$ *for all Γ and N: if $\Gamma \vdash N \Leftarrow S_1$, then $\Gamma \vdash N \Leftarrow S_2$.*

4. $S_1 \leq_4 S_2 \overset{def}{=}$ *for all Γ_L, Γ_R, N, and S: if $\Gamma_L, x{::}S_2{\sqsubset}A, \Gamma_R \vdash N \Leftarrow S$ then $\Gamma_L, x{::}S_1{\sqsubset}A, \Gamma_R \vdash N \Leftarrow S$*

5. $S_1 \leq_5 S_2 \overset{def}{=}$ *for all Γ_L, Γ_R, N, S, and N_1: if $\Gamma_L, x{::}S_2{\sqsubset}A, \Gamma_R \vdash N \Leftarrow S$ and $\Gamma_L \vdash N_1 \Leftarrow S_1$, then $\Gamma_L, [N_1/x]^{\gamma}_A\,\Gamma_R \vdash [N_1/x]^{n}_A\,N \Leftarrow [N_1/x]^{s}_A\,S$.*

Then, $S_1 \leq_1 S_2 \iff S_1 \leq_2 S_2 \iff \cdots \iff S_1 \leq_5 S_2$.

Proof. Using the identity and substitution principles along with Lemma 3.21, the commutativity of substitution with η-expansion.

1 $\Longrightarrow$ 2 By rule, $\Gamma, x{::}S_1{\sqsubset}A \vdash x \Rightarrow S_1$. By 1, $\Gamma, x{::}S_1{\sqsubset}A \vdash \eta_A(x) \Leftarrow S_2$.

2 $\Longrightarrow$ 3 Suppose $\Gamma \vdash N \Leftarrow S_1$. By 2, $\Gamma, x{::}S_1{\sqsubset}A \vdash \eta_A(x) \Leftarrow S_2$. By Theorem 3.19 (Substitution), $\Gamma \vdash [N/x]^{n}_A\,\eta_A(x) \Leftarrow S_2$. By Lemma 3.21 (Commutativity), $\Gamma \vdash N \Leftarrow S_2$.

3 $\Longrightarrow$ 4 Suppose $\Gamma_L, x{::}S_2{\sqsubset}A, \Gamma_R \vdash N \Leftarrow S$. By weakening, $\Gamma_L, y{::}S_1{\sqsubset}A, x{::}S_2{\sqsubset}A, \Gamma_R \vdash N \Leftarrow S$. By Theorem 3.23 (Identity), $\Gamma_L, y{::}S_1{\sqsubset}A \vdash \eta_A(y) \Leftarrow S_1$. By 3,

$\Gamma_{\mathrm{L}}, y{::}S_1{\sqsubset}A \vdash \eta_A(y) \Leftarrow S_2$. By Theorem 3.19 (Substitution), $\Gamma_{\mathrm{L}}, y{::}S_1{\sqsubset}A$, $[\eta_A(y)/x]^{\gamma}_A\, \Gamma_{\mathrm{R}} \vdash [\eta_A(y)/x]^{\mathrm{n}}_A\, N \Leftarrow [\eta_A(y)/x]^{\mathrm{s}}_A\, S$. By Lemma 3.21 (Commutativity) and α-conversion, $\Gamma_{\mathrm{L}}, x{::}S_1{\sqsubset}A, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$.

4 $\Longrightarrow$ **5** Suppose $\Gamma_{\mathrm{L}}, x{::}S_2{\sqsubset}A, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$ and $\Gamma_{\mathrm{L}} \vdash N_1 \Leftarrow S_1$. By 4, $\Gamma_{\mathrm{L}}, x{::}S_1{\sqsubset}A$, $\Gamma_{\mathrm{R}} \vdash N \Leftarrow S$. By Theorem 3.19 (Substitution), $\Gamma_{\mathrm{L}}, [N_1/x]^{\gamma}_A\, \Gamma_{\mathrm{R}} \vdash [N_1/x]^{\mathrm{n}}_A\, N \Leftarrow [N_1/x]^{\mathrm{s}}_A\, S$.

5 $\Longrightarrow$ **1** Suppose $\Gamma \vdash R \Rightarrow S_1$. By Theorem 3.22 (Expansion), $\Gamma \vdash \eta_A(R) \Leftarrow S_1$. By Theorem 3.23 (Identity), $\Gamma, x{::}S_2{\sqsubset}A \vdash \eta_A(x) \Leftarrow S_2$. By 5, $\Gamma \vdash [\eta_A(R)/x]^{\mathrm{n}}_A\, \eta_A(x) \Leftarrow S_2$. By Lemma 3.21 (Commutativity), $\Gamma \vdash \eta_A(R) \Leftarrow S_2$. □

If we take "subsorting as η-expansion" to be our *model* of subsorting, we can show the "usual" presentation in Figure 3.2 to be both sound and complete with respect to this model. In other words, subsorting as η-expansion *really is* subsorting (soundness), and it is *no more than* subsorting (completeness). Alternatively, we can say that completeness demonstrates that there are no subsorting rules missing from the usual declarative presentation: Figure 3.2 accounts for everything covered intrinsically by η-expansion. By the end of this section, we will have shown both theorems: if $S \leq T$, then $\Gamma, x{::}S{\sqsubset}A \vdash \eta_A(x) \Leftarrow T$, and vice versa.

Soundness is a straightforward inductive argument.

Theorem 3.25 (Soundness of Declarative Subsorting). *If $S \leq T$, then $\Gamma, x{::}S{\sqsubset}A \vdash \eta_A(x) \Leftarrow T$.*

Proof. By induction on the derivation of $S \leq T$. The alternate formulations given by Theorem 3.24 are useful in many cases. □

The proof of completeness is considerably more intricate. We demonstrate completeness via a detour through an algorithmic subsorting system very similar to the algorithmic typing system from Section 3.2, with judgments $\Delta \leqq S$ and $\Delta \mathbin{@} x{::}\Delta_1{\sqsubset}A_1 = \Delta_2$. To show completeness, we show that intrinsic subsorting implies algorithmic subsorting and that algorithmic subsorting implies declarative subsorting; the composition of these theorems is our desired completeness result.

If $\Gamma, x{::}S{\sqsubset}A \vdash \eta_A(x) \Leftarrow T$, then $\mathrm{split}(S) \leqq T$. (**Theorem 3.39** below.)

If $\mathrm{split}(S) \leqq T$, then $S \leq T$. (**Theorem 3.31** below.)

The following schematic representation of soundness and completeness may help the reader to understand the key theorems.

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{``declarative''}\\ S \leq T\end{array}} \arrow[rr, "\text{soundness}"] & & \boxed{\begin{array}{c}\text{``intrinsic''}\\ \Gamma, x{::}S{\sqsubset}A \vdash \eta_A(x) \Leftarrow T\end{array}} \arrow[ddl] \\
& \text{completeness} & \\
& \boxed{\begin{array}{c}\text{``algorithmic''}\\ \mathrm{split}(S) \leqq T\end{array}} \arrow[uul] &
\end{tikzcd}$$

$$\boxed{\Delta \leqq S}$$

$$\frac{}{\Delta \leqq \top} \qquad \frac{\Delta \leqq S_1 \qquad \Delta \leqq S_2}{\Delta \leqq S_1 \wedge S_2} \qquad \frac{Q' \in \Delta \qquad Q' \leq Q}{\Delta \leqq Q}$$

$$\frac{\Delta \mathbin{@} x{::}\mathrm{split}(S_1)\sqsubset A_1 = \Delta_2 \qquad \Delta_2 \leqq S_2}{\Delta \leqq \Pi x{::}S_1\sqsubset A_1.\, S_2}$$

$$\boxed{\Delta \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2}$$

$$\frac{}{\cdot \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \cdot} \qquad \frac{\Delta \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2 \qquad \Delta_1 \leqq S_1 \qquad [\eta_{A_1}(x)/y]^{\mathrm{n}}_{A_1}\, S_2 = S_2'}{(\Delta, \Pi y{::}S_1\sqsubset A_1.\, S_2) \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2, \mathrm{split}(S_2')}$$

$$\frac{\Delta \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2 \qquad \Delta_1 \nleqq S_1}{(\Delta, \Pi y{::}S_1\sqsubset A_1.\, S_2) \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2} \qquad \frac{\Delta \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2 \qquad \nexists S_2'.\, [\eta_{A_1}(x)/y]^{\mathrm{s}}_{A_1}\, S_2 = S_2'}{(\Delta, \Pi y{::}S_1\sqsubset A_1.\, S_2) \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2}$$

$$\frac{\Delta \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2}{(\Delta, Q) \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2}$$

Figure 3.3: Algorithmic subsorting.

As mentioned above, the algorithmic subsorting system is characterized by two judgments: $\Delta \leqq S$ and $\Delta \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2$; rules defining them are shown in Figure 3.3. As in Section 3.2, Δ represents an intersection-free list of sorts. The interpretation of the judgment $\Delta \leqq S$, made precise below, is roughly that the intersection of all the sorts in Δ is a subsort of the sort S.

The rule for checking whether Δ is a subsort of a function type makes use of the application judgment $\Delta \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2$ to extract all of the applicable function codomains from the list Δ. As in Section 3.2, care is taken to ensure that this judgment is defined even in seemingly "impossible" scenarios that well-formedness preconditions would rule out, like Δ containing atomic sorts or hereditary substitution being undefined.

First, we must verify that the definitions of the two judgments are well-founded since they involve negative occurrences. We make use of the following ordering: Δ is smaller than S if every element of Δ is a subterm of S, and Δ is smaller than Δ' if every element of Δ is a subterm of some element of Δ'. We combine it with the unordered pair ordering that we saw in the proof of Lemma 3.13 (Composition of Substitutions).

Theorem 3.26. *Algorithmic subsorting is decidable. In particular:*

1. *Given Δ and S, either $\Delta \leqq S$ or $\Delta \nleqq S$.*

2. *Given Δ, Δ_1, and A_1, there is a Δ_2 smaller than Δ such that $\Delta \mathbin{@} x{::}\Delta_1\sqsubset A_1 = \Delta_2$.*

$$\frac{S_1 \leq T_1 \qquad S_2 \leq T_2}{S_1 \wedge S_2 \leq T_1 \wedge T_2}\ (\textbf{S-}\wedge) \qquad\qquad \frac{}{S_1 \wedge (S_2 \wedge S_3) \leq (S_1 \wedge S_2) \wedge S_3}\ (\wedge\textbf{-assoc})$$

$$\frac{S \leq \Pi x{::}T_1.\, T_2 \qquad T_1 \leq S_1}{S \wedge \Pi x{::}S_1.\, S_2 \leq \Pi x{::}T_1.\, (T_2 \wedge S_2)}\ (\wedge/\Pi\textbf{-dist}')$$

Figure 3.4: Useful rules derivable from those in Figure 3.2.

Proof. By mutual induction on the unordered pairs $\{\Delta, S\}$ and $\{\Delta, \Delta_1\}$. For each applicable rule, the premises are either known to be decidable or at a smaller pair. In the second clause, whatever the inputs are, some rule applies, but when $\Delta = \Delta', \Pi y{::}S_1{\sqsubset}A_1.\, S_2$, we require the unordered inductive hypothesis to tell us that either $\Delta_1 \leqq S_1$ or $\Delta_1 \nleqq S_1$. □

Our next task is to demonstrate that the algorithm has the interpretation alluded to above. To that end, we define an operator $\bigwedge(-)$ that transforms a list Δ into a sort S by "folding" $\wedge$ over Δ with unit $\top$.

$$\bigwedge(\cdot) = \top \qquad\qquad \bigwedge(\Delta, S) = \bigwedge(\Delta) \wedge S$$

Now our goal is to demonstrate that if the algorithm says $\Delta \leqq S$, then declaratively $\bigwedge(\Delta) \leq S$. First, we prove some useful properties of the $\bigwedge(-)$ operator.

Lemma 3.27. $\bigwedge(\Delta_1) \wedge \bigwedge(\Delta_2) \leq \bigwedge(\Delta_1, \Delta_2)$

Proof. Straightforward induction on Δ_2. □

Lemma 3.28. $S \leq \bigwedge(\text{split}(S))$.

Proof. Straightforward induction on S. □

Lemma 3.29. *If $Q' \in \Delta$ and $Q' \leq Q$, then $\bigwedge(\Delta) \leq Q$.*

Proof. Straightforward induction on Δ. □

Theorem 3.30 (Generalized Algorithmic ⇒ Declarative).

1. *If $\mathcal{D} :: \Delta \leqq T$, then $\bigwedge(\Delta) \leq T$.*
2. *If $\mathcal{D} :: \Delta \mathbin{@} x{::}\Delta_1{\sqsubset}A_1 = \Delta_2$, then $\bigwedge(\Delta) \leq \Pi x{::}\bigwedge(\Delta_1){\sqsubset}A_1.\ \bigwedge(\Delta_2)$.*

Proof (sketch). By induction on $\mathcal{D}$, using Lemmas 3.27, 3.28, and 3.29. The derivable rules from Figure 3.4 come in handy in the proof of clause 2. (The full proof may be found in Appendix B.5.) □

Theorem 3.30 is sufficient to prove that algorithmic subsorting implies declarative subsorting.

Theorem 3.31 (Algorithmic ⇒ Declarative). *If* $\mathrm{split}(S) \leqq T$*, then* $S \leq T$*.*

Proof. Suppose $\mathrm{split}(S) \leqq T$. Then,

$\bigwedge(\mathrm{split}(S)) \leq T$	By Theorem 3.30.
$S \leq \bigwedge(\mathrm{split}(S))$	By Lemma 3.28.
$S \leq T$	By rule **trans**.

□

Now it remains only to show that intrinsic subsorting implies algorithmic. To do so, we require some lemmas. First, we extend our notion of a sort S refining a type A to an entire list of sorts Δ refining a type A in the obvious way.

$$\frac{}{\Gamma \vdash \cdot \sqsubset A} \qquad \frac{\Gamma \vdash \Delta \sqsubset A \qquad \Gamma \vdash S \sqsubset A}{\Gamma \vdash (\Delta, S) \sqsubset A}$$

This new notion has the following important properties.

Lemma 3.32. *If* $\Gamma \vdash \Delta_1 \sqsubset A$ *and* $\Gamma \vdash \Delta_2 \sqsubset A$*, then* $\Gamma \vdash \Delta_1, \Delta_2 \sqsubset A$*.*

Proof. Straightforward induction on Δ_2. □

Lemma 3.33. *If* $\Gamma \vdash S \sqsubset A$*, then* $\Gamma \vdash \mathrm{split}(S) \sqsubset A$*.*

Proof. Straightforward induction on S. □

Lemma 3.34. *If* $\mathcal{D} :: \Gamma \vdash \Delta \sqsubset \Pi x{:}A_1.\, A_2$ *and* $\mathcal{E} :: \Gamma \vdash \Delta \mathbin{@} N = \Delta_2$ *and* $[N/x]^{\mathrm{a}}_{A_1} A_2 = A_2'$*, then* $\Gamma \vdash \Delta_2 \sqsubset A_2'$*.*

Proof (sketch). By induction on $\mathcal{E}$, using Theorem 3.9 (Soundness of Algorithmic Typing) to appeal to Theorem 3.19 (Substitution), along with Lemmas 3.32 and 3.33. (The full proof may be found in Appendix B.6.) □

We will also require an analogue of subsumption for our algorithmic typing system, which relies on two lemmas about lists of sorts.

Lemma 3.35. *If* $\Gamma \vdash \Delta \sqsubset A$*, then for all* $S \in \Delta$*,* $\Gamma \vdash S \sqsubset A$*.*

Proof. Straightforward induction on Δ. □

Lemma 3.36. *If for all* $S \in \Delta$*,* $\Gamma \vdash N \Leftarrow S$*, then* $\Gamma \vdash N \Leftarrow \bigwedge(\Delta)$*.*

Proof. Straightforward induction on Δ. □

Theorem 3.37 (Algorithmic Subsumption). *If* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\Gamma \vdash \Delta \sqsubset A$ *and* $\Delta \leqq S$*, then* $\Gamma \vdash \eta_A(R) \Lleftarrow S$*.*

Proof. Straightforward deduction, using soundness and completeness of algorithmic typing.

$\forall S' \in \Delta.\, \Gamma \vdash R \Rightarrow S'$ By Theorem 3.9 (Soundness of Alg. Typing).
$\forall S' \in \Delta.\, \Gamma \vdash S' \sqsubset A$ By Lemma 3.35.
$\forall S' \in \Delta.\, \Gamma \vdash \eta_A(R) \Leftarrow S'$ By Theorem 3.22 (Expansion).
$\Gamma \vdash \eta_A(R) \Leftarrow \bigwedge(\Delta)$ By Lemma 3.36.

$\Delta \leqq S$ By assumption.
$\bigwedge(\Delta) \leq S$ By Theorem 3.30 (Generalized Alg. $\Rightarrow$ Decl.).

$\Gamma \vdash \eta_A(R) \Leftarrow S$ By Theorem 3.25 (Soundness of Decl. Subsorting) and Theorem 3.24 (Alternate Formulations of Subsorting).
$\Gamma \vdash \eta_A(R) \Lleftarrow S$ By Theorem 3.11 (Completeness of Alg. Typing).

□

Now we can prove the following main theorem, which generalizes our desired "Intrinsic ⇒ Algorithmic" theorem:

Theorem 3.38 (Generalized Intrinsic ⇒ Algorithmic).

1. *If* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\mathcal{E} :: \Gamma \vdash \eta_A(R) \Lleftarrow S$ *and* $\Gamma \vdash \Delta \sqsubset A$ *and* $\Gamma \vdash S \sqsubset A$*, then* $\Delta \leqq S$.
2. *If* $\Gamma \vdash x \Rrightarrow \Delta_1$ *and* $\mathcal{E} :: \Gamma \vdash \Delta \mathbin{@} \eta_{A_1}(x) = \Delta_2$ *and* $\Gamma \vdash \Delta_1 \sqsubset A_1$ *and* $\Gamma \vdash \Delta \sqsubset \Pi x{:}A_1.\, A_2$*, then* $\Delta \mathbin{@} x{::}\Delta_1{\sqsubset}A_1 = \Delta_2$.

Proof (sketch). By induction on A, S, and $\mathcal{E}$.

Clause 1 is most easily proved by case analyzing the sort S and applying inversion to the derivation $\mathcal{E}$. The case when $S = \Pi x{::}S_1{\sqsubset}A_1.\, S_2$ appeals to the induction hypothesis at an unrelated derivation but at a smaller type, and Lemmas 3.32 and 3.33 are used to satisfy the preconditions of the induction hypotheses.

Clause 2 is most easily proved by case analyzing the derivation $\mathcal{E}$. In one case, we require the contrapositive of Theorem 3.37 (Algorithmic Subsumption) to convert a derivation of $\Gamma \nvdash \eta_{A_1}(x) \Lleftarrow S_1$ into a derivation of $\Delta_1 \nleqq S_1$.

(The full proof may be found in Appendix B.7.) □

Theorem 3.38 along with Theorem 3.11, the Completeness of Algorithmic Typing, gives us our desired result:

Theorem 3.39 (Intrinsic ⇒ Algorithmic). *If* $\Gamma, x{::}S{\sqsubset}A \vdash \eta_A(x) \Leftarrow T$*, then* $\mathrm{split}(S) \leqq T$.

Proof. Suppose $\Gamma, x{::}S{\sqsubset}A \vdash \eta_A(x) \Leftarrow T$. Then,

$\Gamma, x{::}S{\sqsubset}A \vdash x \Rrightarrow \mathrm{split}(S)$ By rule.
$\Gamma, x{::}S{\sqsubset}A \vdash \eta_A(x) \Lleftarrow T$ By Theorem 3.11 (Completeness of Alg. Typing).
$\mathrm{split}(S) \leqq T$ By Theorem 3.38.

□

Finally, we have completeness as a simple corollary:

Theorem 3.40 (Completeness of Declarative Subsorting). *If* $\Gamma, x{::}S{\sqsubset}A \vdash \eta_A(x) \Leftarrow T$, *then* $S \leq T$.

Proof. Corollary of Theorems 3.39 and 3.31. □

3.5 Summary

To be called a logical framework at all, there are certain properties a formal system must possess: decidability, substitution, identity. To claim to support "subtyping", there are certain principles a formal system must respect: closure under subsumption, the usual rules of subtyping.

In this chapter, we have justified LFR as a logical framework by establishing all of the standard important metatheoretic results. And we have justified LFR as a theory of subtyping by showing that despite only defining subsorting at base sorts, the usual principles are derived through η-expansion, including the order-theoretic distributivity properties expected of intersections. A common theme throughout our toil was structural inductions and syntactic methods, a simplification afforded by the modern canonical forms approach.

These metatheoretic results and the simplicity of their proofs demonstrate that LFR is a powerful yet practical theory, containing multitudes not obvious from its definition. In the next chapter, we leverage some of these results to explore the power of LFR further, relating it to another well-studied extension of LF, proof irrelevance.

Chapter 4

Subset Interpretation

We have seen in Chapter 2 that it is natural to use refinements to represent certain subsets of data types. Conversely, refinements can be interpreted as defining subsets. In this chapter, we take a slight detour from our primary development to exhibit an interpretation of LF refinement types which we refer to as the "*subset interpretation*". Although the interpretation will not have much bearing on the technical developments to come, it does provide an alternate understanding of refinement types which the reader may find enlightening. As such, this chapter may be regarded as an optional metatheoretic foray relating our work to previous work on proof irrelevance and subset types.

We call this interpretation the "subset interpretation" since a sort refining a type is interpreted as a predicate embodying the refinement, and the set of terms having that sort is simply the subset of terms of the refined type that also satisfy the predicate.

For a simple example, recall the signature of even and odd natural numbers:

$$\begin{array}{l} nat : \mathsf{type}. \\ z : nat. \\ s : nat \to nat. \end{array}$$

$$\begin{array}{l} even \sqsubset nat :: \mathsf{sort}. \\ odd \sqsubset nat :: \mathsf{sort}. \\ z :: even. \\ s :: even \to odd \\ \quad \wedge\ odd \to even. \end{array}$$

Under the subset interpretation, we translate the refinements *even* and *odd* to predicates on natural numbers. The natural numbers themselves remain unchanged, and the refinement declarations for *z* and *s* turn into constructors for proofs of the *even* and *odd* predicates.

$$\begin{array}{l} even : nat \to \mathsf{type}. \\ odd : nat \to \mathsf{type}. \\ \widehat{z}\ \ : even\ z. \\ \widehat{s_1} : \Pi x{:}nat.\ even\ x \to odd\ (s\ x). \\ \widehat{s_2} : \Pi x{:}nat.\ odd\ x \to even\ (s\ x). \end{array}$$

The successor function's two unrelated sorts translate to proof constructors for two different predicates.

We show that our interpretation is correct by proving, for instance, that a term N has sort S if and only if its translation $\widehat{N}$ has type $\widehat{S}(N)$, where $\widehat{S}(-)$ is the translation of the sort S into a type family representing a predicate; thus, an adequate encoding using refinement types remains adequate after translation. The chief complication in proving correctness is the dependency of types on terms, which forces us to deal with a *coherence* problem [BTCGS91, Rey91].

Normally, subset interpretations are not subject to the issue of coherence—that is, of ensuring that the interpretation of a judgment is independent of its derivation—since the terms in the target of the translation are *the same* as the terms in the source, just with the stipulation that a certain property hold of them. The proofs of these properties are computationally immaterial, so they may simply be ignored. But the presence of full dependent types in LF means that the interpretation of a sort might depend on these proofs, potentially violating the adequacy of representations.

In order to solve the coherence problem we employ *proof irrelevance*, a technique used in type theories to selectively hide the identities of terms representing proofs [Pfe01a, AB04]. In the example, the terms whose identity should be irrelevant are those constructing proofs of $odd(n)$ and $even(n)$, that is, those composed from $\widehat{z}$, $\widehat{s_1}$, and $\widehat{s_2}$.

The subset interpretation completes our intuitive understanding of refinement types as representing subsets of types. It turns out that in the presence of variable binding and dependent types, this understanding is considerably more difficult to attain than it might seem from the small example shown here.

We begin the remainder of the chapter by recapitulating briefly prior work on extending LF with proof irrelevance. We then describe our interpretation piece by piece, focusing in particular on the difficulties introduced by dependencies and subsorting. Finally, we sketch the soundness and completeness of the interpretation, obtaining as a corollary the preservation of adequacy of representations.

4.1 Proof Irrelevance

When constructive type theory is used as a foundation for verified functional programming, we notice that many parts of proofs are *computationally irrelevant*, that is, their structure does not affect the returned value we are interested in. The role of these proofs is only to guarantee that the returned value satisfies the desired specification. For example, from a proof of $\forall x{:}A.\, \exists y{:}B.\, C(x, y)$ we may choose to extract a function $f : A \to B$ such that $C(x, f(x))$ holds for every $x{:}A$, but ignore the proof that this is the case. The proof must be present, but its identity is irrelevant. Proof-checking in this scenario has to ascertain that such a proof is indeed not needed to compute the relevant result.

A similar issue arises when a type theory such as λ^{Π} is used as a logical framework. For example, assume we would like to have an adequate representation of prime numbers, that is, to have a bijection between prime numbers p and closed terms $M : primenum$. It is relatively easy to define a type family $prime : nat \to \mathsf{type}$ such that there exists a closed

$M : \mathit{prime}\ N$ if and only if N is prime. Then $\mathit{primenum} = \Sigma n{:}\mathit{nat}.\, \mathit{prime}\ n$ is a candidate (with members $\langle N, M\rangle$), but it is not actually in bijective correspondence with prime numbers unless the proof M that a number is prime is always unique. Again, we need the existence of M, but would like to ignore its identity. This can be achieved with *subset types* [C+86, SS88] $\{x{:}\mathit{nat} \mid \mathit{prime}(x)\}$ whose members are just the prime numbers p, but if the restricting predicate is undecidable then type-checking would be undecidable, which is not acceptable for a logical framework.

For LF, we further note that Σ is not available as a type constructor, so we instead introduce a new type *primenum* with exactly one constructor, *primenum/i*:

$$\begin{array}{l} \mathit{primenum} : \mathsf{type}. \\ \mathit{primenum/i} : \Pi N{:}\mathit{nat}.\, \mathit{prime}\ N \div\!\!\to \mathit{primenum}. \end{array}$$

Here the second arrow $\div\!\!\to$ represents a function that ignores the identity of its argument. The inhabitants of *primenum*, all of the form $\mathit{primenum/i}\ N\ [M]$, are now in bijective correspondence with prime numbers since $\mathit{primenum/i}\ N\ [M] = \mathit{primenum/i}\ N\ [M']$ for all M and M'.

In the extension of LF with proof irrelevance [Pfe01a, RP08], or LFI, we have a new form of hypothesis $x{\div}A$ (x has type A, but the identity of x should be irrelevant). In the non-dependent case (the only one important for the purposes of this paper), such an assumption is introduced by a λ-abstraction:

$$\frac{\Gamma, x{\div}A \vdash M \Leftarrow B}{\Gamma \vdash \lambda x.\, M \Leftarrow A \div\!\!\to B}\,.$$

We can use such variables only in places where their identity doesn't matter, e.g., in the second argument to the constructor *primenum/i* in the prime number example. More generally, we can only use it in arguments to constructor functions that do not care about the identity of their argument:

$$\frac{\Gamma \vdash R \Rightarrow A \div\!\!\to B \qquad \Gamma^{\oplus} \vdash N \Leftarrow A}{\Gamma \vdash R\ [N] \Rightarrow B}\,.$$

Here, $\Gamma^{\oplus}$ is the *promotion* operator which converts any assumption $x{\div}A$ to $x{:}A$, thereby making x usable in N. Note that there is no direct way to use an assumption $x{\div}A$.

The underlying definitional equality "=" (usually just α-conversion on canonical forms) is extended so that $R\ [N] = R'\ [N']$ if $R = R'$, no matter what N and N' are.

The substitution principle (shown here only in its simplest, non-dependent form) captures the proper typing as well as the irrelevance of assumptions $x{\div}A$:

Principle 4.1 (Irrelevant Substitution). *If $\Gamma, x{\div}A \vdash N \Leftarrow B$ and $\Gamma^{\oplus} \vdash M \Leftarrow A$ then $\Gamma \vdash [M/x]\,N \Leftarrow B$ and $[M/x]\,N = N$ (under definitional equality).*

One typical use of proof irrelevance in type theory is to render the typechecking of subset types [C+86, SS88] decidable. A subset type $\{x{:}A \mid B(x)\}$ represents the set of terms of type A which also satisfy B; typechecking is undecidable because to determine if a term M has this type, you must search for a proof of $B(M)$. One might attempt to recover decidability by using a dependent sum $\Sigma x{:}A.\,B(x)$, representing the set of terms M of type A paired with proofs of $B(M)$; typechecking is decidable, since a proof of $B(M)$ is provided, but equality of terms is overly fine-grained: if there are two proofs of $B(M)$, the two pairs will be considered unequal. Using proof irrelevance, one can find a middle ground with the type $\Sigma x{:}A.\,[B(x)]$, where $[-]$ represents the proof irrelevance modality. Type checking is decidable for such terms, since a proof of the property B is always given, but the identity of that proof is ignored, so all pairs with the same first component will be considered equal.

Our situation with the subset interpretation is similar: we would like to represent proofs of sort-checking judgments without depending on the identities of those proofs. By carefully using proof irrelevance to hide the identities of sort-checking proofs, we are able to make a translation that is sound and complete, preserving the adequacy of representations.

4.2 Overview of the Interpretation

We interpret LFR into LFI by representing sorts as predicates and derivations of sorting as proofs of those predicates. In this section, we endeavor to explain our general translation by way of examples of it in action. The translation is derivation-directed and compositional: for each judgment $\Gamma \vdash \mathcal{J}$, there is a corresponding judgment $\Gamma \vdash \mathcal{J} \rightsquigarrow X$ whose rules mimic the rules of $\Gamma \vdash \mathcal{J}$. The syntactic class of X and its precise interpretation vary from judgment to judgment. For reference, the various forms are listed in Figure 4.1, but we will explain them in turn as they arise in our examples.

Recall our simplest example of refinement types: the natural numbers, where the even and odd numbers are isolated as refinements.

$nat : \mathsf{type}.$
$z : nat.$
$s : nat \to nat.$

$even \sqsubset nat.$
$odd \sqsubset nat.$
$z :: even.$
$s :: even \to odd \;\wedge\; odd \to even.$

As described above, our translation represents *even* and *odd* as *predicates* on natural numbers, and the refinement declarations for z and s become declarations for constants for constructing proofs of those predicates.

$even : nat \to \mathsf{type}.$
$odd : nat \to \mathsf{type}.$
$\widehat{z} \;: even\ z.$
$\widehat{s_1} : \Pi x{:}nat.\ even\ x \to odd\ (s\ x).$
$\widehat{s_2} : \Pi x{:}nat.\ odd\ x \to even\ (s\ x).$

Judgment:	Result:
$\Gamma \vdash L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L}_{\text{f}}(-)$	Type of proofs of the formation family
$K \overset{\text{pred}}{\rightsquigarrow} \widehat{K}_{\text{p}}(-,-)$	Kind of the predicate family
$K \overset{\leq}{\rightsquigarrow} \widehat{K}_{\text{s}}(-,-,-,-,-)$	Type of coercions between families of kind K
$\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}(-)$	Metafunction representing predicate
$\Gamma \vdash Q \sqsubset P \Rightarrow L \rightsquigarrow \widehat{Q}$	Proof that Q is well-formed
$\Gamma \vdash N \Leftarrow S \rightsquigarrow \widehat{N}$	Proof that N has sort S
$\Gamma \vdash R \Rightarrow S \rightsquigarrow \widehat{R}$	Proof that R has sort S
$\Gamma \vdash Q_1 \leq Q_2 \rightsquigarrow F(-,-)$	Metacoercion from proofs of Q_1 to proofs of Q_2
$Q_1 \leq Q_2 \rightsquigarrow \widehat{Q_1\text{-}Q_2}$	Coercion from proofs of Q_1 to proofs of Q_2
$\vdash \Gamma\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma}$	Translated context
$\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}$	Translated signature

Figure 4.1: Judgments of the translation.

Starting simple, the proof constructor declaration for $\widehat{z}$ can be read as an assertion that the constant z satisfies a certain predicate, namely that of being even.

In fact, every sort S will have a representation as a predicate, not just the base sorts like *even* and *odd*. Generally, a predicate is just a *type* with a hole for a term; conventionally, we write the predicate representation of S as a meta-level function $\widehat{S}(-)$, and we say that a term N satisfies such a predicate if the type $\widehat{S}(N)$ is inhabited. Predicates will be the output of the sort translation judgment, $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$, which mirrors the sort formation judgment, adding a translation as an output.

For example, the predicate corresponding to the sort *even* → *odd* is the meta-function $(\Pi x{:}nat.\ even\ x \to odd\ ((-)\ x))$, and we see this predicate applied to the successor constant s in the type of the proof constructor $\widehat{s_1}$. Thus the proof constructor declaration for $\widehat{s_1}$ can *also* be read as an assertion: the constant s satisfies the predicate that, when applied to an even natural number, it yields an odd one.

Our analysis suggests a general strategy for translating a refinement type declaration: translate its sort into a predicate, and yield a declaration of a proof constructor asserting that the predicate holds of the original constant.

$$\frac{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma} \qquad c{:}A \in \Sigma \qquad \cdot \vdash_{\Sigma} S \sqsubset A \rightsquigarrow \widehat{S}}{\vdash \Sigma,\ c{::}S\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma},\ \widehat{c}{:}\widehat{S}(\eta_A(c))}$$

As a reflection of the fact that in general these predicates may be applied to arbitrary terms, not just atomic ones, we fully η-expand the constant before applying the predicate.

How do arrow sorts like *even* → *odd* translate in general? Recall that $S \to T$ is just shorthand for the dependent function sort $\Pi x{::}S.\, T$ when x does not occur in T. The general rule for translating dependent function sorts is:

$$\frac{\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S} \qquad \Gamma, x{::}S{\sqsubset}A \vdash T \sqsubset B \rightsquigarrow \widehat{T}}{\Gamma \vdash \Pi x{::}S{\sqsubset}A.\, T \sqsubset \Pi x{:}A.\, B \rightsquigarrow \boldsymbol{\lambda} N\boldsymbol{.}\ \Pi x{:}A.\, \Pi \widehat{x}{:}\widehat{S}\boldsymbol{(}\eta_A(x)\boldsymbol{)}.\, \widehat{T}\boldsymbol{(}N@x\boldsymbol{)}}\ (\Pi\textbf{-F})$$

There are two points of note in this rule. First, writing predicates as types with holes becomes cumbersome, so we instead write metafunctions explicitly using meta-level abstraction, written as a bold $\boldsymbol{\lambda}$; we continue to write meta-level application using bold **(**parens**)**. Second, since as we noted above, the term argument of a predicate is in general a canonical term, and canonical terms may not appear in application position, we appeal to an auxiliary judgment that applies a canonical term to an atomic one, $N@R = N'$. It is defined by the single clause,

$$(\lambda x.\, N)@R = [R/x]\, N,$$

where the right-hand side is an ordinary non-hereditary substitution. Now we can read the translation output as the predicate of a term N which holds if there is a function from objects $x : A$ satisfying predicate $\widehat{S}$ to proofs that N applied to x satisfies predicate $\widehat{T}$.

But what about the fact that s only had one declaration in the original signature, but there are two proof constructor declarations asserting predicates that hold of it? For compositionality's sake, we would like to translate the single refinement declaration for s into a single proof constructor declaration, but one that can effectively serve the roles of both $\widehat{s_1}$ and $\widehat{s_2}$. To this end, we use a product type.

$\widehat{s} : (\Pi x{:}nat.\ even\ x \rightarrow odd\ (s\ x))$
$\times\ (\Pi x{:}nat.\ odd\ x \rightarrow even\ (s\ x)).$

Now $\pi_i \widehat{s}$ may be used anywhere $\widehat{s_i}$ was used before. Generally, an intersection sort will translate to a conjunction of predicates, represented as a type-theoretic product. Similarly, the nullary intersection $\top$ will translate to a unit type.[1]

$$\frac{\Gamma \vdash S_1 \sqsubset A \rightsquigarrow \widehat{S_1} \qquad \Gamma \vdash S_2 \sqsubset A \rightsquigarrow \widehat{S_2}}{\Gamma \vdash S_1 \wedge S_2 \sqsubset A \rightsquigarrow \boldsymbol{\lambda} N\boldsymbol{.}\ \widehat{S_1}\boldsymbol{(}N\boldsymbol{)} \times \widehat{S_2}\boldsymbol{(}N\boldsymbol{)}}\ (\wedge\textbf{-F}) \qquad \frac{}{\Gamma \vdash \top \sqsubset A \rightsquigarrow \boldsymbol{\lambda} N\boldsymbol{.}\ 1}\ (\top\textbf{-F})$$

What kinds of proofs inhabit these predicates? Such proofs are the output of the term translation judgment $\Gamma \vdash N \Leftarrow S \rightsquigarrow \widehat{N}$, which mirrors the sort checking judgment, adding a translation as an output. Generally, a derivation that a term N has sort S will translate to a proof $\widehat{N}$ that the predicate $\widehat{S}$ holds of N (where $\widehat{S}$ is as usual the interpretation of S as a predicate), or symbolically, if $S \sqsubset A \rightsquigarrow \widehat{S}$ and $N \Leftarrow S \rightsquigarrow \widehat{N}$, then $\widehat{N} \Leftarrow \widehat{S}\boldsymbol{(}N\boldsymbol{)}$—ignoring for a moment the question of what happens to the contexts. This

[1]Strictly speaking, this means our translation targets an extension of LFI with product and unit types. Such an extension is orthogonal to the addition of proof irrelevance, and has been studied by many people over the years, including Schürmann [Sch03] and Sarkar [Sar09]. Alternatively, products may be eliminated after translation by a simple currying transformation, but that is beyond the scope of this article.

expectation begins to hint at the soundness theorem we will demonstrate below, but for now we will use it just to guide our intuitions.

For example, since an intersection sort is represented by a product of predicates, we should expect that a term judged to have an intersection sort should translate to a proof of a product, or a pair. Similarly, since the sort $\top$ translates to a trivially true unit predicate, a term judged to have sort $\top$ should translate to a trivial unit element.

$$\frac{\Gamma \vdash N \Leftarrow S_1 \rightsquigarrow \widehat{N_1} \qquad \Gamma \vdash N \Leftarrow S_2 \rightsquigarrow \widehat{N_2}}{\Gamma \vdash N \Leftarrow S_1 \wedge S_2 \rightsquigarrow \langle \widehat{N_1}, \widehat{N_2} \rangle}\ (\wedge\textbf{-I}) \qquad \frac{}{\Gamma \vdash N \Leftarrow \top \rightsquigarrow \langle\rangle}\ (\top\textbf{-I})$$

Intuitively, knowing that a term has an intersection sort $S_1 \wedge S_2$ gives us two pieces of information about it, while knowing that a term has sort $\top$ tells us nothing new. This aspect of our translation is similar in spirit to Liquori and Ronchi Della Rocca's $\Lambda^{\mathrm{t}}_{\wedge}$ [LRDR07], a Church-style type system for intersections in which derivations are explicitly represented as proofs and intersections as products, though in their setting the proofs are viewed as part of a program rather than the output of a translation.

We can similarly intuit the appropriate proof for an implication predicate by examining the rule for translating $\Pi x{::}S.\,T$ above. We start from the sort-checking rule Π**-I**, which shows that a term $\lambda x.\,N$ has sort $\Pi x{::}S.\,T$. To prove that the corresponding Π predicate holds of $\lambda x.\,N$, we will have to produce a function taking an object x of type A and a proof that x satisfies $\widehat{S}$ and yielding a proof that $(\lambda x.\,N)@x = [x/x]\,N = N$ satisfies $\widehat{T}$. This is easily done: the translation of the body N is precisely the proof we require about N, and we wrap this in two λ-abstractions to get a proof of the Π predicate.

$$\frac{\Gamma, x{::}S{\sqsubset}A \vdash N \Leftarrow T \rightsquigarrow \widehat{N}}{\Gamma \vdash \lambda x.\,N \Leftarrow \Pi x{::}S{\sqsubset}A.\,T \rightsquigarrow \lambda x.\,\lambda\widehat{x}.\,\widehat{N}}\ (\Pi\textbf{-I})$$

Careful examination of the Π**-I** rule reveals a subtlety: it is clear from our understanding of the sort-checking part of the rule that the free variables of N and T may include x, but we seem to have indicated by our λ-abstraction that the proof $\widehat{N}$ may depend not only on the variable x, but also on a variable $\widehat{x}$. Where did this second variable come from?

The answer—as hinted above—is that we have not yet specified with respect to what context the translation of a term is to be interpreted. This context should in fact be the translation of the context Γ associated with the original term N, and by convention we write it as $\widehat{\Gamma}$. The judgment translating contexts is an annotated version of the context-formation judgment, written $\vdash \Gamma\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma}$.

$$\frac{}{\vdash \cdot\ \mathsf{ctx} \rightsquigarrow \cdot} \qquad \frac{\vdash \Gamma\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma} \qquad \Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}}{\vdash \Gamma, x{::}S{\sqsubset}A\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma}, x{:}A, \widehat{x}{:}\widehat{S}\,\mathbf{(}\eta_A(x)\mathbf{)}}$$

The second rule is quite similar to the translation rule we have seen for signature declarations $c{:}A$: each declaration $x{::}S{\sqsubset}A$ splits into a typing declaration $x{:}A$ and a proof

$$\boxed{\Gamma \vdash_\Sigma R^+ \Rightarrow S^- \rightsquigarrow \widehat{R}^-}$$

$$\frac{c{::}S \in \Sigma}{\Gamma \vdash c \Rightarrow S \rightsquigarrow \widehat{c}}\ (\textbf{const}) \qquad \frac{x{::}S{\sqsubset}A \in \Gamma}{\Gamma \vdash x \Rightarrow S \rightsquigarrow \widehat{x}}\ (\textbf{var})$$

$$\frac{\Gamma \vdash R_1 \Rightarrow \Pi x{::}S_2{\sqsubset}A_2.\, S \rightsquigarrow \widehat{R_1} \qquad \Gamma \vdash N_2 \Leftarrow S_2 \rightsquigarrow \widehat{N_2} \qquad [N_2/x]^{\mathsf{s}}_{A_2}\, S = S'}{\Gamma \vdash R_1\, N_2 \Rightarrow S' \rightsquigarrow \widehat{R_1}\, N_2\, \widehat{N_2}}\ (\Pi\textbf{-E})$$

$$\frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2 \rightsquigarrow \widehat{R}}{\Gamma \vdash R \Rightarrow S_1 \rightsquigarrow \pi_1\, \widehat{R}}\ (\wedge\textbf{-E}_1) \qquad \frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2 \rightsquigarrow \widehat{R}}{\Gamma \vdash R \Rightarrow S_2 \rightsquigarrow \pi_2\, \widehat{R}}\ (\wedge\textbf{-E}_2)$$

Figure 4.2: Translation rules for atomic term sort synthesis

declaration $\widehat{x}{:}\widehat{S}(\eta_A(x))$. Now it is easily seen why the proof $\widehat{N}$ in the translation rule Π**-I** may depend on $\widehat{x}$: our soundness criterion will tell us that $\widehat{\Gamma}, x{:}A, \widehat{x}{:}\widehat{S}(\eta_A(x)) \vdash \widehat{N} \Leftarrow \widehat{T}(N)$.

There is just one sort checking rule remaining: the **switch** rule for checking an atomic term at a base sort. This rule appeals to subsorting, so we postpone discussion of it until we discuss the translation of subsorting judgments in Section 4.4. For now, the reader may think of the rule as simply returning the result of the sort synthesis translation judgment, $\Gamma \vdash R \Rightarrow S \rightsquigarrow \widehat{R}$. At the base cases, this judgment returns the hatted proof constants $\widehat{c}$ and variables $\widehat{x}$ we have seen in the translations of signature declarations and contexts. The other rules correspond to elimination forms, and they follow straightforwardly by the same intuitions we used to derive the introduction rules in the sort checking translation. All the rules for this judgment are shown in Figure 4.2.

There is also just one sort formation rule remaining: the rule for translating base sorts Q. Although this translation seems straightforward in the case of simple sorts like *even* and *odd*, it is rather subtle when it comes to dependent sort families due to a problem of coherence. To explain, we return to another early example, the doubling relation on natural numbers.

4.3 Dependent Base Sorts

Recall the double relation defined as a type family in LF:

> $\mathit{double} : \mathit{nat} \to \mathit{nat} \to \mathsf{type}.$
> $\mathit{dbl/z} : \mathit{double}\ z\ z.$
> $\mathit{dbl/s} : \Pi N{:}\mathit{nat}.\ \Pi N_2{:}\mathit{nat}.\ \mathit{double}\ N\ N_2 \to \mathit{double}\ (s\ N)\ (s\ (s\ N_2)).$

As we saw earlier, we can use LFR *refinement kinds*, or *classes*, to express and enforce the property that the second subject of any doubling relation is always even, no matter what properties hold of the first subject. To do so we define a sort *double** which is isomorphic to *double*, but has a more precise class.[2]

$$
\begin{aligned}
&\mathit{double}^* \sqsubset \mathit{double} :: \top \to \mathit{even} \to \mathsf{sort}.\\
&\mathit{dbl/z} :: \mathit{double}^*\; z\; z.\\
&\mathit{dbl/s} :: \Pi N{::}\top.\; \Pi N_2{::}\mathit{even}.\; \mathit{double}^*\; N\; N_2 \to \mathit{double}^*\; (s\; N)\; (s\; (s\; N_2)).
\end{aligned}
$$

Successfully sort-checking the declarations for *dbl/z* and *dbl/s* demonstrates that whenever $\mathit{double}^*\; M\; N$ is inhabited, the second argument, N, is even.

There is a crucial difference between refinements like *even* or *odd* and refinements like *double**: while *even* and *odd* denote particular subsets of the natural numbers, the inhabitants of the refinement $\mathit{double}^*\; M\; N$ are identical to those of the ordinary type $\mathit{double}\; M\; N$. What is important is not whether a particular instance $\mathit{double}^*\; M\; N$ is inhabited, but rather whether it is *well-formed at all*.

For this reason, we separate the *formation* of a dependent refinement type family from its *inhabitation*. Simple sorts like *even* and *odd* are always well-formed, but we would like a way to explicitly represent the formation of an indexed sort like $\mathit{double}^*\; M\; N$. Therefore, we translate *double** into two parts: a *formation family*, written $\widehat{\mathit{double}}^*$, and a *predicate family*, written using the original name of the sort, *double**.

There are two declarations involving the formation family. First, the declaration of the formation family itself:

$$\widehat{\mathit{double}}^* : \mathit{nat} \to \mathit{nat} \to \mathsf{type}.$$

The formation family has the same kind as the original refined type. Intuitively, the formation family $\widehat{\mathit{double}}^*\; M\; N$ should be inhabited whenever the sort $\mathit{double}^*\; M\; N$ would have been a well-formed sort pre-translation. For example, $\widehat{\mathit{double}}^*\; z\; z$ will be inhabited, since $\mathit{double}^*\; z\; z$ was a well-formed sort.

Next, we have a constructor for the formation family:

$$\widehat{\mathit{double}}^*/i : \Pi x{:}\mathit{nat}.\; \Pi y{:}\mathit{nat}.\; \mathit{even}\; y \to \widehat{\mathit{double}}^*\; x\; y.$$

The constructor takes all the arguments to *double** along with evidence that they have the appropriate sorts and yields a member of the formation family, i.e., a proof that *double** applied to those arguments was well-formed pre-translation. For example, $\widehat{\mathit{double}}^*/i\; z\; z\; \widehat{z}$ is a proof that $\mathit{double}^*\; z\; z$ was well-formed, since it contains the necessary evidence: a proof that the second argument z is *even*.

Finally, we have a declaration for the predicate family itself:

$$\mathit{double}^* : \Pi x{:}\mathit{nat}.\; \Pi y{:}\mathit{nat}.\; \widehat{\mathit{double}}^*\; x\; y \dashrightarrow \mathit{double}\; x\; y \to \mathsf{type}.$$

For any M and N, the predicate family will be inhabited by proofs that derivations of $\mathit{double}\; M\; N$ have the refinement $\mathit{double}^*\; M\; N$, provided that $\mathit{double}^*\; M\; N$ is well-formed in the first place. In our doubling example, all derivations of $\mathit{double}\; M\; N$ satisfy the refinement $\mathit{double}^*\; M\; N$, so the predicate family will have one inhabitant for each of them.

[2]Earlier, we used the name *double* for both the type family and the sort family refining it, but in what follows it will be important to distinguish the two.

As before, these inhabitants come from the translation of the refinement declarations for *dbl/z* and *dbl/s*. Writing arguments in irrelevant position in [*square brackets*], we get:

$$
\begin{array}{l}
\widehat{dbl/z} : \mathit{double^*}\ z\ z \\
\qquad [\ \widehat{\mathit{double}}{}^*\!/i\ z\ z\ \widehat{z}\ ] \\
\qquad \mathit{dbl/z}. \\
\widehat{dbl/s} : \Pi N{:}\mathit{nat}.\ \Pi N2{:}\mathit{nat}.\ \Pi \widehat{N2}{:}\mathit{even}\ N2.\ \Pi D{:}\mathit{double}\ N\ N2. \\
\qquad \mathit{double^*}\ N\ N2\ [\ \widehat{\mathit{double}}{}^*\!/i\ N\ N2\ \widehat{N2}\ ]\ D \\
\quad \rightarrow \mathit{double^*}\ (s\ N)\ (s\ (s\ N2)) \\
\qquad [\ \widehat{\mathit{double}}{}^*\!/i\ (s\ N)\ (s\ (s\ N2))\ (\widehat{s}_2\ (s\ N2)\ (\widehat{s}_1\ N2\ \widehat{N2}))\ ] \\
\qquad (\mathit{dbl/s}\ N\ N2\ D).
\end{array}
$$

As is evident even from this short and abbreviated example, the interpretation leads to a significant blowup in the size and complexity of a signature, underscoring the importance of a primitive understanding of refinement types.

Note that in the declaration of the predicate family *double**, the proof of well-formedness is made irrelevant using a proof-irrelevant function space $A \div\!\!\to B$, representing functions from A to B that are insensitive to the identity of their argument. Using irrelevance ensures that a given sort has a unique translation, up to equivalence. We elaborate on this below.

Generalizing from the above example, a sort declaration translates into three declarations: one for the *formation family*, one for the *proof constructor* for the formation family, and one for the *predicate family*.

$$
\frac{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma} \qquad a{:}K \in \Sigma \qquad \cdot \vdash_\Sigma L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L}_\mathrm{f} \qquad K \overset{\text{pred}}{\rightsquigarrow} \widehat{K}_\mathrm{p}}{\vdash \Sigma, s{\sqsubset}a{::}L\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma},\ \widehat{s}{:}K,\ \widehat{s}/i{:}\widehat{L}_\mathrm{f}(\widehat{s}),\ s{:}\widehat{K}_\mathrm{p}(\widehat{s}, a)}
$$

The class formation judgment $\Gamma \vdash L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L}_\mathrm{f}$ yields a metafunction describing the type of proofs of formation family, while an auxiliary kind translation judgment $K \overset{\text{pred}}{\rightsquigarrow} \widehat{K}_\mathrm{p}$ yields a metafunction describing the kind of the predicate family. As in the example, the kind of the formation family is the same as the kind of the refined type, K.

