Contextual modal refinement types

Antoine Gaulin

School of Computer Science

McGill University

Montreal, Quebec, Canada

January 2024

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

Contents

Abstract

We develop an extension of the proof environment Beluga with datasort refinement types and study its impact on mechanized proofs. In particular, we introduce *refinement schemas*, which provide fine-grained classification for the structures of contexts and binders. Refinement schemas are helpful in concisely representing certain proofs that rely on relations between contexts. Our formulation of refinements combines the type checking and sort checking phases into one by viewing typing derivations as outputs of sorting derivations. This allows us to cleanly state and prove the conservativity of our extension.

Résumé

Nous développons une extension de l'environnement de preuve Beluga avec des types de rafinements, suivant l'approche des sortes de données, puis étudions l'impact sur les preuves mécanisées. En particulier, nous introduisons les *schèmes de rafinement*, qui permettent une classification fines des structures de contextes et des variables. Les schèmes de rafinement sont utiles pour représenter certaines preuves reposant sur les relations entre contextes. Nous suggérons une formulation du rafinement qui combine les phases de typage et de sortage et interprète les dérivations de typage comme un produit des dérivations de sortage. Cette formulation nous permet de spécifier et de prouver la conservativité de l'extension de façon concise et claire.

Acknowledgements

I would be nothing without the various people that surrounded me during the last few decades that have composed my life. All of you deserve as much credit for this work as I do, and so thanks are very much required. To those that I inevitably forget mentioning here, please accept my most sincere apologies and know that I appreciated your contribution.

First, I must thank my supervisor, Brigitte Pientka, for the formidable guidance that she provided along the way. Your expert's insights in the academic world have been invaluable to me, and certainly played a crucial role in any success that I've had (or that I will ever have, for that matter).

I am eternally grateful for the support, both material and emotional, of my close friends. I doubt any of you will ever read this, but if you do, you can hopefully recognize yourselves. I owe special thanks to those who encouraged (or, more accurately, forced) me to enroll in university.

Next, I thank my father for passing along to me his thirst for knowledge. Words cannot express just how important your influence makes a better person.

I am also grateful to many people who will likely never know me, especially the artists whose work has made life enjoyable. In particular, the chaotic music of Ornette Coleman has helped me structure my ideas tremendously.

Last, but not least, I wish to thank anyone who reads this work. After all, the main purpose of research is not to establish new results, but to communicate them.

List of Figures

List of abbreviations

CBV call-by-value

CMTT contextual modal type theory

FOL first-order logic

HOAS higher-order abstract syntax

OL object language

STLC simply-typed λ-calculus

Chapter 1

Introduction

Proof mechanization provides strong trust guarantees towards the validity of theorems. Contrary to informal proofs, expressing a theorem and its proof formally requires absolute precision. The resulting statement can thus become riddled with technicalities, which obscures their relation to their informal counterparts. This work combines two approaches to type systems that significantly reduce the added complexity from formalization, namely refinement types and higher-order abstract syntax (HOAS).

Datasort refinement types (Freeman and Pfenning, 1991; Freeman, 1994) provide ways to define subtypes (called *datasorts* or just *sorts*) by imposing constraints on the constructors of (inductive) types. Intuitively, a sort S refines a types A if it is defined by a subset of its constructors. The idea originated in the simply-typed setting, where refinements enhance the expressive power of the type system. Later, Lovas and Pfenning (2010); Lovas (2010) extended datasort refinements to the dependently-typed Edinburgh logical framework LF (Harper et al., 1993). They provide an equivalence between their system of refinements,

LFR, and another extension of LF with proof-irrelevance. An immediate conclusion here is that refinements do not increase the expressive power of dependently-typed calculi. Rather, Lovas (2010) observes that refinements may significantly reduce the verbosity of mechanized proofs, which is demonstrated through several case studies.