The metafunction $\widehat{L}_\mathrm{f}$ takes as input the formation family so far, initially just $\widehat{s}$. The translation of Π classes adds an argument, and the base case returns the formation family so constructed.

$$
\frac{\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S} \qquad \Gamma, x{::}S{\sqsubset}A \vdash L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L}}{\Gamma \vdash \Pi x{::}S{\sqsubset}A.\, L \sqsubset \Pi x{:}A.\, K \overset{\text{form}}{\rightsquigarrow} \lambda Q_\mathrm{f}.\ \Pi x{:}A.\ \Pi \widehat{x}{:}\widehat{S}(\eta_A(x)).\ \widehat{L}(Q_\mathrm{f}\ \eta_A(x))}
$$

$$
\frac{}{\Gamma \vdash \mathsf{sort} \sqsubset \mathsf{type} \overset{\text{form}}{\rightsquigarrow} \lambda Q_\mathrm{f}.\ Q_\mathrm{f}}
$$

Employing a similar trick as we did with intersection sorts, we will translate intersection and $\top$ classes to unit and product types.

$$\frac{\Gamma \vdash L_1 \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L_1} \qquad \Gamma \vdash L_2 \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L_2}}{\Gamma \vdash L_1 \wedge L_2 \sqsubset K \overset{\text{form}}{\rightsquigarrow} \boldsymbol{\lambda} Q_{\mathrm{f}}\boldsymbol{.}\, \widehat{L_1}\boldsymbol{(}Q_{\mathrm{f}}\boldsymbol{)} \times \widehat{L_2}\boldsymbol{(}Q_{\mathrm{f}}\boldsymbol{)}} \qquad \frac{}{\Gamma \vdash \top \sqsubset K \overset{\text{form}}{\rightsquigarrow} \boldsymbol{\lambda} Q_{\mathrm{f}}\boldsymbol{.}\, 1}$$

Intersection classes give multiple ways for a sort to be well-formed, and a product of formation families gives multiple ways to project out a proof of well-formedness.

The metafunction $\widehat{K}_{\mathrm{p}}$ takes two arguments: one for the formation family so far (initially $\widehat{s}$) and one for the refined type so far (initially a). The rule for Π kinds just adds an argument to each:

$$\frac{K \overset{\text{pred}}{\rightsquigarrow} \widehat{K}}{\Pi x{:}A.\, K \overset{\text{pred}}{\rightsquigarrow} \boldsymbol{\lambda(}Q_{\mathrm{f}}\boldsymbol{,} P\boldsymbol{).}\ \Pi x{:}A.\, \widehat{K}\boldsymbol{(}Q_{\mathrm{f}}\ \eta_A(x)\boldsymbol{,}\ P\ \eta_A(x)\boldsymbol{)}}$$

while the translation is really characterized by its behavior on the base kind, type:

$$\frac{}{\mathsf{type} \overset{\text{pred}}{\rightsquigarrow} \boldsymbol{\lambda(}Q_{\mathrm{f}}\boldsymbol{,} P\boldsymbol{).}\ Q_{\mathrm{f}} \div\!\!\to P \to \mathsf{type}}$$

The kind of the predicate family for a base sort Q refining P is essentially a one-place judgment on terms of type P, along with an irrelevant argument belonging to the formation family of Q.

Finally, we are able to make sense of the rule for translating base sorts:

$$\frac{\Gamma \vdash Q \sqsubset P' \Rightarrow L \rightsquigarrow \widehat{Q} \qquad P' = P \qquad L = \mathsf{sort}}{\Gamma \vdash Q \sqsubset P \rightsquigarrow \boldsymbol{\lambda} N\boldsymbol{.}\ Q\ [\widehat{Q}]\ N}\ (Q\text{-}\mathbf{F})$$

The class synthesis translation judgment $\Gamma \vdash Q \sqsubset P \Rightarrow L \rightsquigarrow \widehat{Q}$ (similar to the sort synthesis judgment; see Figure 4.3) yields a proof of Q's formation family; thus the predicate for a base sort Q, given an argument N, is simply the predicate family Q applied to an irrelevant proof $\widehat{Q}$ that Q is well-formed and the argument itself, N.

What if we hadn't made the proofs of formation irrelevant? Then if there were more than one proof that Q were well-formed, a soundness problem would arise. To see how, let us return to the doubling example. Imagine extending our encoding of natural numbers with a sort distinguishing zero as a refinement.

$\quad zero \sqsubset nat.$
$\quad z :: even\ \wedge\ zero.$

As with *even* and *odd*, the sort *zero* turns into a predicate. Now that z has two sorts, it translates to two proof constructors.[3]

[3]For the sake of simplicity, we will continue our example with the slightly unfaithful assumptions we've been making all along. Strictly speaking, *zero* should also have a formation family with a single trivial member, and the two declarations $\widehat{z}_1$ and $\widehat{z}_2$ should be one declaration of product type. The point we wish to make will be the same nonetheless.

$$\boxed{\Gamma \vdash_\Sigma Q^+ \sqsubset P^- \Rightarrow L^- \rightsquigarrow \widehat{Q}^-}$$

$$\frac{s{\sqsubset}a{::}L \in \Sigma}{\Gamma \vdash s \sqsubset a \Rightarrow L \rightsquigarrow \widehat{s}/i}$$

$$\frac{\Gamma \vdash Q \sqsubset P \Rightarrow \Pi x{::}S{\sqsubset}A.\, L \rightsquigarrow \widehat{Q} \qquad \Gamma \vdash N \Leftarrow S \rightsquigarrow \widehat{N} \qquad [N/x]^{\mathrm{l}}_A\, L = L'}{\Gamma \vdash Q\, N \sqsubset P\, N \Rightarrow L' \rightsquigarrow \widehat{Q}\, N\, \widehat{N}}$$

$$\frac{\Gamma \vdash Q \sqsubset P \Rightarrow L_1 \wedge L_2 \rightsquigarrow \widehat{Q}}{\Gamma \vdash Q \sqsubset P \Rightarrow L_1 \rightsquigarrow \pi_1\, \widehat{Q}} \qquad \frac{\Gamma \vdash Q \sqsubset P \Rightarrow L_1 \wedge L_2 \rightsquigarrow \widehat{Q}}{\Gamma \vdash Q \sqsubset P \Rightarrow L_2 \rightsquigarrow \pi_2\, \widehat{Q}}$$

Figure 4.3: Translation rules for base sort class synthesis

$zero : nat \to \mathsf{type}.$
$\widehat{z}_1 : even\ z.$
$\widehat{z}_2 : zero\ z.$

Next, we can observe that zero always doubles to itself and augment the declaration of *double** using an intersection class:

$$\begin{aligned} double^* \sqsubset double :: &\ \top \to even \to \mathsf{sort} \\ &\wedge\ zero \to zero \to \mathsf{sort}. \end{aligned}$$

After translation, since there are potentially two ways for $double^*\ x\ y$ to be well-formed, there are two introduction constants for the formation family.

$\widehat{double}^*/i_1 : \Pi x{:}nat.\ \Pi y{:}nat.\ even\ y \to \widehat{double}^*\ x\ y.$
$\widehat{double}^*/i_2 : \Pi x{:}nat.\ zero\ x \to \Pi y{:}nat.\ zero\ y \to \widehat{double}^*\ x\ y.$

The declarations for $\widehat{double}^*$ and $double^*$ remain the same.

Now recall the refinement declaration for doubling zero,

$dbl/z :: double^*\ z\ z\ ,$

and observe that it is valid for two reasons, since $double^*\ z\ z$ is well-formed for two reasons. Consequently, after translation, there will be two proofs inhabiting the formation family $\widehat{double}^*\ z\ z$, but only one of them will be used in the translation of the *dbl/z* declaration. Supposing it is the first one, we'll have

$\widehat{dbl/z} : double^*\ z\ z\ [\ \widehat{double}^*/i_1\ z\ z\ \widehat{z}_1\ ]\ dbl/z\ ,$

but our soundness criterion will still require that the constant $\widehat{dbl/z}$ check at the type $double^*\ z\ z\ [\ \widehat{double}^*/i_2\ z\ \widehat{z}_2\ z\ \widehat{z}_2\ ]\ dbl/z$, the other possibility. The apparent mismatch is resolved by the fact that the formation proofs are irrelevant, and so the two types are considered equal. Without proof irrelevance, the two types would be distinct and we would have a counterexample to the soundness theorem (Theorem 4.2) we prove below.

4.4 Subsorting

We now return to the question of how the translation handles subsorting. Recall that an LFR signature can include subsorting declarations between sort family constants, $s_1 \leq s_2$. For instance, continuing with our running example of the natural numbers, we might note that any *nat* that is *zero* is *even* by declaring:

$$\mathit{zero} \leq \mathit{even}.$$

Such a declaration may seem redundant, since the only thing declared to have sort *zero* has *already* been declared to have sort *even*, but it may be necessary given the inherently open-ended nature of an LF signature. We may find ourselves later in a situation where we have a new hypothesis $x : \mathit{zero}$, and without the inclusion, we would not be able to conclude that $x : \mathit{even}$. For example, the derivation of $\cdot \vdash \lambda x.\, x \Leftarrow \mathit{zero} \to \mathit{even}$ requires the inclusion to satisfy the second premise of the **switch** rule.

$$\dfrac{\dfrac{\dfrac{}{x{:}\mathit{zero} \vdash x \Rightarrow \mathit{zero}}\,\textbf{var} \qquad \dfrac{\mathit{zero} {\leq} \mathit{even} \in \Sigma}{\mathit{zero} \leq \mathit{even}}}{x{:}\mathit{zero} \vdash x \Leftarrow \mathit{even}}\,\textbf{switch}}{\cdot \vdash \lambda x.\, x \Leftarrow \mathit{zero} \to \mathit{even}}\,\Pi\textbf{-I}$$

How should we translate that derivation into a proof? As we saw earlier, the representation of $\mathit{zero} \to \mathit{even}$ as a predicate is $\boldsymbol{\lambda} N.\, \Pi x{:}\mathit{nat}.\, \mathit{zero}\ x \to \mathit{even}\ (N \mathbin{@} x)$, and applying this predicate to $\lambda x.\, x$ yields the type we need the proof to have: $\Pi x{:}\mathit{nat}.\, \mathit{zero}\ x \to \mathit{even}\ x$. It is not much of a leap of the imagination to see that one solution is simply to posit a constant of the appropriate type, an explicit coercion from proofs of "zero-ness" to proofs of "even-ness":

$$\mathit{zero\text{-}even} : \Pi x{:}\mathit{nat}.\, \mathit{zero}\ x \to \mathit{even}\ x.$$

Now the translation of $\lambda x.\, x \Leftarrow \mathit{zero} \to \mathit{even}$ can be simply the η-expansion of this constant: $\lambda x.\, \lambda \widehat{x}.\, \mathit{zero\text{-}even}\ x\ \widehat{x}$. This makes intuitive sense: the constant *zero-even* witnesses the meaning of the declaration $\mathit{zero} \leq \mathit{even}$ under the subset interpretation.

Our example leads us to a rule: a subsorting declaration $s_1 {\leq} s_2$ will translate into a declaration for a coercion constant $s_1\text{-}s_2$.

$$\dfrac{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma} \qquad s_1 {\sqsubset} a{::}L \in \Sigma \qquad s_2 {\sqsubset} a{::}L \in \Sigma \qquad a{:}K \in \Sigma \qquad K \overset{\leq}{\rightsquigarrow} \widehat{K}_{\mathrm{s}}}{\vdash \Sigma, s_1 {\leq} s_2\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}, s_1\text{-}s_2{:}\widehat{K}_{\mathrm{s}}(a, \widehat{s_1}, s_1, \widehat{s_2}, s_2)}$$

The auxiliary judgment $K \overset{\leq}{\rightsquigarrow} \widehat{K}_{\mathrm{s}}$ yields a metafunction describing the type of proof coercions between sorts that refine a type family of kind K. The metafunction $\widehat{K}_{\mathrm{s}}$ takes five arguments: the refined type, the formation family and predicate family for the domain of the coercion, and the formation family and predicate family for the codomain of the coercion. As before, the Π translation adds an argument to each of the meta-arguments.

$$\frac{K \overset{\leq}{\rightsquigarrow} \widehat{K}}{\Pi x{:}A.\, K \overset{\leq}{\rightsquigarrow} \boldsymbol{\lambda}(P, Q_{1\mathrm{f}}, Q_1, Q_{2\mathrm{f}}, Q_2).\, \Pi x{:}A.\, \widehat{K}(P', Q_{1\mathrm{f}}', Q_1', Q_{2\mathrm{f}}', Q_2')}$$
$$\text{(where, for each } P,\ P' = P\ \eta_A(x))$$

At the base kind type, the rule outputs the type of the coercion:

$$\frac{}{\mathsf{type} \overset{\leq}{\rightsquigarrow} \boldsymbol{\lambda}(P, Q_{1\mathrm{f}}, Q_1, Q_{2\mathrm{f}}, Q_2).\, \Pi f_1{:}Q_{1\mathrm{f}}.\, \Pi f_2{:}Q_{2\mathrm{f}}.\, \Pi x{:}P.\, Q_1\ [f_1]\ x \to Q_2\ [f_2]\ x}$$

Essentially, this is the type of coercions, given x, from proofs of $Q_1\ x$ to proofs of $Q_2\ x$, but in the general case, we must pass the predicates Q_1 and Q_2 evidence that they are well-formed, so the coercion requires formation proofs as inputs as well.

How do these coercions work? Recall that subsorting need only be defined at base sorts Q, and there, it is simply the application-compatible, reflexive, transitive closure of the declared relation. For the purposes of the translation, we employ an equivalent algorithmic formulation of subsorting. Following the inspiration of bidirectional typing, there are two judgments: a *checking* judgment that takes two base sorts as inputs and a *synthesis* judgment that takes one base sort as input and outputs another base sort that is one step higher in the subsort hierarchy.

The synthesis judgment constructs a coercion from the new coercion constants in the signature.

$$\frac{s_1 {\leq} s_2 \in \Sigma}{s_1 \leq s_2 \rightsquigarrow s_1\text{-}s_2} \qquad \frac{Q_1 \leq Q_2 \rightsquigarrow \widehat{Q_1\text{-}Q_2}}{Q_1\ N \leq Q_2\ N \rightsquigarrow \widehat{Q_1\text{-}Q_2}\ N}$$

The checking judgment, on the other hand, constructs a *meta-level* coercion between proofs of the two sorts. It is defined by two rules: a rule of reflexivity and a rule to climb the subsort hierarchy.

$$\frac{Q_1 = Q_2}{\Gamma \vdash Q_1 \leq Q_2 \rightsquigarrow \boldsymbol{\lambda}(R, R_1).\, R_1}\ (\textbf{refl})$$

$$\frac{\begin{array}{cc} Q_1 \leq Q' \rightsquigarrow \widehat{Q_1\text{-}Q'} & \Gamma \vdash Q_1 \sqsubset P \Rightarrow \mathsf{sort} \rightsquigarrow \widehat{Q_1} \\ \Gamma \vdash Q' \leq Q_2 \rightsquigarrow F & \Gamma \vdash Q' \sqsubset P \Rightarrow \mathsf{sort} \rightsquigarrow \widehat{Q'} \end{array}}{\Gamma \vdash Q_1 \leq Q_2 \rightsquigarrow \boldsymbol{\lambda}(R, R_1).\, F(R, \widehat{Q_1\text{-}Q'}\ \widehat{Q_1}\ \widehat{Q'}\ R\ R_1)}\ (\textbf{climb})$$

The reflexivity rule's metacoercion simply returns the proof it is given, while the climb rule composes the actual coercion $\widehat{Q_1\text{-}Q'}$ with the metacoercion F. Two extra premises generate the necessary formation proofs.

Finally, we have described enough of the translation to explain the rule most central to the design of LFR, the **switch** rule.

$$\frac{\Gamma \vdash R \Rightarrow Q' \rightsquigarrow \widehat{R} \qquad \Gamma \vdash Q' \leq Q \rightsquigarrow F}{\Gamma \vdash R \Leftarrow Q \rightsquigarrow F(R, \widehat{R})}\ (\textbf{switch})$$

The first premise produces a proof $\widehat{R}$ that R satisfies property Q', and the second premise generates the meta-level proof coercion that transforms such a proof into a proof that R satisfies property Q.

Having sketched the translation and the role of proof irrelevance, we now review some metatheoretic results.

4.5 Correctness

Our translation is both sound and complete with respect to the original system of LF with refinement types, and so our correctness criteria will come in two flavors.

Soundness theorems tell us that the result of a translation is well-formed. But even more importantly than telling us that our translation is on some level correct, they serve as an independent means of understanding the translation. In a sense, a soundness theorem can be read as the meta-level type of a translation judgment—a specification of its intended behavior—and just as types serve as an organizing principle for the practicing programmer, so too do soundness theorems serve the thoughtful theoretician. We explain our soundness theorems, then, not only to demonstrate the sensibility of our translation, but also to aid the reader in understanding its purpose.

In what follows, form(Q) represents the formation family for a base sort Q.

$$\text{form}(s) = \widehat{s} \qquad\qquad \text{form}(Q\ N) = \text{form}(Q)\ N$$

Theorem 4.2 (Soundness). *Suppose* $\vdash \Gamma\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma}$ *and* $\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}$. *Then:*

1. *If* $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$ *and* $\Gamma \vdash N \Leftarrow S \rightsquigarrow \widehat{N}$, *then* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{N} \Leftarrow \widehat{S}(N)$.

2. *If* $\Gamma \vdash R \Rightarrow S \rightsquigarrow \widehat{R}$, *then* $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$ *and* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{R} \Rightarrow \widehat{S}(\eta_A(R))$ *(for some* A *and* $\widehat{S}$*).*

3. *If* $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$ *and* $\Gamma \vdash N \Leftarrow A$, *then* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{S}(N) \Leftarrow \mathsf{type}$.

4. *If* $\Gamma \vdash Q \sqsubset P \Rightarrow L \rightsquigarrow \widehat{Q}$, *then for some* K, $\widehat{L}_{\mathrm{f}}$, *and* $\widehat{K}_{\mathrm{p}}$,

   - $\Gamma \vdash L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L}_{\mathrm{f}}$ *and* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{Q} \Rightarrow \widehat{L}_{\mathrm{f}}(\text{form}(Q))$, *and*
   - $K \overset{\text{pred}}{\rightsquigarrow} \widehat{K}_{\mathrm{p}}$ *and* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} Q \Rightarrow \widehat{K}_{\mathrm{p}}(\text{form}(Q), P)$.

5. *If* $\Gamma \vdash L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L_{\text{f}}}$ *and* $\Gamma \vdash P \Rightarrow K$, *then* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{L_{\text{f}}}(P) \Leftarrow \mathsf{type}$.

6. *If* $K \overset{\text{pred}}{\rightsquigarrow} \widehat{K_{\text{p}}}$, $\Gamma \vdash Q_{\text{f}} \Rightarrow K$, *and* $\Gamma \vdash P \Rightarrow K$, *then* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{K_{\text{p}}}(Q_{\text{f}}, P) \Leftarrow \mathsf{kind}$.

7. *If* $Q_1 \leq Q_2 \rightsquigarrow \widehat{Q_1\text{-}Q_2}$, $\Gamma \vdash Q_1 \sqsubset P \Rightarrow L$, $\Gamma \vdash P \Rightarrow K$, *and* $K \overset{\leq}{\rightsquigarrow} \widehat{K_{\text{s}}}$, *then* $\Gamma \vdash Q_2 \sqsubset P \Rightarrow L$ *and* $\widehat{\Gamma} \vdash \widehat{Q_1\text{-}Q_2} \Rightarrow \widehat{K_{\text{s}}}(P, \mathsf{form}(Q_1), Q_1, \mathsf{form}(Q_2), Q_2)$.

8. *If* $\Gamma \vdash R \Rightarrow P$, $\Gamma \vdash Q_i \sqsubset P \rightsquigarrow \widehat{Q_i}$, $\Gamma \vdash Q_1 \leq Q_2 \rightsquigarrow F$, *and* $\widehat{\Gamma} \vdash R_1 \Rightarrow \widehat{Q_1}(R)$, *then* $\widehat{\Gamma} \vdash F(R, R_1) \Rightarrow \widehat{Q_2}(R)$.

9. *If* $K \overset{\leq}{\rightsquigarrow} \widehat{K_s}$, $K \overset{\text{pred}}{\rightsquigarrow} \widehat{K_{\text{p}}}$, $\Gamma \vdash P \Rightarrow K$, $\Gamma \vdash Q_{i\text{f}} \Rightarrow K$, *and* $\widehat{\Gamma} \vdash Q_i \Rightarrow \widehat{K_{\text{p}}}(Q_{i\text{f}}, P)$, *then* $\widehat{\Gamma} \vdash \widehat{K_s}(P, Q_{1\text{f}}, Q_1, Q_{2\text{f}}, Q_2) \Leftarrow \mathsf{type}$.

Proof. By induction on each clause's main input derivation. Several clauses must be proved mutually; for instance, clauses 1, 2, 8, and 4 are all mutual, since the rules for translating terms refer to the translation of subsorting, the rules for translating subsorting refer to the class synthesis translation, and since sorts may be dependent, the rules for class synthesis translation refer back to the term translation. □

The proofs use entirely standard syntactic methods, but they appeal to several key lemmas about the structure of the translation.

Lemma 4.3 (Erasure). *If* $\Gamma \vdash \mathcal{J} \rightsquigarrow X$, *then* $\Gamma \vdash \mathcal{J}$.

Proof. Straightforward induction on the structure of the translation derivation. The translation rules are premise-wise strictly more restrictive than the original LFR rules, except for the subsorting rules, which are also more restrictive in the sense that they force rules to be applied in a certain order. □

Lemma 4.4 (Reconstruction). *If* $\Gamma \vdash \mathcal{J}$, *then for some* X, $\Gamma \vdash \mathcal{J} \rightsquigarrow X$.

Proof. By induction on the structure of the LFR derivation. The cases for the subsorting rules require us to demonstrate that an LFR subsorting derivation can be put into "algorithmic form", with all uses of reflexivity and transitivity outermost and right-nested, like the algorithmic translation rules **refl** and **climb**. We also make use of the tacit assumption that the judgment $\Gamma \vdash \mathcal{J}$ itself is well-formed, e.g. if $\mathcal{J} = N \Leftarrow S$, then $\Gamma \vdash S \sqsubset A$, which ensures that we will have the necessary formation premises when we need to apply the **climb** rule. □

Erasure and reconstruction substantiate the claim that our translation is derivation-directed by allowing us to move freely between translation judgments and ordinary ones. Using erasure and reconstruction, we can leverage all of the LFR metatheory without reproving it for translation judgments. For example, several cases require us to substitute into a translation derivation: we can apply erasure, appeal to LFR's substitution theorem, and invoke reconstruction to get the output we require.

But since reconstruction only gives us *some* output X, we may not know that it is the one that suits our needs. Therefore, we usually require another lemma, compositionality, to tell us that the translation commutes with substitution. There are several such lemmas; we show here the ones for sort and class translation. Note that compositionality is a purely syntactic property relating the translation and substitution, and as such, the lemma and its proof ignore the sort of the variable being substituted for, in this case x.

Lemma 4.5 (Compositionality). *Let σ denote $[M/x]_A$.*

1. *If $\Gamma_{\mathrm{L}}, x{::}\_, \Gamma_{\mathrm{R}} \vdash S \sqsubset A \rightsquigarrow \widehat{S}$ and $\Gamma_{\mathrm{L}}, \sigma\Gamma_{\mathrm{R}} \vdash \sigma S \sqsubset \sigma A \rightsquigarrow \widehat{S'}$, then $\sigma\widehat{S}(N) = \widehat{S'}(\sigma N)$ (for any N such that σN is defined),*

2. *If $\Gamma_{\mathrm{L}}, x{::}\_, \Gamma_{\mathrm{R}} \vdash L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L}$ and $\Gamma, \sigma\Gamma_{\mathrm{R}} \vdash \sigma L \sqsubset \sigma K \overset{\text{form}}{\rightsquigarrow} \widehat{L'}$, then $\sigma\widehat{L}(P) = \widehat{L'}(\sigma P)$ (for any P such that σP is defined),*

and similarly for $K \overset{\leq}{\rightsquigarrow} \widehat{K}_s$ and $K \overset{\text{pred}}{\rightsquigarrow} \widehat{K}_{\mathrm{p}}$.

Proof. Straightforward induction using functionality of hereditary substitution. The base case of the first clause leverages the irrelevance introduced in the Q-**F** translation rule: both sort formation derivations will have a premise outputting evidence for the well-formedness of the sort, and there is no guarantee they will output the *same* evidence, but since the evidence is relegated to an irrelevant position, its identity is ignored. The second clause's Π case appeals to the first clause, since Π classes contain sorts. □

Finally, there is a lemma demonstrating that proof variables only ever occur irrelevantly, so substituting for them cannot change the identity of a sort or class meta-function output by the translation.

Lemma 4.6 (Proof Variable Substitution).

1. *If $\Gamma_{\mathrm{L}}, x{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash S \sqsubset A \rightsquigarrow \widehat{S}$ then $[M/\widehat{x}]^{\mathrm{a}}_{A_0}\, \widehat{S}(N) = \widehat{S}([M/\widehat{x}]^{\mathrm{n}}_{A_0}\, N)$.*

2. *If $\Gamma_{\mathrm{L}}, x{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L}$ then $[M/\widehat{x}]^{\mathrm{a}}_{A_0}\, \widehat{L}(P) = \widehat{L}([M/\widehat{x}]^{\mathrm{p}}_{A_0}\, P)$.*

Proof. Straightforward induction, noting in the base case, the Q-**F** rule, the only term that could depend on $\widehat{x}$ is in an irrelevant position. □

Completeness theorems tell us that our target is not too rich: that everything we find evidence of in the codomain of the translation actually holds true in its domain. While important for establishing general correctness, completeness theorems are not as informative as soundness theorems, so we give here only the cases for terms—and in any case, those are the only cases we require to fulfill our goal of preserving adequacy.

In stating completeness, we syntactically isolate the set of terms that could represent proofs using metavariables $\widehat{R}$ and $\widehat{N}$.

$$\begin{aligned}
\widehat{R} &::= \widehat{c} \mid \widehat{x} \mid \widehat{R}\; N\; \widehat{N} \mid \pi_1\, \widehat{R} \mid \pi_2\, \widehat{R} \\
\widehat{N} &::= \widehat{F} \mid \lambda x.\, \lambda\widehat{x}.\, \widehat{N} \mid \langle \widehat{N_1}, \widehat{N_2} \rangle \mid \langle\rangle \\
\widehat{F} &::= \widehat{R} \mid \widehat{Q_1\text{-}Q_2}\; \widehat{Q_1}\; \widehat{Q_2}\; R\; F \\
\widehat{Q_1\text{-}Q_2} &::= s_1\text{-}s_2 \mid \widehat{Q_1\text{-}Q_2}\; N \\
\widehat{Q} &::= \widehat{s}/i \mid \widehat{Q}\; N\; \widehat{N} \mid \pi_1\, \widehat{Q} \mid \pi_2\, \widehat{Q}
\end{aligned}$$

Theorem 4.7 (Completeness). *Suppose* $\vdash \Gamma\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma}$ *and* $\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}$. *Then:*

1. *If* $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$ *and* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{N} \Leftarrow \widehat{S}(N)$, *then* $\Gamma \vdash N \Leftarrow S$.

2. *If* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{R} \Rightarrow B$, *then* $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$, $B = \widehat{S}(\eta_A(R))$, *and* $\Gamma \vdash R \Rightarrow S$ *(for some* S, A, $\widehat{S}$, *and* R*).*

3. *If* $\widehat{\Gamma} \vdash \widehat{F} \Rightarrow Q\ [\widehat{Q}]\ R$, *then* $\widehat{\Gamma} \vdash \widehat{R} \Leftarrow Q$.

4. *If* $\widehat{\Gamma} \vdash \widehat{Q_1\text{-}Q_2} \Rightarrow B$, *then* $K \overset{\le}{\rightsquigarrow} \widehat{K}_s$, $B = \widehat{K}_s(P, \mathrm{form}(Q_1), Q_1, \mathrm{form}(Q_2), Q_2)$, *and* $Q_1 \le Q_2$ *(for some* K, $\widehat{K}_s$, P, Q_1, *and* Q_2*).*

5. *If* $\widehat{\Gamma} \vdash \widehat{Q} \Rightarrow B$, *then* $\Gamma \vdash L \sqsubset K \overset{\mathrm{form}}{\rightsquigarrow} \widehat{L}_\mathrm{f}$, $B = \widehat{L}_\mathrm{f}(\mathrm{form}(Q))$, *and* $\Gamma \vdash Q \sqsubset P \Rightarrow L$ *(for some* L, K, $\widehat{L}_\mathrm{f}$, *and* Q*).*

Proof. By induction over the structure of the proof term. □

Adequacy of a representation is generally shown by exhibiting a compositional bijection between informal entities and terms of certain LFR sorts. Since we have undertaken a subset interpretation, the set of terms of any LFR sort are unchanged by translation, and so any bijective correspondence between those terms and informal entities remains after translation. Furthermore, soundness and completeness tell us that our interpretation preserves and reflects the derivability of any refinement type judgments over those terms. Thus, we have achieved our main goal: any adequate LFR representation can be translated to an adequate LFI representation.

4.6 Summary

In this chapter we have described a sound and complete way of embedding the language of LF with refinement types into the language of LF with proof irrelevance. Treating properties as predicates is an obvious idea to any practitioner of LF, but the technical subtleties involved in applying just enough proof irrelevance in just the right way to achieve the desired correspondence are significant. The comparatively high expressivity-to-cost ratio afforded by refinements makes them attractive for the subset of properties they can represent: although in some sense less powerful than proof irrelevance, refinements are more practical in many situations.

Moving forward, we will see how refinement types can be made even more practical through an account of *sort reconstruction* akin to the type reconstruction enjoyed by even the most novice user of Twelf.

Chapter 5

Sort Reconstruction

In order to be practical as a tool for formalizing deductive systems, an implementation of a logical framework like LF needs to have some form of type reconstruction to relieve the user from the burdens of specifying the types of metavariables and writing numerous redundant parameters when they are unambiguously clear from context. This need is even greater in a logical framework with refinement types like LFR since intersection sorts and classes can be used to encode a great deal of information about the various forms a judgment may take. We would like to keep the user of our framework from having to enter nearly identical versions of rules for various judgment forms.

In this chapter, we explore the problem of sort reconstruction. We start off with a brief overview of the entire process to set the stage for what is to come. Then, we describe some preliminaries necessary to fully specify the sort reconstruction problem, including a formulation of LFR known as *spine form* and an algorithm for first reconstructing the ordinary types appearing in a refinement declaration. After that, we show that the sort reconstruction problem for LFR—unlike the type reconstruction problem for LF—is decidable, at least in principle. The naive solution is wildly impractical, though, due to the exponential growth in the number of sorts that refine a type as new base refinements are introduced.

Therefore, we then move on to describe a more practical algorithm for sort reconstruction that works in two phases: first, we generate a set of constraints that must hold for a declaration to be well-sorted, and then we determine whether those constraints have an easy-to-find most general solution. The algorithm is sound but, by design, only partially complete: if it succeeds, it returns the most general reconstruction, and if no reconstruction exists, it reports an error, but the algorithm may also give up if it determines that a declaration does not have sufficient structure to induce a solution without enumerating the sorts that refine a type. In the last case, the user may make the declaration more explicit with sort annotations to help point the way to the desired reconstruction.

5.1 Overview

The Twelf implementation of LF has a permissive concrete syntax in which metavariables of a declaration may be left free, types and terms may be expressly omitted, and constants may have arbitrarily many "implicit" parameters that are expected to be inferred from context. It is the job of type reconstruction to tame this syntax, transforming it in the most general way possible into the fully-explicit syntax we have assumed for all of our metatheory [Pie10].

The type reconstruction problem for LF can be stated as follows: given a declaration with some free term variables, omitted types, and omitted terms, compute most general closed, well-formed instance of that declaration by (1) synthesizing appropriate implicit parameters for constants that require them, (2) computing instantiations for omitted types and terms, and (3) Π-quantifying over the explicit free term variables and any other variables introduced along the way. The variables quantified over represent the implicit parameters that must be synthesized when reconstructing later declarations in which the constant is used. The meaning of "most general" is that as many things as possible should be left as parameters, so that the constant declared may be used in the widest possible variety of contexts. The problem is undecidable in general due to its relationship to higher-order unification [Dow93], but in practice, the Twelf implementation is able to compute most general reconstructions for many typical examples.

The *sort reconstruction problem for LFR* can be stated as follows: given a fully type-reconstructed declaration with some omitted sorts, compute the most general instance of that declaration. The meaning of "most general" here is roughly that the declaration's parameters should be given the most general sorts possible, which will ensure that the constant declared can be used in the widest possible variety of contexts. Interestingly, this problem *is* decidable, thanks to the refinement restriction: once type reconstruction succeeds the overall structure of the solution to sort reconstruction is determined, and since there are finitely many sorts that refine any given type we can find the most general solution by enumeration. In particular, after LF type reconstruction we no longer have to perform any higher-order unification.

We begin by examining a simple example to help to illustrate the essential ideas; in the sections that follow, we will describe the phases involved more formally. Recall the signature of even and odd natural numbers:

$nat : \mathsf{type}.$
$z : nat.$
$s : nat \to nat.$

$even \sqsubset nat :: \mathsf{sort}.$
$odd \sqsubset nat :: \mathsf{sort}.$

$z :: even.$
$s :: even \to odd$
$\quad \wedge\, odd \to even.$

We can define the successor relation as a judgment. Although this is a rather trivial judgment—it has only one rule, and it is not recursive—it will serve well to highlight the key ideas of sort reconstruction.

$$\begin{aligned}&\mathit{succ} : \mathit{nat} \to \mathit{nat} \to \mathsf{type}.\\ &\mathit{succ/i} : \mathit{succ}\ X\ (s\ X).\end{aligned}$$

After LF type reconstruction, the free variable X becomes quantified at type nat.

$$\mathit{succ/i} : \Pi X{:}\mathit{nat}.\ \mathit{succ}\ X\ (s\ X).$$

In this particular example, the set of variables that must be quantified over—just X—is syntactically evident, but in general this may not be the case due to the presence of implicit dependencies in the types of free variables like X. The Twelf implementation treats such dependencies as implicit parameters of *succ* and inserts placeholders for them during the first phase of type reconstruction [Pfe91, PS99, Pie10]. It then uses a higher-order unification algorithm [DHKP96, Ree09] to determine the most general identities for the placeholders; any that remain uninstantiated become additional implicit parameters of *succ/i*.

We can define a refinement *succ′* of *succ* with a more precise class describing its parity-changing behavior:

$$\begin{aligned}\mathit{succ}' \sqsubset \mathit{succ} :: {}&\mathit{even} \to \mathit{odd} \to \mathsf{sort}\\ \wedge\ {}&\mathit{odd} \to \mathit{even} \to \mathsf{sort}.\end{aligned}$$

A user of LFR might populate the refinement *succ′* by giving a sort declaration for the *succ/i* rule:

$$\mathit{succ/i} :: \mathit{succ}'\ X\ (s\ X).$$

Since the *succ′* judgment relates even numbers to odd ones and odd numbers to even ones, the variable X may have either sort *even* or sort *odd*, and the job of sort reconstruction is to infer this fact without the user having to write down two *succ′* rules.

Before getting to the problem of sort reconstruction proper, we have to solve a problem analogous to LF type reconstruction to come up with a "fully type-reconstructed" LFR declaration, i.e., one in which all variables are explicitly quantified and all type information is present, but some sort information is omitted. Leveraging the refinement restriction and the type-reconstructed LF declaration for *succ/i*, we can construct the following fully type-reconstructed LFR declaration:

$$\mathit{succ/i} :: \Pi X{::}?{\sqsubset}\mathit{nat}.\ \mathit{succ}'\ X\ (s\ X).$$

in which the ? represents an unknown sort. This step proceeds simply by matching the incomplete LFR declaration against the type-reconstructed LF declaration that it refines, leaving a ? wherever complete sort information is unavailable.

Next, we go through the type-reconstructed declaration and fill in any unknown sorts ? with fresh sort variables like σ.

$$\mathit{succ/i} :: \Pi X{::}\sigma{\sqsubset}\mathit{nat}.\ \mathit{succ}'\ X\ (s\ X).$$

Now, we are ready to tackle the problem of sort reconstruction proper, which amounts to finding the most general sort we can fill in for the sort variables such as σ. There may be multiple solutions of incomparable generality, in which case we intersect all solutions together to yield the most precise classifier for the *succ/i* constant—the most precise classifier for a constant is the one that will allow the constant to be applied in the widest variety of contexts.

At this point, the impractical algorithm resorts to the finiteness of refinements to construct a solution by enumeration: there are only four refinements of the type *nat*, so we can fill in σ with each one in turn and see which instances are well-formed; if we intersect all of the results together, we will have the most precise possible reconstruction. The four refinements are $\top$, *even*, *odd*, and *even* $\wedge$ *odd*, and all but the first yield a well-formed sort when substituted for the sort variable σ in the above declaration:

$$\begin{array}{l} \Pi X{::}\mathit{even}{\sqsubset}\mathit{nat}.\ \mathit{succ}'\ X\ (s\ X) \\ \Pi X{::}\mathit{odd}{\sqsubset}\mathit{nat}.\ \mathit{succ}'\ X\ (s\ X) \\ \Pi X{::}\mathit{even}{\wedge}\mathit{odd}{\sqsubset}\mathit{nat}.\ \mathit{succ}'\ X\ (s\ X) \end{array}$$

The last one is redundant, though, since the two previous ones are subsorts of it. Abbreviating *succ′ X* (*s X*) as Q, we have

$$\begin{array}{l} \Pi X{::}\mathit{even}{\sqsubset}\mathit{nat}.\ Q \ \leq\ \Pi X{::}\mathit{even}{\wedge}\mathit{odd}{\sqsubset}\mathit{nat}.\ Q \ \text{ and} \\ \Pi X{::}\mathit{odd}{\sqsubset}\mathit{nat}.\ Q \ \leq\ \Pi X{::}\mathit{even}{\wedge}\mathit{odd}{\sqsubset}\mathit{nat}.\ Q\ . \end{array}$$

Thus, up to equivalence, we have the following as the most general reconstruction:

$$\begin{array}{l} \mathit{succ/i} :: \Pi X{::}\mathit{even}{\sqsubset}\mathit{nat}.\ \mathit{succ}'\ X\ (s\ X) \\ \qquad\quad \wedge\ \Pi X{::}\mathit{odd}{\sqsubset}\mathit{nat}.\ \mathit{succ}'\ X\ (s\ X). \end{array}$$

In many situations, including this example, there is a more efficient possibility than enumeration: we can perform sort checking on the fully type-reconstructed declaration, generating a constraint that the sort variables must obey for sort checking to succeed. Then finding the most general solution to that constraint corresponds to finding the most general reconstruction. Here, we generate the constraint that either $\sigma \leq \mathit{even}$ or $\sigma \leq \mathit{odd}$, a constraint that has two incomparable "most general" solutions, namely $\sigma = \mathit{even}$ and $\sigma = \mathit{odd}$.[1] Instantiating and intersecting the two solutions together, we arrive at the same solution as the impractical algorithm above.

In the following sections, we explain formally the above-outlined ideas. First we give a formal account of the entire top-level signature reconstruction process (Section 5.2). Then, we take a brief detour to describe some preliminary notions, including a term representation known as *spine form* (Section 5.3.1) which we will be using for the remainder of the chapter, the "matching" algorithm we use for LFR type reconstruction (Section 5.3.2), and the role of sort variables and how they come into existence (Section 5.3.3). Finally, we describe sort reconstruction proper, first as a complete but impractical algorithm (Section 5.4), primarily to demonstrate the decidability of the problem, and then as an incomplete algorithm which works well in practice (Section 5.5).

[1] As above, there is a third solution, $\sigma = \mathit{even} \wedge \mathit{odd}$, but this solution is "less general" than both solutions given in that it makes more commitments than necessary to resolve the constraint. We return to this point below in Section 5.5.2 when we discuss generality.

$$\boxed{\vdash \Sigma\ \mathsf{sig} \blacktriangleright \Sigma'}$$

$$\frac{}{\vdash \cdot\ \mathsf{sig} \blacktriangleright \cdot} \qquad \frac{\vdash \Sigma\ \mathsf{sig} \blacktriangleright \Sigma' \qquad \cdot \vdash_{\Sigma'} K \Leftarrow \mathsf{kind} \blacktriangleright (K', i)}{\vdash \Sigma, a{:}K\ \mathsf{sig} \blacktriangleright \Sigma', (a{:}K', i)}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \blacktriangleright \Sigma' \qquad \cdot \vdash_{\Sigma'} A \Leftarrow \mathsf{type} \blacktriangleright (A', i)}{\vdash \Sigma, c{:}A\ \mathsf{sig} \blacktriangleright \Sigma', (c{:}A', i)}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \blacktriangleright \Sigma' \qquad (a{:}K, i) \in \Sigma' \qquad \vdash^{i}_{\Sigma'} L \mathrel{\underset{\sim}{\sqsubset}} K \blacktriangleright L_1 \qquad L_1 \blacktriangleright^{-} (\Xi \vdash L_2) \qquad \Xi \vdash_{\Sigma'} L_2 \sqsubset K \blacktriangleright L'}{\vdash \Sigma, s{\sqsubset}a{::}L\ \mathsf{sig} \blacktriangleright \Sigma', (s{\sqsubset}a{::}L', i)}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \blacktriangleright \Sigma' \qquad (c{:}A, i) \in \Sigma' \qquad \vdash^{i}_{\Sigma'} S \mathrel{\underset{\sim}{\sqsubset}} A \blacktriangleright S_1 \qquad S_1 \blacktriangleright^{-} (\Xi \vdash S_2) \qquad \Xi \vdash_{\Sigma'} S_2 \sqsubset A \blacktriangleright S'}{\vdash \Sigma, c{::}S\ \mathsf{sig} \blacktriangleright \Sigma', (c{::}S', i)}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \blacktriangleright \Sigma' \qquad (s_1{\sqsubset}a{::}L, i) \in \Sigma' \qquad (s_2{\sqsubset}a{::}L, i) \in \Sigma'}{\vdash \Sigma, s_1{\leq}s_2\ \mathsf{sig} \blacktriangleright \Sigma', s_1{\leq}s_2}$$

Figure 5.1: Signature reconstruction.

5.2 Top-level Signature Reconstruction

As shown in the overview above, we reconstruct a signature one declaration at a time: for each declaration, we know that we have fully reconstructed the entirety of the signature before that declaration. For LFR declarations, we then proceed first by performing LFR type reconstruction, next by filling unknown sorts with sort variables, and finally by performing sort reconstruction proper.

The signature reconstruction judgment in Figure 5.1 makes this intuition formal. Each declaration is reconstructed only after the declarations preceding it, and with respect to those reconstructed declarations. There are a number of reconstruction judgments, which we preview briefly.

First, we assume a pair of LF type reconstruction judgments, $\Gamma \vdash_\Sigma K \Leftarrow \mathsf{kind} \blacktriangleright (K', i)$ and $\Gamma \vdash_\Sigma A \Leftarrow \mathsf{type} \blacktriangleright (A', i)$. These elaborate the classifier of an LF declaration in concrete, "implicit" syntax to fully-explicit internal form, and correspond to what is implemented in the Twelf system. Each judgment has the possibility of introducing some number i of implicit parameters, and this number of implicit parameters for each constant is recorded in its corresponding declaration in the fully reconstructed signature. The signature reconstruction rules that refer to these judgments are set in gray in Figure 5.1 to indicate that we do not explain them further in this work; for a formal treatment of LF type reconstruction, we refer the reader to Pientka [Pie10].

Next, we have two LFR type reconstruction judgments, $\vdash^{i}_{\Sigma} L \mathrel{\underset{\sim}{\sqsubset}} K \blacktriangleright L'$ and $\vdash^{i}_{\Sigma} S \mathrel{\underset{\sim}{\sqsubset}} A \blacktriangleright$

S'. These verify the refinement restriction and elaborate LFR classifiers to fully-explicit form. The LFR constants inherit their implicit parameters from the LF constants they refine, so LFR type reconstruction takes the number of implicit arguments as an input and fills in the implicit parameters based on the reconstruction of the refined declaration. These judgments are described in more detail below in Section 5.3.2.

After type reconstruction, there is a filling pass, with judgments $L \blacktriangleright^t (\Xi \vdash L')$ and $S \blacktriangleright^t (\Xi \vdash S')$, which replace unknown sorts with fresh sort variables of an appropriate *tendency*, t, either maximizing $(+)$ or minimizing $(-)$. The goal of sort reconstruction is to come up with the most precise possible classifier for each constant that is still an instance of the incomplete classifier that the user wrote down, and the tendency of a sort variable tells whether that variable is in a position that needs to be maximized or minimized in order to attain that goal, taking contravariance of functions into account. Since at the top-level we want to minimize the classifier, filling is done at tendency "$-$". The filling judgments also return a sort variable context Ξ containing all the free sort variables in the result. Filling and tendencies are described in more detail below in Section 5.3.3.

Finally, we have two LFR sort reconstruction judgments, $\Xi \vdash_\Sigma L \sqsubset K \blacktriangleright L'$ and $\Xi \vdash_\Sigma S \sqsubset A \blacktriangleright S'$. These judgments reconstruct any unknown sort information remaining in a type-reconstructed LFR classifier. They are the main focus of this chapter, and two different implementations are discussed below, the impractical one in Section 5.4 and the practical one in Section 5.5.

5.3 Preliminaries

5.3.1 Spine Form LFR

The Twelf implementation of LF does not use the technology of atomic and canonical terms we have described for its internal representation; instead, it uses an isomorphic presentation known as *spine form* after Cervesato and Pfenning's "linear spine calculus" [CP03]. The essential feature of spine form is that application has "reversed associativity": the head of an application is immediately exposed and all of the arguments follow in a list called the "spine". From a practical standpoint, spine form is useful because it allows for much more efficient implementation of operations like unification, which are frequently the bottleneck in logic programming and theorem proving applications. Cervesato and Pfenning studied spine form not only as an efficient representation technique, but also because of its connection to uniform proofs [MNPS91]. Herbelin [Her95] touched on many of the key insights in his study of a similar calculus he called the $\overline{\lambda}$-calculus, designed to extend the propositions-as-types correspondence to (a focused fragment of) Gentzen's intuitionistic sequent calculus.

Since spine form is the representation best suited for an implementation of a logical framework, we study the problem of sort reconstruction using a spine form framework. The correspondence between spine form and "spineless" atomic/canonical form is well-understood, so we omit a detailed discussion of it here. Instead, we simply describe the basic spine form framework and relate it informally to the framework we have been using until now.

As mentioned above, the essential feature of spine form is the representation of applicative terms as a *head*, which is either a variable or a constant, and a *spine*, which is a list of canonical arguments that the head is applied to. In canonical terms as we have in a logical framework, the spine must include *all* of the arguments, since heads must be fully applied to be η-long—there is no such thing as a "partial application" in canonical spine form. We have three syntactic categories: heads h, which are just variables or constants, normal terms N, which correspond to normal terms from earlier, and spines Sp, which correspond roughly to atomic terms R without their heads.

$$\begin{array}{rlr} h & ::= c \mid x & \text{heads} \\ N & ::= h \cdot Sp \mid \lambda x.\, N & \text{normal terms} \\ Sp & ::= () \mid (N; Sp) & \text{spines} \end{array}$$

Rules for sort-checking spine form terms are shown in Figure 5.2. Normal terms N are sort-checked by a judgment $\Gamma \vdash N \Leftarrow S$, which has a similar interpretation—and similar rules—to the identically-written judgment in the atomic/canonical system. Spines Sp are sort-checked by a judgment $\Gamma \vdash Sp :: S < Q$, which can be read "under context Γ, spine Sp can eliminate a term of sort S yielding a result that checks at sort Q".

The two judgments come together at the rules for checking at base sorts Q. In the atomic/canonical presentation of LF, only atomic terms may be checked at base sorts. In spine form, "atomic terms" are terms of the form $h \cdot Sp$, where h is either a constant or a variable. The rules first determine the type of h from its entry in either the signature or the context, and then check that Sp can eliminate h to yield a result that checks at sort Q.

Spines can be thought of as something like continuations, but with an explicit result type, Q. Under this reading, the nil spine () is something like the identity continuation: it can accept something of sort Q' and yield a result of sort Q, provided that $Q' \leq Q$. The spine $(N; Sp)$ corresponds to a continuation which, when given a function, first applies that function to N and then continues according to the continuation Sp. The typing rule reflects this interpretation: $(N; Sp)$ can take a function to a result sort Q as long as the argument N belongs to the function's domain and the continuation Sp can take the function's co-domain to the result Q. Finally, a continuation can accept an intersection $S_1 \wedge S_2$ as long as it can accept either conjunct.

In spine form, every rule has the property that one of its subjects becomes smaller from the conclusion to the premises. This property did not hold of atomic/canonical form: while spine form derivations always proceed from the root of the derivation towards the leaves, in atomic/canonical form the checking judgment proceeds from the root upward and the synthesis judgment proceeds from the leaves downward, and they meet in the middle at the **switch** rule. A pleasant consequence of the spine presentation's "uni-directionality" is that it can be read as a straightforward decision procedure.

Theorem 5.1 (Decidability, spine form). *Spine form sort checking is decidable.*

1. *Given Γ, N, and S, either $\Gamma \vdash N \Leftarrow S$ or $\Gamma \nvdash N \Leftarrow S$.*

$$\boxed{\Gamma \vdash N \Leftarrow S}$$

$$\frac{x{::}S{\sqsubset}A \in \Gamma \qquad \Gamma \vdash Sp :: S < Q}{\Gamma \vdash x \cdot Sp \Leftarrow Q}\ (\mathbf{var}) \qquad \frac{c{::}S \in \Sigma \qquad \Gamma \vdash Sp :: S < Q}{\Gamma \vdash c \cdot Sp \Leftarrow Q}\ (\mathbf{const})$$

$$\frac{\Gamma, x{::}S{\sqsubset}A \vdash N \Leftarrow T}{\Gamma \vdash \lambda x.\, N \Leftarrow \Pi x{::}S{\sqsubset}A.\, T}\ (\Pi\textbf{-R})$$

$$\frac{}{\Gamma \vdash N \Leftarrow \top}\ (\top\textbf{-R}) \qquad \frac{\Gamma \vdash N \Leftarrow S_1 \qquad \Gamma \vdash N \Leftarrow S_2}{\Gamma \vdash N \Leftarrow S_1 \wedge S_2}\ (\wedge\textbf{-R})$$

$$\boxed{\Gamma \vdash Sp :: S < Q}$$

$$\frac{Q' \leq Q}{\Gamma \vdash () :: Q' < Q}\ (\mathbf{switch})$$

$$\frac{[N/x]^{\mathsf{s}}_{A}\, T = T' \qquad \Gamma \vdash Sp :: T' < Q \qquad \Gamma \vdash N \Leftarrow S}{\Gamma \vdash (N; Sp) :: \Pi x{::}S{\sqsubset}A.\, T < Q}\ (\Pi\textbf{-L})$$

$$\frac{\Gamma \vdash Sp :: S_1 < Q}{\Gamma \vdash Sp :: S_1 \wedge S_2 < Q}\ (\wedge\textbf{-L}_1) \qquad \frac{\Gamma \vdash Sp :: S_2 < Q}{\Gamma \vdash Sp :: S_1 \wedge S_2 < Q}\ (\wedge\textbf{-L}_2)$$

Figure 5.2: The main judgments of LFR in spine form.

2. *Given* Γ, Sp, S, *and* Q, *either* $\Gamma \vdash Sp :: S < Q$ *or* $\Gamma \nvdash Sp :: S < Q$.

Proof. Straightforward lexicographic induction on N and S (for clause 1), and Sp and S (for clause 2). □

To give a hint of the equivalence between spine form and atomic/canonical form, we briefly sketch the translations between the two.

$$\begin{aligned}
\mathsf{sp}(\lambda x.\, N) &= \lambda x.\, \mathsf{sp}(N) & \mathsf{ac}(\lambda x.\, N) &= \lambda x.\, \mathsf{ac}(N)\\
\mathsf{sp}(R) &= \mathsf{sp}'(R, ()) & \mathsf{ac}(h \cdot Sp) &= \mathsf{ac}'(h, Sp)\\
\mathsf{sp}'(h, Sp) &= h \cdot Sp & \mathsf{ac}'(R, ()) &= R\\
\mathsf{sp}'(R\,N, Sp) &= \mathsf{sp}'(R, (\mathsf{sp}(N); Sp)) & \mathsf{ac}'(R, (N; Sp)) &= \mathsf{ac}'(R\,\mathsf{ac}(N), Sp)
\end{aligned}$$

The function $\mathsf{sp}(-)$ translates atomic/canonical form to spine form, and the function $\mathsf{ac}(-)$ translates spine form to atomic/canonical form. Both translations make use of auxiliary functions that essentially reverse the associativity of applications one argument at a time.