Beluga (Pientka and Dunfield, 2010) is a two-level programming language based on contextual modal type theory (CMTT) (Nanevski et al., 2008). It uses the Edinburgh logical framework LF (Harper et al., 1993) as a specification logic (data-level), with an intuitionistic first-order reasoning logic (computation-level). The data-level is embedded in the computation-level via a (contextual) box modality similar to the one in the modal logic S4. From a logical point of view, the formula $\Box A$ (read box A) expresses that A is true under no assumptions, i.e. in the empty context. The contextual box modality generalizes this idea to arbitrary contexts, yielding formulas of the form $[\Psi \vdash A]$ expressing that A holds in context Ψ. This allows us to represent LF objects (and types) together with a context in which they are meaningful. To handle this representation, LF contexts are restricted using a notion of *schema* that acts as classifiers of contexts, similarly to how types classify terms. In addition, LF substitutions are first-class objects of Beluga and they can be used to move objects from one context to another while preserving their meaningfulness. These features allow the expression of an object language (OL) using HOAS (Pfenning and Elliott, 1988) and provide several substitution lemmas for free in our mechanizations.

1.1 Contributions

Broadly, the main contribution of this thesis is the theoretical development of a refinement type system for Beluga. Specifically, we introduce the notion of refinement schemas, which allow expressing more precise properties of contexts, much like type refinements allow expressing more precise properties of objects. We also extend Lovas and Pfenning (2010)'s refinements for LF to contextual refinement types, which requires including a refinement relation on contexts as well as types. With refinement schemas and contextual refinement types in our hand, we can carry our refinement type system to Beluga's computation-level. This means mainly that we allow refinements to cross the box modality that separates the data- and computation-levels.

One of the objectives of refinement types is to provide multiple classifiers to objects, which is why we can view them as extrinsic properties. Likewise, a context can be classified with multiple refinement schemas, but only one type schema. To achieve this flexibility, we had to review our understanding of contexts. Specifically, the presence of annotations to variables in a context can restrict the schemas that classify that context. Our solution is to view schemas as specifications of how a context can be generated, similarly to how atomic types explain how terms may be generated correctly. In doing so, we can remove most (but not all) type information from a context, leaving a specification of contexts that relies (almost) exclusively on terms. Such contexts can more naturally be interpreted as objects existing independently of their classifiers, so that we maintain a view that refinements are extrinsic properties of objects.

Finally, we observe that the two-step process used by previous work on datasort refinements is redundant for our purposes. That is, performing type-checking before sort-checking is unnecessary (although it provides certain benefits). We establish this fact by defining refinement-level judgments that produce type-level derivations of their analogous type-level judgment. Then, we prove that whenever we have a refinement-level derivation, the type-level derivation that it outputs is correct in conventional Beluga.

The work presented here covers roughly the same material as Gaulin and Pientka (2023), albeit in more detail. In particular, we give here a complete definition of the language, as well as a proof that the extension with refinements is conservative. Moreover, we include subsorting, which was omitted from Gaulin and Pientka (2023) due to space considerations. Finally, we provide additional case studies that further exemplify the benefits of refinement types in proof environments.

1.2 Background and related work

Before we move on to a more formal definition of our language, we discuss some related work. We focus here on presenting an overview of the development surrounding refinement types. The previous work that went into the design of Beluga was also highly influential to the present thesis, and will be discussed thoroughly at a later point.

Refinements were originally introduced by Freeman and Pfenning (1991) as a way to validate totality of programs in the presence of non-exhaustive pattern-matching. Intuitively, a program containing non-exhaustive pattern-matching has unspecified behaviour on some of its potential inputs. Running the program on such an input would naturally cause a runtime

error. To avoid such undesirable consequences, a compiler might raise an error, thereby forcing the programmer to specify behaviour on all possible patterns. However, if no input matching the unspecified pattern is ever passed to the function, then the non-exhaustive pattern-matching never causes an error. Accordingly, a preferable solution would consist of ensuring that none of the missing patterns can ever be encountered.