Theorem 5.2 (Atomic/canonical ⇒ Spine form).

1. *If* $\Gamma \vdash_{\mathsf{ac}} N \Leftarrow S$, *then* $\Gamma \vdash_{\mathsf{sp}} \mathsf{sp}(N) \Leftarrow S$.
2. *If* $\Gamma \vdash_{\mathsf{ac}} R \Rightarrow S$ *and* $\Gamma \vdash_{\mathsf{sp}} Sp :: S < Q$, *then* $\Gamma \vdash_{\mathsf{sp}} \mathsf{sp}'(R, Sp) \Leftarrow Q$.

Proof. By mutual induction on the derivations of $\Gamma \vdash_{\mathsf{ac}} N \Leftarrow S$ and $\Gamma \vdash_{\mathsf{ac}} R \Rightarrow S$. □

Theorem 5.3 (Spine form ⇒ Atomic/canonical).

If $\Gamma \vdash_{\mathsf{sp}} N \Leftarrow S$ *then* $\Gamma \vdash_{\mathsf{ac}} \mathsf{ac}(N) \Leftarrow S$.

If $\Gamma \vdash_{\mathsf{sp}} Sp :: S < Q$, *and* $\Gamma \vdash_{\mathsf{ac}} R \Rightarrow S$ *then* $\Gamma \vdash_{\mathsf{ac}} \mathsf{ac}'(R, Sp) \Leftarrow Q$.

Proof. By mutual induction on the derivations of $\Gamma \vdash_{\mathsf{sp}} N \Leftarrow S$ and $\Gamma \vdash_{\mathsf{sp}} Sp :: S < Q$. □

Spine form's regular structure also makes it easy to prove a strengthening theorem for hypotheses that are not used.

Theorem 5.4 (Strengthening, spine form).

1. *If* $\Gamma_{\mathrm{L}}, x{::}\_, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$ *and* $x \notin \mathrm{FV}(\Gamma_{\mathrm{R}}) \cup \mathrm{FV}(N) \cup \mathrm{FV}(S)$, *then* $\Gamma_{\mathrm{L}}, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$.
2. *If* $\Gamma_{\mathrm{L}}, x{::}\_, \Gamma_{\mathrm{R}} \vdash Sp :: S < Q$ *and* $x \notin \mathrm{FV}(\Gamma_{\mathrm{R}}) \cup \mathrm{FV}(Sp) \cup \mathrm{FV}(S) \cup \mathrm{FV}(Q)$, *then* $\Gamma_{\mathrm{L}}, \Gamma_{\mathrm{R}} \vdash Sp :: S < Q$.

Proof. Straightforward mutual induction. □

Since spines are something like continuations, the sort that a spine is checked at is in a contravariant position and behaves "backwards" with respect to subsumption. Using strengthening, we can partially extend Theorem 3.24 as follows:

Lemma 5.5 (Spine subsumption). *If $S \leq T$ and $\Gamma \vdash Sp :: T < Q$, then $\Gamma \vdash Sp :: S < Q$.*

Proof. Suppose $S \leq T$ and $\Gamma \vdash Sp :: T < Q$. By weakening, $\Gamma, x{::}T \vdash Sp :: T < Q$ and then by rule **var**, $\Gamma, x{::}T \vdash x \cdot Sp \Leftarrow Q$. By Theorem 3.24, $\Gamma, x{::}S \vdash x \cdot Sp \Leftarrow Q$. Then by inversion, $\Gamma, x{::}S \vdash Sp :: S < Q$ and by strengthening, $\Gamma \vdash Sp :: S < Q$. □

Recall that when we presented atomic/canonical form, we gave a "mode" interpretation of the two judgments: we said that the checking judgment $\Gamma \vdash N \Leftarrow S$ treated Γ, N, and S as inputs, while the synthesis judgment $\Gamma \vdash R \Rightarrow S$ treated Γ and R as inputs but produced S as an output. In the presentation of spine form we give here, all subjects of *both* judgments are considered inputs, a design choice which bears some discussion.

Often in spine form calculi, the result type of the spine-checking judgment $\Gamma \vdash Sp :: S < Q$ is considered an *output*.[2] If we had made that choice, the subsort check would move from the **switch** rule to the **var** and **const** rules: instead of performing the subsort check when checking the nil spine, we would perform it after synthesizing a type from an atomic term $h \cdot S$. Such an organization would seem to be more analogous to the rules we presented for atomic/canonical form, so why did we make the opposite choice when formulating our spine presentation?

The reason for our choice is efficiency: when thought of as a proof search procedure, the spine form rules can be more efficient if the result sort of the spine checking judgment is an input. Imagine checking a spine, and consider in particular the Π**-L** rule. If we search for proofs of the premises in the order we have written them in the rule, we have a chance to fail quickly in the *Sp*-checking premise before even bothering to check that the argument N is appropriate. Failing quickly in this way can have a dramatic impact on the performance of sort checking. Consider the following signature:

$$\begin{aligned} c :: {} & \Pi x_1{::}S_1. \ldots \Pi x_k{::}S_k.\ Q' \\ & \wedge \Pi x_1{::}T_1. \ldots \Pi x_k{::}T_k.\ Q. \end{aligned}$$

Suppose we would like to check that $\cdot \vdash c \cdot (N_1; \ldots; N_k) \Leftarrow Q$, where $Q' \not\leq Q$. If, while checking the spine of arguments, we choose the first conjunct in the sort of c, then we may check all of $\cdot \vdash N_1 \Leftarrow S_1$ through $\cdot \vdash N_k \Leftarrow S_k$ before learning that all our work was for naught, since $Q' \not\leq Q$. We will then backtrack and re-check all of the N_i, now at the sorts T_i. If however we formulate the rules as we have, we will encounter failure with the first conjunct before bothering to check any of the N_i at all—a factor of two savings in time. If applications are deeply right-nested, this factor of two increase becomes an exponential increase.

For a concrete example, it is much more efficient to use the strategy suggested by our rules to check that $\cdot \vdash 64 \Leftarrow$ *even* than it would be using the alternate formulation where the spine-checking result is an output. This kind of optimization is important even in LF to avoid needless proof search and type checking, but its importance is underscored in the presence of intersections due to the nondeterminism they introduce into the sort checking process.

[2]In such presentations, the spine checking judgment is usually written $\Gamma \vdash Sp :: S > Q$, a notation which suggests that the result type of the spine is produced as an output. Since we make opposite choice here, we write the judgment suggestively as $\Gamma \vdash Sp :: S < Q$.

We have now described and justified a spine form presentation of LFR suitable as the basis of an implementation of sort reconstruction. Strictly speaking, several loose ends remain beyond what we have shown here. For instance, we have neglected to give translations for types and sorts and all of the requisite judgments, but a complete account of spine form equivalence would have to treat them. In addition, we should define hereditary substitution on spine form terms and reprove all of the relevant theorems. But the machinery for explicating these matters precisely is well-understood, so we have refrained from recapitulating it here, relying instead on intuition and a small sampling of metatheoretic results.

Interestingly, it will turn out that we don't really need substitution in our implementation anyway: thanks to the refinement restriction, after performing type reconstruction, each base sort's term indices are completely determined, and the only thing left for us to discover is the identity of its head. We shall now move on to discuss the type reconstruction process that makes this simplification possible.

5.3.2 Type Reconstruction

Just as the concrete syntax of LF accepted by the Twelf implementation allows incomplete declarations with free metavariables, implicit parameters, and omitted type or term information, the concrete syntax of LFR allows such incomplete declarations as well. Fortunately, we are able to leverage Twelf's implementation of type reconstruction heavily to fill in most of the implicit or omitted information, freeing us to focus our attention on reconstructing just the missing sort information. A familiar pattern repeats itself here: thanks to the refinement restriction, we can leverage LF-based technology essentially off-the-shelf without having to recapitulate its functionality in the context of our new, richer type theory.

In this section, we show just precisely how we leverage Twelf's type reconstruction to perform a kind of type reconstruction on incomplete LFR declarations. Our type reconstruction process serves several purposes. Primarily, it verifies a weakened form of the refinement restriction: the sort or class in any declaration must refine the corresponding type or kind. In verifying this restriction, our algorithm additionally fills in any omitted type information, determines the identity of any omitted index parameters to dependent sorts, and adds implicit arguments where necessary in both declarations and uses of constants.

Why a "weakened form" of the refinement restriction? In the rules we gave in Chapter 2, the refinement judgments $\Gamma \vdash S \sqsubset A$ and $\Gamma \vdash L \sqsubset K$ guaranteed well-formedness of the main subject by sort-checking any embedded terms. The type reconstruction process does not guarantee well-formedness in this sense—we defer sort-checking until the sort reconstruction phase. Instead, it guarantees only the "structural" part of the refinement relation: the sort produced must have an appropriate *shape* for the type it is meant to refine. At base sorts, this means that the head of the sort must refine the head of the type and their argument spines must be identical. The definition of this "weak refinement" relation is shown in Figure 5.3. Observe that it is essentially the refinement relation

$$\boxed{\vdash_\Sigma S \mathrel{\underset{\sim}{\sqsubset}} A}$$

$$\frac{s \sqsubset a :: L \in \Sigma}{\vdash s \cdot Sp \mathrel{\underset{\sim}{\sqsubset}} a \cdot Sp} \qquad \frac{\vdash S \mathrel{\underset{\sim}{\sqsubset}} A \qquad \vdash T \mathrel{\underset{\sim}{\sqsubset}} B}{\vdash \Pi x{::}S{\sqsubset}A.\, T \mathrel{\underset{\sim}{\sqsubset}} \Pi x{:}A.\, B}$$

$$\frac{\vdash S_1 \mathrel{\underset{\sim}{\sqsubset}} A \qquad \vdash S_2 \mathrel{\underset{\sim}{\sqsubset}} A}{\vdash S_1 \wedge S_2 \mathrel{\underset{\sim}{\sqsubset}} A} \qquad \frac{}{\vdash \top \mathrel{\underset{\sim}{\sqsubset}} A} \qquad \frac{}{\vdash ?_A \mathrel{\underset{\sim}{\sqsubset}} A}$$

Figure 5.3: Weak refinement for sorts.

from before stripped of the checks on the index arguments of a base sort, modulo one addition: the syntax $?_A$ represents an unknown sort refining the type A.[3]

LFR type reconstruction is spread across several judgments. The main judgments are $\vdash^i S \mathrel{\underset{\sim}{\sqsubset}} A \blacktriangleright S'$ and $\vdash S \mathrel{\underset{\sim}{\sqsubset}} A \blacktriangleright S'$, and are shown in Figure 5.4. The latter of these may be read, "match concrete-syntax sort S against internal-syntax type A, elaborating to internal-syntax sort S'." The former is similar, but it specifies that the type A contains i implicit arguments which should be added to the sort during elaboration; once i becomes zero, it appeals to the i-less judgment. Most of the rules proceed as one would expect, matching the structure of the concrete sort against that of the internal type and inserting unknowns $?_A$ whenever necessary. We call attention to two features of note: the treatment of intersections and the handling of base sorts.

First, intersections are taken care of in the implicit-arguments judgment: no implicit arguments will be inserted in front of an intersection, because the intersections are broken down eagerly. Consider a concrete declaration like $c :: p\ X \wedge q\ X$, where the constant c is known to take only one implicit parameter—clearly, this parameter must be X, but there are two ways one might reconstruct the declaration. One is:

$$c :: \Pi X{::}?.\ (p\ X \wedge q\ X).$$

The other, preferred by the rules we've given, is:

$$\begin{aligned} c :: {} & \Pi X{::}?.\ p\ X \\ & \wedge\ \Pi X{::}?.\ q\ X. \end{aligned}$$

At first glance, these two reconstructions appear equivalent because of distributivity, but remember that eventually we must fill in the unknown sorts, and the second reconstruction gives us more leeway in how we do so. Suppose $p :: s \to \mathsf{sort}$ and $q :: t \to \mathsf{sort}$, where the sorts s and t are unrelated. Then given the first type reconstruction, the best well-formed instance we can construct is:

$$c :: \Pi X{::}s{\wedge}t.\ (p\ X \wedge q\ X).$$

The metavariable X must be appropriate for both p and q. Given the second, though, we can fill in the omitted sorts as follows:

[3]We frequently omit the subscript A from $?_A$ when it is either unimportant or clear from context.

$$\boxed{\vdash^i_\Sigma S \underset{\sim}{\sqsubset} A \blacktriangleright S'}$$

$$\frac{\vdash^i S \underset{\sim}{\sqsubset} B \blacktriangleright S'}{\vdash^{i+1} S \underset{\sim}{\sqsubset} \Pi x{:}A.\, B \blacktriangleright \Pi x{::}\, ?_A \sqsubset A.\, S'} \qquad \frac{\vdash S \underset{\sim}{\sqsubset} A \blacktriangleright S'}{\vdash^0 S \underset{\sim}{\sqsubset} A \blacktriangleright S'}$$

$$\frac{}{\vdash^i \top \underset{\sim}{\sqsubset} A \blacktriangleright \top} \qquad \frac{\vdash^i S_1 \underset{\sim}{\sqsubset} A \blacktriangleright S_1' \qquad \vdash^i S_2 \underset{\sim}{\sqsubset} A \blacktriangleright S_2'}{\vdash^i S_1 \wedge S_2 \underset{\sim}{\sqsubset} A \blacktriangleright S_1' \wedge S_2'}$$

$$\boxed{\vdash_\Sigma S \underset{\sim}{\sqsubset} A \blacktriangleright S'}$$

$$\frac{\vdash S \underset{\sim}{\sqsubset} A' \blacktriangleright S' \qquad \vdash A \approxeq A' \qquad \vdash T \underset{\sim}{\sqsubset} B \blacktriangleright T'}{\vdash \Pi x{::}S{\sqsubset}A.\, T \underset{\sim}{\sqsubset} \Pi x{:}A'.\, B \blacktriangleright \Pi x{::}S'{\sqsubset}A'.\, T'} \qquad \frac{(s{\sqsubset}a{::}L, i) \in \Sigma \qquad \vdash^i Sp \approxeq Sp'}{\vdash s \cdot Sp \underset{\sim}{\sqsubset} a \cdot Sp' \blacktriangleright s \cdot Sp'}$$

$$\frac{}{\vdash \_ \underset{\sim}{\sqsubset} A \blacktriangleright ?_A}$$

Figure 5.4: Type reconstruction of LFR sorts.

$$\begin{aligned} c &:: \Pi X{::}s.\; p\; X \\ &\;\wedge \Pi X{::}t.\; q\; X. \end{aligned}$$

This second reconstruction is more precise than the first:

$$\begin{aligned} (\Pi X{::}s.\, p\; X) \wedge (\Pi X{::}t.\, q\; X) &\leq (\Pi X{::}s{\wedge}t.\, p\; X) \wedge (\Pi X{::}s{\wedge}t.\, q\; X) && \text{by } \wedge\text{-compatibility} \\ &\leq \Pi X{::}s{\wedge}t.\, (p\; X \wedge q\; X) && \text{by distributivity} \end{aligned}$$

It is for this reason that our rules and our implementation prefer to push intersections as far to the outside as possible: it allows for more precise sort reconstructions down the road. (The principle of distributing intersections outward will arise again in Section 5.4.1 in the context of enumerating refinements.)

Next, we examine the handling of base sorts. As expected, to match $s \cdot Sp$ against $a \cdot Sp'$, the head s must refine the head a. More interesting though is the handling of argument spines. There are three basic concerns: the concrete spine may contain omitted subterms _, it will lack any implicit parameters, and it might just be plain ill-typed. All three concerns can be taken care of by elaborating using the internal spine Sp', and indeed, that's what the rule does: it throws out the user's input and instead uses its own trusted internal reconstruction. This behavior may be somewhat alarming to the user, so the implementation also performs a consistency check to verify that the concrete spine matches that of the refined type. We will return to the discussion of this consistency check below, but since what we have is already sufficient for our formal development, we pause a moment to reflect.

Theorem 5.6 (Soundness, LFR type reconstruction).

1. *If* $\vdash^i S \sqsubseteq\!\!\sim A \blacktriangleright S'$, *then* $\vdash S' \sqsubseteq\!\!\sim A$.

2. *If* $\vdash S \sqsubseteq\!\!\sim A \blacktriangleright S'$ *then* $\vdash S' \sqsubseteq\!\!\sim A$.

Proof. Straightforward induction on the given derivation. This theorem holds even if we delete the consistency check from the rule for elaborating base sorts. □

Syntactically, what we've accomplished through LFR type reconstruction is to fill in all omitted information and implicit parameters except that which pertains to unknown sorts. To be clear, the external, concrete syntax includes productions for omitted types, terms, and sorts (all written as an underscore _), while the internal syntax allows only for omitted sorts (written $?_A$). Semantically, concrete terms may have free variables and potentially ill-typed subterms, while the internal elaboration is fully quantified and well-typed. At this point, we may proceed assuming the refinement restriction is in effect: sorts have the correct structure and terms have already been typechecked.

We now return to the matter of consistency checking during elaboration of base sorts. Although formally all of our problems are solved by ignoring the user's input, we would like to behave in accordance with the principle of least surprise and give some kind of error in the case that their input disagrees with our desired reconstruction. It's not enough to just check the argument spines for equality: the concrete spine may have omitted terms and lack implicit parameters. We instead have two *spine consistency* judgments, $\vdash^i Sp \cong Sp'$ and $\vdash Sp \cong Sp'$, and a *term consistency* judgment $\vdash N \cong N'$, all of whose rules are shown in Figure 5.5.

The basic spine consistency judgment $\vdash Sp \cong Sp'$ is just defined pointwise, calling out to term consistency. The i-ful judgment $\vdash^i Sp \cong Sp'$ drops i implicit arguments and then compares the rest of the spines for consistency—the implicit parameters are accounted for by type reconstruction, and the user cannot get them wrong. Term consistency $\vdash N \cong N'$ is mostly pointwise, but an omitted term is consistent with anything, and the rule for checking consistency of $c \cdot Sp$ makes sure to drop an appropriate number of implicit arguments. All consistency judgments must be relative to the signature Σ in order to have access to information about constants' implicit parameters.

As we saw above, after comparing the argument spines for consistency, the rule for matching base sorts returns an elaborated output consisting of the head of the sort applied to the internal spine from the refined type. In this way, an LFR base sort gets its implicit parameters from its refined type. But how do we match the free variables from the concrete declaration with the bound variables introduced for implicit parameters? In the formal system, the answer is α-conversion: the rule that inserts a Π for an implicit parameter can always "choose" its bound variables to match the free occurrences in the concrete input sort if they are in fact used consistently therein. We give a sample derivation to illustrate the idea, matching $p\ x$ against $\Pi x{:}A.\, b\ x$, where the first argument is implicit, relative to the signature $b : A \to \mathsf{type}$, $p \sqsubset b :: \top \to \mathsf{sort}$. For clarity, we

$$\boxed{\vdash^{i}_{\Sigma} Sp \approxeq Sp'}$$

$$\frac{\vdash^{i} Sp \approxeq Sp'}{\vdash^{i+1} Sp \approxeq (N; Sp')} \qquad \frac{\vdash Sp \approxeq Sp'}{\vdash^{0} Sp \approxeq Sp'}$$

$$\boxed{\vdash_{\Sigma} Sp \approxeq Sp'}$$

$$\frac{}{\vdash () \approxeq ()} \qquad \frac{\vdash N \approxeq N' \qquad \vdash Sp \approxeq Sp'}{\vdash (N; Sp) \approxeq (N'; Sp')}$$

$$\boxed{\vdash_{\Sigma} N \approxeq N'}$$

$$\frac{\vdash Sp \approxeq Sp'}{\vdash x \cdot Sp \approxeq x \cdot Sp'} \qquad \frac{(c{::}S, i) \in \Sigma \qquad \vdash^{i} Sp \approxeq Sp'}{\vdash c \cdot Sp \approxeq c \cdot Sp'} \qquad \frac{\vdash N \approxeq N'}{\vdash \lambda x.\, N \approxeq \lambda x.\, N'} \qquad \frac{}{\vdash \_ \approxeq N'}$$

Figure 5.5: Spine and term consistency.

abbreviate a one-element spine $(N; ())$ as just (N).

$$\cfrac{(p \sqsubset b :: \top \to \mathsf{sort}, 0) \in \Sigma \qquad \cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\vdash () \approxeq ()}}{\vdash x \cdot () \approxeq x \cdot ()} \qquad \cfrac{}{\vdash () \approxeq ()}}{\vdash (x \cdot ()) \approxeq (x \cdot ())}}{\vdash^{0} (x \cdot ()) \approxeq (x \cdot ())}}{}}{\cfrac{\cfrac{\vdash p \cdot (x \cdot ()) \sqsubseteq_{\sim} b \cdot (x \cdot ()) \blacktriangleright p \cdot (x \cdot ())}{\vdash^{0} p \cdot (x \cdot ()) \sqsubseteq_{\sim} b \cdot (x \cdot ()) \blacktriangleright p \cdot (x \cdot ())}}{\vdash^{1} p \cdot (x \cdot ()) \sqsubseteq_{\sim} \Pi x{:}A.\, b \cdot (x \cdot ()) \blacktriangleright \Pi x{::}\,?_{A}\,.\, p \cdot (x \cdot ())}}$$

Note how the bound variable is chosen such that when stripped away, the free variables from the concrete external sort match those of the internal refined type.

Implementation notes. Our implementation of LFR type reconstruction is almost identical to the judgments presented here except for the use of α-conversion to ensure consistency between concrete free variables and internal bound ones. In the implementation, we instead lazily maintain a renaming that maps free variables to the bound ones they seem to correspond to. If the consistency check ever finds something that contradicts the current renaming, then the concrete sort cannot be made to match the internal type, so we signal an error.

Two other small differences are that (1) concrete terms are actually in atomic/canonical form in the implementation, not spine form, so the consistency check must reassociate applicative terms in order to compare them, and (2) we allow η-contracted functions in the external syntax,[4] expanding them on the fly during consistency checking; a suggestive notation for the appropriate rule might look something like this:

$$\frac{\vdash R\ x \cong N}{\vdash R \cong \lambda x.\, N}$$

Observe that this rule requires us to be able to add an argument to a partial application, which is easy in the atomic/canonical form of the concrete syntax.

5.3.3 Sort Variables

LFR type reconstruction resolves all missing type and term information in an LFR declaration, but leaves markers $?_A$ in places where sort information is still missing. The next step after sort reconstruction is to replace these markers with sort variables, a step called "filling". Although this sounds like an entirely trivial operation, we treat it as a separate step because of one small but important complication: *tendencies*.

Recall that the goal of sort reconstruction is to infer the most precise possible sort or class for every constant declaration. By most precise, we mean smallest in the derived higher-subsort relation. Since functions are contravariant and metavariables wind up being implicitly quantified at the outermost level, sort reconstruction must compute the *largest* possible sort for each metavariable. This requirement agrees with our intuition from LF type recostruction, where the requirement is to compute the *most general* type for each metavariable.

Of course, since function sorts may be nested, we have to be careful to come up with the *smallest* possible sort for any sort variable that occurs as the domain of a function sort in the sort of a metavariable. The observation goes "all the way down", of course, so we must take care to record in which direction a sort variable would like to be optimized. We do so by annotating each sort variable with a *tendency* to either maximize (+) or minimize (−). We also annotate each sort variable with the type that it refines in order to allow us to leverage the refinement restriction later on. Thus sort variables are written either σ_A^+ or σ_A^-. Other metavariables we use for sort variables are τ, ρ, and ξ.

We write the filling judgment $S \blacktriangleright^t (\Xi \vdash S')$, where t stands for an arbitrary tendency and Ξ is a sort variable context containing the free sort variables generated for S'.

$$t ::= + \mid - \qquad\qquad \Xi ::= \cdot \mid \Xi, \sigma_A^t$$

Following the usual conventions, when we extend a context Ξ, σ_A^t, we tacitly assume that σ is not already in Ξ, and when we join two contexts Ξ_1, Ξ_2 we tacitly assume that

[4] Perhaps we should refer to it instead as "atomic/normal" syntax to reflect this fact.

$$\boxed{S \blacktriangleright^{t} (\Xi \vdash S')}$$

$$\frac{}{s \cdot Sp \blacktriangleright^{t} (\cdot \vdash s \cdot Sp)} \qquad \frac{S \blacktriangleright^{\sim t} (\Xi_1 \vdash S') \qquad T \blacktriangleright^{t} (\Xi_2 \vdash T')}{\Pi x{::}S{\sqsubset}A.\, T \blacktriangleright^{t} (\Xi_1, \Xi_2 \vdash \Pi x{::}S'{\sqsubset}A.\, T')}$$

$$\frac{S_1 \blacktriangleright^{t} (\Xi_1 \vdash S_1') \qquad S_2 \blacktriangleright^{t} (\Xi_2 \vdash S_2')}{S_1 \wedge S_2 \blacktriangleright^{t} (\Xi_1, \Xi_2 \vdash S_1' \wedge S_2')} \qquad \frac{}{\top \blacktriangleright^{t} (\cdot \vdash \top)} \qquad \frac{}{?_A \blacktriangleright^{t} (\sigma_A^t \vdash \sigma_A^t)}$$

Figure 5.6: Filling of sort variables.

they do not overlap. The parameter t represents the tendency of the current position in a sort, initially $-$, and the rules essentially just replace every unknown $?_A$ with a fresh sort variable of the current tendency. Since functions are contravariant, the rule for filling a function sort makes use of a tendency-flipping function $\sim t$, defined by $\sim + = -$ and $\sim - = +$. See Figure 5.6.

To capture the appropriate invariant on filling, we extend weak refinement to be with respect to a sort variable context in the obvious way, writing $\Xi \vdash S \mathrel{\underset{\sim}{\sqsubset}} A$. This judgment admits weakening.

Theorem 5.7 (Soundness, filling). *If $\vdash S \mathrel{\underset{\sim}{\sqsubset}} A$ and $S \blacktriangleright^{t} (\Xi \vdash S')$, then $\Xi \vdash S' \mathrel{\underset{\sim}{\sqsubset}} A$.*

Proof. Straightforward induction over the filling derivation, using inversion on the weak refinement derivation. To combine two inductive hypotheses, we appeal to weakening. □

Another important invariant of the filling judgment—and one that we will leverage in proving principality of our algorithm below—is that it produces a sort with the appropriate initial tendency. The judgment $\mathsf{tendency}(S, t)$ says that the complete sort S is consistent with tendency t.

$$\boxed{\mathsf{tendency}(S, t)}$$

$$\frac{}{\mathsf{tendency}(s \cdot Sp, t)} \qquad \frac{\mathsf{tendency}(S, \sim t) \qquad \mathsf{tendency}(T, t)}{\mathsf{tendency}(\Pi x{::}S.\, T, t)} \qquad \frac{}{\mathsf{tendency}(\top, t)}$$

$$\frac{\mathsf{tendency}(S_1, t) \qquad \mathsf{tendency}(S_2, t)}{\mathsf{tendency}(S_1 \wedge S_2, t)} \qquad \frac{}{\mathsf{tendency}(\sigma_A^t, t)}$$

Theorem 5.8. *If $S \blacktriangleright^{t} (\Xi \vdash S')$, then $\mathsf{tendency}(S', t)$.*

Proof. By induction on the given filling derivation. □

Both the impractical algorithm and the practical algorithm we describe below manipulate sort substitutions.

$$\theta ::= [\,] \mid [\theta,\, S/\sigma_A]$$

Substitutions are typed with sort variable contexts using the extended weak refinement judgment.

$$\frac{}{\Xi_0 \vdash [\,] : \cdot} \qquad \frac{\Xi_0 \vdash \theta : \Xi \qquad \Xi_0 \vdash S \sqsubseteq A}{\Xi_0 \vdash [\theta,\, S/\sigma_A] : \Xi, \sigma_A}$$

We elide the standard definition of applying a substitution—but note that since terms and spines do not mention sorts, it is always the case that $\theta N = N$ and $\theta Sp = Sp$. We also freely "reorder" substitutions to equivalent ones as convenient.

We are now prepared to tackle the problem of sort reconstruction proper. In the next section, we give an impractical but complete and total algorithm for sort reconstruction in order to demonstrate that the problem is decidable in principle. In the one that follows, we describe a practical algorithm that reduces the problem to one of constraint solving. We will have much to say about sort variable substitutions along the way, but we will not touch on sort variable tendencies again until Section 5.5.2 when we explain our algorithm for constraint solving and its properties.

5.4 Decidability in Principle by Enumeration

After LFR type reconstruction, an LFR declaration is a fully-quantified and explicitly-typed declaration with some sort variables representing missing sort information. Since all that is missing is some sort information, and since we know the type refined by every sort variable, if we knew there were only finitely many refinements of a type, then we could simply enumerate the possibilities and filter out the non-sensical ones. As it turns out, there *are* only finitely many refinements of a type, and describing the enumerative solution is precisely the goal of this section.

The enumerative "algorithm" is quite inefficient in practice, though, even on relatively small examples. Why do we bother studying it then? There are two essential reasons. First, the existence of an enumerative algorithm demonstrates that sort reconstruction is in principle decidable, an important contrast with the undecidability of LF type reconstruction and one which highlights the benefits of the refinement methodology. Second, the enumerative solution also serves to provide us a kind of specification for the sort reconstruction problem: since it is a complete procedure, any solution produced by our (soon to be discussed) efficient algorithm should match the solution given by enumeration.

We begin by showing that, up to equivalence of sorts, there are finitely many refinements of a type. Then, having established finiteness, we outline the enumerative algorithm and demonstrate its correctness.

5.4.1 Finiteness of Refinements

As discussed in Section 3.4, although our framework only defines subsorting at base sorts, all of the usual structural rules (Figure 3.2) are in some sense admissible, including the rules for distributing intersection sorts over function sorts. The derived higher-sort subsorting $S \leq T$ induces an equivalence: $S \equiv T$ if and only if both $S \leq T$ and $T \leq S$. Up to this notion equivalence, there are finitely many refinements of any given type.

To begin, we note that in the presence of the distributivity rules, every sort can be put into a normal form by distinguishing "basic" (non-intersection, non-$\top$) sorts from "composite" ones. The essential idea is to restrict all function codomains to be basic—any intersections in a codomain can be distributed outward by the equivalence $S \to (T_1 \wedge T_2) \equiv (S \to T_1) \wedge (S \to T_2)$ [BCDC83].

$$\begin{array}{rcll} \mathbb{S}, \mathbb{T} & ::= & Q \mid \Pi x{::}\mathbf{S}.\, \mathbb{T} & \qquad\qquad \text{basic sorts} \\ \mathbf{S}, \mathbf{T} & ::= & \mathbb{S} \mid \mathbf{S}_1 \wedge \mathbf{S}_2 \mid \top & \qquad\qquad \text{composite sorts} \end{array}$$

This normal form is well-known: it is essentially the same as the "normalized types" of Coppo, et al. [CDCV81] and the "normal type schemes" of Hindley [Hin82]. Pierce [Pie97] made use of a similar notion to demonstrate semi-decidability of subtyping in $F_\wedge$, and Reynolds [Rey96] used another to show how to compute the subtype relation of Forsythe. The embedding of general sorts into this restricted normal form is given by the following normalization function norm$(-)$, which appeals to an auxiliary function pi$(x{::}\mathbf{S}.\,-)$ to carry out the distributivities described above.

$$\boxed{\mathrm{norm}(S) = \mathbf{S}}$$

$$\begin{aligned} \mathrm{norm}(Q) &= Q \\ \mathrm{norm}(\top) &= \top \\ \mathrm{norm}(S_1 \wedge S_2) &= \mathrm{norm}(S_1) \wedge \mathrm{norm}(S_2) \\ \mathrm{norm}(\Pi x{::}S.\, T) &= \mathrm{pi}(x{::}\,\mathrm{norm}(S).\ \mathrm{norm}(T)) \end{aligned}$$

$$\boxed{\mathrm{pi}(x{::}\mathbf{S}.\,\mathbf{T}) = \mathbf{S}'}$$

$$\begin{aligned} \mathrm{pi}(x{::}\mathbf{S}.\,\mathbb{T}) &= \Pi x{::}\mathbf{S}.\,\mathbb{T} \\ \mathrm{pi}(x{::}\mathbf{S}.\,\top) &= \top \\ \mathrm{pi}(x{::}\mathbf{S}.\,\mathbf{T}_1 \wedge \mathbf{T}_2) &= \mathrm{pi}(x{::}\mathbf{S}.\,\mathbf{T}_1) \wedge \mathrm{pi}(x{::}\mathbf{S}.\,\mathbf{T}_2) \end{aligned}$$

Normalization always produces an equivalent sort.

Theorem 5.9 (Equivalence of normal forms).

1. *For all* $\mathbf{S}$ *and* $\mathbf{T}$, *we have* $\mathrm{pi}(x{::}\mathbf{S}.\,\mathbf{T}) \equiv \Pi x{::}\mathbf{S}.\,\mathbf{T}$.

2. *For all S, we have* $\mathrm{norm}(S) \equiv S$*, and*

Proof. Straightforward (non-mutual) induction on S and $\mathbf{T}$, using the rules $\top/\Pi$**-dist** and $\wedge/\Pi$**-dist** to establish the equivalences in the first clause. □

Following this normal form, we can construct a pair of functions that enumerate the basic and composite refinements of a type. Basic refinements are generated following the structure of the refined type, while composite refinements are generated by considering the n-way intersection of every possible subset of the set of basic refinements. We abuse our earlier notation slightly by conflating sets of sorts with lists of sorts, but our meaning should be clear.

$$\boxed{\mathrm{refs}_\Sigma(A) = \{\mathbf{S}_1, \ldots, \mathbf{S}_n\}}$$

$$\mathrm{refs}(A) = \{\bigwedge(\Delta) \mid \Delta \in \mathcal{P}(\mathrm{brefs}(A))\}$$

$$\boxed{\mathrm{brefs}_\Sigma(A) = \{\mathbb{S}_1, \ldots, \mathbb{S}_n\}}$$

$$\begin{aligned}\mathrm{brefs}(a \cdot Sp) &= \{s \cdot Sp \mid s \sqsubset a{::}\_ \in \Sigma\}\\ \mathrm{brefs}(\Pi x{:}A.\, B) &= \{\Pi x{::}S.\, T \mid S \in \mathrm{refs}(A), T \in \mathrm{brefs}(B)\}\end{aligned}$$

Note that the basic refinements of a base type must be generated with respect to a particular signature Σ, which we leave implicit as usual. We do not however need a context Γ, since we do not check argument spines for well-formedness at this stage—we will filter out ill-formed refinements later in the process.

Analyzing these functions gives us an upper-bound on the number of refinements of a type. For instance, given a base type a_n with n base refinements, there are no more than $2^{n \cdot 2^n}$ refinements of the type $a_n \to a_n$. This is not a tight upper-bound, though, as many subsets of brefs(A) may represent equivalent composite sorts. For example, brefs($nat \to nat$) will contain both $even \to even$ and $\top \to even$, but their intersection is equivalent to just the latter. Experiments with an implementation that filters by sort equivalence can pin down the true number of distinct refinements for small examples, and we summarize a few results in Figure 5.7. It should be clear that even the actual number of refinements of a type grows too fast to be the basis of a practical implementation.[5]

Although they are somewhat naive, our enumeration functions are complete: they generate all weak refinements of a type.

Theorem 5.10 (Completeness of refinement enumeration).

[5]A few searches of the On-Line Encyclopedia of Integer Sequences [OEI10] suggests a connection between the refinements of a type and monotone functions over power sets, but a full exploration of the connection is unnecessary for the purposes of demonstrating the infeasibility of enumeration. We therefore defer it to future work.

Type	Estimated	Actual
$a_0 \to a_0$	1	1
$a_1 \to a_1$	4	3
$a_2 \to a_2$	256	36
$a_3 \to a_3$	16,777,216	8,000

Type	Estimated	Actual
$a_0 \to a_1$	2	2
$a_1 \to a_1$	4	3
$a_2 \to a_1$	16	6
$a_3 \to a_1$	256	20
$a_4 \to a_1$	65,536	168
$a_5 \to a_1$	4,294,967,296	7,581

Figure 5.7: Estimated and actual number of refinements of various types, where the base type a_n is assumed to have n base refinements.

1. *If* $\vdash \mathbf{S} \mathrel{\underset{\sim}{\sqsubset}} A$, *then for some* $S' \equiv \mathbf{S}$, *we have* $S' \in \mathrm{refs}(A)$.

2. *If* $\vdash \mathbb{S} \mathrel{\underset{\sim}{\sqsubset}} A$, *then for some* $S' \equiv \mathbb{S}$, *we have* $S' \in \mathrm{brefs}(A)$.

Proof. By mutual induction on $\mathbf{S}$ and $\mathbb{S}$, using inversion on the given weak refinement derivation. Some notes:

1. The result follows in each case from basic facts about the power set: that it contains the empty set (case $\mathbf{S} = \top$), that it contains singleton sets (case $\mathbf{S} = \mathbb{S}$), and that it is closed under binary unions (case $\mathbf{S} = \mathbf{S}_1 \wedge \mathbf{S}_2$).

2. In the base case, inverting the weak refinement derivation gives us the fact $s \sqsubset a :: \_ \in \Sigma$ necessary to show the required result. □

Since every sort has an equivalent normal form, we have the obvious corollary.

Corollary 5.11. *If* $\vdash S \mathrel{\underset{\sim}{\sqsubset}} A$, *then for some* $S' \equiv S$, *we have* $S \in \mathrm{refs}(A)$.

5.4.2 Impractical Sort Reconstruction Algorithm

Armed with enumeration of refinements, we can formulate a procedure for solving sort reconstruction completely and exactly, though inefficiently. A high-level overview of the algorithm is as follows:

1. From an incomplete declaration, extract the sort variables and their refined types,

2. For each sort variable, enumerate all refinements of its refined type,

3. Collect the enumerated refinements to enumerate all possible grounding substitutions,

4. Try each substitution in turn and check whether it yields a well-formed instance of the declaration, and

5. Intersect together all well-formed instances, yielding the principal reconstruction.

Recall from the discussion of signature reconstruction above (Section 5.2) that declarations c::S are reconstructed by appeal to a (yet-to-be-discussed) sort reconstruction judgment, $\Xi \vdash S \sqsubset A \blacktriangleright S'$, where $\Xi \vdash S \underset{\sim}{\sqsubset} A$. The above procedure can be formally represented by the following single rule, which provides a complete characterization of the sort reconstruction judgment, though not the one we will ultimately adopt.

$$\frac{\begin{array}{c}\Xi = \{\sigma_{1\,A_1}, \ldots, \sigma_{n\,A_n}\} \\ \mathrm{refs}(A_1) = \mathcal{S}_1 \quad \ldots \quad \mathrm{refs}(A_n) = \mathcal{S}_n \\ \mathbf{\Theta} = \{[S_1/\sigma_1, \ldots, S_n/\sigma_n] \mid S_1 \in \mathcal{S}_1, \ldots, S_n \in \mathcal{S}_n\} \\ \mathcal{S} = \{\theta S \mid \theta \in \mathbf{\Theta},\ \cdot \vdash \theta S \sqsubset A\}\end{array}}{\Xi \vdash S \sqsubset A \blacktriangleright \bigwedge(\mathcal{S})}$$

The result of reconstruction is well-formed, the most basic correctness criterion we require.

Theorem 5.12 (Soundness, impractical algorithm). *If $\Xi \vdash S \sqsubset A \blacktriangleright S'$, then $\cdot \vdash S' \sqsubset A$.*

Proof. By inversion, $S' = \bigwedge(\mathcal{S})$, and by construction every $S_i \in \mathcal{S}$ has the property that $\cdot \vdash S_i \sqsubset A$. It follows by a series of applications of $\wedge$-**F** rules that $\cdot \vdash S' \sqsubset A$. □

Furthermore, the result sort is the principal reconstruction since any other conceivable grounding instance of the sort S is subsumed by it.

Theorem 5.13 (Principality, impractical algorithm). *Suppose $\Xi \vdash S \sqsubset A \blacktriangleright S'$. For any $\vdash \theta : \Xi$ such that $\cdot \vdash \theta S \sqsubset A$, we have $S' \leq \theta S$.*

Proof. By construction. □

Principality may seem like a trivial result since it merely recapitulates a property which we specifically crafted the reconstruction algorithm to have, but its implications run deep. If we recast the conclusion $S' \leq \theta S$ using one of the alternate formulations of subsorting from Theorem 3.24, it lets us conclude that in any context expecting a term of a well-formed substitution instance of S, a term of sort S' will suffice: if $\Gamma_{\mathrm{L}}, x{::}\theta S{\sqsubset}A, \Gamma_{\mathrm{R}} \vdash M \Leftarrow T$ and $\Gamma_{\mathrm{L}} \vdash N \Leftarrow S'$, then $\Gamma_{\mathrm{L}}, [N/x]^{\gamma}_{A}\, \Gamma_{\mathrm{R}} \vdash [N/x]^{\mathrm{n}}_{A} M \Leftarrow [N/x]^{\mathrm{s}}_{A} T$.

In other words, if we reconstruct a declaration c::S to c::S' using the procedure sketched above, the constant c will be maximally applicable: any place we could possibly expect it to work, it will.

5.5 Practical Sort Reconstruction

As the previous section demonstrates, the sort reconstruction problem is in principle solvable in a most general fashion. But an enumerative solution becomes infeasible quite quickly as problem size increases. Can the problem be solved efficiently, to a sufficient degree as to be useful in practice? We now show that it can be in many practical cases by giving an algorithm that terminates efficiently with one of three outcomes: either it finds the most general reconstruction, or it reports an error if no reconstruction exists, or it gives up because not enough information is available to compute the most general reconstruction efficiently.

The algorithm we give here follows a typical pattern: we sort-check a term as usual, but generate constraints when we come to questions involving sort variables, and then we find a most general solution to those constraints to determine the least commitments we must make regarding the identities of the sort variables. The ”generate and solve constraints” algorithm is essentially the algorithm used for ML type inference and LF type reconstruction, though there are a few key differences in our setting.

Chief among the differences is the nature of the constraints. In type reconstruction for languages like ML and LF, the constraints are equality constraints between types and can be solved by unification. By contrast, in our setting, the constraints are *subsorting* constraints, and so the algorithm for solving them bears little resemblance to unification. Typically in implementations of ML-like or LF-like languages, constraint generation is interleaved with unification, yielding a one-pass algorithm that seeks solutions as eagerly as possible. Here, we maintain a split between the generation of constraints and their solution, since it is difficult to solve subsorting constraints in an eager fashion.[6]

Another important difference is the existence of solutions. As discussed in the previous section, most general solutions can always be constructed for sort reconstruction problems, though perhaps not efficiently. Thus, we find ourselves in between the worlds of ML and LF: like ML, our sort reconstruction problems are always solvable, but since they may be computationally infeasible, we must adopt a partial point-of-view like LF, sometimes giving up instead of pressing on.

We first describe the constraint generation process, which effectively reduces a sort checking problem to a constraint solving problem by finding a constraint whose satisfiability is both necessary and sufficient for the sort checking problem to have a solution. Then we explain how we efficiently solve a generated constraint without resorting to enumeration, all the while making the least commitments possible. Finally, we present the full sort reconstruction algorithm itself, a composition of constraint generation and constraint solving, and we argue for its correctness based on the correctness of its components.

For clarity's sake, we restrict our attention to the case of simple sorts in this section. The generalization to full dependent types is a straightforward one thanks to the refinement restriction: the indices to any dependent sort family are completely determined by the indices of its refined type, so we just have to ensure that at the end of the day all of those indices are appropriate to the sort families they apply to. We additionally focus only on the term- and spine-checking judgments, but again, the generalization to sort and class formation judgments is not difficult.

[6]Although it has proven beneficial to eagerly *simplify* the constraints using various logical equivalences. See the “implementation notes” below for details.

5.5.1 Constraint Generation

Following Pierce's textbook [Pie02] and drawing inspiration from Pottier and Rémy's ATTAPL article on HM(X) [PR05], we present constraint generation as a pair of judgments that follow the structure of sort checking. We have two judgments, $\Xi; \Gamma \vdash N \Leftarrow S \mid C$ and $\Xi; \Gamma \vdash Sp :: S < Q \mid C$, for generating a constraint by sort checking a term or by sort checking a spine, respectively; the rules are shown in Figure 5.8. The context Ξ is intended to comprise the sort variables appearing free in the remainder of the judgment, and we maintain this property in the rules. All subjects of both judgments are thought of as inputs, except for the final generated constraint C. Note that now both general sorts S and base sorts Q may include sort variables σ.[7]

A constraint generation judgment $\Xi; \Gamma \vdash \mathcal{J} \mid C$ effectively gives meaning to a judgment $\Gamma \vdash \mathcal{J}$ that involves free sort variables among Ξ. Such an "open" judgment may or may not be derivable after substituting its sort variables by concrete sorts, but the judgment $\Xi; \Gamma \vdash \mathcal{J} \mid C$ gives us a handle on which substitutions have which effect: roughly, $\Gamma \vdash \mathcal{J}$ should hold under a substitution if and only if that substitution satisfies the constraint C. Formally, our interpretation is justified by a soundness theorem establishing the sufficiency of generated constraints and a completeness theorem establishing their minimality, both of which we describe below.

The language of constraints we generate is a finitary fragment of first-order logic, where the domain of quantification is ground sorts. We have one atomic constraint, subsorting $S_1 \leq S_2$. In addition we have conjunction, disjunction, the always true and always false constraints, implication, and universal and existential quantification.

$$C ::= S_1 \leq S_2 \mid \mathbf{tt} \mid C_1 \owedge C_2 \mid \mathbf{ff} \mid C_1 \ovee C_2 \mid C_1 \supset C_2 \mid \forall \sigma_A . C \mid \exists \sigma_A . C$$

In order to avoid confusion with the intersection sort $S_1 \wedge S_2$, we write the constraint conjunction and disjunction as $C_1 \owedge C_2$ and $C_1 \ovee C_2$.

Before discussing the semantics of constraints, we take a moment to sketch some of the intuitions behind the rules in Figure 5.8. Most of the rules are straightforward. Moderately noteworthy are the rules $\top$**-L** and $\wedge$**-L**—where before we had no spine-checking rules for the sort $\top$, now we have one that returns the always-false constraint, and where before we had two spine-checking rules for intersection sorts, we now have just one that returns a disjunction constraint. Most interesting though are the rules for checking functions and application spines at sort variables, $\rightarrow$**-R/sv** and $\rightarrow$**-L/sv**.

The rule $\rightarrow$**-L/sv** generates a constraint that must hold for an application spine $(N; Sp)$ to take a head of variable sort to some base sort. Intuitively, in order to check that an application spine $(N; Sp)$ takes some sort S to Q, the sort S must be an intersection of function sorts, one of which has the property that N is a member of its domain and that Sp takes its codomain to sort Q. This intuition is captured by an existential constraint: to check $(N; Sp)$ at a variable ρ, there must be some function sort $\sigma \rightarrow \tau$ that is above ρ, and σ and τ must satisfy the constraints required for N and Sp to be well-formed.

[7]It is a slight abuse of notation to let Q range over sort variables, but it turns out to be more convenient than troublesome.

$$\boxed{\Xi;\Gamma \vdash N \Leftarrow S \mid C}$$

$$\frac{x{::}S{\sqsubset}A \in \Gamma \qquad \Xi;\Gamma \vdash Sp :: S < Q \mid C}{\Xi;\Gamma \vdash x \cdot Sp \Leftarrow Q \mid C}\ (\textbf{var}) \qquad \frac{c{::}S \in \Sigma \qquad \Xi;\Gamma \vdash Sp :: S < Q \mid C}{\Xi;\Gamma \vdash c \cdot Sp \Leftarrow Q \mid C}\ (\textbf{const})$$

$$\frac{\Xi;(\Gamma, x{::}S) \vdash N \Leftarrow T \mid C}{\Xi;\Gamma \vdash \lambda x.\, N \Leftarrow S \to T \mid C}\ (\to\textbf{-R})$$

$$\frac{(\Xi,\sigma,\tau);(\Gamma, x{::}\sigma) \vdash N \Leftarrow \tau \mid C}{\Xi;\Gamma \vdash \lambda x.\, N \Leftarrow \rho^{t}_{A\to B} \mid \forall\sigma^{\sim t}_{A}.\, \forall\tau^{t}_{B}.\, (\rho \leq \sigma \to \tau) \supset C}\ (\to\textbf{-R/sv})$$

$$\frac{}{\Xi;\Gamma \vdash N \Leftarrow \top \mid \textbf{tt}}\ (\top\textbf{-R}) \qquad \frac{\Xi;\Gamma \vdash N \Leftarrow S_1 \mid C_1 \qquad \Xi;\Gamma \vdash N \Leftarrow S_2 \mid C_2}{\Xi;\Gamma \vdash N \Leftarrow S_1 \wedge S_2 \mid C_1 \mathbin{\textbf{∧}} C_2}\ (\wedge\textbf{-R})$$

$$\boxed{\Xi;\Gamma \vdash Sp :: S < Q \mid C}$$

$$\frac{}{\Xi;\Gamma \vdash () :: Q' < Q \mid Q' \leq Q}\ (\textbf{switch})$$

$$\frac{\Xi;\Gamma \vdash Sp :: T < Q \mid C_1 \qquad \Xi;\Gamma \vdash N \Leftarrow S \mid C_2}{\Xi;\Gamma \vdash (N; Sp) :: S \to T < Q \mid C_1 \mathbin{\textbf{∧}} C_2}\ (\to\textbf{-L})$$

$$\frac{(\Xi,\sigma,\tau);\Gamma \vdash Sp :: \tau < Q \mid C_1 \qquad (\Xi,\sigma,\tau);\Gamma \vdash N \Leftarrow \sigma \mid C_2}{\Xi;\Gamma \vdash (N; Sp) :: \rho^{t}_{A\to B} < Q \mid \exists\sigma^{\sim t}_{A}.\, \exists\tau^{t}_{B}.\, (\rho \leq \sigma \to \tau) \mathbin{\textbf{∧}} C_1 \mathbin{\textbf{∧}} C_2}\ (\to\textbf{-L/sv})$$

$$\frac{}{\Xi;\Gamma \vdash Sp :: \top < Q \mid \textbf{ff}}\ (\top\textbf{-L}) \qquad \frac{\Xi;\Gamma \vdash Sp :: S_1 < Q \mid C_1 \qquad \Xi;\Gamma \vdash Sp :: S_2 < Q \mid C_2}{\Xi;\Gamma \vdash Sp :: S_1 \wedge S_2 < Q \mid C_1 \mathbin{\textbf{∨}} C_2}\ (\wedge\textbf{-L})$$

Figure 5.8: Constraint generation.

Dually, in order to check a function $\lambda x. N$ at a sort, that sort must be an intersection of function sorts, *all* of which correctly describe the body of the function. The rule **→-R/sv** captures this intuition with a universal constraint: for $\lambda x. N$ to check at a variable ρ, that variable must be such that *any* function sort $\sigma \to \tau$ that it can be promoted to is an appropriate sort for the function. Our interpretation of function-checking as a universal constraint is novel, and we will see that it is both necessary and sufficient below.