An atomic type is generated by specifying new (unique) constants (i.e. names) for constructing objects and a way to use them (the type of the constant). Since the constants are always new, types provide unique syntactic classifiers of objects formed with constructors. In this sense, atomic types can be considered intrinsic properties of objects, an idea reinforced by the fact that we can synthesize a unique type for every neutral terms (and atomic types are only inhabited by neutral objects in a canonical form presentation). A similar idea applies to function spaces, provided that variables bound by λ-abstractions have a type annotation.

Sorts, on the other hand, can tell us more specific information about the order in which constructors are used in an expression. This is achieved by removing the burden that every constructor name be unique when declaring a new sort. Instead, a user defining a new sort must specify a previously defined type and use only a subset of its constructors. In addition, they may assign new sorts to the selected constructors. This provides the tools to isolate the fragment of objects of a given type whose structure satisfies some regularity condition. Freeman (1994) discusses how datasort definition corresponds to regular tree automata (Gécseg and Steinby, 2015), which generalize regular expressions to trees (instead of lists). Importantly, sorts characterize objects that exist independently of them, and are

therefore extrinsinc properties. We refer to Pfenning (2008) for a discussion of this two-layered approach to unify the intrinsic and extrinsic views of typing.

1.2.1 Datasort refinements

Our notion of refinement falls in the datasort tradition that was initiated by Freeman and Pfenning (1991); Freeman (1994) for MiniML, a monomorphic fragment of Standard ML's core language. Their system uses refinement type inference so that users don't need to provide annotations, but the inferred refinements are usually intersections with some undesired components. Davies (2005), who coined the term datasort, extended this work to the full Standard ML language (including modules). They gave up sort inference in favor of a bi-directional sort-checking algorithm, so that only the desired sorts are used by the compiler. Unfortunately, even sort-checking is untenable in the presence of intersections: the compiler needs to choose the correct branch of an intersection when synthesizing sorts for neutral expressions, making sort-checking PSPACE-hard (Reynolds, 1997).

Later, Lovas and Pfenning (2010); Lovas (2010) designed LFR, an extension of LF with datasort refinement types. They provide an equialence between LFR and another variant of LF, called LFI, that is equipped with a proof irrelevance modality. Intuitively, this means that a datasort S refining a type A can be interpreted as a type family indexed by objects of type A, say $\hat{S}\ M$ for all $M : A$, and such that any two objects of type $\hat{S}\ M$ are equal (i.e. the exact proof is irrelevant, we only care that it exists). Through several case studies, Lovas (2010) demonstrates that proofs in LFR are more concise and more intuitive than their corresponding LFI proofs. Thus, they conclude that datasorts are beneficial in the

setting of proof mechanization. The present thesis investigates the benefits of datasorts in the richer setting of Beluga.

More recenty, Jones and Ramsay (2021) used refinement types to validate termination in the presence of non-exhaustive pattern matching. Their notion of an *intensional* refinement is obtained by removing some of the constructors from a datatype, but the remaining constructors cannot be assigned new sorts. Instead, a constructor $\mathbf{c} : A$ selected for the sort $\mathbf{s} \sqsubset \mathbf{a}$ is assigned the sort S obtained by replacing every occurrence of $\mathbf{a}$ in A by $\mathbf{s}$. Moreover, the simplicity of intensional refinements allows Jones and Ramsay (2021) to provide a practical language featuring full type and refinement inference. In a sense, intensional refinements appear weaker than datasorts since users are not allowed to specify new sorts for constructors. In particular, they cannot control the order in which constructors are applied. On the other hand, Jones and Ramsay (2021) do not consider that the absence of base cases in a refinement leads to the refinement being empty. Rather, they view such refinements as describing infinite structures. For instance, the refinement of the type of lists that only maintains the list extension constructor is viewed as the refinement type of infinite lists instead, whereas it would be an empty sort in our system.

1.2.2 Index refinements and liquid types

In parallel to datasorts, another important form of refinement types, known as *index* refinements, was developped for the purpose of including dependent types in practical programming languages. These were first introduced by Xi (1998); Xi and Pfenning (1999), who designed a family of dependently-typed ML-style languages parameterized by an arbitrary

index domain C, called DML(C). Refinements are obtained by allowing quantification over the index domain, which intuitively corresponds to having a refinement relation $\Pi x{:}S.A \sqsubset A$, where $S \in C$. In this way, most difficulties of dependent types can be avoided, similarly to how we avoid them in Beluga's computation-level by restricting dependencies to data-level terms. Datasort refinements and index refinements were combined by Dunfield (2007), essentially yielding an extension of DML with intersections.

An important development of this approach came in the form of logically qualified (or liquid) types (Rondon et al., 2008), this time as an extension of OCaml. In this methodology, a refinement is expressed as $\{x : \tau \mid P(x)\}$, where τ is a type and P is a boolean-valued function over τ. The type τ can then be seen as the refinement $\{x : \tau \mid \texttt{true}\}$ and this allows combining typing and sorting into one judgment, much like we have done for datasort refinements. Liquid types offer two important advantages over datasorts: Firstly, they are written in a style that is reminiscent of common mathematical notations for defining sets, which allows the (mathematically inclined) programmer to refer to their previously acquired intuition. Secondly, the properties expressible by liquid types are all the first-order boolean formulas, while datasorts are limited to those formulas whose validity can be established by a regular-tree automaton. This second advantage, however, comes at a cost in complexity, since type-checking now requires the use of an SMT solver. SMT solvers perform extremely well in practice (despite their theoretical difficulties), so one could argue that the benefits outweigh the cost in performance. Indeed, liquid types have been integrated to several languages since their introduction by Rondon et al. (2008), such as C (Rondon et al., 2010), Haskell (Vazou, 2016), and Rust (Lehmann et al., 2023).

1.3 Structure of the thesis

The remainder of the thesis is separated as follows. We first present Beluga's type system in Chapter 2. For the most part, this chapter is a review of previous work, although we make some adjustments to the contexts and schemas of the language. We will also discuss a simple example to illustrate how one mechanizes an OL in Beluga. Next, Chapter 3 discusses the refinement system that we define on top of the type system. Here, we revisit our example from the previous chapter to see how refinements can help us improve the clarity of our mechanization, especially when it comes to theorem statements. We prove in Chapter 4 that the extension is conservative, meaning that any well-sorted program is also well-typed, despite not performing a type-checking phase. Chapter 5 provides two additional case studies that exemplify more aspects of our newly acquired power and show that the benefit of refinements are even stronger when mechanizing more complex properties. Finally, we conclude in Chapter 6.

All the work presented in this thesis was developed by myself, with advisory help from Brigitte Pientka along the way.

Chapter 2

Review of Beluga

Beluga (Pientka and Dunfield, 2010) is a dependently-typed functional programming language inspired by contextual modal type theory (CMTT) (Nanevski et al., 2008). It allows direct manipulation of higher-order abstract syntax (HOAS) representation of object languages (OLs). In particular, this means that the mechanized OL inherits the correct behaviour of substitutions from Beluga's specification language. This provides important benefits in concisely mechanizing the meta-theory of programming languages, since substitution properties tend to require long, technical proofs.

This chapter describes the structure of Beluga's type system and programs, with the goal of making our presentation of refinement types for Beluga more accessible. Beluga consists of two levels: a specification language (also called the data-level), and a computation language (also called the computation-level). The data-level is an extension of the Edinburgh logical framework LF (Harper et al., 1993) with contextual types, which we therefore call Contextual LF. The computation-level is an ML-style functional language in which types can

depend on Contextual LF objects, but not other computation-level objects. The ability to manipulate contextual objects allows pattern matching on open code, which further enhances the clarity of proofs. We will start by reviewing LF itself, before extending it to Contextual LF, and finally presenting the computation-level.

The formulation of Beluga has varied somewhat significantly over the years of its development. Here, we present a further variation closely inspired by the one of Pientka and Abel (2015), although we will discuss, in this chapter, how the definition has evolved over time. Throughout the presentation, we include a mechanization of the simply-typed λ-calculus and some of its meta-theory. Later, we will revisit this mechanization to see how it can be improved with refinement types.