The semantics of constraints is given by a constraint satisfaction judgment $\vDash C$, which lifts a ground constraint (one with no free sort variables) to a proposition of our metalanguage. The interpretation of the quantifiers shows that they range over ground sorts refining the appropriate types.

$$\begin{array}{lll}
\vDash S_1 \mathrel{\boldsymbol{\le}} S_2 & \textit{iff} & S_1 \le S_2 \\
\vDash \mathbf{tt} & \textit{always} & \\
\vDash C_1 \owedge C_2 & \textit{iff} & \vDash C_1 \textit{ and} \vDash C_2 \\
\vDash \mathbf{ff} & \textit{never} & \\
\vDash C_1 \ovee C_2 & \textit{iff} & \vDash C_1 \textit{ or} \vDash C_2 \\
\vDash C_1 \supset C_2 & \textit{iff} & \textit{if} \vDash C_1\textit{, then} \vDash C_2 \\
\vDash \forall \sigma_A. C & \textit{iff} & \textit{for every ground } S \sqsubset A\textit{, we have} \vDash [S/\sigma]C \\
\vDash \exists \sigma_A. C & \textit{iff} & \textit{for some ground } S \sqsubset A\textit{, we have} \vDash [S/\sigma]C
\end{array}$$

Since there are finitely many refinements of a type, the domain of quantified constraints is finite, and thus constraint satisfaction is decidable.

Theorem 5.14 (Decidability, constraint satisfaction). *Given a ground constraint C, either $\vDash C$ or $\nvDash C$.*

Proof. By induction on the structure of C, resorting to enumeration in the quantifier cases and the decidability of subsorting at the atomic constraint. □

Our presentation of constraint generation differs from Pierce's [Pie02] in several ways. First, our treatment of names is different: instead of "generating" fresh names over the course of a derivation and ensuring that the names generated by two subderivations do not overlap, we simply make use of standard variable-binding conventions and α-conversion to ensure "freshness". Furthermore, by introducing quantifiers into the language of constraints, we are able to maintain the property that the free variables of a generated constraint are contained in the free variables of the inputs to the judgment, a property that will simplify our completeness theorem significantly.

Finally, we specialize rules to expect either a sort of the appropriate form or a sort variable. See for example the rules **→-R** and **→-R/sv**; alternatively, we could have a single generic rule which does not inspect the form of the sort:

$$\frac{(\Xi, \sigma, \tau); (\Gamma, x{::}\sigma) \vdash N \Leftarrow \tau \mid C}{\Xi; \Gamma \vdash \lambda x. N \Leftarrow S \mid \forall \sigma_A^{\sim t}. \forall \tau_B^{t}. (S \mathrel{\boldsymbol{\le}} \sigma \to \tau) \supset C} \ (\to\textbf{-R}')$$

It may seem that by formulating the rules the way we have, we have lost the property that constraint generation always succeeds, a property that justifies our claim that we can reduce sort reconstruction to constraint solving. But in fact, thanks to the refinement restriction, the rules we have given are indeed total, and since they result in simpler constraints, we choose to adopt them. We capture totality with the following theorem, in which Γ^* erases refinement information from the context Γ and the judgment $\Xi \vdash \Gamma\ \widetilde{\mathsf{ctx}}$ extends the sort variable-sensitive weak refinement relation to context formation.

$$\frac{}{\Xi \vdash \cdot\ \widetilde{\mathsf{ctx}}} \qquad \frac{\Xi \vdash \Gamma\ \widetilde{\mathsf{ctx}} \qquad \Xi \vdash S \mathrel{\widetilde{\sqsubseteq}} A}{\Xi \vdash \Gamma, x{::}S{\sqsubset}A\ \widetilde{\mathsf{ctx}}}$$

Theorem 5.15 (Totality of constraint generation). *Constraint generation succeeds on all well-typed inputs. Suppose $\Xi \vdash \Gamma\ \widetilde{\mathsf{ctx}}$. Then:*

1. *If $\Gamma^* \vdash N \Leftarrow A$ and $\Xi \vdash S \mathrel{\widetilde{\sqsubseteq}} A$, then $\Xi; \Gamma \vdash N \Leftarrow S \mid \mathcal{C}$, for some $\mathcal{C}$, and*

2. *If $\Gamma^* \vdash Sp :: A < P$ and $\Xi \vdash S \mathrel{\widetilde{\sqsubseteq}} A$ and $\Xi \vdash Q \mathrel{\widetilde{\sqsubseteq}} P$, then $\Xi; \Gamma \vdash Sp :: S < Q \mid \mathcal{C}$, for some $\mathcal{C}$.*

Proof. By lexicographic induction on the typing derivation and the refinement derivation for S. In the cases where the sort S is an intersection, we induct on the same typing derivation and subderivations of the refinement derivation, while in the cases where the sort S is a variable, we induct on subderivations of the typing derivation and new variable refinement derivations. □

From the totality result, we can extract a functional specification of constraint generation. We include it in Figure 5.9 for the interested reader. The function $\mathsf{recon}(\Gamma, N, S)$ implements the judgment $\Xi; \Gamma \vdash N \Leftarrow S \mid \mathcal{C}$ while the function $\mathsf{reconSp}(\Gamma, Sp, S, Q)$ implements the judgment $\Xi; \Gamma \vdash Sp :: S < Q \mid \mathcal{C}$. In the functional code, we leave the sort variable context Ξ implicit.

As alluded to above, the meaning of constraint generation $\Xi; \Gamma \vdash \mathcal{J} \mid \mathcal{C}$ is given by soundness and completeness results for the generated constraint $\mathcal{C}$. The soundness result establishes that the generated constraint *sufficiently* captures the derivability of the judgment $\Gamma \vdash \mathcal{J}$: a solution to the constraint will serve as a solution to the judgment. Conversely, the completeness result establishes that the generated constraint *necessarily* captures the derivability of $\Gamma \vdash \mathcal{J}$: a solution to the judgment must necessarily provide a solution to the constraint.

In both cases, a "solution" is of course a grounding substitution. From the definition of $\vdash \theta : \Xi$, we can learn something about the form of a sort being substituted for a variable that refines a function type. The following lemma will be useful in proving both soundness and completeness.

Lemma 5.16. *If $\sigma_{A \to B} \in \Xi$ and $\vdash \theta : \Xi$, then $\theta\sigma \equiv S_1 \to T_1 \wedge \ldots \wedge S_n \to T_n$ where for each i between 1 and n, we have $\cdot \vdash S_i \sqsubset A$ and $\cdot \vdash T_i \sqsubset B$.*

Proof. Straightforward induction on the derivation of $\vdash \theta : \Xi$ gives us that $\cdot \vdash \theta\sigma \sqsubset A \to B$, from which the result follows by a straightforward structural rule induction. □

Theorem 5.17 (Soundness, constraint generation). *Suppose $\vdash \theta : \Xi$.*

$\mathsf{recon}(\Gamma, N, \top) = \mathbf{tt}$
$\mathsf{recon}(\Gamma, N, S_1 \wedge S_2) = \mathsf{recon}(\Gamma, N, S_1) \mathbin{\textcircled{\wedge}} \mathsf{recon}(\Gamma, N, S_2)$
$\mathsf{recon}(\Gamma, \lambda x. N, S \to T) = \mathsf{recon}((\Gamma, x{::}S), N, T)$
$\mathsf{recon}(\Gamma, \lambda x. N, \rho^t_{A\to B}) = \forall \sigma^{\sim t}_A. \forall \tau^t_B. (\rho \leq \sigma \to \tau) \supset \mathsf{recon}((\Gamma, x{::}S), N, \tau)$
$\mathsf{recon}(\Gamma, x \cdot Sp, Q) = \mathsf{reconSp}(\Gamma, Sp, \Gamma(x), Q)$
$\mathsf{recon}(\Gamma, c \cdot Sp, Q) = \mathsf{reconSp}(\Gamma, Sp, \Sigma(c), Q)$

$\mathsf{reconSp}(\Gamma, Sp, \top, Q) = \mathbf{ff}$
$\mathsf{reconSp}(\Gamma, Sp, S_1 \wedge S_2, Q) = \mathsf{reconSp}(\Gamma, Sp, S_1, Q) \mathbin{\textcircled{\vee}} \mathsf{reconSp}(\Gamma, Sp, S_2, Q)$
$\mathsf{reconSp}(\Gamma, (), Q', Q) = Q' \leq Q$
$\mathsf{reconSp}(\Gamma, (N; Sp), S \to T, Q) = \mathsf{reconSp}(\Gamma, Sp, T, Q) \mathbin{\textcircled{\wedge}} \mathsf{recon}(\Gamma, N, S)$
$\mathsf{reconSp}(\Gamma, (N; Sp), \rho^t_{A\to B}, Q) = \exists \sigma^{\sim t}_A. \exists \tau^t_B. (\rho \leq \sigma \to \tau) \mathbin{\textcircled{\wedge}} \mathsf{reconSp}(\Gamma, Sp, \tau, Q) \mathbin{\textcircled{\wedge}} \mathsf{recon}(\Gamma, N, \sigma)$

Figure 5.9: Functional algorithm for generating constraints.

1. *If* $\Xi; \Gamma \vdash N \Leftarrow S \mid C$ *and* $\vDash \theta C$*, then* $\theta\Gamma \vdash N \Leftarrow \theta S$.
2. *If* $\Xi; \Gamma \vdash Sp :: S < Q \mid C$ *and* $\vDash \theta C$*, then* $\theta\Gamma \vdash Sp :: \theta S < \theta Q$.

Proof. By induction on the given constraint generation derivation. We give here the cases for the $\to$**-R/sv** and $\to$**-L/sv** rules; the rest are straightforward.

Case:

$$\frac{(\Xi, \sigma, \tau); (\Gamma, x{::}\sigma) \vdash N \Leftarrow \tau \mid C}{\Xi; \Gamma \vdash \lambda x. N \Leftarrow \rho^t_{A\to B} \mid \forall \sigma^{\sim t}_A. \forall \tau^t_B. (\rho \leq \sigma \to \tau) \supset C}\ (\to\text{-}\mathbf{R/sv})$$

$\vDash \forall \sigma. \forall \tau. (\theta\rho \leq \sigma \to \tau) \supset \theta C$ By assumption.
For every $S \sqsubset A$ and $T \sqsubset B$, if $\theta\rho \leq S \to T$, then $\vDash [S/\sigma, T/\tau]\theta C$ By definition.
$\theta\rho \equiv S_1 \to T_1 \wedge \ldots \wedge S_n \to T_n$, where $\cdot \vdash S_i \sqsubset A$ and $\cdot \vdash T_i \sqsubset B$ By Lemma 5.16.
For each i between 1 and n:
  $\vdash [\theta, S_i/\sigma, T_i/\tau] : \Xi, \sigma_A, \tau_B$ By substitution formation rules.
  $\theta\rho \leq S_i \to T_i$ By subsorting rules.
  $\vDash [\theta, S_i/\sigma, T_i/\tau] C$ By above implication.
  $\theta\Gamma, x{::}S_i \vdash N \Leftarrow T_i$ By i.h. and freshness reasoning ($\sigma, \tau \notin \mathrm{FSV}(\Gamma)$).
  $\theta\Gamma \vdash \lambda x. N \Leftarrow S_i \to T_i$ By rule.
$\theta\Gamma \vdash \lambda x. N \Leftarrow S_i \to T_i \wedge \ldots \wedge S_n \to T_n$ By rules.
$\theta\Gamma \vdash \lambda x. N \Leftarrow \theta\rho$ By equivalence.

Case:

$$\frac{(\Xi, \sigma, \tau); \Gamma \vdash Sp :: \tau < Q \mid C_1 \qquad (\Xi, \sigma, \tau); \Gamma \vdash N \Leftarrow \sigma \mid C_2}{\Xi; \Gamma \vdash (N; Sp) :: \rho^t_{A\to B} < Q \mid \exists \sigma^{\sim t}_A. \exists \tau^t_B. (\rho \leq \sigma \to \tau) \mathbin{\textcircled{\wedge}} C_1 \mathbin{\textcircled{\wedge}} C_2}\ (\to\text{-}\mathbf{L/sv})$$

$\vDash \exists\sigma^{\sim t}_{A}.\, \exists\tau^{t}_{B}.\, (\rho \leq \sigma \to \tau) \owedge C_1 \owedge C_2$ By assumption.
For some $S \sqsubset A$ and $T \sqsubset B$,
$\theta\rho \leq S \to T$ and $\vDash [\theta, S/\sigma, T/\tau]C_1$ and $\vDash [\theta, S/\sigma, T/\tau]C_2$ By definition.
$\vdash [\theta, S/\sigma, T/\tau] : \Xi, \sigma_A, \tau_B$ By substitution formation rules.
$\theta\Gamma \vdash Sp :: T < \theta Q$ and $\theta\Gamma \vdash N \Leftarrow S$ By i.h. and freshness reasoning.
$\theta\Gamma \vdash (N; Sp) :: S \to T < \theta Q$ By rule.
$\theta\Gamma \vdash (N; Sp) :: \theta\rho < \theta Q$ By subsumption (Lemma 5.5). □

Theorem 5.18 (Completeness, constraint generation). *Suppose* $\cdot \vdash \theta : \Xi$.

1. *If* $\Xi; \Gamma \vdash N \Leftarrow S \mid C$ *and* $\theta\Gamma \vdash N \Leftarrow \theta S$, *then* $\vDash \theta C$.

2. *If* $\Xi; \Gamma \vdash Sp :: S < Q \mid C$ *and* $\theta\Gamma \vdash Sp :: \theta S < \theta Q$, *then* $\vDash \theta C$.

Proof. By induction on the given constraint generation derivation, using inversion on the other given derivation. We give here the cases for the **→-R/sv** and **→-L/sv** rules; the rest are straightforward.

Case:

$$\frac{(\Xi, \sigma, \tau); (\Gamma, x{::}\sigma) \vdash N \Leftarrow \tau \mid C}{\Xi; \Gamma \vdash \lambda x.\, N \Leftarrow \rho^{t}_{A \to B} \mid \forall\sigma^{\sim t}_{A}.\, \forall\tau^{t}_{B}.\, (\rho \leq \sigma \to \tau) \supset C}\ (\textbf{→-R/sv})$$

$\theta\Gamma \vdash N \Leftarrow \theta\rho$ By assumption.
Let $S \sqsubset A$ and $T \sqsubset B$, and assume $\theta\rho \leq S \to T$:
$\theta\Gamma \vdash N \Leftarrow S \to T$ By subsumption.
$\theta\Gamma, x{::}S \vdash N \Leftarrow T$ By inversion.
$[\theta, S/\sigma, T/\tau](\Gamma, x{::}\sigma) \vdash N \Leftarrow [\theta, S/\sigma, T/\tau]\tau$ By equality and freshness reasoning.
$\vdash [\theta, S/\sigma, T/\tau] : \Xi, \sigma_A, \tau_B$ By substitution formation rules.
$\vDash [\theta, S/\sigma, T/\tau]C$ By i.h.
For every $S \sqsubset A$ and $T \sqsubset B$, if $\theta\rho \leq S \to T$, then $\vDash [\theta, S/\sigma, T/\tau]C$.
$\vDash \theta(\forall\sigma^{\sim t}_{A}.\, \forall\tau^{t}_{B}.\, (\rho \leq \sigma \to \tau) \supset C)$ By definition.

Case:

$$\frac{(\Xi, \sigma, \tau); \Gamma \vdash Sp :: \tau < Q \mid C_1 \qquad (\Xi, \sigma, \tau); \Gamma \vdash N \Leftarrow \sigma \mid C_2}{\Xi; \Gamma \vdash (N; Sp) :: \rho^{t}_{A \to B} < Q \mid \exists\sigma^{\sim t}_{A}.\, \exists\tau^{t}_{B}.\, (\rho \leq \sigma \to \tau) \owedge C_1 \owedge C_2}\ (\textbf{→-L/sv})$$

$\theta\Gamma \vdash (N; S)p :: \theta\rho < Q$ By assumption.
$\theta\rho \equiv S_1 \to T_1 \wedge \ldots \wedge S_n \to T_n$, where $\cdot \vdash S_i \sqsubset A$ and $\cdot \vdash T_i \sqsubset B$ By Lemma 5.16.
For some i between 1 and n:
$\theta\Gamma \vdash Sp :: T_i < Q$ and $\theta\Gamma \vdash N \Leftarrow S_i$ By multi-step inversion.
Let $\theta' = [\theta, S_i/\sigma, T_i/\tau]$:
$\theta'\Gamma \vdash Sp :: \theta'\tau < \theta' Q$ and $\theta'\Gamma \vdash N \Leftarrow \theta'\sigma$ By equality and freshness reasoning.
$\vdash \theta' : \Xi, \sigma_A, \tau_B$ By substitution formation rules.
$\vDash \theta' C_1$ and $\vDash \theta' C_2$ By i.h.
$\theta\rho \leq S_i \to T_i$ By subsorting rules.
$\vDash \theta(\exists\sigma^{\sim t}_{A}.\, \exists\tau^{t}_{B}.\, (\rho \leq \sigma \to \tau) \owedge C_1 \owedge C_2)$ By definition. □

The proof of the completeness theorem draws attention to an important distinction between our treatment of constraint generation and that of Pottier and Rémy [PR05]. Although we follow them in using quantification over sort variables to manage "freshness", we diverge by including universal quantification and not just existential. In our **→-R/sv** rule, we generate a universally quantified constraint, while Pottier and Rémy generate an existentially quantified one. Porting their rule to our setting and our notation, it might be written something like this:

$$\frac{(\Xi, \sigma, \tau); (\Gamma, x{::}\sigma) \vdash N \Leftarrow \tau \mid C}{\Xi; \Gamma \vdash \lambda x.\, N \Leftarrow \rho^{t}_{A\to B} \mid \exists \sigma^{\sim t}_{A}.\, \exists \tau^{t}_{B}.\, (\sigma \to \tau \leq \rho) \owedge C}\ (\to\textbf{-R/sv}')$$

Observe not only that the quantifiers differ (and correspondingly, that their rule has a conjunction instead of an implication), but also that the direction of subsorting is reversed: the Pottier-Rémy rule posits that $\sigma \to \tau$ be below ρ. Although this approach works in the setting of HM(X), the rule is incomplete in our setting due to the presence of intersection sorts.

The intuition behind the rule **→-R/sv′** is roughly that in order for $\lambda x.\, N$ to have sort ρ, it must have some function sort $\sigma \to \tau$ which is a subsort of ρ. In LFR, a calculus with intersection sorts, we can see immediately that this is not necessarily the case: perhaps it has sort ρ by virtue of the fact that ρ is an *intersection* of function sorts, each one of which is a *supersort* of the sort ρ. Following this line of reasoning, we can construct a counterexample to completeness. Consider a signature with two unrelated base sorts u and v and a constant $c :: u \to u \wedge v \to v$. If we generate a constraint $\rho_{A\to B}; \cdot \vdash \lambda x.\, c\, x \Leftarrow \rho \mid C$ using the new rule **→-R/sv′**, we get:

$$C = \exists \sigma_A.\, \exists \tau_B.\, \sigma \to \tau \leq \rho \owedge \Big((\sigma \leq u \owedge u \leq \tau) \ovee (\sigma \leq v \owedge v \leq \tau)\Big)$$

It is easily seen that for $\theta = [u \to u \wedge v \to v/\rho]$, we can derive $\cdot \vdash \lambda x.\, c\, x \Leftarrow \theta\rho$, so by completeness, we should expect that $\vDash \theta C$. But this is impossible: suppose we could find some $S \sqsubset A$ and $T \sqsubset B$ such that $S \to T \leq u \to u \wedge v \to v$, and either $S \leq u$ and $u \leq T$, or $S \leq v$ and $v \leq T$. In the case that $S \leq u$ and $u \leq T$, we can immediately derive $u \to u \leq S \to T$. By transitivity, we discover that $u \to u \leq v \to v$, which contradicts our original assumption that u and v were unrelated. In the other case, we symmetrically learn that $v \to v \leq u \to u$, an equally contradictory conclusion. Therefore, it must be the case that $\nvDash \theta C$.

Implementation notes. Our implementation of constraint generation follows the system presented here closely, but with a few important optimizations.

- We maintain constraints in disjunctive normal form, which serves two purposes: (1) it will make the constraints easier to solve later, and (2) it cuts down the search space by making conjunction automatically "short-circuit" when one conjunct or the other is unsatisfiable.

- We furthermore apply a number of on-the-fly logical simplifications when computing constraints: the conjunction of two clauses deletes any redundant constraints ($[\mathcal{A} \owedge \mathcal{B}] \owedge [\mathcal{A}] = \mathcal{A} \owedge \mathcal{B}$), and the disjunction of two clauses keeps only the weaker one, if one implies the other ($[\mathcal{A} \owedge \mathcal{B}] \ovee [\mathcal{A}] = \mathcal{A}$). The disjunctive normal form of a constraint can be quite large, but these simplifications help keep constraints from blowing up needlessly.

- We do not generate a "suspended" constraint $Q_1 \leq Q_2$ when Q_1 and Q_2 are ground base sorts; instead we just perform the subsorting check and emit either **tt** or **ff** accordingly, culling the search space further still.

- Finally, we fail instead of generating universally-quantified constraints—our constraint solving algorithm does not handle them, and we argue below that they are uncommon in practice.

5.5.2 Solving the Constraints

We now give an algorithm for determining the identity of each sort variable given a constraint. We restrict our attention to purely existential constraints—ones with no universal quantifiers—for several reasons. For one, purely existential constraints can be put into prenex form, essentially allowing us to forget about the quantification altogether and just solve propositional constraints. This approach corresponds fairly closely to what has been studied in the literature. For another, purely existential constraints can often be solved without resorting to enumeration, a case that cannot be made for universal constraints since they quantify over all ground sorts in a non-parametric fashion. As we have seen, enumeration marks the downfall of practicality, and should be treated as a last resort.

Most importantly, though, universal constraints do not seem to arise very frequently in practice. Moreover, we can make a general argument explaining why. Recall from Figure 5.8 that universal constraints are only generated when we attempt to check an abstraction $\lambda x. N$ at a sort variable. Such a situation in turn arises only when an abstraction appears in the spine of an application $h \cdot (\ldots; \lambda x. N; \ldots)$ whose head is of unknown sort. What kinds of heads have unknown sort? Certainly not term or type constants from earlier in the signature, since as we have seen, all such constants would have already had their sorts fully reconstructed at this point.

In order to be of unknown sort, the guilty head must be a variable, either an implicitly quantified metavariable or an explicitly bound local variable. For such a variable to take a function as one of its arguments, the containing clause would as a whole have to be third-order or higher:

$c :: \Pi E{::}(S{\to}T){\to}U.\ \ p\ (E\ (\lambda x.N)).$

$c :: q\ A\ B\ C$
$\quad \leftarrow (\Pi y{::}(S{\to}T){\to}U.\ \ p\ (y\ (\lambda x.N))).$

Such examples are not unheard of among LF wizards, but neither are they typical. Thus we choose not to worry ourselves too much over focusing on a fragment that excludes them for the present.

Every pure existential constraint can be recast as a logically equivalent one with only prenex quantification in disjunctive normal form. So for the purposes of constraint solving, we consider a constraint to be some number of existential quantifiers whose body is the disjunction of a set of clauses, where each clause is a conjunction of atomic subsorting constraints. Even for the atomic constraints, we can restrict their form further based on the kinds of constraints generated by our algorithm in the previous section: a subsorting constraint is either between two base-or-variable sorts or between a variable and a function sort made of variables of appropriate tendencies. We write $\mathcal{A}$ for atomic constraints, $\mathcal{C}$ for conjunctive clauses, $\mathcal{D}$ for disjunctions of clauses, and $\mathcal{E}$ for prenex-quantified disjunctions.

$$\begin{aligned}
\mathcal{A} &::= Q \leq Q' \mid \sigma \leq Q \mid Q \leq \sigma \mid \sigma \leq \tau \mid \rho^{t} \leq \sigma^{\sim t} \to \tau^{t} \\
\mathcal{C} &::= \mathbf{tt} \mid \mathcal{A} \owedge \mathcal{C} \\
\mathcal{D} &::= \mathbf{ff} \mid \mathcal{C} \ovee \mathcal{D} \\
\mathcal{E} &::= \exists \sigma^{t}_{A}.\, \mathcal{E} \mid \mathcal{D}
\end{aligned}$$

In this section, we mean for Q to range over ground base sorts, in order to make the distinction between different $\mathcal{A}$ productions meaningful.

The top-level algorithm simply solves by clauses after adding the prenex-quantified variables to the local context, producing a list of substitutions $\mathbf{\Theta} ::= \cdot \mid \theta, \mathbf{\Theta}$.

$$\boxed{\mathsf{solve}(\Xi, \mathcal{E}) = \mathbf{\Theta}}$$

$$\frac{\mathsf{solve}((\Xi, \sigma), \mathcal{E}) = \mathbf{\Theta}}{\mathsf{solve}(\Xi, \exists \sigma.\, \mathcal{E}) = \mathbf{\Theta} \setminus \sigma} \qquad \frac{}{\mathsf{solve}(\Xi, \mathbf{ff}) = \cdot}$$

$$\frac{\mathsf{solveClause}(\Xi, \mathcal{C}) = \theta \qquad \mathsf{solve}(\Xi, \mathcal{D}) = \mathbf{\Theta}}{\mathsf{solve}(\Xi, \mathcal{C} \ovee \mathcal{D}) = \theta, \mathbf{\Theta}} \qquad \frac{\mathsf{unsatClause}(\Xi, \mathcal{C}) \qquad \mathsf{solve}(\Xi, \mathcal{D}) = \mathbf{\Theta}}{\mathsf{solve}(\Xi, \mathcal{C} \ovee \mathcal{D}) = \mathbf{\Theta}}$$

When we solve an existential constraint, we remove the bound variable from the substitutions returned, written $\mathbf{\Theta} \setminus \sigma$, since it is no longer free. After the quantifiers, each clause yields a single substitution which satisfies $\mathcal{C}$, and we package them all together in a list of substitutions $\mathbf{\Theta}$. Any substitution in the list is a solution to the entire disjunction of constraints, but we must generate them all in order to achieve principality. If a clause is unsatisfiable then it does not contribute to the result. If for some clause $\mathcal{C}$ neither $\mathsf{solveClause}(\Xi, \mathcal{C}) = \theta$ nor $\mathsf{unsatClause}(\Xi, \mathcal{C})$, then the clause is too underconstrained for our algorithm to find its solution, and there will be no solve derivation.

It is in solving a single clause that we finally come to care about the tendency of a sort variable. Every sort variable σ in a constraint has some upper bounds $\sigma \leq S$ and some lower bounds $T \leq \sigma$. If the variable is a maximizing one, then we solve it by setting it to the greatest lower bound of its upper bounds, i.e., their intersection. By definition, this is the largest possible sort that satisfies the constraints bounding σ from above. But of course, we must check that this solution satisfies the constraints bounding σ from below as well. If the variable is a minimizing one, then we follow the dual strategy: σ is solved by the least upper bound of its lower bounds, and we check that this also satisfies the upper bounds. Though we do not have union types in our calculus, the least upper bound of two sorts is definable, and we describe how below.

The strategy above is surprisingly effective for how intuitive and simple it sounds, but sometimes it can fail to produce a most general solution. The reason is that two sort variables may be in tension: a maximizing variable may be constrained to be below a minimizing one, $\sigma^+ \leq \tau^-$. Such constraints are intuitively problematic: a maximizing variable wants to be as large as possible, but it is bounded from above by a minimizing variable, one that wants to be as small as possible, and vice versa. More concretely, such constraints are problematic because they match both forms of constraint mentioned above, a maximizing variable bounded from above and a minimizing variable bounded from below. Choosing to treat it wholly as one or the other amounts to preferring the tendency of one of the variables over that of the other, a strategy which sacrifices generality.

Before we discuss the solution to such constraints, though, we should convince ourselves that they arise in practice—if they are as uncommon as universally quantified constraints, perhaps we can just ignore the issue. But as it turns out, constraints with tension are rather common: they arise anytime one variable of unknown maximizing sort is applied to another. The following constraint-generation derivation for $f\,x$ illustrates the pattern:

$$\dfrac{f{::}\rho^+ \in \Gamma \qquad \dfrac{\dfrac{}{\ldots;\Gamma \vdash () :: \tau^+ < Q \mid \tau^+ \leq Q} \qquad \dfrac{x{::}\xi^+ \in \Gamma \qquad \dfrac{}{\ldots;\Gamma \vdash () :: \xi^+ < \sigma^- \mid \xi^+ \leq \sigma^-}}{\ldots;\Gamma \vdash x \cdot () \Leftarrow \sigma^- \mid \xi^+ \leq \sigma^-}}{\ldots;\Gamma \vdash (x) :: \rho^+ < Q \mid \exists\sigma^-.\,\exists\tau^+.\,(\rho^+ \leq \sigma^- \to \tau^+) \owedge \ldots}}{\ldots;\Gamma \vdash f \cdot (x \cdot ()) \Leftarrow Q \mid \ldots}$$

The offending constraint $\xi^+ \leq \tau^-$ appears in the upper right of the derivation. Intuitively, what happens is that x must check at the domain sort of f, which means that the sort of x must be a subsort of the domain sort of f. If the sort of f is maximizing, its domain sort will be minimizing due to contravariance. So assuming the sort of x is maximizing, we generate a constraint asserting that a maximizing sort variable be below a minimizing one.

But how often does it happen in LFR that we write an application $f\,x$, where the sorts of both f and x are unknown and maximizing? Well, recall that the most general reconstruction for a declaration must maximize the sorts of its metavariables, and metavariables are precisely the parts of the input which begin life with unknown sorts, so anytime we apply one metavariable to another, we create a constraint with tension. This situation is quite common in LF code using higher-order abstract syntax, where application encodes object-language substitution. We have already seen one such example, in fact, in the rule for evaluating an application in the call-by-value λ-calculus:

$$\begin{aligned} &\textit{ev-app} :: \textit{eval}'\ (\textit{app}\ E_1\ E_2)\ V \\ &\qquad \leftarrow \textit{eval}'\ E_1\ (\textit{lam}\ \lambda x.E_1'\ x) \\ &\qquad \leftarrow \textit{eval}'\ E_2\ V_2 \\ &\qquad \leftarrow \textit{eval}'\ (E_1'\ V_2)\ V. \end{aligned}$$

The application $E_1'\ V_2$ representing the substitution $[v_2/x]\,e_1'$ leads to a tense constraint in just the manner described above. We will explore this example more closely below.

First, though, let us try to gain some intuition by examining the implications of tense constraints more carefully. Consider for the sake of discussion the following constraint:

$$t \leq \tau^- \mathbin{\textcircled{\scriptsize$\wedge$}} \sigma^+ \leq \tau^- \mathbin{\textcircled{\scriptsize$\wedge$}} \sigma^+ \leq s.$$

The maximizing variable σ^+ has one ground upper bound, s, while the minimizing variable τ^- has one ground lower bound, t. The middle constraint $\sigma^+ \leq \tau^-$ can be thought of either as placing an upper bound on σ^+ or a lower bound on τ^-, but the two choices lead to different results. If we think of it as an upper bound on σ^+, then we will set $\sigma^+ = s \wedge \tau^-$, the intersection of its upper bounds, and $\tau^- = t$, the least upper bound of its lower bounds, resulting in the substitution

$$[s \wedge t/\sigma^+, t/\tau^-].$$

But if we think of $\sigma^+ \leq \tau^-$ as a lower bound on τ, then we will set $\sigma^+ = s$ and τ^- to the least upper bound of σ^+ and t. Assuming s and t are unrelated, their upper bound is $\top$, so the end result is the following substitution:

$$[s/\sigma^+, \top/\tau^-].$$

Both substitutions make the constraint clause true, but the first does a better job of minimizing τ^- while the second does a better job maximizing σ^+. Neither substitution represents a better solution than the other. Furthermore, in general, finding solutions to constraints involving such tension might require enumeration of all sorts between two bounds.

If we imagine that constraints determine linear intervals in which sort variables must fall, we can visualize the problem as in Figure 5.10. In the figure, σ^+ and τ^- are both constrained to certain intervals, but σ^+, being a maximizing variable, would like to live at the top of his interval, on the right edge, while τ^-, being a minimizing variable, would like to live at the bottom of his interval, on the left edge. Unfortunately, their intervals overlap: any point inside the grey area represents a most general solution, with σ^+ as large as he can be and τ^- as small as he can be, subject to the constraint that $\sigma^+ \leq \tau^-$. This enumeration we deem impractical and so we would reject the clause that generated the constraint.

If on the other hand the situation is more like that of Figure 5.11, then there is a *single* most general solution, and it is easy to find: σ^+ gets to be the top edge of his interval just like he wants, while τ^- enjoys the view from the bottom edge of his interval. Since σ^+'s

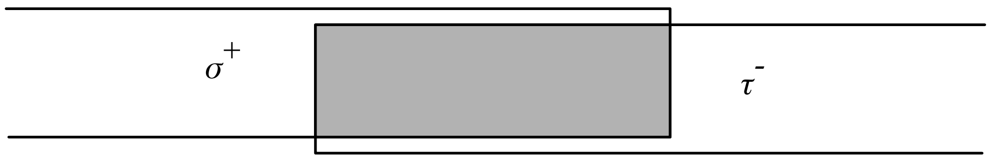

Figure 5.10: Overlapping constraints in tension.

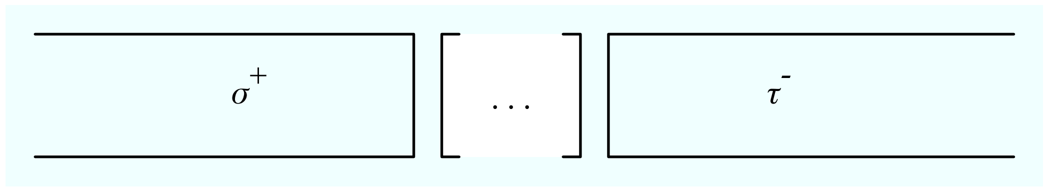

Figure 5.11: Non-overlapping constraints in tension.

interval is entirely beneath τ^-'s interval, their wishes are not at odds and they need not reach a compromise—forces beyond both of them have determined the way of things.

Our story suggests a strategy: use only the ground upper and lower bounds to determine the identities of sort variables, treating variable-variable constraints like $\sigma \leq \tau$ as after-the-fact checks that everything works out all right in the end. In this particular case, we set σ^+ to s and τ^- to t and then check to see if $s \leq t$: if so, then we have found a solution that everyone can agree on! If not, then we have found a pair of incompatible preferences and a non-trivial interval that we would have to enumerate to find the most general solution.

This is the strategy that we adopt, with one twist: we must first compute the transitive closure of the constraints in order to ensure that we find *all* of a variable's ground bounds, even those that act at a distance. Computing this closure is useful in its own right, since it can lead us to discover that a clause is unsatisfiable when it contains a pair of constraints like $s \leq \sigma$ and $\sigma \leq t$, where $s \nleq t$.

We do *not* apply this strategy to constraints of the form $\rho \leq \sigma \to \tau$, though: for these types of constraints, we can be more eager. A quick scan of the constraint generation rules reveals that any variable-variable constraint $\sigma \leq \tau$ will come from the **switch** rule, and thus be a constraint on base sorts by the refinement restriction, so tense variable-variable constraints are never on variables refining function type. Furthermore, note that we only generate constraints of the form $\rho \leq \sigma \to \tau$ when σ and τ are freshly made up to have the correct tendencies. So constraints on sort variables representing functions will never involve tension and can be solved eagerly using our original naive strategy.

Before elucidating the technical details, we turn to a real example of the algorithm in action. Recall the rule for evaluating an application in the call-by-value λ-calculus. After LFR type reconstruction and filling it reads as follows:

$$\begin{aligned}&\mathit{ev\text{-}app} :: \Pi E_1{::}\sigma_1^+.\ \Pi E_2{::}\sigma_2^+.\ \Pi E_1'{::}\rho_1^+.\ \Pi V_2{::}\tau_2^+.\ \Pi V{::}\tau^+.\\ &\qquad \mathit{eval}'\ (\mathit{app}\ E_1\ E_2)\ V\end{aligned}$$

$\leftarrow$ *eval'* E_1 (*lam* $\lambda x.E'_1\ x$)
$\leftarrow$ *eval'* $E_2\ V_2$
$\leftarrow$ *eval'* $(E'_1\ V_2)\ V$.

The constraint generation algorithm generates the following constraint for this declaration, which has a single clause. We have organized them by the first line in the program that induces them, and deleted redundant conjuncts.

$$\begin{aligned}
&\sigma_1^+ \leq \mathit{cmp} \owedge \sigma_2^+ \leq \mathit{cmp} \owedge \tau^+ \leq \mathit{val} \\
\owedge\ &\rho_1^+ \leq \xi_1^- \to \xi_2^+ \owedge \xi_2^+ \leq \mathit{cmp} \owedge \mathit{val} \leq \xi_1^- \\
\owedge\ &\tau_2^+ \leq \mathit{val} \\
\owedge\ &\rho_1^+ \leq \xi_3^- \to \xi_4^+ \owedge \xi_4^+ \leq \mathit{cmp} \owedge \tau_2^+ \leq \xi_3^-
\end{aligned}$$

In the last line, we see a tense constraint $\tau_2^+ \leq \xi_3^-$ which is generated by the line *eval* $(E'_1\ V_2)\ V$: the variable ξ_3^- represents one possible domain of the metavariable E'_1, while the variable τ_2^+ represents the unknown sort of the metavariable V_2, the argument to E'_1.

First, we can solve for the arrow constraints: though we don't know their identities completely, it is safe to set them to their upper bound and move on.

$$[\ \xi_1^- \to \xi_2^+ \wedge \xi_3^- \to \xi_4^+\ /\ \rho_1^+\ ]$$

$$\begin{aligned}
&\sigma_1^+ \leq \mathit{cmp} \owedge \sigma_2^+ \leq \mathit{cmp} \owedge \tau^+ \leq \mathit{val} \\
\owedge\ &\xi_2^+ \leq \mathit{cmp} \owedge \mathit{val} \leq \xi_1^- \\
\owedge\ &\tau_2^+ \leq \mathit{val} \\
\owedge\ &\xi_4^+ \leq \mathit{cmp} \owedge \tau_2^+ \leq \xi_3^-
\end{aligned}$$

When we compute the transitive closure of the constraints to find all bounds, we get two new conjuncts:

$$\begin{aligned}
\tau_2^+ &\leq \xi_1^- && \text{from } \tau_2^+ \leq \mathit{val} \text{ and } \mathit{val} \leq \xi_1^- \\
\owedge\ \tau^+ &\leq \xi_1^- && \text{from } \tau^+ \leq \mathit{val} \text{ and } \mathit{val} \leq \xi_1^-
\end{aligned}$$

Now we can solve the constraints that represent ground bounds, yielding the following composite substitution and remaining constraints:

$$\begin{aligned}
[\ &\mathit{cmp}/\sigma_1^+\ ,\ \mathit{cmp}/\sigma_2^+\ ,\ \mathit{val}/\tau_2^+\ ,\ \mathit{val}/\tau^+\ , \\
&\mathit{val} \to \mathit{cmp} \wedge \mathit{val} \to \mathit{cmp}\ /\ \rho_1^+\ , \\
&\mathit{val}/\xi_1^-\ ,\ \mathit{cmp}/\xi_2^+\ ,\ \mathit{cmp}/\xi_3^+\ ,\ \mathit{cmp}/\xi_4^+\ ]
\end{aligned}$$

$$\tau_2^+ \leq \xi_3^- \owedge \tau_2^+ \leq \xi_1^- \owedge \tau^+ \leq \xi_1^-$$

Two points are worth noting. First, in the entry for ρ_1^+, we have substituted further for its variables as we learned their identities because it is always safe to do so for an arrow variable, but we have *not* yet substituted into the remaining constraints, because

$$Q_1 \sqcup Q_2 = \bigwedge \{Q \mid Q_1 \leq Q \text{ and } Q_2 \leq Q\}$$

$$\begin{aligned} \bot_P &= \bigwedge \{Q \mid \cdot \vdash Q \sqsubset P\} \\ \bot_{A \to B} &= \top \to \bot_B \end{aligned}$$

Figure 5.12: Base sort least upper bounds and least refinements.

doing so might have led to a non-general solution—recall the example of tension we gave above. Second, although the variable ξ_3^- had no ground bounds at all we have determined its identity: since it is a minimizing variable, we solved it with the smallest sort refining *exp*, in this case *val*.

Now we can check to make sure we haven't created any overlapping intervals by applying the substitution to the remaining constraints and making sure they are satisfied.

$$\mathit{val} \leq \mathit{val} \ \owedge \ \mathit{val} \leq \mathit{val} \ \owedge \ \mathit{val} \leq \mathit{val}$$

Everything checks out! But remember that $\mathit{val} \leq \mathit{cmp}$. Were that not the case, things could have turned out differently. The smallest sort refining *exp* would not just be *val*, but rather the intersection of both refinements: $\mathit{val} \wedge \mathit{cmp}$. Had we set ξ_3^- appropriately, the remaining constraints would substitute as follows:

$$\mathit{val} \leq \mathit{val} \wedge \mathit{cmp} \ \owedge \ \mathit{val} \leq \mathit{val} \ \owedge \ \mathit{val} \leq \mathit{val}$$

This constraint is not satisfiable, so our algorithm would give up instead of enumerating the sorts between $\mathit{val} \wedge \mathit{cmp}$ and *cmp*.

Since $\mathit{val} \leq \mathit{cmp}$ does hold, though, we find the first solution. We can then substitute into the original declaration to obtain the final, most generally reconstructed declaration:

$$\begin{aligned} &\mathit{ev\text{-}app} :: \Pi E_1{::}\mathit{cmp}.\ \Pi E_2{::}\mathit{cmp}.\ \Pi E_1'{::}\mathit{val} \to \mathit{cmp}.\ \Pi V_2{::}\mathit{val}.\ \Pi V{::}\mathit{val}. \\ &\qquad \mathit{eval}'\ (\mathit{app}\ E_1\ E_2)\ V \\ &\qquad \leftarrow \mathit{eval}'\ E_1\ (\mathit{lam}\ \lambda x. E_1'\ x) \\ &\qquad \leftarrow \mathit{eval}'\ E_2\ V_2 \\ &\qquad \leftarrow \mathit{eval}'\ (E_1'\ V_2)\ V. \end{aligned}$$

Note in particular that the reconstructed sort of E_1' is most general: it is the largest sort that makes the rest of the declaration well-sorted.

In the formal development that follows, we restrict our attention once more to simple sorts so we can focus on the core issues. As outlined above, we require a definition of least upper bounds. We only need binary least upper bounds on base sorts: to compute the *least* upper bound of Q_1 and Q_2, we simply intersect *all* of their upper bounds together. For "nullary" least upper bounds, or least sorts refining a type, we proceed by induction on the refined type. Both definitions are shown in Figure 5.12.

It is not difficult to verify that these definitions are correct.

Theorem 5.19 (Base sort least upper bounds). *$Q_1 \sqcup Q_2$ is the least upper bound of Q_1 and Q_2. That is,*

1. *$Q_1 \leq Q_1 \sqcup Q_2$ and $Q_2 \leq Q_1 \sqcup Q_2$, and*
2. *for any Q, if $Q_1 \leq Q$ and $Q_2 \leq Q$, then $Q_1 \sqcup Q_2 \leq Q$.*

Proof.

1. By definition, $Q_1 \sqcup Q_2$ is an intersection of upper-bounds of both Q_i. Since each Q_i is below every conjunct, each is also below the intersection.
2. Suppose $Q_1 \leq Q$ and $Q_2 \leq Q$. Then we have that $Q_1 \sqcup Q_2 = \bigwedge\{\ldots, Q, \ldots\}$, and therefore $Q_1 \sqcup Q_2 \leq Q$. □

Theorem 5.20 (Least sorts). *$\bot_A$ is the least sort refining A. That is, if $\cdot \vdash S \sqsubset A$, then $\bot_A \leq S$.*

Proof. By straightforward induction on the derivation of $\cdot \vdash S \sqsubset A$ using the derived rules for subsorting at higher sorts. □

It will be convenient in what follows to have metavariables for both base and non-base sorts that may be variables, and we write these in a calligraphic face.

$$\mathcal{Q} ::= Q \mid \sigma$$
$$\mathcal{S} ::= S \mid \sigma$$

As already mentioned, for base sort constraints we proceed by saturating to transitivity. This is done with a judgment $\mathcal{C} \vdash_{\mathsf{sat}} \mathcal{Q}_1 \leq \mathcal{Q}_2$.

$$\boxed{\mathcal{C} \vdash_{\mathsf{sat}} \mathcal{Q}_1 \leq \mathcal{Q}_2}$$

$$\frac{\mathcal{Q}_1 \leq \mathcal{Q}_2 \in \mathcal{C}}{\mathcal{C} \vdash_{\mathsf{sat}} \mathcal{Q}_1 \leq \mathcal{Q}_2} \qquad \frac{\mathcal{C} \vdash_{\mathsf{sat}} \mathcal{Q}_1 \leq \mathcal{Q}' \quad \mathcal{C} \vdash_{\mathsf{sat}} \mathcal{Q}' \leq \mathcal{Q}_2}{\mathcal{C} \vdash_{\mathsf{sat}} \mathcal{Q}_1 \leq \mathcal{Q}_2}$$

The key property of the transitivity saturation judgment is that any solution to the original constraints also validates any subsorting judgment entailed by saturation.

Lemma 5.21 (Soundness, saturation). *If $\mathcal{C} \vdash_{\mathsf{sat}} \mathcal{Q}_1 \leq \mathcal{Q}_2$ and $\vDash \theta\mathcal{C}$, then $\theta\mathcal{Q}_1 \leq \theta\mathcal{Q}_2$.*

Proof. Straightforward induction on the derivation of $\mathcal{C} \vdash_{\mathsf{sat}} \mathcal{Q}_1 \leq \mathcal{Q}_2$. □

We split clause solving into three judgments: $\mathsf{solveBases}(\Xi, \mathcal{C}) = \theta$ which solves a constraint containing only base sort variables, $\mathsf{solveArrows}(\Xi, \mathcal{C}) = \theta$ which solves a constraint containing only arrow sort variables, and the actual $\mathsf{solveClause}(\Xi, \mathcal{C}) = \theta$ which puts the results of the other two together. The split enables a proof of completeness by taking care not to be too "eager" when solving base sort constraints that may involve tension; as described above, though, since no tension can arise in arrow sort constraints, we *can* be eager in solveArrows.

$\boxed{\mathsf{solveBases}(\Xi, C) = \theta}$

$$\frac{}{\mathsf{solveBases}(\cdot, \mathbf{tt}) = []} \qquad \frac{S = \bigwedge\{Q \mid C \vdash_{\mathrm{sat}} \sigma \leq Q\} \qquad \mathsf{solveBases}(\Xi, C) = \theta}{\mathsf{solveBases}((\Xi, \sigma_P^+), C) = [\theta,\ S/\sigma]}$$

$$\frac{T = \bigsqcup\{Q \mid C \vdash_{\mathrm{sat}} Q \leq \tau\} \qquad \mathsf{solveBases}(\Xi, C) = \theta}{\mathsf{solveBases}((\Xi, \tau_P^-), C) = [\theta,\ T/\tau]}$$

$\boxed{\mathsf{solveArrows}(\Xi, C) = \theta}$

$$\frac{}{\mathsf{solveArrows}(\cdot, \mathbf{tt}) = []}$$

$$\frac{U = S_1 \to T_1 \wedge \ldots \wedge S_n \to T_n \qquad \mathsf{solveArrows}(\Xi, [U/\rho]C) = \theta}{\mathsf{solveArrows}((\Xi, \rho_{A\to B}^+), \rho \leq S_1 \to T_1 \owedge \ldots \owedge \rho \leq S_n \to T_n \owedge C) = [\theta,\ U/\rho]}$$

$$\frac{U = \bot_{A\to B} \qquad \mathsf{solveArrows}(\Xi, [U/\rho]C) = \theta}{\mathsf{solveArrows}((\Xi, \rho_{A\to B}^-), C) = [\theta,\ U/\rho]}$$

In the definition of solveClause, we check the result of solveBases: it is possible that it may return an invalid solution if the constraint given involves tension, as shown by the example above; in such cases, our algorithm gives up rather than enumerating all sorts in an interval. We write $\theta \circ \theta'$ for the concatenation of two non-overlapping substitutions.

$\boxed{\mathsf{solveClause}(\Xi, C) = \theta}$

$$\frac{\mathsf{solveBases}(\Xi_{\mathrm{base}}, C_{\mathrm{base}}) = \theta_{\mathrm{base}} \qquad \vDash \theta_{\mathrm{base}} C_{\mathrm{base}} \qquad \mathsf{solveArrows}(\Xi_{\to}, \theta_{\mathrm{base}} C_{\to}) = \theta_{\to}}{\mathsf{solveClause}((\Xi_{\mathrm{base}}, \Xi_{\to}), C_{\mathrm{base}} \owedge C_{\to}) = \theta_{\mathrm{base}} \circ \theta_{\to}}$$

A clause is unsatisfiable if saturating it yields a contradiction.

$\boxed{\mathsf{unsatClause}(\Xi, C)}$

$$\frac{C \vdash_{\mathrm{sat}} Q_1 \leq Q_2 \quad Q_1 \not\leq Q_2}{\mathsf{unsatClause}(\Xi, C)}$$

Lemma 5.22 (Soundness, solveBases). *If* solveBases$(\Xi, C) = \theta$*, then* $\vdash \theta : \Xi$.

Proof. Straightforward induction on the derivation of solveBases$(\Xi, C) = \theta$. □

Lemma 5.23 (Soundness, $\mathsf{solveArrows}$). *If* $\mathsf{solveArrows}(\Xi, C) = \theta$, *then* $\vdash \theta : \Xi$ *and* $\vDash \theta C$.

Proof. By induction on the derivation of $\mathsf{solveArrows}(\Xi, C) = \theta$, making use of the fact that $[\theta, S/\sigma]\, C = \theta\,([S/\sigma]\, C)$. ◻

Theorem 5.24 (Soundness, clause solving). *If* $\mathsf{solveClause}(\Xi, C) = \theta$, *then* $\vdash \theta : \Xi$ *and* $\vDash \theta C$.

Proof. Using Lemmas 5.22 and 5.23, along with the premise $\vDash \theta_{\text{base}} C_{\text{base}}$. ◻

Theorem 5.25 (Soundness, constraint solving). *If* $\mathsf{solve}(\Xi, \mathcal{E}) = \mathbf{\Theta}$, *then for every* $\theta \in \mathbf{\Theta}$, *we have* $\vdash \theta : \Xi$ *and* $\vDash \theta\mathcal{E}$.

Proof. By induction on the solve derivation, appealing to Theorem 5.24 when a new substitution is added. ◻

The completeness direction requires a bit more care to state and prove. First, since we do not expect our constraint solving algorithm to always succeed, we do not want a "total" completeness result saying that if a solution exists, we will find it. Instead, we want to say that if our algorithm *does* find a solution, it is at least as good as any other.

Assuming $\mathsf{solve}(\Xi, \mathcal{E})$ does return a solution $\mathbf{\Theta}$, we would like to say something like "any substitution satisfying $\mathcal{E}$ is in $\mathbf{\Theta}$". That's too imprecise, though, because $\mathsf{solve}(\Xi, -)$ only finds *most general* solutions: not every solution to $\mathcal{E}$ will be output by $\mathsf{solve}(\Xi, \mathcal{E})$, but every solution to $\mathcal{E}$ should be *subsumed* by one in $\mathsf{solve}(\Xi, \mathcal{E})$. We formalize this with a relation $\theta_1 \preccurlyeq \theta_2$, which can be read "$\theta_1$ is better than θ_2", "θ_1 is more principal than θ_2", or "θ_1 is more general than θ_2". One substitution is better than another if it provides a better solution for every variable, and a better solution is one that tends more towards the tendency of the variable, maximizing or minimizing. Formally:

$$\begin{aligned}\theta \preccurlyeq \theta' \text{ if and only if for every } \sigma^{+} \in \mathsf{dom}(\theta'),\ \theta(\sigma) \geq \theta'(\sigma)\\ \text{and for every } \sigma^{-} \in \mathsf{dom}(\theta'),\ \theta(\sigma) \leq \theta'(\sigma).\end{aligned}$$

There is an important connection between the "better" relation and tendencies which we will exploit in the proofs below.

Theorem 5.26. *The definition of the relation* $\theta \preccurlyeq \theta'$ *lifts to complete sorts.*

1. *If* $\theta \preccurlyeq \theta'$ *and* $\mathsf{tendency}(S, -)$, *then* $\theta S \leq \theta' S$.

2. *If* $\theta \preccurlyeq \theta'$ *and* $\mathsf{tendency}(S, +)$, *then* $\theta S \geq \theta' S$.

Proof. By induction on the given tendency derivation using the definition of $\theta \preccurlyeq \theta'$ at sort variables and the rules of higher-sort subsorting elsewhere. ◻

Lemma 5.27 (Completeness, $\mathsf{solveBases}$). *If* $\mathsf{solveBases}(\Xi, C) = \theta$, *then for every* $\vdash \theta' : \Xi$ *such that* $\vDash \phi(\theta' C)$ *for some* ϕ, *we have* $\theta \preccurlyeq \theta'$.

Proof. By induction on the derivation of $\mathsf{solveBases}(\Xi, C) = \theta$. In the base case, $\theta' = []$ by inversion, and $[] \preccurlyeq []$. The case for the rule where we form an intersection proceeds as follows:

$\Xi = \Xi_1, \sigma^+$	This case.
$\theta' = [\theta'_1,\, S'/\sigma^+]$, where $\vdash \theta'_1 : \Xi_1$	By inversion.
$\vDash \phi([\theta'_1,\, S'/\sigma^+]\,C)$	By assumption.
For each Q such that $C \vdash_{\mathsf{sat}} \sigma^+ \leq Q$: $S' \leq Q$	By Lemma 5.21.
$S' \leq \bigwedge\{Q \mid C \vdash_{\mathsf{sat}} \sigma^+ \leq Q\}$	By iterated $\wedge$-**R**.
$\vDash [\phi,\, S'/\sigma^+]\,(\theta' C)$	By reordering.
$\theta \preccurlyeq \theta'$	By i.h.
$[\theta,\, S/\sigma^+] \preccurlyeq [\theta',\, S'/\sigma^+]$	By definition.

The case for the remaining rule is similar, but it makes use of Lemmas 5.19 and 5.20 to show that $\bigsqcup\{Q \mid C \vdash_{\mathsf{sat}} Q \leq \tau^-\} \leq S'$. □

The completeness argument for solveArrows relies on the fact that generated arrow constraints are *non-tense*: for every generated $\rho^t \boldsymbol{\leq} \mathcal{S}$, we have $\mathsf{tendency}(\mathcal{S}, t)$.

Lemma 5.28 (Completeness, solveArrows**).** *Suppose $\vdash \phi : \Xi_0$ and $\vdash \phi' : \Xi_0$, and that $\phi \preccurlyeq \phi'$. If $\mathsf{solveArrows}(\Xi, \phi C) = \theta$ and C is non-tense, then for every $\vdash \theta' : \Xi$ such that $\vDash \theta'(\phi' C)$, we have $\theta \preccurlyeq \theta'$.*

Proof. By induction on the derivation of $\mathsf{solveArrows}(\Xi, C) = \theta$. In the base case, $\theta' = []$ by inversion, and $[] \preccurlyeq []$. The case for the rule where we form an intersection proceeds as follows:

Case:
$$\frac{U = S_1 \to T_1 \wedge \ldots \wedge S_n \to T_n \qquad \mathsf{solveArrows}(\Xi_1, [U/\rho]\,\phi C_1) = \theta_1}{\mathsf{solveArrows}(\underbrace{(\Xi_1, \rho^+_{A\to B})}_{\Xi}, \underbrace{\rho \boldsymbol{\leq} S_1 \to T_1 \owedge \ldots \owedge \rho \boldsymbol{\leq} S_n \to T_n \owedge \phi C_1}_{\phi C}) = \underbrace{[\theta_1,\, U/\rho]}_{\theta}}$$

$C = \rho^+ \boldsymbol{\leq} \mathcal{S}_1 \owedge \ldots \owedge \rho^+ \boldsymbol{\leq} \mathcal{S}_n \owedge C_1$, where $\phi\mathcal{S}_i = S_i \to T_i$	By equality reasoning.
$\theta' = [\theta'_1,\, U'/\rho^+]$, where $\vdash \theta'_1 : \Xi_1$	By inversion.
$\vDash [\theta'_1,\, U'/\rho]\,(\phi' C)$	By assumption.
For each i from 1 to n:	
$\quad [\theta'_1,\, U'/\rho]\,(\phi(\rho \boldsymbol{\leq} \mathcal{S}_i)) = U' \boldsymbol{\leq} \phi'\mathcal{S}_i$	Since $\vdash \phi', \phi : \Xi_0$ and ϕ is grounding.
$\quad \vDash U' \boldsymbol{\leq} \phi'\mathcal{S}_i$	By definition and equality reasoning.
$\quad U' \leq \phi'\mathcal{S}_i$	By definition.
$\quad \mathsf{tendency}(\mathcal{S}_i, +)$	Since C is non-tense.
$\quad \phi'\mathcal{S}_i \leq \phi\mathcal{S}_i$	By Theorem 5.26, since $\phi \preccurlyeq \phi'$.
$\quad U' \leq \phi\mathcal{S}_i$	By transitivity.
$U' \leq \phi\mathcal{S}_1 \wedge \ldots \wedge \phi\mathcal{S}_n$	By iterated $\wedge$-**R**.
$U' \leq U$	By equality reasoning.

$\vdash [\phi,\, U/\rho] : (\Xi_0, \rho)$ and $\vdash [\phi',\, U'/\rho] : (\Xi_0, \rho)$,
and $[\phi,\, U/\rho] \preccurlyeq [\phi',\, U'/\rho]$ — By definition.
$\mathsf{solveArrows}(\Xi_1, [\phi,\, U/\rho]\, C_1) = \theta_1$ — By equality reasoning on subderivation.
$\vDash \theta_1'([\phi',\, U'/\rho]\, C_1)$ — By reordering.
$\theta_1 \preccurlyeq \theta_1'$ — By i.h.

$[\theta_1,\, U/\rho] \preccurlyeq [\theta_1',\, U'/\rho]$ — By definition.

The case for the remaining rule is similar, but it makes use of Lemma 5.20 to show that $\bot_{A\to B} \leq U'$. □

Theorem 5.29 (Completeness, clause solving). *If* $\mathsf{solveClause}(\Xi, C) = \theta$, *then for every* $\vdash \theta' : \Xi$ *such that* $\vDash \theta' C$, *we have* $\theta \preccurlyeq \theta'$.

Proof. Suppose $\vdash \theta' : (\Xi_{\text{base}}, \Xi_{\to})$ and $\vDash \theta'(C_{\text{base}} \otimes C_{\to})$. By inversion, θ' must have the form $\theta'_{\text{base}} \circ \theta'_{\to}$, where $\vdash \theta'_{\text{base}} : \Xi_{\text{base}}$ and $\vdash \theta'_{\to} : \Xi_{\to}$. Note that by Lemma 5.22, $\vdash \theta_{\text{base}} : \Xi_{\text{base}}$. By Lemma 5.27 with $\phi = \theta'_{\to}$, we have $\theta_{\text{base}} \preccurlyeq \theta'_{\text{base}}$. Since $C_{\to}$ is non-tense, by Lemma 5.28, $\theta_{\to} \preccurlyeq \theta'_{\to}$. Putting it all together, we have $(\theta_{\text{base}} \circ \theta_{\to}) \preccurlyeq (\theta'_{\text{base}} \circ \theta'_{\to})$, or $\theta \preccurlyeq \theta'$. □

Theorem 5.30 (Soundness, clause unsatisfiability). *If* $\mathsf{unsatClause}(\Xi, C)$, *then there is no* $\vdash \theta : \Xi$ *such that* $\vDash \theta C$.

Proof. Suppose that $\mathsf{unsatClause}(\Xi, C)$ but that $\vdash \theta : \Xi$ and $\vDash \theta C$. By inversion, $C \vdash_{\text{sat}} Q_1 \leq Q_2$, and by Lemma 5.21, $Q_1 \leq Q_2$, a contradiction. □

Theorem 5.31 (Completeness, constraint solving). *If* $\mathsf{solve}(\Xi, \mathcal{E}) = \boldsymbol{\Theta}$, *then for every* $\vdash \theta' : \Xi$ *such that* $\vDash \theta' \mathcal{E}$, *there is some* $\theta \in \boldsymbol{\Theta}$ *such that* $\theta \preccurlyeq \theta'$.

Proof. By induction on the solve derivation. In the case where we solve a clause, Theorem 5.29 (Completeness, clause solving) and the inductive hypothesis give us the desired result. In the case where we find an unsatisfiable clause, Theorem 5.30 (Soundness, clause unsatisfiability) and the inductive hypothesis give us the desired result. □

5.5.3 Practical Sort Reconstruction Algorithm

Now that we have described practical algorithms for constraint generation and constraint solving, we can chain them together to get a practical algorithm for sort reconstruction. Recall from Section 5.2 that top-level reconstruction of declarations $s \sqsubset a :: L$ and $c :: S$ proceeds by first performing LFR type reconstruction, then by filling unknown sorts with fresh appropriate-tendency sort variables, and finally by calling out to "sort reconstruction proper" to find the best possible instantiations for those variables. We recapitulate the two key rules here.

$$\frac{\vdash \Sigma\ \mathsf{sig} \blacktriangleright \Sigma' \qquad (a{:}K, i) \in \Sigma' \qquad \vdash^{i}_{\Sigma'} L \underset{\sim}{\sqsubseteq} K \blacktriangleright L_1 \qquad L_1 \blacktriangleright^{-} (\Xi \vdash L_2) \qquad \Xi \vdash_{\Sigma'} L_2 \sqsubset K \blacktriangleright L'}{\vdash \Sigma, s{\sqsubset}a{::}L\ \mathsf{sig} \blacktriangleright \Sigma', (s{\sqsubset}a{::}L', i)}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \blacktriangleright \Sigma' \qquad (c{:}A, i) \in \Sigma' \qquad \vdash^{i}_{\Sigma'} S \underset{\sim}{\sqsubseteq} A \blacktriangleright S_1 \qquad S_1 \blacktriangleright^{-} (\Xi \vdash S_2) \qquad \Xi \vdash_{\Sigma'} S_2 \sqsubset A \blacktriangleright S'}{\vdash \Sigma, c{::}S\ \mathsf{sig} \blacktriangleright \Sigma', (c{::}S', i)}$$

In Section 5.4.2, we saw one way of specifying the sort reconstruction judgment $\Xi \vdash_{\Sigma'} S \sqsubset A \blacktriangleright S'$ (and by analogy the class reconstruction $\Xi \vdash_{\Sigma'} L \sqsubset K \blacktriangleright L'$), a single rule that explicitly enumerated and intersected together all possible instantiations.

We can similarly define a practical algorithm for sort reconstruction as a single rule that composes together the constraint generation and constraint solving judgments. As before, we focus on the rule for reconstructing a sort; the rule for reconstructing a class is analogous.

$$\frac{\Xi; \cdot \vdash S \sqsubset A \mid C \qquad \mathsf{solve}(\Xi, C) = \mathbf{\Theta}}{\Xi \vdash S \sqsubset A \blacktriangleright \bigwedge\{\theta S \mid \theta \in \mathbf{\Theta}\}}$$

In words, given a fully type-reconstructed declaration $c{::}S$ which may have some free sort variables, we compute the best instantiation by (1) generating a constraint which soundly and completely captures the requirements imposed on those sort variables by the structure of the declaration, (2) solving that constraint to obtain a sound and complete set of valid instantiations, and (3) intersecting together the results of applying all of the instantiations to the original declaration.

We can easily show that the reconstructed sort produced in this way is well-formed.

Theorem 5.32 (Soundness, practical algorithm). *If $\Xi \vdash S \sqsubset A \blacktriangleright S'$, then $\cdot \vdash S' \sqsubset A$.*

Proof. By inversion, we know $\Xi; \cdot \vdash S \sqsubset A \mid C$ and $\mathsf{solve}(\Xi, C) = \mathbf{\Theta}$, and we must show that $\cdot \vdash \bigwedge\{\theta S \mid \theta \in \mathbf{\Theta}\} \sqsubset A$.

By Theorem 5.25 (Soundness, constraint solving), for every $\theta \in \mathbf{\Theta}$, we have $\vDash \theta C$. Then for each θ, by Theorem 5.17 (Soundness, constraint generation), we have $\cdot \vdash \theta S \sqsubset A$. Finally, by several applications of the intersection refinement rule $\wedge$**-F**, we have $\cdot \vdash \bigwedge\{\theta S \mid \theta \in \mathbf{\Theta}\} \sqsubset A$, which is the desired result. □

We furthermore know that the reconstructed sort is principal, since our constraint solving algorithm makes the least possible commitments in every case. We may assume that the result of sort reconstruction has minimizing tendency since the sort reconstruction judgment is called from top-level signature reconstruction on the result of filling at minimizing tendency ("$-$") (c.f., Theorem 5.8).

Theorem 5.33 (Principality, practical algorithm). *Suppose $\Xi \vdash S \sqsubset A \blacktriangleright S'$ and that* $\mathsf{tendency}(S, -)$. *For any $\vdash \theta' : \Xi$ such that $\cdot \vdash \theta' S \sqsubset A$, we have $S' \leq \theta' S$.*

Proof. Suppose $\vdash \theta' : \Xi$ such that $\cdot \vdash \theta' S \sqsubset A$. By inversion, we know $\Xi; \cdot \vdash S \sqsubset A \mid C$ and $\mathsf{solve}(\Xi, C) = \mathbf{\Theta}$, and we must show that $\bigwedge\{\theta S \mid \theta \in \mathbf{\Theta}\} \leq \theta' S$.

By Theorem 5.18 (Completeness, constraint generation), we have $\vDash \theta' C$. Then by Theorem 5.31 (Completeness, constraint solving), there is some $\theta \in \mathbf{\Theta}$ such that $\theta \preccurlyeq \theta'$. By Theorem 5.26, since $\mathsf{tendency}(S, -)$, we have $\theta S \leq \theta' S$. Finally, by several applications of $\wedge$-$\mathbf{L}_1$ and $\wedge$-$\mathbf{L}_2$, we have $\bigwedge\{\theta S \mid \theta \in \mathbf{\Theta}\} \leq \theta' S$, which is the desired result. □

5.6 Summary

We have seen in this chapter decidability results and practical algorithms for reconstructing LFR signatures given in an implicit concrete syntax similar to that of Twelf. We explained sort reconstruction by a simple and elegant reduction to constraint satisfiability for a finitary fragment of first-order logic, and we gave an efficient algorithm for solving a class of constraints representative of many typical applications, bolstering our claim that refinement types are a practical addition to the logical framework LF.

Just how practical is our algorithm? In terms of algorithmic complexity, we cannot hope for very much, since as mentioned in Chapter 2, even the problem of sort *checking* for LFR is PSPACE-hard. By focusing on situations that seem to arise in practice and ruling out enumerative cases that would obviously be expensive, we believe we have constructed an algorithm that hits a certain "sweet spot": efficient enough to handle typical situations despite being decidedly partial. Our preliminary experiments bear this out, with our algorithm returning the expected reconstruction on almost all of the examples we've considered, and requiring only minimal annotations to cover the remainder.

Now that we can work in a practical concrete syntax similar to Twelf's, we are ready to explore the expressive power of refinements through case studies more substantial than the small examples we have restricted ourselves to until now.

Chapter 6

Case Studies

In this chapter, we present a few larger, more realistic case studies showing how we expect LFR to be used in practice. All of the case studies are inspired by representational challenges that have come up during the course of actual research.

The examples we give are divided into three sections. First, we explore several encodings related to the field of programming languages, where refinements primarily help us to get a handle on subsyntax and invariants captured by the forms of judgments. Next, we cover a variety of examples from logic and proof theory, where refinements are particularly useful for isolating subsets of derivations. Finally, we present a series of challenges that arise specifically in the study of logical frameworks, several of which come from this very dissertation.

6.1 Programming Languages

6.1.1 Fragments of Polymorphism

The study of polymorphism is a rich and beautiful subject, seemingly simple but with deep implications. As first studied indepently by Girard [Gir72] and Reynolds [Rey74], the essence of polymorphism was to extend the type structure of the λ-calculus with universally quantified types:

$$t ::= \alpha \mid b \mid t_1 \to t_2 \mid \forall\alpha.\, t$$

Polymorphic types are easy to represent in LF using higher-order abstract syntax:

$base$: type.
tp : type.

? : $base \to tp$.
➪ : $tp \to tp \to tp$.
∀ : $(tp \to tp) \to tp$.

Since polymorphism was first invented, people have studied various fragments of it that have important properties like decidable type inference. Here, we consider two of them: the prenex polymorphism of ML [DM82] and rank-2 polymorphism [KT92].

Damas and Milner's [DM82] type inference algorithm for ML is based around the idea of prenex polymorphism, or *type schemes*, where all of the quantifiers are outermost:

$$\begin{array}{lr}
\tau ::= \alpha \mid b \mid \tau_1 \to \tau_2 & \text{simple types} \\
\sigma ::= \tau \mid \forall\alpha.\,\sigma & \text{type schemes}
\end{array}$$

We can represent type schemes succinctly in LFR, using a sort family *simple* for the simple types and a sort family *prenex* for the type schemes:

$simple \sqsubset tp.$

$? :: base \to simple.$
$⇨ :: simple \to simple \to simple.$

$prenex \sqsubset tp.$

$simple \leq prenex.$
$\forall :: (simple \to prenex) \to prenex.$

As we have seen before, the inclusion of simple types into prenex type schemes is captured by a subsorting declaration.

ML polymorphism is celebrated for its combination of simplicity and power, but it is not the only decidable fragment of polymorphism. Kfoury and Tiuryn [KT92] showed that type inference for rank-2 polymorphism is polynomial-time equivalent to ML type inference. Polymorphism of rank k, intuitively, is the fragment where quantified types may only appear in positions that are not to the left of more than k arrows. Kfoury and Tiuryn's work is based on a system slightly more restrictive than but ultimately equivalent to rank-2 polymorphism, in which quantifiers are pushed as far to the front as possible:

$$\begin{array}{lr}
\tau ::= \alpha \mid b \mid \tau_1 \to \tau_2 & \text{simple types} \\
\sigma ::= \tau \mid \forall\alpha.\,\sigma & \text{type schemes} \\
\rho ::= \tau \mid \sigma \to \tau & \text{rank-2 types} \\
\pi ::= \rho \mid \forall\alpha.\,\pi & \text{rank-2 type schemes}
\end{array}$$

As is evident, the system of rank-2 types is a strict superset of prenex polymorphism. A rank-2 type scheme has the form $\forall\alpha_1.\ \ldots\forall\alpha_n.\,\sigma_1 \to \ldots\sigma_m \to \tau$, where each σ_i is a type scheme and τ is a simple type. Using a feature of our implementation wherein multiple declarations for a constant are interpreted as conjuncts of an intersection declaration, we can extend our encoding of prenex polymorphism to one of rank-2 polymorphism, writing *rank2* for the rank-2 types and *prenex2* for the rank-2 type schemes.

$rank2 \sqsubset tp.$

prenex2 $\sqsubset$ *tp*.

simple $\leq$ *rank2*.
➪ :: *prenex* $\rightarrow$ *rank2* $\rightarrow$ *rank2*.

rank2 $\leq$ *prenex2*.
∀ :: (*simple* $\rightarrow$ *prenex2*) $\rightarrow$ *prenex2*.

Suppose we wanted to formalize the equivalence of the simplified rank-2 system and the full strength rank-2 system. The full system's type language has a regular structure that captures the intuition that a rank-k arrow type can only have a domain of rank $k-1$ or lower.

$R(0) ::= \alpha \mid b \mid R(0) \rightarrow R(0)$	rank-0 types
$R(1) ::= R(0) \mid R(0) \rightarrow R(1) \mid \forall\alpha.\, R(1)$	rank-1 types
$R(2) ::= R(1) \mid R(1) \rightarrow R(2) \mid \forall\alpha.\, R(2)$	rank-2 types

Observe that the full system is indeed a generalization of the restricted one: every rank-2 type scheme π is a member of $R(2)$, and every ML type scheme σ is a member of $R(1)$, but in neither case does the converse hold. We can easily define the full system in LFR, opening the way for a formalization of Kfoury and Tiuryn's results.

r0 $\sqsubset$ *tp*.

? :: *base* $\rightarrow$ *r0*.
➪ :: *r0* $\rightarrow$ *r0* $\rightarrow$ *r0*.

r1 $\sqsubset$ *tp*.

r0 $\leq$ *r1*.
➪ :: *r0* $\rightarrow$ *r1* $\rightarrow$ *r1*.
∀ :: (*r0* $\rightarrow$ *r1*) $\rightarrow$ *r1*.

r2 $\sqsubset$ *tp*.

r1 $\leq$ *r2*.
➪ :: *r1* $\rightarrow$ *r2* $\rightarrow$ *r2*.
∀ :: (*r0* $\rightarrow$ *r2*) $\rightarrow$ *r2*.

6.1.2 Values and Computations

A typical practice in the study of the operational semantics of programming languages is to distinguish values from arbitrary computations. Earlier, we saw an encoding of values and expressions along with their evaluation judgment. We recap it here and extend it with pairing and projections.

```
exp : type.

lam : (exp → exp) → exp.
app : exp → exp → exp.
pair : exp → exp → exp.

val ⊏ exp.
cmp ⊏ exp.

val ≤ cmp.

lam :: (val → cmp) → val.
app :: cmp → cmp → cmp.
pair :: val → val → val
    ∧  cmp → cmp → cmp.
pi1 :: cmp → cmp.
pi2 :: cmp → cmp.

eval :: cmp → val → type.

ev/lam  :: eval (lam E) (lam E).

ev/app :: eval (app E1 E2) V
       ← eval E1 (lam E1′)
       ← eval E2 V2
       ← eval (E1′ V2) V.

ev/pair :: eval (pair E1 E2) (pair V1 V2)
        ← eval E1 V1
        ← eval E2 V2.

ev/pi1 :: eval (pi1 E) V1
       ← eval E (pair V1 V2).

ev/pi2 :: eval (pi2 E) V2
       ← eval E (pair V1 V2).
```

We present this extended example to highlight the expressive power of intersections: the *pair* constructor can be applied either to two values, with the result being a value, or to two computations, with the result being a computation. The duality is exploited in the *ev/pair* rule, where the first pair is a computation and the second a value.

Contrast this with the way the example might be represented in ordinary LF, with two separate types for values and computations.

```
val : type.
```

```
cmp : type.

% ...

pair : cmp → cmp → cmp.
vpair : val → val → val.

% ...

eval : cmp → val → type.

% ...

ev/pair : eval (pair E1 E2) (vpair V1 V2)
        ← eval E1 V1
        ← eval E2 V2.
```

In the LF encoding, we have been forced to introduce a second constructor for pairs, one which takes in values and returns a value. If we were to define other judgments on λ-expressions, we would have to define them twice—once for computations and once for values—with duplicated cases for the pair constructs. If we were to prove theorems about expressions, we would have to prove them twice as well. The situation can be quite cumbersome if our encoded language's values and computations overlap significantly. Using the LFR approach, we encode just one language and isolate whatever fragments we wish.

Of course, we always have the option to work in pure LF and use judgments to isolate fragments of the language, and this is also frequently done in practice, but it comes with all the caveats we mentioned in Section 2.2: the burden of proof is now on the user to verify that representations are used correctly.

6.1.3 Other Evaluation Strategies

"Values and expressions" is a typical theme that comes up frequently in the literature. In Harper and Lillibridge's study of explicit polymorphism and CPS [HL93], they delineate several evaluation strategies for an extension of F^ω with control operators. For each evaluation strategy, they give a grammar of values, redexes, and evaluation contexts. We can isolate each of these sub-languages as refinements of a larger language. Ignoring the kind and constructor levels, the syntax of the larger language is given as follows:

$$\begin{array}{rl} \text{kinds} & K ::= \dots \\ \text{constructors} & A ::= \dots \\ \text{terms} & M ::= x \mid \lambda x{:}A.\, M \mid M_1\, M_2 \mid \Lambda u{:}K.\, M \mid M\{A\} \mid \mathit{callcc}_A(M) \mid \mathit{abort}_A(M) \end{array}$$

It is encoded in LF in the usual manner:

knd : type.
con : type.

tm : type.

$lam : con \to (tm \to tm) \to tm.$
$app : tm \to tm \to tm.$
$Lam : knd \to (con \to tm) \to tm.$
$App : tm \to con \to tm.$
$callcc : con \to tm \to tm.$
$abort : con \to tm \to tm.$

A complete grammar subsuming all manners of evaluation contexts is never given in the paper, but we can induce what it must be from the sub-languages:

$$\text{evaluation contexts} \;\; E ::= [] \mid E\,M \mid M\,E \mid \Lambda u{:}K.\,E \mid E\{A\}$$

Again, it is not difficult to encode this grammar in LF:

evctx : type.

$\langle\rangle : evctx.$
$capp1 : evctx \to tm \to evctx.$
$capp2 : tm \to evctx \to evctx.$
$cLam : knd \to (con \to evctx) \to evctx.$
$cApp : evctx \to con \to evctx.$

Some auxiliary notions will turn out to be useful below. First, when encoding redexes, it is useful to isolate immediate abstractions as refinements of terms.

$lambda \sqsubset tm.\;\; lambda :: \text{sort}.$
$lam :: \top \to (\top \to \top) \to lambda.$

$Lambda \sqsubset tm.\;\; Lambda :: \text{sort}.$
$Lam :: \top \to (\top \to \top) \to Lambda.$

Second, it will turn out that the control operators are always redexes, so it is useful to isolate them as a refinement *control*.

$control \sqsubset tm.\;\; control :: \text{sort}.$
$abort :: \top \to \top \to control.$
$callcc :: \top \to \top \to control.$

Call-by-Value (CBV) Strategy. Harper and Lillibridge give the syntax of CBV values, redexes, and expressions as follows:

$$V ::= x \mid \lambda x{:}A.\,M \mid \Lambda u{:}K.\,M$$
$$R ::= (\lambda x{:}A.\,M)\;V \mid (\Lambda u{:}K.\,M)\{A\} \mid abort_A(M) \mid callcc_A(M)$$
$$E ::= [] \mid E\,M \mid V\,E \mid E\{A\}$$

We encode this grammar in LFR by delineating three new refinements.

$v/value \sqsubset tm.\ \ v/value :: \mathsf{sort}.$ % CBV values V.
$v/redex \sqsubset tm.\ \ v/redex :: \mathsf{sort}.$ % CBV redexes R.
$v/evctx \sqsubset evctx.\ \ v/evctx :: \mathsf{sort}.$ % CBV eval contexts E.

We must take care to define a new "immediate abstraction" refinement, since in the call-by-value language, variables represent values. (Compare with the declarations above for the sort *lambda*.)

$v/lambda \sqsubset tm.\ \ v/lambda :: \mathsf{sort}.$
$lam :: \top \to (v/value \to \top) \to v/lambda.$

The only values are CBV λ-abstractions and ordinary Λ-abstractions.

$v/lambda \leq v/value.$
$Lambda \leq v/value.$

Redexes are either applications of an appropriate abstraction to an appropriate argument or control operators.

$app :: v/lambda \to v/value \to v/redex.$
$App :: Lambda \to \top \to v/redex.$
$control \leq v/redex.$

The evaluation contexts follow in a straightforward manner.

$\langle\rangle :: v/evctx.$
$capp1 :: v/evctx \to \top \to v/evctx.$
$capp2 :: v/value \to v/evctx \to v/evctx.$
$cApp :: v/evctx \to \top \to v/evctx.$

The remaining sub-languages follow similar techniques, so we simply give the grammars and their encodings. Two "ML-like" strategies are interesting in that they allow reduction under Λ-abstractions.

Call-by-Name (CBN) Strategy.

$$\begin{aligned} V &::= \lambda x{:}A.\, M \mid \Lambda u{:}K.\, M \\ R &::= (\lambda x{:}A.\, M_1)\, M_2 \mid (\Lambda u{:}K.\, M)\{A\} \mid abort_A(M) \mid callcc_A(M) \\ E &::= [\,] \mid E\, M \mid E\{A\} \end{aligned}$$

$n/value \sqsubset tm.\ \ n/value :: \mathsf{sort}.$ % CBN values V.
$n/redex \sqsubset tm.\ \ n/redex :: \mathsf{sort}.$ % CBN redexes R.
$n/evctx \sqsubset evctx.\ \ n/evctx :: \mathsf{sort}.$ % CBN eval contexts E.

$lambda \leq n/value.$
$Lambda \leq n/value.$

$app :: lambda \to \top \to n/redex.$
$App :: Lambda \to \top \to n/redex.$
$control \leq n/redex.$

$\langle\rangle :: n/evctx.$
$capp1 :: n/evctx \to \top \to n/evctx.$
$cApp :: n/evctx \to \top \to n/evctx.$

ML-like Call-by-Value (ML-CBV) Strategy .

$$
\begin{aligned}
V &::= x \mid \lambda x{:}A.\, M \mid \Lambda u{:}K.\, V \\
R &::= (\lambda x{:}A.\, M)\ V \mid (\Lambda u{:}K.\, V)\{A\} \mid abort_A(M) \mid callcc_A(M) \\
E &::= [] \mid E\ M \mid V\ E \mid \Lambda u{:}K.\, E \mid E\{A\}
\end{aligned}
$$

$mlv/value \sqsubset tm.$ $mlv/value ::$ sort. % ML−CBV values V.
$mlv/redex \sqsubset tm.$ $mlv/redex ::$ sort. % ML−CBV redexes R.
$mlv/evctx \sqsubset evctx.$ $mlv/evctx ::$ sort. % ML−CBV eval contexts E.

$mlv/lambda \sqsubset tm.$ $mlv/lambda ::$ sort.
$lam :: \top \to (mlv/value \to \top) \to mlv/lambda.$
$mlv/Lambda \sqsubset tm.$ $mlv/Lambda ::$ sort.
$Lam :: \top \to (\top \to mlv/value) \to mlv/Lambda.$

$mlv/lambda \leq mlv/value.$
$mlv/Lambda \leq mlv/value.$

$app :: mlv/lambda \to mlv/value \to mlv/redex.$
$App :: mlv/Lambda \to \top \to mlv/redex.$
$control \leq mlv/redex.$

$\langle\rangle :: mlv/evctx.$
$capp1 :: mlv/evctx \to \top \to mlv/evctx.$
$capp2 :: mlv/value \to mlv/evctx \to mlv/evctx.$
$cLam :: \top \to (\top \to mlv/evctx) \to mlv/evctx.$
$cApp :: mlv/evctx \to \top \to mlv/evctx.$

ML-like Call-by-Name (ML-CBN) Strategy.

$$
\begin{aligned}
V &::= \lambda x{:}A.\, M \mid \Lambda u{:}K.\, V \\
R &::= (\lambda x{:}A.\, M_1)\ M_2 \mid (\Lambda u{:}K.\, V)\{A\} \mid abort_A(M) \mid callcc_A(M) \\
E &::= [] \mid E\ M \mid \Lambda u{:}K.\, E \mid E\{A\}
\end{aligned}
$$

$mln/value \sqsubset tm.\ \ mln/value :: \mathsf{sort}.$ % ML−CBN values V.
$mln/redex \sqsubset tm.\ \ mln/redex :: \mathsf{sort}.$ % ML−CBN redexes R.
$mln/evctx \sqsubset evctx.\ \ mln/evctx :: \mathsf{sort}.$ % ML−CBN eval contexts E.

$mln/Lambda \sqsubset tm.\ \ mln/Lambda :: \mathsf{sort}.$
$Lam :: \top \to (\top \to mln/value) \to mln/Lambda.$

$lambda \leq mln/value.$
$mln/Lambda \leq mln/value.$

$app :: lambda \to \top \to mln/redex.$
$App :: mln/Lambda \to \top \to mln/redex.$
$control \leq mln/redex.$

$\langle\rangle :: mln/evctx.$
$capp1 :: mln/evctx \to \top \to mln/evctx.$
$cLam :: \top \to (\top \to mln/evctx) \to mln/evctx.$
$cApp :: mln/evctx \to \top \to mln/evctx.$

6.1.4 Weak Head Normal Types in Higher-Order Subtyping

Compagnoni and Goguen [CG03] present a typed operational semantics for an $F^{\omega}_{\leq}$-like language with higher-order subtyping. Typed operational semantics is a way of presenting type theories similar to the canonical forms/hereditary substitutions methodology, and it leads to similar reductions in the complexity of metatheoretic arguments.

The constructors of the language they define is given as follows:

$$K ::= \ldots$$
$$A ::= X \mid A \to A \mid \forall X{\leq}A{:}K.\, A \mid \Lambda X{\leq}A{:}K.\, A \mid A\ A \mid \mathrm{T}_\star$$

Its encoding in LF is typical:

$kd : \mathsf{type}.$
$tp : \mathsf{type}.$

$T_\star : tp.$
$arrow : tp \to tp \to tp.$
$all : tp \to kd \to (tp \to tp) \to tp.$
$Lam : tp \to kd \to (tp \to tp) \to tp.$
$App : tp \to tp \to tp.$

The work makes use of a notion of weak head normal form for type constructors, which is described by the following definition [CG03]:

DEFINITION 3.1 (*Weak-Head Normal*)
$T_\star$, $A_1 \to A_2$, $\forall X{\le}A{:}K.\, B$, and $\Lambda X{\le}A{:}K.\, B$ are weak head normal.
$X(A_1, \ldots, A_n)$ is weak head normal if $A_1, \ldots A_n$ are in normal form.

To encode this notion in LFR, we first encode normal types via the usual "atomic/normal" strategy:

$atp \sqsubset tp.$ $atp :: \mathsf{sort}.$
$ntp \sqsubset tp.$ $ntp :: \mathsf{sort}.$

$T_\star :: ntp.$
$arrow :: ntp \to ntp \to ntp.$
$all :: ntp \to \top \to (atp \to ntp) \to ntp.$
$Lam :: ntp \to \top \to (atp \to ntp) \to ntp.$
$App :: atp \to ntp \to atp.$
$atp \le ntp.$

We can then isolate the weak head normal types by directly following the definition. Note that we already have a way of describing a variable type applied to a series of normal arguments, *atp*, so we use a subsort inclusion to declare that such types are weak head normal.

$whntp \sqsubset tp.$ $whntp :: \mathsf{sort}.$
$T_\star :: whntp.$
$arrow :: \top \to \top \to whntp.$
$all :: \top \to \top \to (atp \to \top) \to whntp.$
$Lam :: \top \to \top \to (atp \to \top) \to whntp.$
$atp \le whntp.$

6.1.5 Singleton Kind Elimination

Singleton kinds can be used in type-preserving compilers to track the partial identities of type variables. Crary [Cra07] gives a sound and complete procedure for compiling away singleton kinds, in the form of a translation from a language with singletons to a language without. Using sorts, we can encode the singleton-free language as a refinement of the full language, both in terms of the syntactic elements and in terms of the rules defining judgments over those elements. In doing so, we highlight the expressive power afforded by the uniformity of the LFR approach.

The kinds of the singleton calculus are the base kind "type", singletons of constructors, and dependent products and sums of kinds. We focus our attention on the kind layer, but note that the constructor layer includes the variables bound by dependent product and sum kinds.

$$\begin{aligned} \text{kinds}\ \ K &::= T \mid S(c) \mid \Pi\alpha{:}K_1.\, K_2 \mid \Sigma\alpha{:}K_1.\, K_2 \\ \text{constructors}\ \ c &::= \alpha \mid \ldots \end{aligned}$$

As usual, the encoding in LF is typical.

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash T = T}\ \textbf{(14)} \qquad \frac{\Gamma \vdash c_1 = c_2 : T}{\Gamma \vdash S(c_1) = S(c_2)}\ \textbf{(15)}$$

$$\frac{\Gamma \vdash K_2' = K_1' \qquad \Gamma, \alpha{:}K_1' \vdash K_1'' = K_2''}{\Gamma \vdash \Pi\alpha{:}K_1'.\, K_1'' = \Pi\alpha{:}K_2'.\, K_2''}\ \textbf{(16)} \qquad \frac{\Gamma \vdash K_1' = K_2' \qquad \Gamma, \alpha{:}K_1' \vdash K_1'' = K_2''}{\Gamma \vdash \Sigma\alpha{:}K_1'.\, K_1'' = \Sigma\alpha{:}K_2'.\, K_2''}\ \textbf{(17)}$$

Figure 6.1: Kind equivalence in the singleton calculus.

kd : type.
tp : type.

$t : kd.$
$sing : tp \to kd.$
$pi : kd \to (tp \to kd) \to kd.$
$sigma : kd \to (tp \to kd) \to kd.$

One of the judgments of the singleton calculus is the equivalence of kinds, written $\Gamma \vdash K_1 = K_2$. Typically kind equivalence is trivial, but in the singleton calculus, it must depend on constructor equivalence. The four rules defining kind equivalence are shown in Figure 6.1, exactly as they appear in Crary's article. We can encode the rules of the kind equivalence judgment in LF using the usual technique of higher-order judgments. We declare, but do not define, judgments for type equivalence (*eq*) and kinding of constructors (*kof*).

$eq : tp \to tp \to kd \to$ type.
$kof : tp \to kd \to$ type.
% ...

$keq : kd \to kd \to$ type.

$rule14 : keq\ t\ t.$

$rule15 : keq\ (sing\ C_1)\ (sing\ C_2)$
$\quad \leftarrow eq\ C_1\ C_2\ t.$

$rule16 : keq\ (pi\ K_1'\ \lambda a.K_1''\ a)\ (pi\ K_2'\ \lambda a.K_2''\ a)$
$\quad \leftarrow keq\ K_2'\ K_1'$
$\quad \leftarrow (\Pi a.\ kof\ a\ K_1' \to keq\ (K_1''\ a)\ (K_2''\ a)).$

$rule17 : keq\ (sigma\ K_1'\ \lambda a.K_1''\ a)\ (sigma\ K_2'\ \lambda a.K_2''\ a)$
$\quad \leftarrow keq\ K_1'\ K_2'$
$\quad \leftarrow (\Pi a.\ kof\ a\ K_1' \to keq\ (K_1''\ a)\ (K_2''\ a)).$

Crary defines the singleton-free target language in terms of its differences from the singleton calculus: its kinds and constructors may not mention any singleton kinds, and any rules dealing with singleton kinds are omitted, including rule 15 above. To encode the syntax of the singleton-free language, we declare two refinements *sf/kd* and *sf/tp*. The constructor for singletons, *sing*, does not belong to the refinement *sf/kd*.

$sf/kd \sqsubset kd.$
$sf/tp \sqsubset tp.$

$t :: sf/kd.$
% no "sing" declaration
$pi :: sf/kd \to (sf/tp \to sf/kd) \to sf/kd.$
$sigma :: sf/kd \to (sf/tp \to sf/kd) \to sf/kd.$

The singleton-free judgments are defined as refinements of the ordinary ones, noting that they apply only to singleton-free syntactic elements. Rule 15 is omitted.

$sf/eq \sqsubset eq :: sf/tp \to sf/tp \to sf/kd \to \mathsf{sort}.$
$sf/kof \sqsubset kof :: sf/tp \to sf/kd \to \mathsf{sort}.$
% ...

$sf/keq \sqsubset keq :: sf/kd \to sf/kd \to \mathsf{sort}.$

$rule14 :: sf/keq\ t\ t.$

% no "rule15" declaration

$rule16 :: sf/keq\ (pi\ K_1'\ \lambda a.K_1''\ a)\ (pi\ K_2'\ \lambda a.K_2''\ a)$
$\quad\leftarrow sf/keq\ K_2'\ K_1'$
$\quad\leftarrow (\Pi a.\ sf/kof\ a\ K_1' \to sf/keq\ (K_1''\ a)\ (K_2''\ a)).$

$rule17 :: sf/keq\ (sigma\ K_1'\ \lambda a.K_1''\ a)\ (sigma\ K_2'\ \lambda a.K_2''\ a)$
$\quad\leftarrow sf/keq\ K_1'\ K_2'$
$\quad\leftarrow (\Pi a.\ sf/kof\ a\ K_1' \to sf/keq\ (K_1''\ a)\ (K_2''\ a)).$

The singleton-free rules are all well-sorted since they only apply to singleton-free syntax. We now have an encoding of the target singleton-free calculus suitable for formalizing Crary's translation.

It is worth noting that the LFR framework prevents one from accidentally including rule 15 in the singleton-free calculus, since the singleton-free judgments should only apply to singleton-free syntax. Since there is no singleton-free declaration for the *sing* constructor, the following declaration would produce a sort error:

% *** sort error: ***
$rule15 :: sf/keq\ (sing\ C_1)\ (sing\ C_2)$
$\quad\leftarrow sf/eq\ C_1\ C_2\ t.$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash * : \Box} \qquad \frac{\Gamma \vdash T_1 : s_1 \qquad \Gamma, x{:}T_1 \vdash T_2 : s_2}{\Gamma \vdash \Pi x{:}T_1.\, T_2 : s_2}\,(s_1, s_2)$$

$$\frac{\Gamma \vdash U_1 : s \qquad \Gamma, x{:}U_1 \vdash T : U_2}{\Gamma \vdash \lambda x{:}U_1.\, T : \Pi x{:}U_1.\, U_2} \qquad \frac{\Gamma \vdash T_1 : \Pi x{:}U_2.\, U \qquad \Gamma \vdash T_2 : U_2}{\Gamma \vdash T_1\ T_2 : [T_2/x]\, U}$$

Figure 6.2: The λ-cube

So not only does LFR allow one to easily isolate a syntactic and judgmental subset of a language, but it also protects one from the mistake of doing so inconsistently.

6.1.6 The λ-Cube

In this section we present an encoding of a system similar to Barendregt's cube of typed λ-calculi [Bar92] which makes extensive use of intersections as finitary polymorphism. The rules are given first without distinguishing the levels of objects, type families, and kinds, and then refinements are given that make all distinctions apparent.

The syntax of our system is the same as Barendregt's, an undifferentiated mass of terms representing objects, families, and constructors.

$$T, U ::= x \mid * \mid \Box \mid T_1\ T_2 \mid \lambda x{:}U.\, T \mid \Pi x{:}U_1.\, U_2$$

The rules we encode are shown in Figure 6.2. They are not precisely the same as Barendregt's: in particular, the rule for checking λ abstractions does not check the well-formedness of its entire classifier, but rather just the domain. In the PTS presentation of the cube, the check on the classifier serves to limit the kinds of λ-abstractions we consider to those that should be present in the current point on the cube; we will encode the limitation directly using intersections. We also omit the conversion rule for brevity.

The rule labelled "s_1, s_2" is what Barendregt referred to as the "specific rules": there would be one for each pair (s_1, s_2) currently in effect, where s_1 and s_2 are either $*$ or $\Box$. Our encoding represents the syntax and rules in the usual manner, but with one perhaps alarming change: instead of distinguishing between $*$ and $\Box$, we have just one term representing both, written *tp*. This change is alarming because the rule which previously said $* : \Box$ now says *tp* : *tp*, a looming paradox! But we will sort things out shortly with refinements.

$term : \mathsf{type}.$

$tp : term.$
$pi : term \to (term \to term) \to term.$
$lm : term \to (term \to term) \to term.$
$ap : term \to term \to term.$

$of : term \to term \to \mathsf{type}.$

$of\text{-}tp : of\ tp\ tp.$

$of\text{-}pi : of\ (pi\ T_1\ \lambda x.T_2\ x)\ tp$
$\quad \leftarrow of\ T_1\ tp$
$\quad \leftarrow (\Pi x{:}term.\ of\ x\ T_1 \to of\ (T_2\ x)\ tp).$

$of\text{-}lm : of\ (lm\ U_1\ \lambda x.T\ x)\ (pi\ U_1\ \lambda x.U_2\ x)$
$\quad \leftarrow of\ U_1\ tp$
$\quad \leftarrow (\Pi x{:}term.\ of\ x\ U_1 \to of\ (T\ x)\ (U_2\ x)).$

$of\text{-}ap : of\ (ap\ T_1\ T_2)\ (U\ T_2)$
$\quad \leftarrow of\ T_1\ (pi\ U_2\ \lambda x.U\ x)$
$\quad \leftarrow of\ T_2\ U_2.$

We sort things out by declaring four refinements of terms: objects, families, kinds, and hyperkinds. Hyperkinds classify kinds, but we will see there is only one, namely $\square$.

$hyp \sqsubset term :: \mathsf{sort}.$
$knd \sqsubset term :: \mathsf{sort}.$
$fam \sqsubset term :: \mathsf{sort}.$
$obj \sqsubset term :: \mathsf{sort}.$

The constant *tp* represents both $*$ and $\square$. Since $*$ is a kind, the kind "type", and $\square$ is a hyperkind, the hyperkind "kind", the constant *tp* is classified as both.

$tp :: hyp \wedge knd.$

The remainder of the constructors are given intersection types that bake in all eight points of the λ-cube.

```
pi :: fam → (obj → fam) → fam      % dependent function types, {x:A} B.
  ∧ fam → (obj → knd) → knd        % type family kinds, {x:A} K.
  ∧ knd → (fam → fam) → fam        % polymorphic function types, {a:K} A.
  ∧ knd → (fam → knd) → knd.       % type operator kinds, {a:K1} K2.

lm :: fam → (obj → obj) → obj      % functions, [x:A] M.
  ∧ fam → (obj → fam) → fam        % type families, [x:A] B.
  ∧ knd → (fam → obj) → obj        % polymorphic abstractions, [a:K] M.
  ∧ knd → (fam → fam) → fam.       % type operators [a:K] A.

ap :: obj → obj → obj              % ordinary application, M N
  ∧ fam → obj → fam                % type family application, A M
  ∧ obj → fam → obj                % polymorphic instantiation, M [A]
  ∧ fam → fam → fam.               % type operator instatiation, A B
```

Then we define a stratified typing judgment *of′*, which tells us that kinds are classified by hyperkinds, families by kinds, and objects by families.

$$\begin{aligned}&\mathit{of}' \sqsubset \mathit{of}\\ &\quad :: \mathit{knd} \to \mathit{hyp} \to \mathsf{sort}\\ &\quad \wedge \mathit{fam} \to \mathit{knd} \to \mathsf{sort}\\ &\quad \wedge \mathit{obj} \to \mathit{fam} \to \mathsf{sort}.\end{aligned}$$

Now we simply recapitulate the rules from before—but they mean much more, now, since the form of the *of′* judgment are much more stringent than the form of the *of* judgment. Note in particular that the alarming declaration *of′ tp tp* is no longer worrisome: the left *tp* must have sort *knd* and the right one *hyp*, so we are in no danger of paradox.

$$\mathit{of\text{-}tp} :: \mathit{of}'\ \mathit{tp}\ \mathit{tp}.$$

$$\begin{aligned}&\mathit{of\text{-}pi} :: \mathit{of}'\ (\mathit{pi}\ T_1\ \lambda x.T_2\ x)\ \mathit{tp}\\ &\qquad \leftarrow \mathit{of}'\ T_1\ \mathit{tp}\\ &\qquad \leftarrow (\Pi x.\ \mathit{of}'\ x\ T_1 \to \mathit{of}'\ (T_2\ x)\ \mathit{tp}).\end{aligned}$$

$$\begin{aligned}&\mathit{of\text{-}lm} :: \mathit{of}'\ (\mathit{lm}\ U_1\ \lambda x.T\ x)\ (\mathit{pi}\ U_1\ \lambda x.U_2\ x)\\ &\qquad \leftarrow \mathit{of}'\ U_1\ \mathit{tp}\\ &\qquad \leftarrow (\Pi x.\ \mathit{of}'\ x\ U_1 \to \mathit{of}'\ (T\ x)\ (U_2\ x)).\end{aligned}$$

$$\begin{aligned}&\mathit{of\text{-}ap} :: \mathit{of}'\ (\mathit{ap}\ T_1\ T_2)\ (U\ T_2)\\ &\qquad \leftarrow \mathit{of}'\ T_1\ (\mathit{pi}\ U_2\ \lambda x.U\ x)\\ &\qquad \leftarrow \mathit{of}'\ T_2\ U_2.\end{aligned}$$

In theory, sort reconstruction should be able to automatically infer which instances of these rules are valid given the declarations we gave for the term constructors. Then we should be able to obtain any point on the cube by deleting some conjuncts from our constructor declarations above.

Unfortunately, the problem turns out to be just beyond the limits of our algorithms. First, we reconstruct *of-pi*. Focusing on just the parts of the declaration introduced by sort reconstruction and eliding the remainder with ellipses, we get the following output:

$$\begin{aligned}&\mathit{of\text{-}pi} :: \Pi T_1{::}\mathit{fam}.\ \Pi T_2{::}\mathit{obj} \to \mathit{fam}.\ ... \leftarrow (\Pi x{::}\mathit{hyp}{\wedge}\mathit{knd}{\wedge}\mathit{fam}{\wedge}\mathit{obj}.\ ...)\\ &\qquad \wedge\ \Pi T_1{::}\mathit{fam}.\ \Pi T_2{::}\mathit{obj} \to \mathit{knd}.\ ... \leftarrow (\Pi x{::}\mathit{hyp}{\wedge}\mathit{knd}{\wedge}\mathit{fam}{\wedge}\mathit{obj}.\ ...)\\ &\qquad \wedge\ \Pi T_1{::}\mathit{knd}.\ \Pi T_2{::}\mathit{fam} \to \mathit{fam}.\ ... \leftarrow (\Pi x{::}\mathit{hyp}{\wedge}\mathit{knd}{\wedge}\mathit{fam}{\wedge}\mathit{obj}.\ ...)\\ &\qquad \wedge\ \Pi T_1{::}\mathit{knd}.\ \Pi T_2{::}\mathit{fam} \to \mathit{knd}.\ ... \leftarrow (\Pi x{::}\mathit{hyp}{\wedge}\mathit{knd}{\wedge}\mathit{fam}{\wedge}\mathit{obj}.\ ...).\end{aligned}$$

So far, things seem roughly as expected: we see the four sorts we expect for functions in this far corner of the cube. The sort of the bound variable in the subgoal is somewhat surprising—we might expect it to be *obj* in the first two conjuncts and *fam* in the last two—but after a moment's thought, we realize that the sort assigned to x is in a minimizing position, so indeed, this represents a better solution than the expected one.

Moving forward, we try *of-lm*. Again, we focus our attention on the newly introduced sort information and elide the remainder with ellipses:

of-lm ::

$\Pi U_1{::}fam.\ \Pi U_2{::}obj \to fam.\ \Pi T{::}obj \to obj.\ ... \leftarrow (\Pi z{::}hyp{\wedge}knd{\wedge}fam{\wedge}obj.\ ...)$
$\wedge\ \Pi U_1{::}fam.\ \Pi U_2{::}obj \to knd.\ \Pi T{::}obj \to fam.\ ... \leftarrow (\Pi z{::}hyp{\wedge}knd{\wedge}fam{\wedge}obj.\ ...)$
$\wedge\ \Pi U_1{::}fam{\wedge}knd.\ \Pi U_2{::}fam \to fam.\ \Pi T{::}obj \to obj.\ ... \leftarrow (\Pi z{::}hyp{\wedge}knd{\wedge}fam{\wedge}obj.\ ...)$
$\wedge\ \Pi U_1{::}knd{\wedge}fam.\ \Pi U_2{::}obj \to fam.\ \Pi T{::}fam \to obj.\ ... \leftarrow (\Pi z{::}hyp{\wedge}knd{\wedge}fam{\wedge}obj.\ ...)$
$\wedge\ \Pi U_1{::}knd.\ \Pi U_2{::}fam \to fam.\ \Pi T{::}fam \to obj.\ ... \leftarrow (\Pi z{::}hyp{\wedge}knd{\wedge}fam{\wedge}obj.\ ...)$
$\wedge\ \Pi U_1{::}fam{\wedge}knd.\ \Pi U_2{::}fam \to knd.\ \Pi T{::}obj \to fam.\ ... \leftarrow (\Pi z{::}hyp{\wedge}knd{\wedge}fam{\wedge}obj.\ ...)$
$\wedge\ \Pi U_1{::}knd{\wedge}fam.\ \Pi U_2{::}obj \to knd.\ \Pi T{::}fam \to fam.\ ... \leftarrow (\Pi z{::}hyp{\wedge}knd{\wedge}fam{\wedge}obj.\ ...)$
$\wedge\ \Pi U_1{::}knd.\ \Pi U_2{::}fam \to knd.\ \Pi T{::}fam \to fam.\ ... \leftarrow (\Pi z{::}hyp{\wedge}knd{\wedge}fam{\wedge}obj.\ ...)$

Here, we see that something is definitely amiss. Recall the roles of the metavariables in this rule: we are checking $\lambda x{:}U_1.\,T$ at the classifier $\Pi x{:}U_1.\,U_2$. The first conjunct of the reconstruction seems correct enough: it corresponds to the rule for checking an object-level function $\lambda x{:}A_1.\,M$ at the type $\Pi x{:}A_1.\,A_2$. The second conjunct is expected, as well: it corresponds to the rule for checking a family-level λ-abstraction $\lambda x{:}A_1.\,B$ at the dependent kind $\Pi x{:}A_1.\,K_2$.

When we reach the third conjunct, things begin to break down. The sort assigned to T suggests an object-level function, just as in the first conjunct, but the sort assigned to U_2 suggests a polymorphic function type, and of course the only way to make this work is for U_1 to be both a family and a kind! If such a thing were to exist, perhaps written as Ak_1, we could say that this conjunct represents the rule for checking the object function $\lambda x{:}Ak_1.\,M$ at the polymorphic type $\forall\alpha{:}Ak_1.\,A_2$. This is clearly not something we intended, and the other conjuncts involving intersections in the sort of U_1 are equally meaningless.

The situation worsens even further when we run our sort reconstruction algorithm on the *of-ap* rule: reconstruction gives up because a constraint in tension would require enumeration. What can we do now? Can we recover our modular presentation without being forced to give the fully explicit four-conjunct rules we expect?

Fortunately, we can get by with giving two-conjunct rules that include some annotations. The role of the annotations is to rule out pathological intersected classifiers like we saw in the reconstruction of *of-lm*. They also have the effect of simplifying the strange sorts given to the bound variable x we saw above.

$$\begin{aligned}
&\textit{of-pi} :: \textit{of}'\ (\textit{pi}\ T_1\ \lambda x.T_2\ x)\ \textit{tp}\\
&\qquad \leftarrow \textit{of}'\ T_1\ \textit{tp}\\
&\qquad \leftarrow (\Pi x{::}obj.\ \textit{of}'\ x\ T_1 \to \textit{of}'\ (T_2\ x)\ \textit{tp}).\\
&\quad \wedge\ \textit{of}'\ (\textit{pi}\ T_1\ \lambda x.T_2\ x)\ \textit{tp}\\
&\qquad \leftarrow \textit{of}'\ T_1\ \textit{tp}\\
&\qquad \leftarrow (\Pi x{::}fam.\ \textit{of}'\ x\ T_1 \to \textit{of}'\ (T_2\ x)\ \textit{tp}).
\end{aligned}$$

$$\begin{aligned}
&\textit{of-lm} :: \textit{of}'\ (\textit{lm}\ (U_1 :: fam)\ \lambda x.T\ x)\ (\textit{pi}\ U_1\ \lambda x.U_2\ x)\\
&\qquad \leftarrow \textit{of}'\ U_1\ \textit{tp}\\
&\qquad \leftarrow (\Pi x{::}obj.\ \textit{of}'\ x\ U_1 \to \textit{of}'\ (T\ x)\ (U_2\ x)).\\
&\quad \wedge\ \textit{of}'\ (\textit{lm}\ (U_1 :: knd)\ \lambda x.T\ x)\ (\textit{pi}\ U_1\ \lambda x.U_2\ x)\\
&\qquad \leftarrow \textit{of}'\ U_1\ \textit{tp}
\end{aligned}$$

$\leftarrow (\Pi x{::}\mathit{fam}.\ \mathit{of}'\ x\ U_1 \rightarrow \mathit{of}'\ (T\ x)\ (U_2\ x)).$

$\mathit{of\text{-}ap} :: \mathit{of}'\ (\mathit{ap}\ T_1\ (T_2 :: \mathit{obj}))\ (U\ T_2)$
$\leftarrow \mathit{of}'\ T_1\ (\mathit{pi}\ U_2\ \lambda x.U\ x)$
$\leftarrow \mathit{of}'\ T_2\ U_2.$
$\wedge\ \mathit{of}'\ (\mathit{ap}\ T_1\ (T_2 :: \mathit{fam}))\ (U\ T_2)$
$\leftarrow \mathit{of}'\ T_1\ (\mathit{pi}\ U_2\ \lambda x.U\ x)$
$\leftarrow \mathit{of}'\ T_2\ U_2.$

These annotations—on the bound variable for *of-pi*, on the bound variable and the domain type for *of-lm*, and on the argument of the application for *of-app*, are enough to coax our sort reconstruction algorithm into producing the expected four-clause reconstructions that follow:

$\mathit{of\text{-}pi} :: \Pi T_1{::}\mathit{fam}.\ \Pi T_2{::}\mathit{obj} \rightarrow \mathit{fam}.$
$\mathit{of}'\ (\mathit{pi}\ T_1\ \lambda x.T_2\ x)\ \mathit{tp}$
$\leftarrow \mathit{of}'\ T_1\ \mathit{tp}$
$\leftarrow (\Pi x{::}\mathit{obj}.\ \mathit{of}'\ x\ T_1 \rightarrow \mathit{of}'\ (T_2\ x)\ \mathit{tp}).$
$\wedge\ \Pi T_1{::}\mathit{fam}.\ \Pi T_2{::}\mathit{obj} \rightarrow \mathit{knd}.$
$\mathit{of}'\ (\mathit{pi}\ T_1\ \lambda x.T_2\ x)\ \mathit{tp}$
$\leftarrow \mathit{of}'\ T_1\ \mathit{tp}$
$\leftarrow (\Pi x{::}\mathit{obj}.\ \mathit{of}'\ x\ T_1 \rightarrow \mathit{of}'\ (T_2\ x)\ \mathit{tp}).$
$\wedge\ \Pi T_1{::}\mathit{knd}.\ \Pi T_2{::}\mathit{fam} \rightarrow \mathit{fam}.$
$\mathit{of}'\ (\mathit{pi}\ T_1\ \lambda x.T_2\ x)\ \mathit{tp}$
$\leftarrow \mathit{of}'\ T_1\ \mathit{tp}$
$\leftarrow (\Pi x{::}\mathit{fam}.\ \mathit{of}'\ x\ T_1 \rightarrow \mathit{of}'\ (T_2\ x)\ \mathit{tp}).$
$\wedge\ \Pi T_1{::}\mathit{knd}.\ \Pi T_2{::}\mathit{fam} \rightarrow \mathit{knd}.$
$\mathit{of}'\ (\mathit{pi}\ T_1\ \lambda x.T_2\ x)\ \mathit{tp}$
$\leftarrow \mathit{of}'\ T_1\ \mathit{tp}$
$\leftarrow (\Pi x{::}\mathit{fam}.\ \mathit{of}'\ x\ T_1 \rightarrow \mathit{of}'\ (T_2\ x)\ \mathit{tp}).$

$\mathit{of\text{-}lm} :: \Pi U_1{::}\mathit{fam}.\ \Pi U_2{::}\mathit{obj} \rightarrow \mathit{fam}.\ \Pi T{::}\mathit{obj} \rightarrow \mathit{obj}.$
$\mathit{of}'\ (\mathit{lm}\ U_1\ \lambda x.T\ x)\ (\mathit{pi}\ U_1\ \lambda x.U_2\ x)$
$\leftarrow \mathit{of}'\ U_1\ \mathit{tp}$
$\leftarrow (\Pi x{::}\mathit{obj}.\ \mathit{of}'\ x\ U_1 \rightarrow \mathit{of}'\ (T\ x)\ (U_2\ x)).$
$\wedge\ \Pi U_1{::}\mathit{fam}.\ \Pi U_2{::}\mathit{obj} \rightarrow \mathit{knd}.\ \Pi T{::}\mathit{obj} \rightarrow \mathit{fam}.$
$\mathit{of}'\ (\mathit{lm}\ U_1\ \lambda x.T\ x)\ (\mathit{pi}\ U_1\ \lambda x.U_2\ x)$
$\leftarrow \mathit{of}'\ U_1\ \mathit{tp}$
$\leftarrow (\Pi x{::}\mathit{obj}.\ \mathit{of}'\ x\ U_1 \rightarrow \mathit{of}'\ (T\ x)\ (U_2\ x)).$
$\wedge\ \Pi U_1{::}\mathit{knd}.\ \Pi U_2{::}\mathit{fam} \rightarrow \mathit{fam}.\ \Pi T{::}\mathit{fam} \rightarrow \mathit{obj}.$
$\mathit{of}'\ (\mathit{lm}\ U_1\ \lambda x.T\ x)\ (\mathit{pi}\ U_1\ \lambda x.U_2\ x)$
$\leftarrow \mathit{of}'\ U_1\ \mathit{tp}$
$\leftarrow (\Pi x{::}\mathit{fam}.\ \mathit{of}'\ x\ U_1 \rightarrow \mathit{of}'\ (T\ x)\ (U_2\ x)).$
$\wedge\ \Pi U_1{::}\mathit{knd}.\ \Pi U_2{::}\mathit{fam} \rightarrow \mathit{knd}.\ \Pi T{::}\mathit{fam} \rightarrow \mathit{fam}.$

$$\begin{array}{l}
of'\ (lm\ U_1\ \lambda x.T\ x)\ (pi\ U_1\ \lambda x.U_2\ x) \\
\quad \leftarrow of'\ U_1\ tp \\
\quad \leftarrow (\Pi x{::}fam.\ of'\ x\ U_1 \rightarrow of'\ (T\ x)\ (U_2\ x)).
\end{array}$$

$$\begin{array}{l}
of\text{-}ap :: \Pi T_1{::}obj.\ \Pi T_2{::}obj.\ \Pi U_2{::}fam.\ \Pi U{::}obj \rightarrow fam. \\
\qquad of'\ (ap\ T_1\ T_2)\ (U\ T_2) \\
\qquad \leftarrow of'\ T_1\ (pi\ U_2\ \lambda x.U\ x) \\
\qquad \leftarrow of'\ T_2\ U_2. \\
\quad \wedge\ \Pi T_1{::}obj.\ \Pi T_2{::}fam.\ \Pi U_2{::}knd.\ \Pi U{::}fam \rightarrow fam. \\
\qquad of'\ (ap\ T_1\ T_2)\ (U\ T_2) \\
\qquad \leftarrow of'\ T_1\ (pi\ U_2\ \lambda x.U\ x) \\
\qquad \leftarrow of'\ T_2\ U_2. \\
\quad \wedge\ \Pi T_1{::}fam.\ \Pi T_2{::}obj.\ \Pi U_2{::}fam.\ \Pi U{::}obj \rightarrow knd. \\
\qquad of'\ (ap\ T_1\ T_2)\ (U\ T_2) \\
\qquad \leftarrow of'\ T_1\ (pi\ U_2\ \lambda x.U\ x) \\
\qquad \leftarrow of'\ T_2\ U_2. \\
\quad \wedge\ \Pi T_1{::}fam.\ \Pi T_2{::}fam.\ \Pi U_2{::}knd.\ \Pi U{::}fam \rightarrow knd. \\
\qquad of'\ (ap\ T_1\ T_2)\ (U\ T_2) \\
\qquad \leftarrow of'\ T_1\ (pi\ U_2\ \lambda x.U\ x) \\
\qquad \leftarrow of'\ T_2\ U_2.
\end{array}$$

An interesting question for future work arises: could we have managed with fewer annotations? For each declaration, we had to give two conjuncts that only varied in small ways. Perhaps we could do better by introducing a form of quantification analogous to the generalized λ-abstractions of Reynolds's Forsythe [Rey96] or an explicit alternation construct like the *for* construct of Pierce's $F_\wedge$ [Pie97].

Better still, perhaps there is some general modification we can make to the sort reconstruction process to allow us to encode certain invariants, like for instance the fact that we never intend for there to be any terms of sort $fam \wedge knd$. Armed with such extra information, perhaps in the form of somehing like a **%worlds** declaration, maybe sort reconstruction can be made to handle a broader class of representation needs that arise in common practice.

6.2 Proof Theory

We now turn our attention to a few case studies in the realm of proof theory and logic. Of course, many of the ideas apply equally well to programming languages, thanks to the propositions-as-types correspondence.

In this section, we will work relative to the following signature of implicational propositions.

$atom$: type.
o : type.

$? : atom \to o.$
$\supset : o \to o \to o.$
%**infix** *right* 10 $\supset$.

6.2.1 Cut-free Sequent Calculi

One of the earliest formal developments done in Twelf was Pfenning's structural proof of cut elimination for intuitionistic and classical first-order sequent calculi [Pfe94]. Pfenning's essential insight was to first prove the cut rule admissible in a cut-free variant of the sequent calculus, and then to show via a simple structural induction leveraging that admissibility that every proof in the calculus *with* cut can be converted to a proof in the calculus *without* cut.

We show here the LF encoding of the implicational fragment of the intuitionistic sequent calculus with cut.

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

$init : conc\ A$
$\quad \leftarrow hyp\ A.$

$\supset R : conc\ (A \supset B)$
$\quad \leftarrow (hyp\ A \to conc\ B).$

$\supset L : (hyp\ (A \supset B) \to conc\ C)$
$\quad \leftarrow conc\ A$
$\quad \leftarrow (hyp\ B \to conc\ C).$

$cut : conc\ C$
$\quad \leftarrow conc\ A$
$\quad \leftarrow (hyp\ A \to conc\ C).$

The type family *hyp* represents hypotheses on the left of a sequent, while the type family *conc* represents the distinguished conclusion of a sequent.

Next, Pfenning encoded the rules for a cut-free version of the same calculus, using a different name for the conclusion judgment and different names for all of the inference rules. Unsurprisingly, we can do better using refinements: instead of cut-free proofs being a different type family altogether, they become simply a refinement of proofs with cut.

$cutfree \sqsubset conc.$

$init :: cutfree\ A$
$\quad \leftarrow hyp\ A.$

$$\begin{array}{l} \supset\!R :: \mathit{cutfree}\ (A \supset B) \\ \quad \leftarrow (\mathit{hyp}\ A \rightarrow \mathit{cutfree}\ B). \end{array}$$

$$\begin{array}{l} \supset\!L :: (\mathit{hyp}\ (A \supset B) \rightarrow \mathit{cutfree}\ C) \\ \quad \leftarrow \mathit{cutfree}\ A \\ \quad \leftarrow (\mathit{hyp}\ B \rightarrow \mathit{cutfree}\ C). \end{array}$$

Cut-free proofs are fundamentally just a subset of ordinary cut-full proofs, and the LFR encoding captures the inclusion intrinsically without needing two separate types of proofs.

Although we still have to explicitly write out all of the rules for the cut-free version of the calculus, there are a number of advantages to the LFR representation. First, the refinement restriction guarantees that the cut-free calculus is indeed a fragment of the original: every rule still has to have essentially the same form it did previously. In this way, the refinement restriction prevents certain kinds of encoding mistakes where one accidentally makes a calculus that is *not* a sub-calculus of another, a mistake that would typically not be caught until much later in the proof of some interesting metatheorem. Additionally, the refinement approach saves the user from the burden of having to invent entirely new names for all of the constants representing the sequent calculus rules. Even if we used a module system to isolate the two type families in separate namespaces, we would still lose the fundamental connection between cut-free proofs and proofs with cut—namely that every cut-free proof can be regarded as a proof with zero uses of cut—unless we encoded it post-hoc as another metatheorem.[1]

6.2.2 Normal Natural Deductions

Just as we can represent the cut-free sequent calculus as a refinement, we can represent normal natural deductions as refinements of unconstrained ones. The problem is a little bit more interesting because the "bidirectional" nature of natural deduction proofs—introducing upwards, eliminating downwards—gives an intricate structure to their normal forms.

Recall the encoding of natural deductions and the refinements of neutral and normal ones we saw earlier, based on the rules in Figure 2.4:

$$\mathit{true} : o \rightarrow \mathsf{type}.$$

$$\begin{array}{l} \supset\!I\ : (\mathit{true}\ A \rightarrow \mathit{true}\ B) \rightarrow \mathit{true}\ (A \supset B). \\ \supset\!E : \mathit{true}\ (A \supset B) \rightarrow \mathit{true}\ A \rightarrow \mathit{true}\ B. \end{array}$$

$$\begin{array}{l} \mathit{normal} \sqsubset \mathit{true}. \\ \mathit{neutral} \sqsubset \mathit{true}. \end{array}$$

[1]The signature morphisms of Rabe and Schümann [RS09] can go a long way towards automating the correspondence, though.

$\mathit{neutral} \leq \mathit{normal}$.   % judgmental inclusion

$\supset\! I$ :: $(\mathit{neutral}\ A \rightarrow \mathit{normal}\ B) \rightarrow \mathit{normal}\ (A \supset B)$.
$\supset\! E$ :: $\mathit{neutral}\ (A \supset B) \rightarrow \mathit{normal}\ A \rightarrow \mathit{neutral}\ B$.

Since hypotheses are neutral and proofs are normal, there must be some way of mediating between the two. In Chapter 2, we represented a judgmental inclusion of neutral derivations into normal ones by way of a "phantom" rule of inference labelled $*$ whose use we supressed when reasoning about the adequacy of our encoding. In our encoding, the inclusion is represented by subsorting.

Another interesting refinement of natural deductions is *canonical* derivations, or *verifications*. They are structured precisely the same way as normal derivations modulo one extra constraint: the judgmental inclusion rule $*$ is only applicable at atomic propositions P:

$$\frac{P\ \textbf{neutral}}{P\ \textbf{normal}}\,*$$

This paradigm should of course sound familiar, since it is the basis of the canonical forms of LF that we have use throughout this dissertation. Can we represent it in LFR?

Unfortunately the answer is no. What we would like of our encoding is a sort representing atomic propositions, *atomic*, and a way to say that for any $P \Leftarrow \mathit{atomic}$, we should have $\mathit{neutral}\ P \leq \mathit{normal}\ P$. It is easy enough to represent atomic propositions as a refinement:

$\mathit{atomic} \sqsubset o$.
$? :: \top \rightarrow \mathit{atomic}$.

But we have no way of expressing our desired subsorting relationship. A plausible suggestion might be something like the following:

$\mathit{neutral} \leq \mathit{normal} :: \mathit{atomic} \rightarrow \mathsf{sort}$.

We have not thus far specified the behavior of such a declaration: our subsorting declarations are bare, $s_1 \leq s_2$. In the signature formation rules, we require that to declare $s_1 \leq s_2$, it must be the case that s_1 and s_2 refine the same type and have the same class. Perhaps there is some notion of compatibility between classes that would allow subsorting declarations to be at a particular class, $s_1 \leq s_2 :: L$, provided that it is compatible with the classes of the two subjects.

Relatedly, perhaps there is a way of relaxing the restriction on subsorting declarations that the two sorts have the same class. If we allowed $s_1 \leq s_2$ when $s_1 \sqsubset a :: L_1$ and $s_2 \sqsubset a :: L_2$, we would still want to check some notion of compatibility between L_1 and L_2. Using the relaxed semantics, it may be possible to encode canonical derivations without using classed subsorting declarations by doing something like the following:

$\mathit{neutral\text{-}atomic} \sqsubset \mathit{true} :: \mathit{atomic} \rightarrow \mathsf{sort}$.

$\mathit{neutral} \leq \mathit{neutral\text{-}atomic}$.
$\mathit{neutral\text{-}atomic} \leq \mathit{neutral}$.

neutral-atomic $\leq$ *normal*.

The idea here is that neutral derivations should be equivalent to neutral-atomic ones—i.e., one can be converted to the other and vice-versa—but neutral-atomic derivations have a more stringent requirement on their subject, namely that it be atomic. Then we permit only neutral-atomic derivations to be promoted to normal ones. Given a derivation of *neutral P* where $P \Leftarrow$ *atomic*, we should be able to promote it to *neutral-atomic P*, since that sort is also well-formed, and then to *normal P* as desired.

The subset interpretation of Chapter 4 would be an ideal vehicle for studying classed subsorting declarations since its subsorting judgment $\Gamma \vdash Q_1 \leq Q_2 \rightsquigarrow F(-,-)$ is required to sort-check the arguments of Q_1 and Q_2 anyway, for the purposes of the translation. A careful study of this aspect of the subset interpretation might lend some insight into how LFR might be extended along these lines.

6.2.3 Intuitionistic and Classical Proofs

It is well known that intuitionistic logic proves fewer theorems than classical logic, and conversely that classical logic makes fewer distinctions than intuitionistic. There are many interesting results to study relating the two, for instance double-negation translations and their relationship to CPS conversion. We can represent intuitionistic logic as a refinement of classical logic so that such investigations can take place with respect to a single language of proof terms. This case study is courtesy of Frank Pfenning and Ron Garcia.

Here we use the type o of propositions to represent the object-language types of an intrinsically typed encoding of a classical λ-calculus. We extend it with one more member, the false proposition:

$ff : o.$

In classical logic, there are thre kinds of terms: expressions having a type, continuations accepting a type, and contradictions.

$exp : o \rightarrow \mathsf{type}.$
$cont : o \rightarrow \mathsf{type}.$
$impossible : \mathsf{type}.$

Abstraction, application, and falsehood-elimination have their usual rules introducing expressions:

$lam : (exp\ A \rightarrow exp\ B) \rightarrow exp\ (A \supset B).$
$app : exp\ (A \supset B) \rightarrow exp\ A \rightarrow exp\ B.$
$abort : exp\ ff \rightarrow exp\ A.$

Continuations serve as the hypotheses introduced by a control operator, and they can be thrown an appropriately-typed expression to create an impossible state.

$ctrl : (cont\ A \rightarrow impossible) \rightarrow exp\ A.$
$throw : cont\ A \rightarrow exp\ A \rightarrow impossible.$

Now we can isolate as refinements an intuitionistic calculus *iexp* that never uses the control operator:

```
iexp ⊏ exp :: ⊤ → sort.

lam :: (iexp A → iexp B) → iexp (A ⊃ B).
app :: iexp (A ⊃ B) → iexp A → iexp B.
abort :: iexp ff → iexp A.
```

And a classical calculus *cexp* freely making use of all rules:

```
cexp ⊏ exp :: ⊤ → sort.
ccont ⊏ cont :: ⊤ → sort.
cimpossible ⊏ impossible.

lam :: (cexp A → cexp B) → cexp (A ⊃ B).
app :: cexp (A ⊃ B) → cexp A → cexp B.
abort :: cexp ff → cexp A.

throw :: ccont A → cexp A → cimpossible.
ctrl :: (ccont A → cimpossible) → cexp A.
```

Using this encoding, we can write an intrinsically correct CPS transformation that translates classical expressions to (parametrically double-negated) intuitionistic ones. We have not run this example in our prototype sort reconstructor, and it almost certainly would not reconstruct successfully due to its third-order nature, but we list it here anyway to illustrate the richness of properties that can be expressed through refinements. Perhaps it will inspire the reader to imagine other similarly rich encodings.

```
% ... e2t, f2t, c2t ...
e2t′ ⊏ e2t :: cexp A → (Πc. (Πc′. iexp A → iexp c′) → iexp c) → sort.
f2t′ ⊏ f2t :: cimpossible → (Πc. iexp c) → sort.
c2t′ ⊏ c2t :: ccont A → (Πc. iexp A → iexp c) → sort.

e2t-lam :: e2t′ (lam λx.E x)
           (λc.λk.k c (lam λx′.M x′ B (λc′.λz.k c′ (lam λq.z))))
   ← (Πx:cexp A. Πx′:iexp A.
        e2t′ x (λc.λk.k c x′)
      → e2t′ (E x) (λc.λk.M x′ c λa.λe.k a e)).

e2t-app :: e2t′ (app E1 E2)
           (λc.λk.M_1 c (λc1.λf.M2 c1 (λc2.λz.k c2 (app f z))))
   ← e2t′ E2 (λc.λk.M_2 c λa.λe.k a e)
   ← e2t′ E1 (λc.λk.M_1 c λa.λe.k a e).

e2t-abort :: e2t′ (abort E)
             (λc.λk.M c (λc′.λx.abort x))
```

$$\boxed{\Gamma \Longrightarrow C}$$

$$\frac{}{\Gamma, A \Longrightarrow A}\ (\mathbf{ax}) \qquad \frac{\Gamma, A \Longrightarrow B}{\Gamma \Longrightarrow A \supset B}\ (\supset\textbf{-C}) \qquad \frac{\Gamma \Longrightarrow A \supset B \qquad \Gamma \Longrightarrow A \qquad \Gamma, B \Longrightarrow C}{\Gamma \Longrightarrow C}\ (\supset\textbf{-D})$$

Figure 6.3: Natural deduction with general eliminations

$\quad \leftarrow e2t'\ E\ (\lambda c.\lambda k.M\ c\ \lambda a.\lambda e.k\ a\ e).$

$e2t\text{-}ctrl :: e2t'\ (ctrl\ \lambda u.F\ u)\ (\lambda c.\lambda k.M\ c\ (\lambda c' \lambda x.k\ c'\ x))$
$\quad \leftarrow (\Pi u{:}ccont\ A.\ \Pi k{:}\Pi c.\ iexp\ A \to iexp\ c.$
$\qquad c2t'\ u\ (\lambda a.\lambda e.k\ a\ e)$
$\quad\ \ \to f2t'\ (F\ u)\ (\lambda c.M\ c\ \lambda a.\lambda e.k\ a\ e)).$

$f2t\text{-}throw :: f2t'\ (throw\ U\ E)\ (\lambda c.M\ c\ (\lambda c'.K\ c'))$
$\quad \leftarrow c2t'\ U\ K$
$\quad \leftarrow e2t'\ E\ (\lambda c.\lambda k.M\ c\ k).$

6.2.4 General Eliminations and the Uniform Calculus

Rules for natural deduction can be given in a style where the eliminations are always toward a general conclusion like the usual rule for eliminating disjunctions, a style sometimes referred to as "closed-scope eliminations". Figure 6.3 gives one such formulation, which is similar to von Plato's system of natural deduction with general eliminations [vP01] and Negri and von Plato's "uniform calculus" [NvP01]. An interesting property of this inference system is that it contains both the usual natural deduction system and the sequent calculus as subsystems, i.e., subsets of derivations with a particular form. In this section, we will show how to encode those subsystems as refinements.

First, we give an LF encoding of natural deduction with general eliminations. In both Figure 6.3 and its LF encoding, we give the rules names which suggest neither natural deduction nor sequent calculus. The rule **ax** uses an assumption, while the rules ⊃**-C** and ⊃**-D** construct and destruct implications, respectively.

$left : o \to \mathsf{type}.$
$right : o \to \mathsf{type}.$

$ax : left\ A \to right\ A.$

% implication constructor
$\supset C : (left\ A \to right\ B) \to right\ (A \supset B).$
% implication destructor, with general conclusion

$\supset\!D : right\ (A \supset B) \rightarrow right\ A \rightarrow (left\ B \rightarrow right\ C) \rightarrow right\ C.$

The encoding shares some aspects of the encoding of natural deduction and some aspects of the encoding of the sequent calculus. Like the sequent calculus, there are two different judgments, one for assumptions to the left of the sequent arrow and one for the conclusion to the right of the sequent arrow. But like natural deduction, the principal formula of the destructor appears on the right in a premise rather than on the left.

Suppose we had a derivation using the generalized rules such that every use of the destructor **⊃-D** derived its first premise by the assumption rule **ax**. It is not difficult to see that such a derivation would correspond precisely to a derivation in the sequent calculus, where each use of the destructor **⊃-D** corresponded to a use of the sequent calculus left rule **⊃-L**. In other words, any derivation using the general rules such that the first premise of each use of the destructor is *initial* corresponds to a sequent calculus proof.

We can encode this restriction on generalized natural deductions easily using refinements.

$seqhyp \sqsubset left.$
$seq \sqsubset right.$

$initial \sqsubset right.$
$initial \leq seq.$

$ax :: seqleft\ A \rightarrow initial\ A.$

% right rule: same as the constructor
$\supset\!C :: (seqhyp\ A \rightarrow seq\ B) \rightarrow seq\ (A \supset B).$
% left rule: restrict proof of $A \supset B$ to be initial
$\supset\!D :: initial\ (A \supset B) \rightarrow seq\ A \rightarrow (seqhyp\ B \rightarrow seq\ C) \rightarrow seq\ C.$

We first define refinements of *left* and *right* called *seqhyp* and *seq*. We then define a special refinement of *right* called *initial* such that every *initial* derivation is also a *seq* derivation, but the only way to conjure up an *initial* derivation is using the *ax* rule. Finally, we require that the first argument to the $\supset\!D$ rule be an initial sequent.

The only difference between standard natural deduction and the generalized rules is the general form of the destructor. Suppose we had a derivation in which each use of the destructor **⊃-D** was an immediate use of the just-introduced assumption, such that the conclusion was the same as the consequent of the implication being destructed. Such a derivation would correspond directly to a standard “open-scope” natural deduction, with uses of the destructor **⊃-D** corresponding to uses of the elimination rule **⊃-E**.

Again, we can encode this restriction using sorts quite easily.

$ndhyp \sqsubset left.$
$nd \sqsubset right.$

$immedhyp \sqsubset left.$

```
immed ⊏ right.

ax :: ndhyp A → nd A
   ∧ immedhyp A → immed A.

% intro rule: same as the constructor
⊃C :: (ndhyp A → nd B) → nd (A ⊃ B).
% elim rule: restrict proof of B ⊢ C to be immediate use of B
⊃D :: nd (A ⊃ B) → nd A → (immedhyp B → immed C) → nd C.
```

As before, we first introduce natural deduction refinements of *left* and *right*, called *ndhyp* and *nd*. Next, we introduce two further refinements of *left* and *right* for immediate hypotheses and immediate uses of them. The assumption rule *ax* has two forms, represented by an intersection: one is the standard natural deduction hypothesis rule that lets you use an assumption of A as a derivation of A; the other converts an immediate hypothesis of A to an immediate proof of A. It is this latter form which captures the appropriate restriction on the destructor: the "continuation" of the derivation must be an immediate use of the just-introduced assumption. Since *immedhyp* hypotheses are not introduced anywhere else in a derivation, and since the *ax* rule is the only way conjure up a proof of *immed A*, we know that the third argument to $\supset D$ must in fact be $\lambda h. ax\ h$.

6.3 Omphaloskepsis

We now examine some case studies close to home. This section concerns examples from the study of logical frameworks, and indeed mostly examples that come from this very dissertation. It is our hope that through these examples, the reader will come to understand the LFR philosophy and begin to see refinements everywhere they look.

In what follows, we will assume the following partial signature representing an encoding of LFR itself.

```
tm : type.
% ... terms ...

ast : type.
% ... atomic sorts ...

st : type.

? : ast → st.
pi : st → (tm → st) → st.
top : st.
inter : st → st → st.
```

6.3.1 Normalized Sorts

Recall the "normalized sorts" from Section 5.4.1, in which intersections are distributed as far outwards as possible, so that they only occur to the left of arrows.

$\mathbb{S}, \mathbb{T} ::= Q \mid \Pi x{::}\mathbf{S}.\, \mathbb{T}$	basic sorts
$\mathbf{S}, \mathbf{T} ::= \mathbb{S} \mid \mathbf{S}_1 \wedge \mathbf{S}_2 \mid \top$	composite sorts

This sort of example is the bread and butter of refinement types:

$basic \sqsubset st.$ $basic$ :: sort.
$composite \sqsubset st.$ $composite$ :: sort.

$? :: \top \to basic.$
$pi :: composite \to (\top \to basic) \to basic.$

$basic \leq composite.$
$top :: composite.$
$inter :: composite \to composite \to composite.$

Armed with this formalization, we can even go on to formalize the normalization procedure, giving it a class that makes it manifestly correct. We omit the LF declarations for the normalization, giving just the LFR ones.

$norm : tp \to tp \to$ type.
$normpi : tp \to (tm \to tp) \to tp \to$ type.
% ...

$norm' \sqsubset norm :: \top \to composite \to$ sort.
$normpi' \sqsubset normpi :: composite \to (tm \to composite) \to composite \to$ sort.

$norm/? :: norm'\ (?\ Q)\ (?\ Q).$
$norm/top :: norm'\ top\ top.$
$norm/inter :: norm'\ (inter\ S_1\ S_2)\ (inter\ S_1'\ S_2')$
$\quad\quad \leftarrow norm'\ S_1\ S_1'$
$\quad\quad \leftarrow norm'\ S_2\ S_2'.$
$norm/pi :: norm'\ (pi\ S\ (\lambda x.T\ x))\ Pi$
$\quad\quad \leftarrow norm'\ S\ S'$
$\quad\quad \leftarrow (\Pi x.\ norm'\ (T\ x)\ (T'\ x))$
$\quad\quad \leftarrow normpi'\ S'\ (\lambda x.T'\ x)\ Pi.$

$normpi/basic :: normpi'\ S\ (\lambda x.T\ x)\ (pi\ S\ (\lambda x.T\ x)).$
$normpi/top :: normpi'\ S\ (\lambda x.top)\ top.$
$normpi/inter :: normpi'\ S\ (\lambda x.inter\ (T_1\ x)\ (T_2\ x))\ (inter\ T_1'\ T_2')$
$\quad\quad \leftarrow normpi'\ S\ (\lambda x.T_1\ x)\ T_1'$
$\quad\quad \leftarrow normpi'\ S\ (\lambda x.T_2\ x)\ T_2'.$

The most interesting declaration is the one for the constant *normpi/basic*: it should correspond to the basic clause from the definition of pi(x::$\mathbf{S}$. $\mathbb{T}$):

$$\text{pi}(x{::}\mathbf{S}.\,\mathbb{T}) = \Pi x{::}\mathbf{S}.\,\mathbb{T}$$

Yet it seems like it would match *any* input, basic or not. How does the clause "know" that the body $T\,x$ is basic? The answer is that it is constrained to be so by the type of the result: if we can form $pi\ S\ (\lambda x.T\ x)$, then it must be because $T\,x$ is basic. Indeed, our prototype implementation returns the following reconstruction for the clause:

$$normpi/basic :: \Pi S{::}composite.\ \Pi T{::}tm \to basic.\ normpi'\ S\ (\lambda x.T\ x)\ (pi\ S\ (\lambda x.T\ x)).$$

In a sense, things seem backwards here. On paper when we wrote $\mathbb{T}$ it was meant as a check on the input that we needed to perform in order to produce well-formed output. But in the LFR code, we just write the equivalent of T, an unconstrained sort, and the framework figures out that we *must* have meant $\mathbb{T}$ *because* of our declaration that the result had to be well-formed. The framework is very trusting, isn't it.

The declaration *normpi/basic* is interesting as well from a logic programming perspective: it is a rule that cannot be run to produce correct output unless it has been sort-reconstructed. The original LF declaration which we elided would not have had the correct operational behavior: only the LFR code will. We have not yet undertaken to study the logic programming interpretation of LFR, but it would make for interesting future work, and it will be important to bear in mind examples such as this one.

6.3.2 Prenex DNF Constraints

Recall the first-order logic of constraints we used to capture the problem of sort reconstruction. It is easily encoded in LF:

$constraint$: type.

$sub : st \to st \to constraint.$
$tt : constraint.$
$and : constraint \to constraint \to constraint.$
$ff : constraint.$
$or : constraint \to constraint \to constraint.$
$imp : constraint \to constraint \to constraint.$
$forall : (st \to constraint) \to constraint.$
$exists : (st \to constraint) \to constraint.$

When we described our procedure for solving these constraints in Section 5.5.2, we made it clear that it only solved a fragment of these constraints, namely the fragment of prenex existential constraints whose propositional bodies were in disjunctive normal form. Not surprisingly, we can encode this fragment succinctly using sorts:

$base \sqsubset st.$
$? :: \top \to base.$

$var \sqsubset st.$

$vararrow \sqsubset st.$
$vararrow :: var \to var \to vararrow.$

$atomic \sqsubset constraint.$

$$\begin{aligned} sub :: {} & var \to base \to atomic \\ \wedge {} & base \to var \to atomic \\ \wedge {} & var \to var \to atomic \\ \wedge {} & var \to vararrow \to atomic. \end{aligned}$$

$clause \sqsubset constraint.$ $clause :: \mathsf{sort}.$

$tt :: clause.$
$and :: atomic \to clause \to clause.$

$dnf \sqsubset constraint.$ $dnf :: \mathsf{sort}.$

$ff :: dnf.$
$or :: clause \to dnf \to dnf.$

$eprenex \sqsubset constraint.$ $eprenex :: \mathsf{sort}.$

$dnf \leq eprenex.$
$exists :: (var \to eprenex) \to eprenex.$

The most interesting aspect of this encoding is its treatment of variables. In order to capture the invariants on the forms of atomic subsorting constraints, we must be able to recognize variables. But the LF encoding of constraints used higher-order abstract syntax to represent variables, so it would seem that we have no way of recognizing them. We can, though, using refinements.

First, we posit a new refinement of *st* called *var* with no constructors and no constants. Then, when we give a refinement for the existential constraint, we make it bind a variable of this new sort. Since the existential quantifier is the only way variables are introduced into a prenex constraint, we now know that every variable in a well-sorted inhabitant of *eprenex* will have sort *var*, and in this way will be recognizable to the *sub* constructor. We get the benefits of an intrinsic encoding of variablehood without weakening our use of higher-order abstract syntax through the introduction of a separate variable type.

$\boxed{Q_1 \leq Q_2}$

$$\frac{s_1{\leq}s_2 \in \Sigma}{s_1 \leq s_2} \qquad \frac{Q_1 \leq Q_2}{Q_1\ N \leq Q_2\ N} \qquad \frac{}{Q \leq Q} \qquad \frac{Q_1 \leq Q' \qquad Q' \leq Q_2}{Q_1 \leq Q_2}$$

Figure 6.4: "Declarative" formulation of base subsorting

6.3.3 Subsorting Derivations

This case study concerns the representation of various forms of subsorting declarations which arose at various stages during our work on LFR itself. Recall from Chapter 2 that originally we gave a relatively "declarative" formulation of subsorting, shown in Figure 6.4. This way of formulating the rules of subsorting is simple and permissive: reflexivity and transitivity may be used at any point in a derivation. It can be faithfully represented in LF by a signature such as the following:

```
stconst : type.

c : stconst → ast.
@ : ast → tm → ast.
%infix left 10 @.

leqconst : stconst → stconst → type.
leq : ast → ast → type.

leq/c : leq (c S1) (c S2)
      ← leqconst S1 S2.

leq/@ : leq (Q1 @ N) (Q2 @ N)
      ← leq Q1 Q2.

leq/refl : leq Q Q.

leq/trans : leq Q1 Q2
         ← leq Q1  Q'
         ← leq Q'  Q2.
```

The type *ast* represents our atomic sorts Q, with the constructor c lifting a sort constant to an atomic sort and the infix constructor @ applying an atomic sort family to an index argument. Subsorting declarations $s_1{\leq}s_2$ are represented by the type family *leqconst* and the relation $Q_1 \leq Q_2$ is represented by the type family *leq*. The rules are essentially a direct transcription of the ones given in Figure 6.4.

$$\boxed{Q_1 \leq_{\mathrm{E}} Q_2}$$

$$\frac{s_1 \leq^{\mathrm{clo}}_{\mathrm{E}} s_2}{s_1 \leq_{\mathrm{E}} s_2} \qquad \frac{Q_1 \leq_{\mathrm{E}} Q_2}{Q_1\; N \leq_{\mathrm{E}} Q_2\; N}$$

$$\boxed{s_1 \leq^{\mathrm{clo}}_{\mathrm{E}} s_2}$$

$$\frac{s_1 {\leq} s_2 \in \Sigma}{s_1 \leq^{\mathrm{clo}}_{\mathrm{E}} s_2} \qquad \frac{}{s \leq^{\mathrm{clo}}_{\mathrm{E}} s} \qquad \frac{s_1 \leq^{\mathrm{clo}}_{\mathrm{E}} s' \qquad s' \leq^{\mathrm{clo}}_{\mathrm{E}} s_2}{s_1 \leq^{\mathrm{clo}}_{\mathrm{E}} s_2}$$

Figure 6.5: “Early” formulation of base subsorting

Such a formulation of subsorting is fine for metatheory, but it says little about how to implement the subsorting relation. In our implementation, we take a cue from the decidability proof: the relation $Q_1 \leq Q_2$ can be implemented by computing the reflexive, transitive closure of the relation $s_1 {\leq} s_2$ on sort constants, verifying that the heads of Q_1 and Q_2 are related by that closure, and checking that all of the index arguments of Q_1 and Q_2 are equal. In order to prove that this algorithm is complete, we can write down alternate rules for the subsorting relation that capture the extra imposed structure and then prove these alternate rules equivalent to the original ones above. We give such rules in Figure 6.5, writing the new relation $Q_1 \leq_{\mathrm{E}} Q_2$. The “E” stands for “early”: this relation requires reflexivity and transitivity to be resolved early, before any arguments are accumulated. It makes use of an auxiliary relation $s_1 \leq^{\mathrm{clo}}_{\mathrm{E}} s_2$ which represents the reflexive, transitive closure of the declared relation on constants.

A moment’s thought allows one to see that this new, “early” formulation of subsorting is really nothing but a restricted form of the original formulation. If we read the rule that lets us pass from $s_1 \leq^{\mathrm{clo}}_{\mathrm{E}} s_2$ to $s_1 \leq_{\mathrm{E}} s_2$ as a judgmental inclusion, then any derivation in “early” form can be trivially erased to one in the original relation. Using refinements, we can make this observation formal by regarding $\leq_{\mathrm{E}}$ and $\leq^{\mathrm{clo}}_{\mathrm{E}}$ to be refinements of the original relation.

% ”early” subsorting: reflexivity and transitivity only at the leaves.
$\mathit{leq}_{\mathrm{E}} \sqsubset \mathit{leq} :: \top \to \top \to \mathsf{sort}.$
$\mathit{leq}^{\mathrm{clo}}_{\mathrm{E}} \sqsubset \mathit{leq} :: \top \to \top \to \mathsf{sort}.$

$\mathit{leq}^{\mathrm{clo}}_{\mathrm{E}} \leq \mathit{leq}_{\mathrm{E}}.$

$\mathit{s/@} :: \mathit{leq}_{\mathrm{E}}\; (Q_1 \mathbin{@} N)\; (Q_2 \mathbin{@} N)$
$\quad \leftarrow \mathit{leq}_{\mathrm{E}}\;\; Q_1\; Q_2.$

% once we pass from $\mathrm{leq}^{\mathrm{clo}}_{\mathrm{E}}$ to $\mathrm{leq}_{\mathrm{E}}$, there’s no going back −−
% reflexivity and transitivity are only allowed at the auxiliary relation.

$$\boxed{Q_1 \leq_{\mathrm{L}} Q_2}$$

$$\frac{}{Q \leq_{\mathrm{L}} Q} \qquad \frac{Q_1 \leq^{\mathrm{syn}}_{\mathrm{L}} Q' \qquad Q' \leq_{\mathrm{L}} Q_2}{Q_1 \leq_{\mathrm{L}} Q_2}$$

$$\boxed{s_1 \leq^{\mathrm{syn}}_{\mathrm{L}} s_2}$$

$$\frac{s_1 {\leq} s_2 \in \Sigma}{s_1 \leq^{\mathrm{syn}}_{\mathrm{L}} s_2} \qquad \frac{Q_1 \leq^{\mathrm{syn}}_{\mathrm{L}} Q_2}{Q_1\, N \leq^{\mathrm{syn}}_{\mathrm{L}} Q_2\, N}$$

Figure 6.6: "Late" formulation of base subsorting

$s/c :: leq^{\mathrm{clo}}_{\mathrm{E}}\ (c\ S_1)\ (c\ S_2)$
    $\leftarrow \top$. % i.e., leqconst S_1 S_2.

$s/refl :: leq^{\mathrm{clo}}_{\mathrm{E}}\ S\ S$.

$s/trans :: leq^{\mathrm{clo}}_{\mathrm{E}}\ S_1\ S_2$
      $\leftarrow leq^{\mathrm{clo}}_{\mathrm{E}}\ S_1\ S'$
      $\leftarrow leq^{\mathrm{clo}}_{\mathrm{E}}\ S'\ S_2$.

As we have seen before, the judgmental inclusion of $s_1 \leq^{\mathrm{clo}}_{\mathrm{E}} s_2$ into the $\leq_{\mathrm{E}}$ relation is handled by a subsorting declaration. In the declaration for *s/c*, we write the type $\top$ to mean *leqconst* S_1 S_2, i.e., the maximal refinement of that type.

In Chapter 4 when we explored the subset interpretation of LFR into LF with proof irrelevance, we found it necessary to recast subsorting yet again. For the purposes of the subset interpretation, it was necessary to split subsorting into a "checking" judgment that only made sense at fully-applied atomic sorts and a "synthesis" judgment that, given a sort, synthesized a sort one step higher in the sort hierarchy. Instead of a transitivity rule, we had a "climb" rule which said that a sort Q_1 was below a sort Q_2 if we could synthesize a sort Q' one step higher than Q_1 and check that Q' was below Q_2, with the whole process ending at an application of reflexivity. Rules defining this relation $Q_1 \leq_{\mathrm{L}} Q_2$ and the auxiliary synthesis relation $Q_1 \leq^{\mathrm{syn}}_{\mathrm{L}} Q_2$ are shown in Figure 6.6. The "L" now stands for "late", indicating that reflexivity and transitivity can only be applied after we have *finished* accumulating arguments.

Once again, after a moment's thought, we realize that this new "late" formulation is in fact just a *different* restricted form of subsorting derivations: appeals to reflexivity and transitivity can only occur near the root of the derivation, and repeated appeals to transitivity must be right-nested. And as before, this restricted form is easily captured using refinement types as a pair of sort families refining the original judgment.

```
% "late" subtyping: reflexivity and transitivity only towards the root,
% right−nested. like the refl and climb rules of the subset interpretation.
```
$leq_{\mathrm{L}} \sqsubset leq :: \top \to \top \to \mathsf{sort}.$
$leq_{\mathrm{L}}^{\mathrm{syn}} \sqsubset leq :: \top \to \top \to \mathsf{sort}.$

$s/refl :: leq_{\mathrm{L}}\ Q\ Q.$

$s/trans :: leq_{\mathrm{L}}\ Q_1\ Q_2$
$\quad \leftarrow leq_{\mathrm{L}}^{\mathrm{syn}}\ Q_1\ Q'$
$\quad \leftarrow leq_{\mathrm{L}}\ Q'\ Q_2.$

$s/c :: leq_{\mathrm{L}}^{\mathrm{syn}}\ (c\ S_1)\ (c\ S_2)$
$\quad \leftarrow \top.$ % i.e., leqconst $S_1\ S_2$.

$s/@ :: leq_{\mathrm{L}}^{\mathrm{syn}}\ (Q_1\ @\ N)\ (Q_2\ @\ N)$
$\quad \leftarrow leq_{\mathrm{L}}^{\mathrm{syn}}\ Q_1\ Q_2.$

If we were to extend LFR to handle Twelf-like metatheoretic reasoning about encodings, we could prove four theorems establishing the equivalence of the three representations: two theorems showing that the new formulations are sound, i.e., every "early" or "late" derivation corresponds to an ordinary one, and two showing that they are complete, i.e., every ordinary derivation can be put into either "early" or "late" form. The soundness theorems would be completely trivial, since they say nothing beyond the refinement restriction: every "early" or "late" form derivation *must* be an ordinary derivation by invariant.

This case study further exemplifies the power of refinement types in specifying various "normal forms" of derivations of judgments. In the next section, we will see an even more dramatic example of what can be achieved using refinements.

6.3.4 Coloring Hypotheses

Finally, we consider a case study not directly related to LFR, but related to metatheorem checking in logical frameworks in general. Consider the signature of the untyped λ-calculus along with a "copying" predicate that recursively copies a term.

$exp : \mathsf{type}.$

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

$cp : exp \to exp \to \mathsf{type}.$

$cp/app : cp\ (app\ E1\ E2)\ (app\ F1\ F2)$
$\quad \leftarrow cp\ E1\ F1$
$\quad \leftarrow cp\ E2\ F2.$

$\mathit{cp/lam} : \mathit{cp}\ (\mathit{lam}\ \lambda x.E\ x)\ (\mathit{lam}\ \lambda y.F\ y)$
$\quad\quad \leftarrow (\Pi x.\ \Pi y.\ \mathit{cp}\ x\ y \rightarrow \mathit{cp}\ (E\ x)\ (F\ y)).$

One can also think of the predicate *cp* as defining a kind of logical-relations-style equivalence on λ-terms. The key idea is that when copying a λ-abstraction, we have to make a copy of its bound variable to use while copying the body.

Intuitively, this predicate is total with mode $(+, -)$: every λ-expression can be copied. Twelf fails to recognize its totality, though, because of an apparent coverage error: the variable y in the higher-order subgoal is not covered by any clause.

We can correct the problem through a procedure called "coloring". If we color all expressions either red or green, then we can specify that the copying predicate takes a red expression as input and yields a green one. If we color the hypotheses correctly, coverage checking ought to succeed. Unfortunately, it is inconvenient to do coloring with the standard tools: one must duplicate the entire syntax of expressions and consequently all judgments and metatheoretic results.

Enter refinements, the usual way of dealing with such problems. We can represent red and green not as separate types, but as refinements of the type of expressions.

$\mathit{red} :: \mathit{exp}.$
$\mathit{green} :: \mathit{exp}.$

$\mathit{app} :: \mathit{red} \rightarrow \mathit{red} \rightarrow \mathit{red}$
$\quad \wedge\ \mathit{green} \rightarrow \mathit{green} \rightarrow \mathit{green}.$

$\mathit{lam} :: (\mathit{red} \rightarrow \mathit{red}) \rightarrow \mathit{red}$
$\quad \wedge\ (\mathit{green} \rightarrow \mathit{green}) \rightarrow \mathit{green}.$

An interesting property of this representation is that red and green are isomorphic, if we restrict our attention to ground terms. This is an example where refinements only start to make interesting distinctions when we work in an open context and start creating new hypotheses.

$\mathit{ccp} \sqsubset \mathit{cp}.\ \ \mathit{ccp} :: \mathit{red} \rightarrow \mathit{green} \rightarrow \mathsf{sort}.$

$\mathit{cp/app} :: \mathit{ccp}\ (\mathit{app}\ E1\ E2)\ (\mathit{app}\ F1\ F2)$
$\quad\quad \leftarrow \mathit{ccp}\ E1\ F1$
$\quad\quad \leftarrow \mathit{ccp}\ E2\ F2.$

$\mathit{cp/lam} :: \mathit{ccp}\ (\mathit{lam}\ \lambda x.E\ x)\ (\mathit{lam}\ \lambda y.F\ y)$
$\quad\quad \leftarrow (\Pi x{::}\mathit{red}.\ \Pi y{::}\mathit{green}.\ \mathit{ccp}\ x\ y \rightarrow \mathit{ccp}\ (E\ x)\ (F\ y)).$

Note the explicit annotations coloring x red and y green—now we have introduced an example of a red sort that is not also green.

Although we have not studied the problem of adding metatheoretic tools to LFR as they exist in Twelf, we conjecture that the above clause would totality check in such an extension. Thus we have a way of distinguishing hypothetical parameters even if we don't distinguish anything about ground terms, illustrating a different way of using refinements.

6.4 Summary

In this chapter, we showed how refinements can be used to represent interesting properties of several real-world deductive systems drawn from a variety of fields. It is our hope that the case studies we've shown speak to the wide applicability of these techniques, supporting our assertion that refinement types are a useful addition to the logical framework LF.

Chapter 7

Future Work and Conclusions

This dissertation began with the following thesis:

Refinement types are a useful and practical extension to the LF logical framework.

In the chapters that followed, we defended this thesis in a variety of ways.

First, in Chapter 2 we exhibited a system of refinement types extending the logical framework LF. The inherent complexity of dependent types, subsorting, and intersections was tamed by following the modern LF methodology of working only with canonical forms. The result was a simple, bidirectional system of sort and class checking in which subsorting was only defined at base sorts. The restrictions imposed by the refinement methodology were instrumental in developing a clear picture of the representation methodology of *properties as sorts*, and we presented several examples of this representation methodology in action.

From the very beginning, we saw a bird's eye view of a refinement type system for LF that was both *practical*, by being based on modern simplifying technology, and *useful*, by solving several representational challenges not otherwise easily captured with existing languages. The remainder of this work further supported the claims of practicality and utility through a combination of theory and practice.

In Chapter 3, we established the fundamental metatheory required of any extension of LF: the decidability of sort checking and the transitivity and reflexivity of LFR's entailment. In addition, we explored some of the implications of the novel combination of refinements and canonical forms, including the fact that more traditional accounts of subtyping at higher types are intrinsically accounted for in a sound and complete manner by our canonical presentation. Together, these metatheoretic arguments establish the sensibility of our extension and justify our referring to it as a logical framework with subtyping.

We then further explored the meaning of LFR by presenting in Chapter 4 an interpretation of it into LF augmented with a notion of proof irrelevance. The soundness and completeness of our interpretation established that LF with refinement types is no more expressive or complex than LF with proof irrelevance, an extension that has been studied on its own for many years. But the inherent complexity of the uniform translation itself bolstered our claims of expressivity: although everything that can be done with refinement types can be done with proof irrelevance, it is only with great cost. The relative simplicity of refinement types as an independent notion demonstrates a high expressivity-to-cost ratio.

Next, in Chapter 5, we outlined some partial algorithms for sort reconstruction, an essential element of any practical implementation. By freeing the LFR user from writing down a great deal of redundant information regarding sorts, we end up with a logical framework which is roughly as simple to use in practice as the Twelf implementation of LF. Although the reconstruction problem is in principle decidable, we argued for the importance of partial algorithms that run quickly in practice, and we argued that our algorithms do run quickly on examples that typify the kinds of uses we expect.

Finally, we further explored the kinds of uses we expect by outlining in Chapter 6 a number of realistic case studies, drawing inspiration from representational challenges that arise in day-to-day research in programming languages and type theory. The expressive power offered by refinements allowed for intrinsic representations of several typical notions of subsets of derivations, and at a level comparable to the usual informal "on paper" representations of such things. While not the end of the story by any means, the representative sampling of case studies establishes the usefulness of refinement types for formalization carried out in a logical framework like LF.

7.1 Future Work

Although we've come a long way, as always, the story doesn't end here. There are several further directions that would be interesting to pursue.

One direction would be to loosen the conditions required for a subsorting declaration to be accepted: as currently formulated, in order to be able to declare that $s_1 {\leq} s_2$, the sort families s_1 and s_2 must have the exact same class. Instead, we could allow them to have any two compatible classes, for some appropriate notion of compatible. Or perhaps more interestingly, we could allow subsorting declarations themselves to be at a particular class, as in $s_1 {\leq} s_2 {::} L$, provided that L is appropriately compatible with the classes of s_1 and s_2. The meaning of such a declaration would be that the sort families are in the subsort relation only if the arguments are well-formed according to the class L. For instance, if we were to declare $s_1 {\leq} s_2 :: S \to \mathsf{sort}$, then $s_1\ N$ would be a subsort of $s_2\ N$ in a context Γ only if $\Gamma \vdash N \Leftarrow S$.

Such extended subsorting declarations would be useful for encoding a variety of deductive systems where a judgmental inclusion is restricted to judgments of a particular form. For example, to encode canonical natural deductions, one must be able to declare an inclusion between atomic derivations and canonical derivations, but one that holds only as long as the derivations are of an atomic proposition. Similarly, one could use classed subsorting declarations to encode the sequent calculus with the "init" rule only at atomic propositions as a refinement of the unrestricted version.

Another direction would be to delineate more precisely the fragment of LFR signatures we can efficiently reconstruct. Ideally, it would be nice to have some simple, syntactic characterization of efficiently-reconstructible signatures analogous to the "pattern fragment" for languages like Twelf and λProlog. Also along these lines would be an exploration of ways a user could annotate a non-reconstructable signature to produce one that is efficiently reconstructable, or ways the language might be extended with first-class constraints in order to capture certain patterns more completely.

Finally, it would be interesting to see how refinements interact with other extensions to the type theory of LF, like the linear logical framework LLF or the concurrent logical framework CLF. Orthogonally, it would also be nice to explore the logic programming interpretation of LFR, and in doing so try to incorporate some of the metatheoretic aspects of the Twelf system so that we could produce machine-checkable proofs of metatheorems about representations like the ones we've seen.

Appendix A

Complete LFR Rules

In the judgment forms below, superscript + and − indicate a judgment's "inputs" and "outputs", respectively.

A.1 Grammar

Kind level

$K ::= \mathsf{type} \mid \Pi x{:}A.\, K$	kinds
$L ::= \mathsf{sort} \mid \Pi x{::}S{\sqsubset}A.\, L \mid \top \mid L_1 \wedge L_2$	classes

Type level

$P ::= a \mid P\, N$	atomic type families
$A ::= P \mid \Pi x{:}A_1.\, A_2$	canonical type families
$Q ::= s \mid Q\, N$	atomic sort families
$S ::= Q \mid \Pi x{::}S_1{\sqsubset}A_1.\, S_2 \mid \top \mid S_1 \wedge S_2$	canonical sort families

Term level

$R ::= c \mid x \mid R\, N$	atomic terms
$N ::= R \mid \lambda x.\, N$	canonical terms

Signatures and contexts

$\Sigma ::= \cdot \mid \Sigma, D$	signatures
$D ::= a{:}K \mid c{:}A \mid s{\sqsubset}a{::}L \mid s_1{\leq}s_2 \mid c{::}S$	declarations
$\Gamma ::= \cdot \mid \Gamma, x{::}S{\sqsubset}A$	contexts

A.2 Expansion and Substitution

All bound variables are tacitly assumed to be sufficiently fresh.

$$\boxed{(A)^- = \alpha}$$

$$\alpha, \beta ::= a \mid \alpha_1 \to \alpha_2$$

$$\begin{aligned}(a)^- &= a \\ (P\ N)^- &= (P)^- \\ (\Pi x{:}A.\, B)^- &= (A)^- \to (B)^-\end{aligned}$$

$$\boxed{\eta_\alpha(R) = N}$$

$$\begin{aligned}\eta_a(R) &= R \\ \eta_{\alpha\to\beta}(R) &= \lambda x.\, \eta_\beta(R\ \eta_\alpha(x))\end{aligned}$$

$$\boxed{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N = N'}$$

$$\frac{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R = (N, a)}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, R = N} \qquad \frac{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R = R'}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, R = R'} \qquad \frac{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N = N'}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, \lambda x.\, N = \lambda x.\, N'}$$

$$\boxed{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R = R'}$$

$$\frac{x \neq x_0}{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, x = x} \qquad \frac{}{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, c = c} \qquad \frac{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R_1 = R_1' \qquad [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_2 = N_2'}{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R_1\, N_2 = R_1'\, N_2'}$$

$$\boxed{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R = (N', \alpha')}$$

$$\frac{}{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, x_0 = (N_0, \alpha_0)}\ \textbf{(subst-rn-var)}$$

$$\frac{\begin{array}{c}[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_1 = (\lambda x.\, N_1, \alpha_2 \to \alpha_1) \\ [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_2 = N_2' \qquad [N_2'/x]^{\mathrm{n}}_{\alpha_2}\, N_1 = N_1'\end{array}}{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_1\, N_2 = (N_1', \alpha_1)}\ \textbf{(subst-rn-}\boldsymbol{\beta}\textbf{)}$$

(Substitution for other syntactic categories (q, p, s, a, l, k, γ) is compositional.)

A.3 Kinding

$$\boxed{\Gamma \vdash_\Sigma L^+ \sqsubset K^+}$$

$$\frac{}{\Gamma \vdash \mathsf{sort} \sqsubset \mathsf{type}} \qquad \frac{\Gamma \vdash S \sqsubset A \qquad \Gamma, x{::}S{\sqsubset}A \vdash L \sqsubset K}{\Gamma \vdash \Pi x{::}S{\sqsubset}A.\, L \sqsubset \Pi x{:}A.\, K}$$

$$\frac{}{\Gamma \vdash \top \sqsubset K} \qquad \frac{\Gamma \vdash L_1 \sqsubset K \qquad \Gamma \vdash L_2 \sqsubset K}{\Gamma \vdash L_1 \wedge L_2 \sqsubset K}$$

$$\boxed{\Gamma \vdash_\Sigma Q^+ \sqsubset P^- \Rightarrow L^-}$$

$$\frac{s{\sqsubset}a{::}L \in \Sigma}{\Gamma \vdash s \sqsubset a \Rightarrow L}$$

$$\frac{\Gamma \vdash Q \sqsubset P \Rightarrow \Pi x{::}S{\sqsubset}A.\, L \qquad \Gamma \vdash N \Leftarrow S \qquad [N/x]^{\mathsf{l}}_A\, L = L'}{\Gamma \vdash Q\, N \sqsubset P\, N \Rightarrow L'}$$

$$\frac{\Gamma \vdash Q \sqsubset P \Rightarrow L_1 \wedge L_2}{\Gamma \vdash Q \sqsubset P \Rightarrow L_1} \qquad \frac{\Gamma \vdash Q \sqsubset P \Rightarrow L_1 \wedge L_2}{\Gamma \vdash Q \sqsubset P \Rightarrow L_2}$$

$$\boxed{\Gamma \vdash_\Sigma S^+ \sqsubset A^+}$$

$$\frac{\Gamma \vdash Q \sqsubset P' \Rightarrow L \qquad P' = P \qquad L = \mathsf{sort}}{\Gamma \vdash Q \sqsubset P}\ (Q\text{-}\mathbf{F})$$

$$\frac{\Gamma \vdash S \sqsubset A \qquad \Gamma, x{::}S{\sqsubset}A \vdash S' \sqsubset A'}{\Gamma \vdash \Pi x{::}S{\sqsubset}A.\, S' \sqsubset \Pi x{:}A.\, A'}\ (\Pi\text{-}\mathbf{F})$$

$$\frac{}{\Gamma \vdash \top \sqsubset A}\ (\top\text{-}\mathbf{F}) \qquad \frac{\Gamma \vdash S_1 \sqsubset A \qquad \Gamma \vdash S_2 \sqsubset A}{\Gamma \vdash S_1 \wedge S_2 \sqsubset A}\ (\wedge\text{-}\mathbf{F})$$

Note: no intro rules for classes $\top$ and $L_1 \wedge L_2$.

A.4 Typing

$$\boxed{\Gamma \vdash_\Sigma R^+ \Rightarrow S^-}$$

$$\frac{c{::}S \in \Sigma}{\Gamma \vdash c \Rightarrow S}\ \textbf{(const)} \qquad \frac{x{::}S{\sqsubset}A \in \Gamma}{\Gamma \vdash x \Rightarrow S}\ \textbf{(var)}$$

$$\frac{\Gamma \vdash R_1 \Rightarrow \Pi x{::}S_2{\sqsubset}A_2.\, S \qquad \Gamma \vdash N_2 \Leftarrow S_2 \qquad [N_2/x]^{\mathsf{s}}_{A_2}\, S = S'}{\Gamma \vdash R_1\, N_2 \Rightarrow S'}\ (\Pi\textbf{-E})$$

$$\frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_1}\ (\wedge\textbf{-E}_1) \qquad \frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_2}\ (\wedge\textbf{-E}_2)$$

$$\boxed{\Gamma \vdash_\Sigma N^+ \Leftarrow S^+}$$

$$\frac{\Gamma \vdash R \Rightarrow Q' \qquad Q' \leq Q}{\Gamma \vdash R \Leftarrow Q}\ \textbf{(switch)}$$

$$\frac{\Gamma, x{::}S{\sqsubset}A \vdash N \Leftarrow S'}{\Gamma \vdash \lambda x.\, N \Leftarrow \Pi x{::}S{\sqsubset}A.\, S'}\ (\Pi\textbf{-I})$$

$$\frac{}{\Gamma \vdash N \Leftarrow \top}\ (\top\textbf{-I}) \qquad \frac{\Gamma \vdash N \Leftarrow S_1 \qquad \Gamma \vdash N \Leftarrow S_2}{\Gamma \vdash N \Leftarrow S_1 \wedge S_2}\ (\wedge\textbf{-I})$$

$$\boxed{Q_1^+ \leq Q_2^+}$$

$$\frac{Q_1 = Q_2}{Q_1 \leq Q_2} \qquad \frac{Q_1 \leq Q' \qquad Q' \leq Q_2}{Q_1 \leq Q_2} \qquad \frac{s_1 {\leq} s_2 \in \Sigma}{s_1 \leq s_2} \qquad \frac{Q_1 \leq Q_2}{Q_1\, N \leq Q_2\, N}$$

A.5 Signatures and Contexts

$$\boxed{\vdash \Sigma \ \mathsf{sig}}$$

$$\frac{}{\vdash \cdot \ \mathsf{sig}} \qquad \frac{\vdash \Sigma \ \mathsf{sig} \qquad \cdot \vdash_{\Sigma^*} K \Leftarrow \mathsf{kind} \qquad a{:}K' \notin \Sigma}{\vdash \Sigma, a{:}K \ \mathsf{sig}}$$

$$\frac{\vdash \Sigma \ \mathsf{sig} \qquad \cdot \vdash_{\Sigma^*} A \Leftarrow \mathsf{type} \qquad c{:}A' \notin \Sigma}{\vdash \Sigma, c{:}A \ \mathsf{sig}}$$

$$\frac{\vdash \Sigma \ \mathsf{sig} \qquad a{:}K \in \Sigma \qquad \cdot \vdash_{\Sigma} L \sqsubset K \qquad s{\sqsubset}a'{::}L' \notin \Sigma}{\vdash \Sigma, s{\sqsubset}a{::}L \ \mathsf{sig}}$$

$$\frac{\vdash \Sigma \ \mathsf{sig} \qquad c{:}A \in \Sigma \qquad \cdot \vdash_{\Sigma} S \sqsubset A \qquad c{::}S' \notin \Sigma}{\vdash \Sigma, c{::}S \ \mathsf{sig}} \qquad \frac{\vdash \Sigma \ \mathsf{sig} \qquad s_1{\sqsubset}a{::}L \in \Sigma \qquad s_2{\sqsubset}a{::}L \in \Sigma}{\vdash \Sigma, s_1{\leq}s_2 \ \mathsf{sig}}$$

$$\boxed{\vdash_{\Sigma} \Gamma \ \mathsf{ctx}}$$

$$\frac{}{\vdash \cdot \ \mathsf{ctx}} \qquad \frac{\vdash \Gamma \ \mathsf{ctx} \qquad \Gamma \vdash S \sqsubset A}{\vdash \Gamma, x{::}S{\sqsubset}A \ \mathsf{ctx}}$$

Appendix B

Full Proofs of Basic Metatheory

B.1 Lemma 3.13 (Composition of Substitutions)

Lemma 3.13 (Composition of Substitutions). *Suppose* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0} N_2 = N_2^{\backprime}$ *and* $x_2 \notin \mathrm{FV}(N_0)$. *Then:*

1. *If* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N^{\backprime}$ *and* $[N_2/x_2]^{\mathrm{n}}_{\alpha_2} N = N'$, *then for some* $N^{\backprime\prime}$,
   $[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2} N^{\backprime} = N^{\backprime\prime}$ *and* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0} N' = N^{\backprime\prime}$,
2. *If* $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R^{\backprime}$ *and* $[N_2/x_2]^{\mathrm{rr}}_{\alpha_2} R = R'$, *then for some* $R^{\backprime\prime}$,
   $[N_2^{\backprime}/x_2]^{\mathrm{rr}}_{\alpha_2} R^{\backprime} = R^{\backprime\prime}$ *and* $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R' = R^{\backprime\prime}$,
3. *If* $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R^{\backprime}$ *and* $[N_2/x_2]^{\mathrm{rn}}_{\alpha_2} R = (N', \beta)$, *then for some* $N^{\backprime\prime}$,
   $[N_2^{\backprime}/x_2]^{\mathrm{rn}}_{\alpha_2} R^{\backprime} = (N^{\backprime\prime}, \beta)$ *and* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0} N' = N^{\backprime\prime}$,
4. *If* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N^{\backprime}, \beta)$ *and* $[N_2/x_2]^{\mathrm{rr}}_{\alpha_2} R = R'$, *then for some* $N^{\backprime\prime}$,
   $[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2} N^{\backprime} = N^{\backprime\prime}$ *and* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R' = (N^{\backprime\prime}, \beta)$,

and similarly for other syntactic categories.

Proof. By lexicographic induction on the unordered pair of α_0 and α_2, and on the first substitution derivation in each clause.

Not all clauses' proofs need be mutually inductive—the four given cases can be proven independently of the ones elided. We give only the proof for the four given cases; the rest are straightforward.

1. Suppose $[N_0/x_0]^{\mathrm{n}}_{\alpha_0} N_2 = N_2^{\backprime}$
   and $\mathcal{D} :: [N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N^{\backprime}$
   and $\mathcal{E} :: [N_2/x_2]^{\mathrm{n}}_{\alpha_2} N = N'$.

   (We want to show:
   $[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2} N^{\backprime} = N^{\backprime\prime}$
   and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0} N' = N^{\backprime\prime}$.)

   Case: $$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ [N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N^{\backprime}, a)\end{array}}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0} R = N^{\backprime}}$$

$$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{[N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R = R'}}{[N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, R = R'}$$ By inversion, using Lemma 3.1.

$[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N^{\backprime} = N^{\backprime\prime}$ and $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R' = (N^{\backprime\prime}, a)$ By i.h. (4) on $\mathcal{D}_1$.

$[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, R' = N^{\backprime\prime}$ By rule.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R = R^{\backprime}}}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, R = R^{\backprime}}$

$$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{[N_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, R = (N', a)}}{[N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, R = N'} \text{ or } \mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{[N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R = R'}}{[N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, R = R'}$$

By inversion.

Subcase: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{[N_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, R = (N', a)}}{[N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, R = N'}$

$[N_2^{\backprime}/x_2]^{\mathrm{rn}}_{\alpha_2}\, R^{\backprime} = (N^{\backprime\prime}, a)$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N' = N^{\backprime\prime}$ By i.h. (3) on $\mathcal{D}_1$.

$[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, R^{\backprime} = N^{\backprime\prime}$ By rule.

Subcase: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{[N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R = R'}}{[N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, R = R'}$

$[N_2^{\backprime}/x_2]^{\mathrm{rr}}_{\alpha_2}\, R^{\backprime} = R^{\backprime\prime}$ and $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R' = R^{\backprime\prime}$ By i.h. (2) on $\mathcal{D}_1$.

$[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, R^{\backprime} = R^{\backprime\prime}$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, R' = R^{\backprime\prime}$ By rule.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N = N^{\backprime}}}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, \lambda x.\, N = \lambda x.\, N^{\backprime}}$

$$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{[N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, N = N'}}{[N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, \lambda x.\, N = \lambda x.\, N'}$$ By inversion.

$[N^{\backprime}_2/x_2]^{\mathrm{n}}_{\alpha_2}\, N^{\backprime} = N^{\backprime\prime}$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N' = N^{\backprime\prime}$ By i.h. (1) on $\mathcal{D}_1$.
$[N^{\backprime}_2/x_2]^{\mathrm{n}}_{\alpha_2}\, \lambda x.\, N^{\backprime} = \lambda x.\, N^{\backprime\prime}$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, \lambda x.\, N' = \lambda x.\, N^{\backprime\prime}$ By rule.

2. Suppose $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_2 = N^{\backprime}_2$ and $\mathcal{D} :: [N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R = R^{\backprime}$ and $\mathcal{E} :: [N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R = R'$. (We want to show: $[N^{\backprime}_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R^{\backprime} = R^{\backprime\prime}$ and $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R' = R^{\backprime\prime}$.)

Case: $\mathcal{D} = \dfrac{x \neq x_0}{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, x = x}$

$\mathcal{E} = \dfrac{x \neq x_2}{[N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, x = x}$ By inversion.

$[N^{\backprime}_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, x = x$ and $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, x = x$ By rule.

Case: $\mathcal{D} = \dfrac{}{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, c = c}$

$\mathcal{E} = \dfrac{}{[N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, c = c}$ By inversion.

$[N^{\backprime}_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, c = c$ and $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, c = c$ By rule.

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ [N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R_3 = R^{\backprime}_3\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_4 = N^{\backprime}_4\end{array}}{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R_3\, N_4 = R^{\backprime}_3\, N^{\backprime}_4}$

$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ [N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R_3 = R'_3\end{array} \qquad \begin{array}{c}\mathcal{E}_2\\ [N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, N_4 = N'_4\end{array}}{[N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R_3\, N_4 = R'_3\, N'_4}$

By inversion.

$[N^{\backprime}_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R^{\backprime}_3 = R^{\backprime\prime}_3$ and $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R'_3 = R^{\backprime\prime}_3$ By i.h. (2) on $\mathcal{D}_1$.
$[N^{\backprime}_2/x_2]^{\mathrm{n}}_{\alpha_2}\, N^{\backprime}_4 = N^{\backprime\prime}_4$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N'_4 = N^{\backprime\prime}_4$ By i.h. (1) on $\mathcal{D}_2$.
$[N^{\backprime}_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R^{\backprime}_3\, N^{\backprime}_4 = R^{\backprime\prime}_3\, N^{\backprime\prime}_4$ and $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R'_3\, N'_4 = R^{\backprime\prime}_3\, N^{\backprime\prime}_4$ By rule.

3. Suppose $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_2 = N^{\backprime}_2$ and $\mathcal{D} :: [N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R = R^{\backprime}$ and $\mathcal{E} :: [N_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, R = (N', \beta)$.

   (We want to show: $[N^{\backprime}_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, R^{\backprime} = (N^{\backprime\prime}, \beta)$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N' = N^{\backprime\prime}$.)

   Case: $\mathcal{D} = \dfrac{x \neq x_0}{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, x = x}$, where $R^{\backprime} = x$

   $\mathcal{E} = \dfrac{}{[N_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, x_2 = (N_2, \alpha_2)}$, where $N' = N_2$ and $x = x_2$

   By inversion.

   $[N^{\backprime}_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, x_2 = (N^{\backprime}_2, \alpha_2)$ By rule.
   $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_2 = N^{\backprime}_2$ By assumption.

   Case: $\mathcal{D} = \dfrac{}{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, c = c}$, where $R = c$

   Impossible: no rule can conclude $\mathcal{E} :: [N_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, c = (N', \beta)$.

   Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R_3 = R^{\backprime}_3} \qquad \overset{\mathcal{D}_2}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_4 = N^{\backprime}_4}}{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\, R_3\, N_4 = R^{\backprime}_3\, N^{\backprime}_4}$

   $$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{[N_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, R_3 = (\lambda x.\, N'_3, \alpha_4 \to \alpha_3)} \qquad \overset{\mathcal{E}_2}{[N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, N_4 = N'_4} \qquad \overset{\mathcal{E}_3}{[N'_4/x]^{\mathrm{n}}_{\alpha_4}\, N'_3 = \hat{N}'_3}}{[N_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, R_3\, N_4 = (N^{\backprime}_3, \alpha_3)}$$

   By inversion.

   We need to show: $[N^{\backprime}_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, R^{\backprime}_3\, N^{\backprime}_4 = (\hat{N}^{\backprime\prime}_3, \alpha_3)$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, \hat{N}'_3 = \hat{N}^{\backprime\prime}_3$.

   $[N^{\backprime}_2/x_2]^{\mathrm{rn}}_{\alpha_2}\, R^{\backprime}_3 = (\lambda x.\, N^{\backprime\prime}_3, \alpha_4 \to \alpha_3)$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, \lambda x.\, N'_3 = \lambda x.\, N^{\backprime\prime}_3$
   By i.h. (3) on $\mathcal{D}_1$.
   $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N'_3 = N^{\backprime\prime}_3$ By inversion.
   $[N^{\backprime}_2/x_2]^{\mathrm{n}}_{\alpha_2}\, N^{\backprime}_4 = N^{\backprime\prime}_4$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N'_4 = N^{\backprime\prime}_4$ By i.h. (1) on $\mathcal{D}_2$.

   $[N^{\backprime\prime}_4/x]^{\mathrm{n}}_{\alpha_4}\, N^{\backprime\prime}_3 = \hat{N}^{\backprime\prime}_3$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, \hat{N}'_3 = \hat{N}^{\backprime\prime}_3$
   By i.h. (1) on (α_0, α_4), using
   $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N'_4 = N^{\backprime\prime}_4$,
   $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N'_3 = N^{\backprime\prime}_3$,

and $[N_4'/x]^{\mathrm{n}}_{\alpha_4}\, N_3' = \hat{N}_3'$

$[N_2^{\backprime}/x_2]^{\mathrm{rn}}_{\alpha_2}\, R_3^{\backprime}\, N_4^{\backprime} = (\hat{N}_3^{\backprime\prime}, \alpha_3)$ By rule, using $[N_2^{\backprime}/x_2]^{\mathrm{rn}}_{\alpha_2}\, R_3^{\backprime} = (\lambda x.\, N_3^{\backprime\prime}, \alpha_4 \to \alpha_3)$, $[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N_4^{\backprime} = N_4^{\backprime\prime}$, and $[N_4^{\backprime\prime}/x]^{\mathrm{n}}_{\alpha_4}\, N_3^{\backprime\prime} = \hat{N}_3^{\backprime\prime}$.

4. Suppose $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_2 = N_2^{\backprime}$
and $\mathcal{D} :: [N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R = (N^{\backprime}, \beta)$
and $\mathcal{E} :: [N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R = R'$.

(We want to show:
$[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N^{\backprime} = N^{\backprime\prime}$
and $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R' = (N^{\backprime\prime}, \beta)$.)

Case: $\mathcal{D} = \dfrac{}{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, x_0 = (N_0, \alpha_0)}$, where $N^{\backprime} = N_0$

$\mathcal{E} = \dfrac{x_0 \neq x_2}{[N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, x_0 = x_0}$, where $R' = x_0$ By inversion.

$[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N_0 = N_0$ By trivial substitution, since $x_2 \notin \mathrm{FV}(N_0)$.
$[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, x_0 = (N_0, \alpha_0)$ By rule.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_3 = (\lambda x.\, N_3, \alpha_4 \to \alpha_3)} \quad \overset{\mathcal{D}_2}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_4 = N_4^{\backprime}} \quad \overset{\mathcal{D}_3}{[N_4^{\backprime}/x]^{\mathrm{n}}_{\alpha_4}\, N_3 = N_3^{\backprime}}}{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_3\, N_4 = (N_3^{\backprime}, \alpha_3)}$

$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{[N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R_3 = R_3'} \quad \overset{\mathcal{E}_2}{[N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, N_4 = N_4'}}{[N_2/x_2]^{\mathrm{rr}}_{\alpha_2}\, R_3\, N_4 = R_3'\, N_4'}$

By inversion.

We need to show: $[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N_3^{\backprime} = N_3^{\backprime\prime}$ and $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_3'\, N_4' = (N_3^{\backprime\prime}, \alpha_3)$

$[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, \lambda x.\, N_3 = \lambda x.\, N_3'$ and $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_3' = (\lambda x.\, N_3', \alpha_4 \to \alpha_3)$
By i.h. (4) on $\mathcal{D}_1$.
$[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N_3 = N_3'$ By inversion.
$[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N_4^{\backprime} = N_4^{\backprime\prime}$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_4' = N_4^{\backprime\prime}$
By i.h. (1) on $\mathcal{D}_2$.

$[N_4^{\backprime\prime}/x]^{\mathrm{n}}_{\alpha_4}\, N_3' = N_3^{\backprime\prime}$ and $[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N_3^{\backprime} = N_3^{\backprime\prime}$

By i.h. (1) on (α_2, α_4), using
$[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N_4^{\backprime} = N_4^{\backprime\prime}$,
$[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N_3 = N_3'$,
and $\mathcal{D}_3 :: [N_4^{\backprime}/x]^{\mathrm{n}}_{\alpha_4}\, N_3 = N_3^{\backprime}$.

$[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_3'\, N_4' = (N_3^{\backprime\prime}, \alpha_3)$

By rule, using
$[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\, R_3' = (\lambda x.\, N_3', \alpha_4 \to \alpha_3)$
$[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_4' = N_4^{\backprime\prime}$,
and $[N_4^{\backprime\prime}/x]^{\mathrm{n}}_{\alpha_4}\, N_3' = N_3^{\backprime\prime}$.

□

B.2 Theorem 3.15 (Proto-Substitution, terms)

Theorem 3.15 (Proto-Substitution, terms).

1. *If*
   - $\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0 A_0$ *(and $\Gamma_{\mathrm{L}}^{*} \vdash N_0 \Leftarrow A_0$), and*
   - $\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash N \Leftarrow SA$ *(and $\Gamma_{\mathrm{L}}^{*}, x_0{:}A_0, \Gamma_{\mathrm{R}}^{*} \vdash N \Leftarrow A$), and*
   - $[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_{\mathrm{R}} = \Gamma_{\mathrm{R}}^{\backprime}$ *, and*
   - $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S^{\backprime}$ *(and $[N_0/x_0]^{\mathrm{a}}_{A_0}\, A = A^{\backprime}$),*

   then

   - $[N_0/x_0]^{\mathrm{n}}_{A_0}\, N = N^{\backprime}$ *, and*
   - $\Gamma_{\mathrm{L}}, \Gamma_{\mathrm{R}}^{\backprime} \vdash N^{\backprime} \Leftarrow S^{\backprime} A^{\backprime}$ *(and $\Gamma_{\mathrm{L}}^{*}, (\Gamma_{\mathrm{R}}^{\backprime})^{*} \vdash N^{\backprime} \Leftarrow A^{\backprime}$).*

2. *If*
   - $\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0 A_0$ *(and $\Gamma_{\mathrm{L}}^{*} \vdash N_0 \Leftarrow A_0$), and*
   - $\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash R \Rightarrow SA$ *(and $\Gamma_{\mathrm{L}}^{*}, x_0{:}A_0, \Gamma_{\mathrm{R}}^{*} \vdash R \Rightarrow A$), and*
   - $[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_{\mathrm{R}} = \Gamma_{\mathrm{R}}^{\backprime}$ *,*

   then

   - $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S^{\backprime}$ *(and $[N_0/x_0]^{\mathrm{a}}_{A_0}\, A = A^{\backprime}$), and*
   - *either*
     - $[N_0/x_0]^{\mathrm{rr}}_{A_0}\, R = R^{\backprime}$ *and*
     - $\Gamma_{\mathrm{L}}, \Gamma_{\mathrm{R}}^{\backprime} \vdash R^{\backprime} \Rightarrow S^{\backprime} A^{\backprime}$ *(and $\Gamma_{\mathrm{L}}^{*}, (\Gamma_{\mathrm{R}}^{\backprime})^{*} \vdash R^{\backprime} \Rightarrow A^{\backprime}$),*

     or
     - $[N_0/x_0]^{\mathrm{rn}}_{A_0}\, R = (N^{\backprime}, (A^{\backprime})^{-})$ *and*

– $\Gamma_L, \Gamma'_R \vdash N' \Leftarrow S'A'$ *(and* $\Gamma^*_L, (\Gamma'_R)^* \vdash N' \Leftarrow A'$*)* .

Proof. By lexicographic induction on $(A_0)^-$ and the derivation $\mathcal{D}$ hypothesizing $x_0{::}S_0{\sqsubset}A_0$.

1. Suppose $\Gamma_L \vdash N_0 \Leftarrow S_0$
   and $\mathcal{D} :: \Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R \vdash N \Leftarrow S$
   and $[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_R = \Gamma'_R$
   and $[N_0/x_0]^{s}_{A_0}\, S = S'$.

Case: $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R \vdash R \Rightarrow Q_1} \qquad \overset{\mathcal{D}_2}{Q_1 \leq Q}}{\Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R \vdash R \Leftarrow Q}$$

$[N_0/x_0]^{s}_{A_0}\, Q_1 = Q'_1$ and either
$([N_0/x_0]^{rr}_{A_0}\, R = R'$ and $\Gamma_L, \Gamma'_R \vdash R' \Rightarrow Q'_1)$, or
$([N_0/x_0]^{rn}_{A_0}\, R = (N', (P'_1)^-)$ and $\Gamma_L, \Gamma'_R \vdash N' \Leftarrow Q'_1)$
By i.h. (2) on $\mathcal{D}_1$.

$S' = Q'$ and $[N_0/x_0]^{q}_{A_0}\, Q = Q'$ — By inversion.
$Q'_1 \leq Q'$ — By Lemma 3.14 (Substitution into Subsorting).

Subcase: $[N_0/x_0]^{rr}_{A_0}\, R = R'$ and $\Gamma_L, \Gamma'_R \vdash R' \Rightarrow Q'_1$
$[N_0/x_0]^{n}_{A_0}\, R = R'$ — By rule **subst-n-atom**.
$\Gamma_L, \Gamma'_R \vdash R' \Leftarrow Q'$ — By rule **switch**.

Subcase: $[N_0/x_0]^{rn}_{A_0}\, R = (N', (P'_1)^-)$ and $\Gamma_L, \Gamma'_R \vdash N' \Leftarrow Q'_1$
$N' = R'$ and $\Gamma_L, \Gamma'_R \vdash R' \Rightarrow Q'_2$ and $Q'_2 \leq Q'_1$ — By inversion.
$(P'_1)^- = a$, for some a — By definition.
$[N_0/x_0]^{n}_{A_0}\, R = R'$ — By rule **subst-n-atom-norm**.
$Q'_2 \leq Q'$ — By rule.
$\Gamma_L, \Gamma'_R \vdash R' \Leftarrow Q'$ — By rule **switch**.

Case: $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R, x{::}S_1{\sqsubset}A_1 \vdash N \Leftarrow S_2}}{\Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R \vdash \lambda x.\, N \Leftarrow \Pi x{::}S_1{\sqsubset}A_1.\, S_2}$$

$S' = \Pi x{::}S'_1{\sqsubset}A'_1.\, S'_2$ and $[N_0/x_0]^{s}_{A_0}\, S_1 = S'_1$ and
$[N_0/x_0]^{a}_{A_0}\, A_1 = A'_1$ and $[N_0/x_0]^{s}_{A_0}\, S_2 = S'_2$ — By inversion.
$[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_R, x{::}S_1{\sqsubset}A_1 = \Gamma'_R, x{::}S'_1{\sqsubset}A'_1$ — By rule.
$[N_0/x_0]^{n}_{A_0}\, N = N'$ and $\Gamma_L, \Gamma'_R, x{::}S'_1{\sqsubset}A'_1 \vdash N' \Leftarrow S'_2$
By i.h. (1) on $\mathcal{D}_1$.
$[N_0/x_0]^{n}_{A_0}\, \lambda x.\, N = \lambda x.\, N'$ — By rule.

$\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash \lambda x.\, N' \Leftarrow \Pi x{::}S'_1{\sqsubset}A'_1.\, S'_2$ By rule.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S_1} \qquad \overset{\mathcal{D}_2}{\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S_2}}{\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S_1 \wedge S_2}$

$S' = S'_1 \wedge S'_2$ and $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S_1 = S'_1$ and $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S_2 = S'_2$
By inversion.
$[N_0/x_0]^{\mathrm{s}}_{A_0}\, N = N'_1$ and $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N'_1 \Leftarrow S'_1$ By i.h. (1) on $\mathcal{D}_1$.
$[N_0/x_0]^{\mathrm{s}}_{A_0}\, N = N'_2$ and $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N'_2 \Leftarrow S'_2$ By i.h. (1) on $\mathcal{D}_2$.
$N'_1 = N'_2$ By Theorem 3.2 (Functionality of Substitution).
Let $N' = N'_1 = N'_2$: $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow S_1 \wedge S_2$ By rule.

Case: $\mathcal{D} = \dfrac{}{\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash N \Leftarrow \top}$

$[N_0/x_0]^{\mathrm{n}}_{A_0}\, N = N'$ By core LF Proto-Substitution Theorem.
$\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow \top$ By rule.

Note: This case is where we make use of the three grey assumptions to clause 1. The remainder of the grey assumptions and conclusions are only required to ensure that these three key assumptions are satisfied on every inductive appeal. (The interested reader may gain great insight into the essential difficulty of the proof by tracing these dependencies; all of the action is in the "application" case of clause 2.)

2. Suppose $\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0$ and $\mathcal{D} :: \Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash R \Rightarrow S$ and $[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_{\mathrm{R}} = \Gamma'_{\mathrm{R}}$.

Case: $\mathcal{D} = \dfrac{c{:}S \in \Sigma}{\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash c \Rightarrow S}$

$\mathrm{FV}(S) = \emptyset$ By signature well-formedness.
$[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S$ By trivial substitution.
$[N_0/x_0]^{\mathrm{rr}}_{A_0}\, c = c$ By rule.
$\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash c \Rightarrow S$ By rule.

Case: $\mathcal{D} = \dfrac{x{::}S{\sqsubset}A \in \Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}}}{\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash x \Rightarrow S}$

Subcase: $x{::}S{\sqsubset}A \in \Gamma_{\mathrm{L}}$

$x_0 \notin \mathrm{FV}(S)$ and $x_0 \neq x$ — By α-conversion convention.
$[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S$ — By trivial substitution.
$[N_0/x_0]^{\mathrm{rr}}_{A_0}\, x = x$ — By rule.
$\Gamma_{\mathrm{L}}, \Gamma^{\backprime}_{\mathrm{R}} \vdash x \Rightarrow S$ — By rule.

Subcase: $x{::}S{\sqsubset}A = x_0{::}S_0{\sqsubset}A_0$

$x_0 \notin \mathrm{FV}(S_0)$ — By α-conversion convention.
$[N_0/x_0]^{\mathrm{s}}_{A_0}\, S_0 = S_0$ — By trivial substitution.
$[N_0/x_0]^{\mathrm{rn}}_{A_0}\, x_0 = (N_0, (A_0)^-)$ — By rule.
$\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0$ — By assumption.
$\Gamma_{\mathrm{L}}, \Gamma^{\backprime}_{\mathrm{R}} \vdash N_0 \Leftarrow S_0$ — By weakening.

Subcase: $x{::}S{\sqsubset}A \in \Gamma_{\mathrm{R}}$

$[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_{\mathrm{R}} = \Gamma^{\backprime}_{\mathrm{R}}$ — By assumption.
$x{::}S^{\backprime}{\sqsubset}A^{\backprime} \in \Gamma^{\backprime}_{\mathrm{R}}$ and $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S^{\backprime}$ — By inversion.
$x_0 \neq x$ — By α-conversion convention.
$[N_0/x_0]^{\mathrm{rr}}_{A_0}\, x = x$ — By rule.
$\Gamma_{\mathrm{L}}, \Gamma^{\backprime}_{\mathrm{R}} \vdash x \Rightarrow S^{\backprime}$ — By rule.

Case:
$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash R_1 \Rightarrow \Pi x{::}S_2{\sqsubset}A_2.\, S_1\end{array} \quad \begin{array}{c}\mathcal{D}_2\\ \Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash N_2 \Leftarrow S_2\end{array} \quad \begin{array}{c}\mathcal{D}_3\\ [N_2/x]^{\mathrm{s}}_{A_2}\, S_1 = S'_1\end{array}}{\Gamma_{\mathrm{L}}, x_0{::}S_0{\sqsubset}A_0, \Gamma_{\mathrm{R}} \vdash R_1\, N_2 \Rightarrow S'_1}$$

$[N_0/x_0]^{\mathrm{s}}_{A_0}\, \Pi x{::}S_2{\sqsubset}A_2.\, S_1 = S^{\backprime}$ and either
$([N_0/x_0]^{\mathrm{rr}}_{A_0}\, R_1 = R^{\backprime}_1$ and $\Gamma_{\mathrm{L}}, \Gamma^{\backprime}_{\mathrm{R}} \vdash R^{\backprime}_1 \Rightarrow S^{\backprime})$, or
$([N_0/x_0]^{\mathrm{rn}}_{A_0}\, R_1 = (N^{\backprime}_1, (\Pi x{:}A^{\backprime}_2.\, A^{\backprime}_1)^-)$ and $\Gamma_{\mathrm{L}}, \Gamma^{\backprime}_{\mathrm{R}} \vdash N^{\backprime}_1 \Leftarrow S^{\backprime})$
By i.h. (2) on $\mathcal{D}_1$.

$S^{\backprime} = \Pi x{::}S^{\backprime}_2{\sqsubset}A^{\backprime}_2.\, S^{\backprime}_1$
and $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S_2 = S^{\backprime}_2$ and $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S_1 = S^{\backprime}_1$ — By inversion.

$[N_0/x_0]^{\mathrm{n}}_{A_0}\, N_2 = N^{\backprime}_2$ and $\Gamma_{\mathrm{L}}, \Gamma^{\backprime}_{\mathrm{R}} \vdash N^{\backprime}_2 \Leftarrow S^{\backprime}_2$ — By i.h. (1) on $\mathcal{D}_2$.
$[N^{\backprime}_2/x]^{\mathrm{s}}_{A_2}\, S^{\backprime}_1 = S^{\backprime\prime}_1$ and $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S'_1 = S^{\backprime\prime}_1$
By Lemma 3.13 (Composition).

Subcase: $[N_0/x_0]^{\mathrm{rr}}_{A_0}\, R_1 = R^{\backprime}_1$ and $\Gamma_{\mathrm{L}}, \Gamma^{\backprime}_{\mathrm{R}} \vdash R^{\backprime}_1 \Rightarrow \Pi x{::}S^{\backprime}_2{\sqsubset}A^{\backprime}_2.\, A^{\backprime}_1$
$[N_0/x_0]^{\mathrm{rr}}_{A_0}\, R_1\, N_2 = R^{\backprime}_1\, N^{\backprime}_2$ — By rule.
$\Gamma_{\mathrm{L}}, \Gamma^{\backprime}_{\mathrm{R}} \vdash R^{\backprime}_1\, N^{\backprime}_2 \Rightarrow S^{\backprime\prime}_1$ — By rule.

Subcase: $[N_0/x_0]^{\mathrm{rn}}_{A_0}\, R_1 = (N'_1, (\Pi x{:}A'_2.\, A'_1)^-)$ and $\Gamma_L, \Gamma'_R \vdash N'_1 \Leftarrow \Pi x{::}S'_2{\sqsubset}A'_2.\, S'_1$

$N'_1 = \lambda x.\, N'$ and $\Gamma_L, \Gamma'_R, x{::}S'_2{\sqsubset}A'_2 \vdash N' \Leftarrow S'_1$ By inversion.

$[N'_2/x]^{\mathrm{n}}_{A_2}\, N' = N''$ and $\Gamma_L, \Gamma'_R \vdash N'' \Leftarrow S''_1$

By i.h. (1) at $(A_2)^-$, using
$\Gamma_L, \Gamma'_R \vdash N'_2 \Leftarrow S'_2$,
$\Gamma_L, \Gamma'_R, x{::}S'_2{\sqsubset}A'_2 \vdash N' \Leftarrow S'_1$,
$[N'_2/x]^{\gamma}_{A'_2}\, \cdot = \cdot$,
and $[N'_2/x]^{\mathrm{s}}_{A'_2}\, S'_1 = S''_1$.

$[N_0/x_0]^{\mathrm{rn}}_{A_0}\, R_1\, N_2 = N''$ By rule, using
$[N_0/x_0]^{\mathrm{rn}}_{A_0}\, R_1 = (\lambda x.\, N', (\Pi x{:}A'_2.\, A'_1)^-)$,
$[N_0/x_0]^{\mathrm{n}}_{A_0}\, N_2 = N'_2$,
and $[N'_2/x]^{\mathrm{n}}_{A_2}\, N' = N''$.

Case: $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R \vdash R \Rightarrow S_1 \wedge S_2}}{\Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R \vdash R \Rightarrow S_1}$$

$[N_0/x_0]^{\mathrm{s}}_{A_0}\, S_1 \wedge S_2 = S'$ and either
$([N_0/x_0]^{\mathrm{rr}}_{A_0}\, R = R'$ and $\Gamma_L, \Gamma'_R \vdash R' \Rightarrow S')$, or
$([N_0/x_0]^{\mathrm{rn}}_{A_0}\, R = (N', (A')^-)$ and $\Gamma_L, \Gamma'_R \vdash N' \Leftarrow S')$

By i.h. (2) on $\mathcal{D}_1$.

$S' = S'_1 \wedge S'_2$ and $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S_1 = S'_1$ By inversion.

Subcase: $[N_0/x_0]^{\mathrm{rr}}_{A_0}\, R = R'$ and $\Gamma_L, \Gamma'_R \vdash R' \Rightarrow S'_1 \wedge S'_2$

$\Gamma_L, \Gamma'_R \vdash R' \Rightarrow S'_1$ By rule.

Subcase: $[N_0/x_0]^{\mathrm{rn}}_{A_0}\, R = (N', (A')^-)$ and $\Gamma_L, \Gamma'_R \vdash N' \Leftarrow S'_1 \wedge S'_2$

$\Gamma_L, \Gamma'_R \vdash N' \Leftarrow S'_1$ By inversion.

Case: $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R \vdash R \Rightarrow S_1 \wedge S_2}}{\Gamma_L, x_0{::}S_0{\sqsubset}A_0, \Gamma_R \vdash R \Rightarrow S_2}$$

Similar. □

B.3 Lemma 3.21 (Commutativity of Substitution and η-expansion)

Lemma 3.21 (Commutativity of Substitution and η-expansion). *Substitution commutes with η-expansion. In particular:*

1. (a) *If* $[\eta_\alpha(x)/x]^{\mathrm{n}}_\alpha N = N'$, *then* $N = N'$,
   (b) *If* $[\eta_\alpha(x)/x]^{\mathrm{rr}}_\alpha R = R'$, *then* $R = R'$,
   (c) *If* $[\eta_\alpha(x)/x]^{\mathrm{rn}}_\alpha R = (N, \beta)$, *then* $\eta_\beta(R) = N$,
2. *If* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0} \eta_\alpha(R) = N'$, *then*
   (a) *if* head$(R) \neq x_0$, *then* $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'$ *and* $\eta_\alpha(R') = N'$,
   (b) *if* head$(R) = x_0$ *and* $x_0{:}\alpha_0 \vdash R : \alpha$, *then* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha)$,

and similarly for other syntactic categories.

Proof. By lexicographic induction on α and the given substitution derivation.

Not all clauses' proofs need be mutually inductive—the two given cases can be proven independently of the ones elided. We give only the proof for the two given cases; the rest are straightforward.

1. (a) Suppose $\mathcal{D} :: [\eta_\alpha(x)/x]^{\mathrm{n}}_\alpha N = N'$.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{[\eta_\alpha(x)/x]^{\mathrm{rr}}_\alpha R = R'}}{[\eta_\alpha(x)/x]^{\mathrm{n}}_\alpha R = R'}$

$R = R'$ By i.h. (1b) on $\alpha, \mathcal{D}_1$.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{[\eta_\alpha(x)/x]^{\mathrm{rn}}_\alpha R = (R', a')}}{[\eta_\alpha(x)/x]^{\mathrm{n}}_\alpha R = R'}$

$\eta_{a'}(R) = R'$ By i.h. (1c) on $\alpha, \mathcal{D}_1$.
$\eta_{a'}(R) = R$ By definition.
$R = R'$ By transitivity of equality.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{[\eta_\alpha(x)/x]^{\mathrm{n}}_\alpha N = N'}}{[\eta_\alpha(x)/x]^{\mathrm{n}}_\alpha \lambda y.\, N = \lambda y.\, N'}$

$N = N'$ By i.h. (1a) on $\alpha, \mathcal{D}_1$.
$\lambda y.\, N = \lambda y.\, N'$ By compatibility of equality.

(b) Suppose $\mathcal{D} :: [\eta_\alpha(x)/x]^{\text{rr}}_\alpha\, R = R'$.

Case: $$\mathcal{D} = \frac{y \neq x}{[\eta_\alpha(x)/x]^{\text{rr}}_\alpha\, y = y}$$

$y = y$ By reflexivity of equality.

Case: $$\mathcal{D} = \frac{}{[\eta_\alpha(x)/x]^{\text{rr}}_\alpha\, c = c}$$

$c = c$ By reflexivity of equality.

Case: $$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1\\ [\eta_\alpha(x)/x]^{\text{rr}}_\alpha\, R_1 = R_1'\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ [\eta_\alpha(x)/x]^{\text{n}}_\alpha\, N_2 = N_2'\end{array}}{[\eta_\alpha(x)/x]^{\text{rr}}_\alpha\, R_1\; N_2 = R_1'\; R_2'}$$

$R_1 = R_1'$ By i.h. (1b) on $\alpha, \mathcal{D}_1$.
$N_2 = N_2'$ By i.h. (1a) on $\alpha, \mathcal{D}_2$.
$R_1\; N_2 = R_1'\; N_2'$ By compatibility of equality.

(c) Suppose $\mathcal{D} :: [\eta_\alpha(x)/x]^{\text{rn}}_\alpha\, R = (N, \beta)$

Case: $$\mathcal{D} = \frac{}{[\eta_\alpha(x)/x]^{\text{rn}}_\alpha\, x = (\eta_\alpha(x), \alpha)}$$

$\eta_\alpha(x) = \eta_\alpha(x)$ By reflexivity of equality.

Case: $$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1\\ [\eta_\alpha(x)/x]^{\text{rn}}_\alpha\, R_1 = (\lambda y.\, N_1, \alpha_2 \to \alpha_1)\\ \mathcal{D}_2 \qquad\qquad\qquad\qquad \mathcal{D}_3\\ [\eta_\alpha(x)/x]^{\text{n}}_\alpha\, N_2 = N_2' \qquad [N_2'/y]^{\text{n}}_{\alpha_2}\, N_1 = N_1'\end{array}}{[\eta_\alpha(x)/x]^{\text{rn}}_\alpha\, R_1\; N_2 = (N_1', \alpha_1)}$$

We need to show: $\eta_{\alpha_1}(R_1\; N_2) = N_1'$.

$\eta_{\alpha_2 \to \alpha_1}(R_1) = \lambda y.\, N_1$ By i.h. (1c) on $\alpha, \mathcal{D}_1$.

$\eta_{\alpha_2\to\alpha_1}(R_1) = \lambda y.\, \eta_{\alpha_1}(R_1\ \eta_{\alpha_2}(y))$	By definition.
$\eta_{\alpha_1}(R_1\ \eta_{\alpha_2}(y)) = N_1$	By compatibility of equality.
$N_2 = N_2'$	By i.h. (1a) on $\alpha, \mathcal{D}_2$.
$\mathcal{D}_3 :: [N_2/y]^{\mathrm{n}}_{\alpha_2}\ \eta_{\alpha_1}(R_1\ \eta_{\alpha_2}(y)) = N_1'$	By replacing equals for equals.
$y \notin \mathrm{FV}(R_1)$ and $\mathrm{head}(R_1) \neq y$	By α-conversion convention.
$[N_2/y]^{\mathrm{rr}}_{\alpha_2}\ R_1\ \eta_{\alpha_2}(y) = R'$ and $\eta_{\alpha_1}(R') = N_1'$	By i.h. (2a) on $\alpha_1, \mathcal{D}_3$.
$R' = R_1'\ N_2''$ and	
$[N_2/y]^{\mathrm{rr}}_{\alpha_2}\ R_1 = R_1'$ and $\mathcal{D}_4 :: [N_2/y]^{\mathrm{n}}_{\alpha_2}\ \eta_{\alpha_2}(y) = N_2''$	By inversion.
$[N_2/y]^{\mathrm{rr}}_{\alpha_2}\ R_1 = R_1$	By trivial substitution.
$R_1 = R_1'$	By functionality of substitution.
$\mathrm{head}(y) = y$	By definition.
$y{:}\alpha_2 \vdash y : \alpha_2$	By rule.
$[N_2/y]^{\mathrm{rn}}_{\alpha_2}\ y = (N_2'', \alpha_2)$	By i.h. (2b) on $\alpha_2, \mathcal{D}_4$.
$N_2'' = N_2$	By inversion.
$R' = R_1\ N_2$	By equality reasoning.
$\eta_{\alpha_1}(R') = N_1'$	From above.
$\eta_{\alpha_1}(R_1\ N_2) = N_1'$	By replacing equals for equals.

2. Suppose $\mathcal{D} :: [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\ \eta_\alpha(R) = N'$.

   (a) Suppose $\mathrm{head}(R) \neq x_0$. We need to show: $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\ R = R'$ and $\eta_\alpha(R') = N'$.

Case: $\alpha = a$.

$\eta_a(R) = R$	By definition.
$\mathcal{D} :: [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\ R = N'$	By equality.
$N' = R'$ and $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\ R = R'$.	By inversion, using Lemma 3.1.
$\eta_a(R') = R'$	By definition.

Case: $\alpha = \alpha_2 \to \alpha_1$.

$\eta_{\alpha_2\to\alpha_1}(R) = \lambda y.\, \eta_{\alpha_1}(R\ \eta_{\alpha_2}(y))$	By definition.
$\mathcal{D} :: [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\ \lambda y.\, \eta_{\alpha_1}(R\ \eta_{\alpha_2}(y)) = N'$	By equality.
$N' = \lambda y.\, N''$ and $\mathcal{D}_1 :: [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\ \eta_{\alpha_1}(R\ \eta_{\alpha_2}(y)) = N''$	By inversion.
$[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\ R\ \eta_{\alpha_2}(y) = R''$ and $\eta_{\alpha_1}(R'') = N''$	By i.h. (2a) on $\alpha_1, \mathcal{D}_1$.

$R'' = R'\ N$ and $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0}\ R = R'$ and $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\ \eta_{\alpha_2}(y) = N$

By inversion.

$N = \eta_{\alpha_2}(y)$ — By trivial substitution and functionality.

$\eta_{\alpha_1}(R'') = \eta_{\alpha_1}(R'\eta_{\alpha_2}(y)) = N''$ — By equality.

$\eta_{\alpha_2\to\alpha_1}(R') = \lambda y.\,\eta_{\alpha_1}(R'\ \eta_{\alpha_2}(y))$ — By definition.

$\quad = \lambda y.\,N'' = N'$ — By equality.

(b) Suppose the head$(R) = x_0$ and $x_0{:}\alpha_0 \vdash R : \alpha$. We need to show: $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\ R = (N', \alpha)$.

Case: $\alpha = a$.

$\eta_a(R) = R$ — By definition.

$\mathcal{D} :: [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\ R = N'$. — By equality.

$[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\ R = (N', a')$ — By inversion, using Lemma 3.1.

$a' = a$ — By Lemma 3.20.

Case: $\alpha = \alpha_2 \to \alpha_1$.

$\eta_{\alpha_2\to\alpha_1}(R) = \lambda y.\,\eta_{\alpha_1}(R\ \eta_{\alpha_2}(y))$ — By definition.

$\mathcal{D} :: [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\ \lambda y.\,\eta_{\alpha_1}(R\ \eta_{\alpha_2}(y)) = N'$ — By equality.

$N' = \lambda y.\,N''$ and $\mathcal{D}_1 :: [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\ \eta_{\alpha_1}(R\ \eta_{\alpha_2}(y)) = N''$ — By inversion.

$x_0{:}\alpha_0 \vdash R\ \eta_{\alpha_2}(y) : \alpha_1$ — By rule.

$\mathcal{E} :: [N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\ R\ \eta_{\alpha_2}(y) = (N'', \alpha_1)$ — By i.h. (2b) on $\alpha_1, \mathcal{D}_1$.

$\mathcal{E}_1 :: [N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\ R = (\lambda y.\,N''', \alpha_2' \to \alpha_1)$ and
$\mathcal{E}_2 :: [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\ \eta_{\alpha_2}(y) = N$ and
$\mathcal{E}_3 :: [N/y]^{\mathrm{n}}_{\alpha_2'}\ N''' = N''$ — By inversion.

$\alpha_2' \to \alpha_1 = \alpha_2 \to \alpha_1$ — By Lemma 3.20.

$N = \eta_{\alpha_2}(y)$ — By trivial substitution and functionality.

$\mathcal{E}_3 :: [\eta_{\alpha_2}(y)/y]^{\mathrm{n}}_{\alpha_2}\ N''' = N''$ — By equality.

$N''' = N''$ — By i.h. (1a) on $\alpha_2, \mathcal{E}_3$.

$\mathcal{E}_1 :: [N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\ R = (\lambda y.\,N'', \alpha_2 \to \alpha_1)$,
i.e., $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0}\ R = (N', \alpha)$ — By equality.

□

B.4 Theorem 3.22 (Expansion)

Theorem 3.22 (Expansion). *If $\Gamma \vdash S \sqsubset A$ and $\Gamma \vdash R \Rightarrow S$, then $\Gamma \vdash \eta_A(R) \Leftarrow S$.*

Proof. By induction on S.

Case: $S = \top$

$\Gamma \vdash \eta_A(R) \Leftarrow \top$ — By rule.

Case: $S = S_1 \wedge S_2$

$\Gamma \vdash S \sqsubset A_1$ and $\Gamma \vdash S \sqsubset A_2$ — By inversion.
$\Gamma \vdash R \Rightarrow S_1$ and $\Gamma \vdash R \Rightarrow S_2$ — By rules $\wedge$-**E**$_1$ and $\wedge$-**E**$_2$.
$\Gamma \vdash \eta_A(R) \Leftarrow S_1$ and $\Gamma \vdash \eta_A(R) \Leftarrow S_2$ — By i.h. on S_1 and S_2.
$\Gamma \vdash \eta_A(R) \Leftarrow S_1 \wedge S_2$ — By rule $\wedge$-**I**.

Case: $S = Q$

$A = P$ — By inversion.
$\eta_A(R) = \eta_P(R) = R$ — By definition.
$Q \leq Q$ — By rule.
$\Gamma \vdash R \Leftarrow Q$ — By rule **switch**.

Case: $S = \Pi x{::}S_1{\sqsubset}A_1.\, S_2$

$A = \Pi x{:}A_1.\, A_2$ and $\Gamma \vdash S_1 \sqsubset A_1$ and $\Gamma, x{::}S_1{\sqsubset}A_1 \vdash S_2 \sqsubset A_2$
— By inversion.
$\eta_A(R) = \eta_{\Pi x:A_1.\, A_2}(R) = \lambda x.\, \eta_{A_2}(R\ \eta_{A_1}(x))$ — By definition.

$\Gamma, y{::}S_1{\sqsubset}A_1 \vdash S_1 \sqsubset A_1$ — By weakening.
$\Gamma, y{::}S_1{\sqsubset}A_1 \vdash y \Rightarrow S_1$ — By rule.
$\Gamma, y{::}S_1{\sqsubset}A_1 \vdash \eta_{A_1}(y) \Leftarrow S_1$ — By i.h. on S_1.

$\Gamma, y{::}S_1{\sqsubset}A_1, x{::}S_1{\sqsubset}A_1 \vdash S_2 \sqsubset A_2$ — By weakening.
$[\eta_{A_1}(y)/x]^{\mathsf{s}}_{A_1}\, S_2 = S_2'$ — By Theorem 3.19 (Substitution).
$S_2' = [y/x]\, S_2$ — By Lemma 3.21 (Commutativity).

$\Gamma, y{::}S_1{\sqsubset}A_1 \vdash R \Rightarrow \Pi x{::}S_1{\sqsubset}A_1.\, S_2$ — By weakening.

$\Gamma, y{::}S_1{\sqsubset}A_1 \vdash R\ \eta_{A_1}(y) \Rightarrow [y/x]\,S_2$ By rule Π-**E**.
$\Gamma, x{::}S_1{\sqsubset}A_1 \vdash R\ \eta_{A_1}(x) \Rightarrow S_2$ By α-conversion.

$\Gamma, x{::}S_1{\sqsubset}A_1 \vdash \eta_{A_2}(R\ \eta_{A_1}(x)) \Leftarrow S_2$ By i.h. on S_2.
$\Gamma \vdash \lambda x.\, \eta_{A_2}(R\ \eta_{A_1}(x)) \Leftarrow \Pi x{::}S_1{\sqsubset}A_1.\, S_2$ By rule Π-**I**.

□

B.5 Theorem 3.30 (Generalized Algorithmic ⇒ Declarative)

Theorem 3.30 (Generalized Algorithmic ⇒ Declarative).

1. *If* $\mathcal{D} :: \Delta \leqq T$, *then* $\bigwedge(\Delta) \leq T$.
2. *If* $\mathcal{D} :: \Delta \mathbin{@} x{::}\Delta_1{\sqsubset}A_1 = \Delta_2$, *then* $\bigwedge(\Delta) \leq \Pi x{::}\bigwedge(\Delta_1){\sqsubset}A_1.\ \bigwedge(\Delta_2)$.

Proof. By induction on $\mathcal{D}$. To reduce clutter, we omit the refined type A_1 from bound variables, since it does not affect declarative subsorting in any significant way.

1. Suppose $\mathcal{D} :: \Delta \leqq T$.

Case: $\mathcal{D} = \dfrac{}{\Delta \leqq \top}$

$\bigwedge(\Delta) \leq \top$ By rule $\top$-**R**.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta \leqq S_1} \qquad \overset{\mathcal{D}_2}{\Delta \leqq S_2}}{\Delta \leqq S_1 \wedge S_2}$

$\bigwedge(\Delta) \leq S_1$ By i.h. (1) on $\mathcal{D}_1$.
$\bigwedge(\Delta) \leq S_2$ By i.h. (1) on $\mathcal{D}_2$.
$\bigwedge(\Delta) \leq S_1 \wedge S_2$ By rule $\wedge$-**R**.

Case: $\mathcal{D} = \dfrac{Q' \in \Delta \qquad Q' \leq Q}{\Delta \leqq Q}$

$\bigwedge(\Delta) \leq Q$ By Lemma 3.29.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta \mathbin{@} x{::}\mathrm{split}(S_1) = \Delta_2} \qquad \overset{\mathcal{D}_2}{\Delta_2 \leqq S_2}}{\Delta \leqq \Pi x{::}S_1.\, S_2}$

$\bigwedge(\Delta) \leq \Pi x{::}\bigwedge(\mathrm{split}(S_1)).\ \bigwedge(\Delta_2)$ By i.h. (2) on $\mathcal{D}_1$.

$S_1 \leq \bigwedge(\mathrm{split}(S_1))$ By Lemma 3.28.
$\bigwedge(\Delta_2) \leq S_2$ By i.h. (1) on $\mathcal{D}_2$.
$\Pi x{::}\bigwedge(\mathrm{split}(S_1)).\ \bigwedge(\Delta)_2 \leq \Pi x{::}S_1.\, S_2$ By rule **S-**Π.

$\bigwedge(\Delta) \leq \Pi x{::}S_1.\, S_2$ By rule **trans**.

2. Suppose $\mathcal{D} :: \Delta \mathbin{@} x{::}\mathrm{split}(S_1) = \Delta_2$.

Case: $\mathcal{D} = \dfrac{}{\cdot \mathbin{@} x{::}\Delta_1 = \cdot}$

$\bigwedge(\cdot) = \top$ By definition.
We need to show $\top \leq \Pi x{::}\bigwedge(\Delta_1).\, \top$.
$\top \leq \Pi x{::}\bigwedge(\Delta_1).\, \top$ By rule $\top/\Pi$**-dist**.

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta \mathbin{@} x{::}\Delta_1 = \Delta_2} \qquad \overset{\mathcal{D}_2}{\Delta_1 \leqq S_1} \qquad \overset{\mathcal{D}_3}{[\eta_A(x)/y]^{\mathrm{s}}_A\, S_2 = S_2'}}{(\Delta, \Pi y{::}S_1{\sqsubset}A.\, S_2) \mathbin{@} x{::}\Delta_1 = \Delta_2, \mathrm{split}(S_2')}$

$\bigwedge(\Delta, \Pi y{::}S_1{\sqsubset}A.\, S_2) = \bigwedge(\Delta) \wedge \Pi y{::}S_1{\sqsubset}A.\, S_2$ By definition.
Want to show: $\bigwedge(\Delta) \wedge \Pi y{::}S_1{\sqsubset}A.\, S_2 \leq \Pi x{::}\bigwedge(\Delta_1).\ \bigwedge(\Delta_2, \mathrm{split}(S_2'))$.
$S_2' = [x/y]\, S_2$ By Lemma 3.21 (Commutativity).
So (α-varied): $\bigwedge(\Delta) \wedge \Pi x{::}S_1{\sqsubset}A.\, S_2' \leq \Pi x{::}\bigwedge(\Delta_1).\ \bigwedge(\Delta_2, \mathrm{split}(S_2'))$.

Note: in the following, we omit some uses of reflexivity (rule **refl**).

$\bigwedge(\Delta) \leq \Pi x{::}\bigwedge(\Delta_1).\ \bigwedge(\Delta_2)$ By i.h. (2) on $\mathcal{D}_1$.
$\bigwedge(\Delta_1) \leq S_1$ By i.h. (1) on $\mathcal{D}_2$.
$\bigwedge(\Delta) \wedge \Pi x{::}S_1.\, S_2' \leq \Pi x{::}\bigwedge(\Delta_1).\big(\bigwedge(\Delta_2) \wedge S_2'\big)$ By rule $\wedge/\Pi$**-dist′**.

$S_2' \leq \bigwedge(\mathrm{split}(S_2'))$ By Lemma 3.28.
$\bigwedge(\Delta_2) \wedge S_2' \leq \bigwedge(\Delta_2) \wedge \bigwedge(\mathrm{split}(S_2'))$ By rule **S-**$\wedge$.
$\bigwedge(\Delta_2) \wedge \bigwedge(\mathrm{split}(S_2')) \leq \bigwedge(\Delta_2, \mathrm{split}(S_2'))$ By Lemma 3.27.
$\bigwedge(\Delta_2) \wedge S_2' \leq \bigwedge(\Delta_2, \mathrm{split}(S_2'))$ By rule **trans**.
$\Pi x{::}\bigwedge(\Delta_1).\big(\bigwedge(\Delta_2) \wedge S_2'\big) \leq \Pi x{::}\bigwedge(\Delta_1).\ \bigwedge(\Delta_2, \mathrm{split}(S_2'))$
By rule **S-**Π.

$\bigwedge(\Delta) \wedge \Pi x{::}S_1.\, S_2' \leq \Pi x{::}\bigwedge(\Delta_1).\ \bigwedge(\Delta_2, \mathrm{split}(S_2'))$ By rule **trans**.

Case: $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta \mathbin{@} x{::}\Delta_1 = \Delta_2} \qquad \Delta_1 \not\leq S_1}{\Delta, \Pi y{::}S_1{\sqsubset}A.\, S_2 \mathbin{@} x :: \Delta_1 = \Delta_2}$$

$\bigwedge(\Delta, \Pi y{::}S_1{\sqsubset}A.\, S_2) = \bigwedge(\Delta) \wedge \Pi y{::}S_1{\sqsubset}A.\, S_2$	By definition.
$\bigwedge(\Delta) \leq \Pi x{::} \bigwedge(\Delta_1).\ \bigwedge(\Delta_2)$	By i.h. (2) on $\mathcal{D}_1$.
$\bigwedge(\Delta) \wedge \Pi y{::}S_1{\sqsubset}A.\, S_2 \leq \Pi x{::} \bigwedge(\Delta_1).\ \bigwedge(\Delta_2)$	By rule $\wedge$-**L**$_1$.

Case: $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta \mathbin{@} x{::}\Delta_1 = \Delta_2} \qquad \nexists S_2'.\, [\eta_A(x)/y]^{\mathrm{s}}_A\, S_2 = S_2'}{\Delta, \Pi y{::}S_1{\sqsubset}A.\, S_2 \mathbin{@} x :: \Delta_1 = \Delta_2}$$

Similar.

Case: $$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta \mathbin{@} x{::}\Delta_1 = \Delta_2}}{\Delta, Q \mathbin{@} x :: \Delta_1 = \Delta_2}$$

Similar. □

B.6 Lemma 3.34

Lemma 3.34. *If $\mathcal{D} :: \Gamma \vdash \Delta \sqsubset \Pi x{:}A_1.\, A_2$ and $\mathcal{E} :: \Gamma \vdash \Delta \mathbin{@} N = \Delta_2$ and $[N/x]^{\mathrm{a}}_{A_1}\, A_2 = A_2'$, then $\Gamma \vdash \Delta_2 \sqsubset A_2'$.*

Proof. By induction on $\mathcal{E}$.

Case: $$\mathcal{E} = \dfrac{}{\Gamma \vdash \cdot \mathbin{@} N = \cdot}$$

$\Gamma \vdash \cdot \sqsubset A_2'$	By rule.

Case: $$\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash \Delta \mathbin{@} N = \Delta_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash N \Leftarrow S_1} \qquad \overset{\mathcal{E}_3}{[N/x]^{\mathrm{s}}_{A_1}\, S_2 = S_2'}}{\Gamma \vdash (\Delta, \Pi x{::}S_1{\sqsubset}A_1.\, S_2) \mathbin{@} N = \Delta_2, \mathrm{split}(S_2')}$$

$\mathcal{D}_1 :: \Gamma \vdash \Delta \sqsubset \Pi x{:}A_1.\, A_2$ and	
$\mathcal{D}_2 :: \Gamma \vdash \Pi x{::}S_1{\sqsubset}A_1.\, S_2 \sqsubset \Pi x{:}A_1.\, A_2$	By inversion on $\mathcal{D}$.
$\Gamma \vdash \Delta_2 \sqsubset A_2'$	By i.h. on $\mathcal{E}_1$.

$\Gamma \vdash S_1 \sqsubset A_1$ and $\Gamma, x{::}S_1{\sqsubset}A_1 \vdash S_2 \sqsubset A_2$ By inversion on $\mathcal{D}_2$.
$\Gamma \vdash N \Leftarrow S_1$ By Theorem 3.9 (Soundness of Alg. Typing).
$\Gamma \vdash S_2' \sqsubset A_2'$ By Theorem 3.19 (Substitution).
$\Gamma \vdash \mathsf{split}(S_2') \sqsubset A_2'$ By Lemma 3.33.

$\Gamma \vdash (\Delta_2, \mathsf{split}(S_2')) \sqsubset A_2'$ By Lemma 3.32.

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash \Delta \mathbin{@} N = \Delta_2} \qquad \Gamma \nvdash N \Leftarrow S_1}{\Gamma \vdash (\Delta, \Pi x{::}S_1{\sqsubset}A_1.\, S_2) \mathbin{@} N = \Delta_2}$

$\Gamma \vdash \Delta \sqsubset \Pi x{:}A_1.\, A_2$ By inversion on $\mathcal{D}$.
$\Gamma \vdash \Delta_2 \sqsubset A_2'$ By i.h. on $\mathcal{E}_1$.

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash \Delta \mathbin{@} N = \Delta_2} \qquad \nexists S_2'.\, [N/y]^{\mathsf{s}}_{A_1}\, S_2 = S_2'}{\Gamma \vdash (\Delta, \Pi x{::}S_1{\sqsubset}A_1.\, S_2) \mathbin{@} N = \Delta_2}$

Similar.

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash \Delta \mathbin{@} N = \Delta_2}}{\Gamma \vdash (\Delta, Q) \mathbin{@} N = \Delta_2}$

Impossible:
$\Gamma \vdash Q \sqsubset \Pi x{:}A_1.\, A_2$ By inversion on $\mathcal{D}$.
But there is no rule that can conclude this. □

B.7 Theorem 3.38 (Generalized Intrinsic ⇒ Algorithmic)

Theorem 3.38 (Generalized Intrinsic ⇒ Algorithmic).

1. *If* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\mathcal{E} :: \Gamma \vdash \eta_A(R) \Lleftarrow S$ *and* $\Gamma \vdash \Delta \sqsubset A$ *and* $\Gamma \vdash S \sqsubset A$*, then* $\Delta \leqq S$.
2. *If* $\Gamma \vdash x \Rrightarrow \Delta_1$ *and* $\mathcal{E} :: \Gamma \vdash \Delta \mathbin{@} \eta_{A_1}(x) = \Delta_2$ *and* $\Gamma \vdash \Delta_1 \sqsubset A_1$ *and* $\Gamma \vdash \Delta \sqsubset \Pi x{:}A_1.\, A_2$*, then* $\Delta \mathbin{@} x{::}\Delta_1{\sqsubset}A_1 = \Delta_2$.

Proof. By induction on A, S, and $\mathcal{E}$. We omit the refined type A_1 from the Δ_1 argument of the application judgement when it is clear from context.

1. Suppose $\mathcal{D} :: \Gamma \vdash R \Rrightarrow \Delta$, $\mathcal{E} :: \Gamma \vdash \eta_A(R) \Lleftarrow S$, $\mathcal{F} :: \Gamma \vdash \Delta \sqsubset A$, and $\mathcal{G} :: \Gamma \vdash S \sqsubset A$.

   Case: $S = Q$

   $A = P$ — By inversion on $\mathcal{G}$.
   $\eta_P(R) = R$ — By definition.

   $$\mathcal{E} = \frac{\Gamma \vdash R \Rrightarrow \Delta \qquad Q' \in \Delta \qquad Q' \leq Q}{\Gamma \vdash R \Lleftarrow Q}$$ By inversion.

   $\Delta \leqq Q$ — By rule.

   Case: $S = \top$

   $\Delta \leqq \top$ — By rule.

   Case: $S = S_1 \wedge S_2$

   $$\mathcal{E} = \frac{\overset{\mathcal{E}_1}{\Gamma \vdash \eta_A(R) \Lleftarrow S_1} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash \eta_A(R) \Lleftarrow S_2}}{\Gamma \vdash \eta_A(R) \Lleftarrow S_1 \wedge S_2}$$ By inversion.

   $\Delta \leqq S_1$ — By i.h. (1) on S_1 and $\mathcal{E}_1$
   $\Delta \leqq S_2$ — By i.h. (1) on S_2 and $\mathcal{E}_2$
   $\Delta \leqq S_1 \wedge S_2$ — By rule.

   Case: $S = \Pi x{::}S_1{\sqsubset}A_1.\, S_2$

   $A = \Pi x{:}A_1.\, A_2$ and
   $\Gamma \vdash S_1 \sqsubset A_1$ and $\Gamma, x{::}S_1{\sqsubset}A_1 \vdash S_2 \sqsubset A_2$ — By inversion on $\mathcal{G}$.
   $\eta_{\Pi x:A_1.\, A_2}(R) = \lambda x.\, \eta_{A_2}(R\ \eta_{A_1}(x))$ — By definition.

   $$\mathcal{E} = \frac{\overset{\mathcal{E}_1}{\Gamma, x{::}S_1{\sqsubset}A_1 \vdash \eta_{A_2}(R\ \eta_{A_1}(x)) \Lleftarrow S_2}}{\Gamma \vdash \lambda x.\, \eta_{A_2}(R\ \eta_{A_1}(x)) \Lleftarrow \Pi x{::}S_1{\sqsubset}A_1.\, S_2}$$ By inversion.

   $\Gamma, x{::}S_1{\sqsubset}A_1 \vdash R \Rrightarrow \Delta$ — By weakening.
   $\Gamma, x{::}S_1{\sqsubset}A_1 \vdash \Delta\ @\ \eta_{A_1}(x) = \Delta_2$ — By Theorem 3.7, clause (3).
   $\Gamma, x{::}S_1{\sqsubset}A_1 \vdash R\ \eta_{A_1}(x) \Rrightarrow \Delta_2$ — By rule.
   $[\eta_{A_1}(x)/A_1]^{\mathsf{a}}_x\, A_2 = A_2$ — By validity of $\Pi x{:}A_1.\, A_2$ and Lemma 3.21.

$\Gamma, x{::}S_1{\sqsubset}A_1 \vdash \Delta_2 \sqsubset A_2$ By Lemma 3.34.
$\Delta_2 \leqq S_2$ By i.h. (1) on A_2, S_2, and $\mathcal{E}_1$.

$\Gamma, x{::}S_1{\sqsubset}A_1 \vdash x \Rrightarrow \mathsf{split}(S_1)$ By rule.
$\Gamma, x{::}S_1{\sqsubset}A_1 \vdash \Delta \mathbin{@} \eta_{A_1}(x) = \Delta_2$ From above.
$\Gamma \vdash \mathsf{split}(S_1) \sqsubset A_1$ By Lemma 3.33.
$\Gamma \vdash \Delta \sqsubset \Pi x{:}A_1.\, A_2$ By assumption.
$\Delta \mathbin{@} x{::}\mathsf{split}(S_1) = \Delta_2$ By i.h. (2) on A_1.

$\Delta \leqq \Pi x{::}S_1{\sqsubset}A_1.\, S_2$ By rule.

2. Suppose $\mathcal{D} :: \Gamma \vdash x \Rrightarrow \Delta_1$ and $\mathcal{E} :: \Gamma \vdash \Delta \mathbin{@} \eta_{A_1}(x) = \Delta_2$ and $\mathcal{F} :: \Gamma \vdash \Delta_1 \sqsubset A_1$ and $\mathcal{G} :: \Gamma \vdash \Delta \sqsubset \Pi x{:}A_1.\, A_2$.

Case: $\mathcal{E} = \dfrac{}{\Gamma \vdash \cdot \mathbin{@} \eta_{A_1}(x) = \cdot}$

$\cdot \mathbin{@} x{::}\Delta_1 = \cdot$ By rule.

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash \Delta \mathbin{@} \eta_{A_1}(x) = \Delta_2} \qquad \overset{\mathcal{E}_2}{\Gamma \vdash \eta_{A_1}(x) \Lleftarrow S_1} \qquad \overset{\mathcal{E}_3}{[\eta_{A_1}(x)/y]^{\mathsf{s}}_{A_1}\, S_2 = S_2'}}{\Gamma \vdash (\Delta, \Pi y{::}S_1{\sqsubset}A_1.\, S_2) \mathbin{@} \eta_{A_1}(x) = \Delta_2, \mathsf{split}(S_2')}$

$\Gamma \vdash \Delta \sqsubset \Pi x{:}A_1.\, A_2$ and
$\Gamma \vdash S_1 \sqsubset A_1$ and $\Gamma, y{::}S_1{\sqsubset}A_1 \vdash S_2 \sqsubset A_2$ By inversion on $\mathcal{G}$.
$\Delta \mathbin{@} x{::}\Delta_1 = \Delta_2$ By i.h. (2) on $\mathcal{E}_1$.
$\Delta_1 \leqq S_1$ By i.h. (1) on $\mathcal{E}_2$.
$[\eta_{A_1}(x)/y]^{\mathsf{s}}_{A_1}\, S_2 = S_2'$ By subderivation $\mathcal{E}_3$.
$(\Delta, \Pi y{::}S_1{\sqsubset}A_1.\, S_2) \mathbin{@} x{::}\Delta_1 = (\Delta_2, \mathsf{split}(S_2'))$ By rule.

Case: $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Gamma \vdash \Delta \mathbin{@} \eta_{A_1}(x) = \Delta_2} \qquad \Gamma \nvdash \eta_{A_1}(x) \Lleftarrow S_1}{\Gamma \vdash (\Delta, \Pi x{::}S_1{\sqsubset}A_1.\, S_2) \mathbin{@} \eta_{A_1}(x) = \Delta_2}$

$\Gamma \vdash \Delta \sqsubset \Pi x{:}A_1.\, A_2$ and
$\Gamma \vdash S_1 \sqsubset A_1$ and $\Gamma, y{::}S_1{\sqsubset}A_1 \vdash S_2 \sqsubset A_2$ By inversion on $\mathcal{G}$.
$\Delta \mathbin{@} x{::}\Delta_1 = \Delta_2$ By i.h. (2) on $\mathcal{E}_1$.
$\Delta_1 \nleqq S_1$ By Theorem 3.37 (Alg. Subsumption), contrapositive.
$(\Delta, \Pi y{::}S_1{\sqsubset}A_1.\, S_2) \mathbin{@} x{::}\Delta_1 = \Delta_2$ By rule.

$$\textbf{Case: } \mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Gamma \vdash \Delta \mathbin{@} \eta_{A_1}(x) = \Delta_2\end{array} \qquad \nexists S_2'.\, [\eta_{A_1}(x)/y]^{\mathsf{s}}_{A_1}\, S_2 = S_2'}{\Gamma \vdash (\Delta, \Pi y{::}S_1{\sqsubset}A_1.\, S_2) \mathbin{@} \eta_{A_1}(x) = \Delta_2}$$

$\Delta \mathbin{@} x{::}\Delta_1 = \Delta_2$ By i.h. (2) on $\mathcal{E}_1$.
$\nexists S_2'.\, [\eta_A(x)/y]^{\mathsf{s}}_{A_1}\, S_2 = S_2'$ By side condition.
$(\Delta, \Pi y{::}S_1{\sqsubset}A_1.\, S_2) \mathbin{@} x{::}\Delta_1 = \Delta_2$ By rule.

$$\textbf{Case: } \mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Gamma \vdash \Delta \mathbin{@} N = \Delta_2\end{array}}{\Gamma \vdash (\Delta, Q) \mathbin{@} N = \Delta_2}$$

Impossible:
$\Gamma \vdash Q \sqsubset \Pi x{:}A_1.\, A_2$ By inversion on $\mathcal{G}$.
But there is no rule that can conclude this. □

Appendix C

Complete Subset Interpretation Rules

In the judgment forms below, superscript + and − indicate a judgment's "inputs" and "outputs", respectively.

C.1 Kinding

$$\boxed{\Gamma \vdash_{\Sigma} L^{+} \sqsubset K^{+} \overset{\text{form}}{\rightsquigarrow} \widehat{L}^{-}}$$

$$\frac{}{\Gamma \vdash \mathsf{sort} \sqsubset \mathsf{type} \overset{\text{form}}{\rightsquigarrow} \boldsymbol{\lambda} Q_{\mathrm{f}}\boldsymbol{.}\; Q_{\mathrm{f}}}$$

$$\frac{\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S} \qquad \Gamma, x{::}S{\sqsubset}A \vdash L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L}}{\Gamma \vdash \Pi x{::}S{\sqsubset}A.\, L \sqsubset \Pi x{:}A.\, K \overset{\text{form}}{\rightsquigarrow} \boldsymbol{\lambda} Q_{\mathrm{f}}\boldsymbol{.}\; \Pi x{:}A.\, \Pi \widehat{x}{:}\widehat{S}\boldsymbol{(}\eta_A(x)\boldsymbol{)}.\, \widehat{L}\boldsymbol{(}Q_{\mathrm{f}}\; \eta_A(x)\boldsymbol{)}}$$

$$\frac{}{\Gamma \vdash \top \sqsubset K \overset{\text{form}}{\rightsquigarrow} \boldsymbol{\lambda} Q_{\mathrm{f}}\boldsymbol{.}\; 1} \qquad \frac{\Gamma \vdash L_1 \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L_1} \qquad \Gamma \vdash L_2 \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L_2}}{\Gamma \vdash L_1 \wedge L_2 \sqsubset K \overset{\text{form}}{\rightsquigarrow} \boldsymbol{\lambda} Q_{\mathrm{f}}\boldsymbol{.}\; \widehat{L_1}\boldsymbol{(}Q_{\mathrm{f}}\boldsymbol{)} \times \widehat{L_2}\boldsymbol{(}Q_{\mathrm{f}}\boldsymbol{)}}$$

$$\boxed{\Gamma \vdash_{\Sigma} Q^{+} \sqsubset P^{-} \Rightarrow L^{-} \rightsquigarrow \widehat{Q}^{-}}$$

$$\frac{s{\sqsubset}a{::}L \in \Sigma}{\Gamma \vdash s \sqsubset a \Rightarrow L \rightsquigarrow \widehat{s/i}}$$

$$\frac{\Gamma \vdash Q \sqsubset P \Rightarrow \Pi x{::}S{\sqsubset}A.\, L \rightsquigarrow \widehat{Q} \qquad \Gamma \vdash N \Leftarrow S \rightsquigarrow \widehat{N} \qquad [N/x]^{\mathrm{l}}_{A}\, L = L'}{\Gamma \vdash Q\, N \sqsubset P\, N \Rightarrow L' \rightsquigarrow \widehat{Q}\, N\, \widehat{N}}$$

$$\frac{\Gamma \vdash Q \sqsubset P \Rightarrow L_1 \wedge L_2 \rightsquigarrow \widehat{Q}}{\Gamma \vdash Q \sqsubset P \Rightarrow L_1 \rightsquigarrow \pi_1\, \widehat{Q}} \qquad \frac{\Gamma \vdash Q \sqsubset P \Rightarrow L_1 \wedge L_2 \rightsquigarrow \widehat{Q}}{\Gamma \vdash Q \sqsubset P \Rightarrow L_2 \rightsquigarrow \pi_2\, \widehat{Q}}$$

$$\boxed{\Gamma \vdash_\Sigma S^+ \sqsubset A^+ \rightsquigarrow \widehat{S}^-}$$

$$\frac{\Gamma \vdash Q \sqsubset P' \Rightarrow L \rightsquigarrow \widehat{Q} \qquad P' = P \qquad L = \mathsf{sort}}{\Gamma \vdash Q \sqsubset P \rightsquigarrow \boldsymbol{\lambda} N.\, Q\ [\widehat{Q}]\ N}\ (Q\text{-}\mathbf{F})$$

$$\frac{\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S} \qquad \Gamma, x{::}S{\sqsubset}A \vdash S' \sqsubset A' \rightsquigarrow \widehat{S'}}{\Gamma \vdash \Pi x{::}S{\sqsubset}A.\, S' \sqsubset \Pi x{:}A.\, A' \rightsquigarrow \boldsymbol{\lambda} N.\, \Pi x{:}A.\, \Pi \widehat{x}{:}\widehat{S}\boldsymbol{(}\eta_A(x)\boldsymbol{)}.\, \widehat{S'}\boldsymbol{(}N@x\boldsymbol{)}}\ (\Pi\text{-}\mathbf{F})$$

$$\frac{}{\Gamma \vdash \top \sqsubset A \rightsquigarrow \boldsymbol{\lambda} N.\, 1}\ (\top\text{-}\mathbf{F}) \qquad \frac{\Gamma \vdash S_1 \sqsubset A \rightsquigarrow \widehat{S_1} \qquad \Gamma \vdash S_2 \sqsubset A \rightsquigarrow \widehat{S_2}}{\Gamma \vdash S_1 \wedge S_2 \sqsubset A \rightsquigarrow \boldsymbol{\lambda} N.\, \widehat{S_1}\boldsymbol{(}N\boldsymbol{)} \times \widehat{S_2}\boldsymbol{(}N\boldsymbol{)}}\ (\wedge\text{-}\mathbf{F})$$

Note: no intro rules for classes $\top$ and $L_1 \wedge L_2$.

$$\boxed{K^+ \overset{\text{pred}}{\rightsquigarrow} \widehat{K}^-}$$

$$\frac{}{\mathsf{type} \overset{\text{pred}}{\rightsquigarrow} \boldsymbol{\lambda(}Q_{\mathsf{f}}, P\boldsymbol{)}.\, Q_{\mathsf{f}} \nrightarrow P \to \mathsf{type}} \qquad \frac{K \overset{\text{pred}}{\rightsquigarrow} \widehat{K}}{\Pi x{:}A.\, K \overset{\text{pred}}{\rightsquigarrow} \boldsymbol{\lambda(}Q_{\mathsf{f}}, P\boldsymbol{)}.\, \Pi x{:}A.\, \widehat{K}\boldsymbol{(}Q_{\mathsf{f}}\ \eta_A(x), P\ \eta_A(x)\boldsymbol{)}}$$

$$\boxed{K \overset{\leq}{\rightsquigarrow} \widehat{K}}$$

$$\frac{}{\mathsf{type} \overset{\leq}{\rightsquigarrow} \boldsymbol{\lambda(}P, Q_{1\mathsf{f}}, Q_1, Q_{2\mathsf{f}}, Q_2\boldsymbol{)}.\, \Pi f_1{:}Q_{1\mathsf{f}}.\, \Pi f_2{:}Q_{2\mathsf{f}}.\, \Pi x{:}P.\, Q_1\ [f_1]\ x \to Q_2\ [f_2]\ x}$$

$$\frac{K \overset{\leq}{\rightsquigarrow} \widehat{K}}{\Pi x{:}A.\, K \overset{\leq}{\rightsquigarrow} \boldsymbol{\lambda(}P, Q_{1\mathsf{f}}, Q_1, Q_{2\mathsf{f}}, Q_2\boldsymbol{)}.\, \Pi x{:}A.\, \widehat{K}\boldsymbol{(}P', Q_{1\mathsf{f}}', Q_1', Q_{2\mathsf{f}}', Q_2'\boldsymbol{)}}$$
$$\text{(where, for each } P,\ P' = P\ \eta_A(x)\text{)}$$

C.2 Typing

$$\boxed{\Gamma \vdash_\Sigma R^+ \Rightarrow S^- \rightsquigarrow \widehat{R}^-}$$

$$\frac{c{::}S \in \Sigma}{\Gamma \vdash c \Rightarrow S \rightsquigarrow \widehat{c}}\ (\mathbf{const}) \qquad \frac{x{::}S{\sqsubset}A \in \Gamma}{\Gamma \vdash x \Rightarrow S \rightsquigarrow \widehat{x}}\ (\mathbf{var})$$

$$\frac{\Gamma \vdash R_1 \Rightarrow \Pi x{::}S_2{\sqsubset}A_2.\, S \rightsquigarrow \widehat{R_1} \qquad \Gamma \vdash N_2 \Leftarrow S_2 \rightsquigarrow \widehat{N_2} \qquad [N_2/x]^{\mathrm{s}}_{A_2}\, S = S'}{\Gamma \vdash R_1\, N_2 \Rightarrow S' \rightsquigarrow \widehat{R_1}\, N_2\, \widehat{N_2}}\ (\Pi\text{-}\mathbf{E})$$

$$\frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2 \rightsquigarrow \widehat{R}}{\Gamma \vdash R \Rightarrow S_1 \rightsquigarrow \pi_1\, \widehat{R}}\ (\wedge\text{-}\mathbf{E}_1) \qquad \frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2 \rightsquigarrow \widehat{R}}{\Gamma \vdash R \Rightarrow S_2 \rightsquigarrow \pi_2\, \widehat{R}}\ (\wedge\text{-}\mathbf{E}_2)$$

$$\boxed{\Gamma \vdash_\Sigma N^+ \Leftarrow S^+ \rightsquigarrow \widehat{N}^-}$$

$$\frac{\Gamma \vdash R \Rightarrow Q' \rightsquigarrow \widehat{R} \qquad \Gamma \vdash Q' \le Q \rightsquigarrow F}{\Gamma \vdash R \Leftarrow Q \rightsquigarrow F\mathbf{(}R, \widehat{R}\mathbf{)}}\ (\mathbf{switch})$$

$$\frac{\Gamma, x{::}S{\sqsubset}A \vdash N \Leftarrow S' \rightsquigarrow \widehat{N}}{\Gamma \vdash \lambda x.\, N \Leftarrow \Pi x{::}S{\sqsubset}A.\, S' \rightsquigarrow \lambda x.\, \lambda \widehat{x}.\, \widehat{N}}\ (\Pi\text{-}\mathbf{I})$$

$$\frac{}{\Gamma \vdash N \Leftarrow \top \rightsquigarrow \langle\rangle}\ (\top\text{-}\mathbf{I}) \qquad \frac{\Gamma \vdash N \Leftarrow S_1 \rightsquigarrow \widehat{N_1} \qquad \Gamma \vdash N \Leftarrow S_2 \rightsquigarrow \widehat{N_2}}{\Gamma \vdash N \Leftarrow S_1 \wedge S_2 \rightsquigarrow \langle \widehat{N}_1, \widehat{N}_2 \rangle}\ (\wedge\text{-}\mathbf{I})$$

$$\boxed{\Gamma \vdash Q_1^+ \le Q_2^+ \rightsquigarrow F}$$

$$\frac{Q_1 = Q_2}{\Gamma \vdash Q_1 \le Q_2 \rightsquigarrow \boldsymbol{\lambda}\mathbf{(}R, R_1\mathbf{)}.\, R_1}\ (\mathbf{refl})$$

$$\frac{\begin{array}{cc} Q_1 \le Q' \rightsquigarrow \widehat{Q_1\text{-}Q'} & \Gamma \vdash Q_1 \sqsubset P \Rightarrow \mathsf{sort} \rightsquigarrow \widehat{Q_1} \\ \Gamma \vdash Q' \le Q_2 \rightsquigarrow F & \Gamma \vdash Q' \sqsubset P \Rightarrow \mathsf{sort} \rightsquigarrow \widehat{Q'} \end{array}}{\Gamma \vdash Q_1 \le Q_2 \rightsquigarrow \boldsymbol{\lambda}\mathbf{(}R, R_1\mathbf{)}.\, F\mathbf{(}R, \widehat{Q_1\text{-}Q'}\ \widehat{Q_1}\ \widehat{Q'}\ R\ R_1\mathbf{)}}\ (\mathbf{climb})$$

$$\boxed{Q_1^+ \le Q_2^- \rightsquigarrow \widehat{Q_1\text{-}Q_2}}$$

$$\frac{s_1{\le}s_2 \in \Sigma}{s_1 \le s_2 \rightsquigarrow s_1\text{-}s_2} \qquad \frac{Q_1 \le Q_2 \rightsquigarrow \widehat{Q_1\text{-}Q_2}}{Q_1\, N \le Q_2\, N \rightsquigarrow \widehat{Q_1\text{-}Q_2}\, N}$$

C.3 Signatures and Contexts

$$\boxed{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}}$$

$$\frac{}{\vdash \cdot\ \mathsf{sig} \rightsquigarrow \cdot} \qquad \frac{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma} \qquad \cdot \vdash_{\Sigma^*} K \Leftarrow \mathsf{kind} \qquad a{:}K' \notin \Sigma}{\vdash \Sigma, a{:}K\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}, a{:}K}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma} \qquad \cdot \vdash_{\Sigma^*} A \Leftarrow \mathsf{type} \qquad c{:}A' \notin \Sigma}{\vdash \Sigma, c{:}A\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}, c{:}A}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma} \qquad a{:}K \in \Sigma \qquad \cdot \vdash_{\Sigma} L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L_{\mathrm{f}}} \qquad K \overset{\text{pred}}{\rightsquigarrow} \widehat{K_{\mathrm{p}}} \qquad s{\sqsubset}a'{::}L' \notin \Sigma}{\vdash \Sigma, s{\sqsubset}a{::}L\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma},\ \widehat{s}{:}K,\ \widehat{s}/i{:}\widehat{L_{\mathrm{f}}}(\widehat{s}),\ s{:}\widehat{K_{\mathrm{p}}}(\widehat{s}, a)}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma} \qquad c{:}A \in \Sigma \qquad \cdot \vdash_{\Sigma} S \sqsubset A \rightsquigarrow \widehat{S} \qquad c{::}S' \notin \Sigma}{\vdash \Sigma, c{::}S\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}, \widehat{c}{:}\widehat{S}(\eta_A(c))}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma} \qquad s_1{\sqsubset}a{::}L \in \Sigma \qquad s_2{\sqsubset}a{::}L \in \Sigma \qquad a{:}K \in \Sigma \qquad K \overset{\leq}{\rightsquigarrow} \widehat{K}}{\vdash \Sigma, s_1{\leq}s_2\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}, s_1\text{-}s_2{:}\widehat{K}(a, \widehat{s_1}, s_1, \widehat{s_2}, s_2)}$$

$$\boxed{\vdash_{\Sigma} \Gamma\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma}}$$

$$\frac{}{\vdash \cdot\ \mathsf{ctx} \rightsquigarrow \cdot} \qquad \frac{\vdash \Gamma\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma} \qquad \Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}}{\vdash \Gamma, x{::}S{\sqsubset}A\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma}, x{:}A, \widehat{x}{:}\widehat{S}(\eta_A(x))}$$

Bibliography

[AB04] Steven Awodey and Andrej Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14(4):447–471, 2004. 4

[AC01] David Aspinall and Adriana B. Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1-2):273–309, 2001. 1.1

[Asp00] David Aspinall. Subtyping with power types. In Peter Clote and Helmut Schwichtenberg, editors, *CSL*, volume 1862 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2000. 1.1

[Bar92] Henk Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 117–309. Oxford University Press, 1992. 6.1.6

[BCDC83] Henk Berendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, December 1983. 5.4.1

[BTCGS91] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991. 4

[C$^+$86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986. 4.1, 4.1

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 27(2-6):45–58, 1981. 5.4.1

[CDG$^+$07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007. 2.5

[Cer96] Iliano Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, February 1996. 1.1

[CG03] Adriana Compagnoni and Healfdane Goguen. Typed operational semantics for higher-order subtyping. *Information and Computation*, 184(2):242–297, August 2003. 1.1, 6.1.4, 6.1.4

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940. 1.1

[CP96] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press. 1.1

[CP02] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information & Computation*, 179(1):19–75, November 2002. 1.1

[CP03] Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003. 5.3.1

[Cra03] Karl Crary. Toward a foundational typed assembly language. In G. Morrisett, editor, *Proceedings of the 30th Annual Symposium on Principles of Programming Languages (POPL '03)*, pages 198–212, New Orleans, Louisiana, January 2003. ACM Press. 1.1

[Cra07] Karl Crary. Sound and complete elimination of singleton kinds. *ACM Transactions on Computational Logic (TOCL)*, 8(2), April 2007. 6.1.5

[Dav05] Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, May 2005. Available as Technical Report CMU-CS-05-110. 1, 2.5

[dB94a] N. G. de Bruijn. The Mathematical Vernacular, a language for mathematics with typed sets. In Nederpelt et al. [NGdV94], pages 865–935. 1.1

[dB94b] N. G. de Bruijn. A survey of the project Automath. In Nederpelt et al. [NGdV94], pages 141–161. 1.1

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press. 5.1

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*, pages 207–212. ACM Press, 1982. 6.1.1

[Dow93] Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 139–145, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664. 5.1

[Dun07] Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, August 2007. Available as Technical Report CMU-CS-07-129. 1

[DZ92] Philip W. Dart and Justin Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, Cambridge, Massachusetts, 1992. 2.5

[Fre94] Tim Freeman. *Refinement Types for ML*. PhD thesis, Carnegie Mellon University, March 1994. Available as Technical Report CMU-CS-94-110. 1, 2.5

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972. 6.1.1

[Gog94] Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994. Available as Technical Report ECS-LFCS-94-304. 1.1

[Gog95] Healfdene Goguen. Typed operational semantics. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Proceedings of the 2nd International Conference on Typed Lambda Calculi and Applications (TLCA '95)*, number 902 in Lecture Notes in Computer Science, pages 186–200, London, UK, 1995. Springer. 1.1, 3.1

[Her95] Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1995. 5.3.1

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993. 1, 1.1, 2, 2.2

[Hin82] J. R. Hindley. The simple semantics for Coppo-Dezani-Sallé types. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 1982. 5.4.1

[HL93] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Proceedings of the 20th Annual Symposium on Principles of Programming Languages (POPL '93)*, pages 206–219, Charleston, SC, January 1993. ACM, ACM. 6.1.3

[HL07] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007. 2, 5, 3, 3.3.1

[HP05] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *Transactions on Computational Logic*, 6:61–101, January 2005. 3.2

[JM03] Felix Joachimski and Ralph Matthes. Short proofs of normalization. *Archive of Mathematical Logic*, 42(1):59–87, 2003. 1.1

[KT92] A. J. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order λ-calculus. *Information and Computation*, 98:228–257, 1992. 6.1.1, 6.1.1

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In Matthias Felleisen, editor, *Proceedings of the 34th Annual Symposium on Principles of Programming Languages (POPL '07)*, pages 173–184, Nice, France, January 2007. ACM Press. 1.1

[LRDR07] Luigi Liquori and Simona Ronchi Della Rocca. Intersection-types à la Church. *Information and Computation*, 205(9):1371–1386, 2007. 4.2

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991. 5.3.1

[MP91] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596. 1.1

[NGdV94] R. D. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors. *Selected Papers on Automath*. Number 133 in Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1994. C.3

[NPP07] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 2007. To appear. 3

[NvP01] Sara Negri and Jan von Plato. *Structural Proof Theory*. Cambridge University Press, 2001. 6.2.4

[OEI10] The on-line encyclopedia of integer sequences. Published electronically at `http://oeis.org`, 2010. 5

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991. 5.1

[Pfe92] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. Technical Report CMU-CS-92-186, Department of Computer Science, Carnegie Mellon University, September 1992. 1.1

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993. 1.1

[Pfe94] Frank Pfenning. A structural proof of cut elimination and its representation in a logical framework. Technical Report CMU-CS-94-218, Department of Computer Science, Carnegie Mellon University, November 1994. 6.2.1

[Pfe00] Frank Pfenning. Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141, 2000. 1.1, 1.1, 2

[Pfe01a] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, Massachusetts, June 2001. IEEE Computer Society Press. 4, 4.1

[Pfe01b] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001. 1.1

[Pie97] Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997. 5.4.1, 6.1.6

[Pie02] Benjamin C. Pierce. Type reconstruction. In *Types and Programming Languages*, chapter 22. MIT Press, 2002. 5.5.1, 5.5.1

[Pie10] Brigitte Pientka. An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. Submitted, August 2010. Available from `http://www.cs.mcgill.ca/~bpientka/papers/recon.pdf`. 5.1, 5.1, 5.2

[PR05] Franois Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005. 5.5.1, 5.5.1

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632. 1, 1.1, 5.1

[Ree09] Jason Reed. Higher-order constraint simplification in dependent type theory. In *Proceedings of the 4th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP '09)*, pages 49–56. ACM, 2009. 5.1

[Rey74] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag. 6.1.1

[Rey89] John C. Reynolds. Even normal forms can be hard to type. Unpublished, marked Carnegie Mellon University, December 1, 1989. 2.1

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700, Berlin, 1991. Springer-Verlag. 4

[Rey96] John C. Reynolds. Design of the programming language Forsythe. Report CMU–CS–96–146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 28, 1996. 2.1, 5.4.1, 6.1.6

[RP08] Jason Reed and Frank Pfenning. Proof irrelevance in a logical framework. Unpublished draft, July 2008. 4.1

[RS09] Florian Rabe and Carsten Schürmann. A practical module system for LF. In *Proceedings of the 4th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP '09)*, pages 40–48. ACM, 2009. 1

[Sar09] Susmit Sarkar. *A Dependently Typed Programming Language, with Applications to Foundational Certified Code Systems*. PhD thesis, Carnegie Mellon University, May 2009. Available as Technical Report CMU-CS-09-128. 1

[Sch03] Carsten Schürmann. Towards practical functional programming with logical frameworks. Unpublished, available at `http://cs-www.cs.yale.edu/homes/carsten/delphin/`, July 2003. 1

[SS88] Anne Salvesen and Jan M. Smith. The strength of the subset type in Martin-Löf's type theory. In *Proceedings of LICS'88*, pages 384–391. IEEE Computer Society Press, 1988. 4.1, 4.1

[vP01] Jan von Plato. Natural deduction with general elimination rules. *Archive for Mathematical Logic*, 40:541–567, 2001. 6.2.4

[WCPW02] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003. 1.1, 3, 3.3.1

[WCPW04] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, pages 355–377. Springer-Verlag LNCS 3085, 2004. Revised selected papers from the *Third International Workshop on Types for Proofs and Programs*, Torino, Italy, April 2003. 1.1

[YS91] Eyal Yardeni and Ehud Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–153, 1991. 2.5, 2.5