2.1 LF

The Edinburgh logical framework LF (Harper et al., 1993) is a dependently-typed specification language specialized in representing formal systems, such as logics and typed λ-calculi. The key idea behind LF is its "judgments-as-types" principle, which identifies LF types with judgments of the mechanized OL. Well-typed LF objects then correspond to proofs of the OL judgments.

LF is separated into three levels, terms (or objects), types, and kinds. Intuitively, terms are programs, types are classifiers of terms, and kinds are classifiers of types. The presence of kinds ensures that LF types are predicative, meaning here that LF types may only depend on objects that do not inhabit them.

We adopt here a canonical form presentation (Watkins et al., 2002), although Harper

et al. (1993)'s original formulation of LF is more permissive in allowing reductions to occur. For our purposes, LF serves only as a specification language, so it is not too restrictive to require from our users that they write normal forms. Moreover, Harper et al. (1993) shows that LF is strongly normalizing, implying that any (well-typed) LF term has a unique normal form. Thus, in requiring normal forms only, we lose none of LF's power.

In order to handle canonical forms properly, we use a bi-directional typing algorithm and hereditary substitutions. As the name suggests, bi-directional typing consists of two phases: type checking and type synthesis. Type checking consists mainly of assigning a (given) type to each variable of a function. Then, we use this typing information to synthesize the type of the function's body, and we verify that the synthesized type coincides with the type we are checking against. Hereditary substitutions are similar to ordinary substitutions, but differ in that they sometimes perform additional reduction steps to ensure that the resulting term is still in canonical form. For instance, $[(\lambda y.y)/x](x\ M)$ produces M instead of $(\lambda y.y)\ M$, which is not in normal form.

One advantage of using a bi-directional type-checking algorithm is that terms themselves do not contain type annotations. Rather, users annotate a program with its type on the outside. This means that types never appear directly within terms, which is compatible with an extrinsic view of typing. Extrinsic typing has limited impact here, but it will play a more important role when we introduce refinement types later on.

2.1.1 Syntax

We begin with a discussion of LF's syntax, starting with terms. We differ from Harper et al. (1993) by using a head and spine syntax. Given the simplicity of LF, this leads to a somewhat verbose presentation. However, the use of heads and spines facilitates extending the language of terms since we only need to add new heads. With this in mind, we define the syntax of terms as follows :

$$\begin{array}{lrl} \text{Head} & H ::= & \mathbf{c} \mid x \\ \text{Spine} & \vec{M} ::= & \texttt{nil} \mid M; \vec{M} \\ \text{Neutral term} & R ::= & H\ \vec{M} \\ \text{Normal term} & M ::= & R \mid \lambda x.M \end{array}$$

The head $\mathbf{c}$ represents constants, while x represents variables. All constants have been defined by the user, and are stored in a globally accessible signature along with their user-specified type. A spine $\vec{M}$ is simply a list of normal terms. We note that neither heads nor spines are valid terms on their own.

To construct a term, we first apply a head H to a spine $\vec{M}$, yielding the neutral term $H\ \vec{M}$. We will write simply H when $\vec{M} = \texttt{nil}$ is the empty spine. Intuitively, heads are functions that cannot be evaluated when applied to arguments: $\mathbf{c}$ is a constructor so it is generative, while x must be substituted by an actual function before we can evaluate it. Spines are used to pass multiple arguments to a head at once.

A normal term is either a neutral term of a function abstraction $\lambda x.M$. The fact that applications may only be used in the neutral phase ensures that a function $\lambda x.M$ is never

directly applied to arguments. Since heads do not represent concrete computations, the separation of terms into neutral and normal ensures that no evaluation is possible.

Now, our syntax of terms encodes an untyped λ-calculus enhanced with constants. Since we only describe normal forms, none of the terms can actually run, but we can still express functions that would run forever if they were to be applied in the wrong ways. For instance, the term $\lambda x.x\ x$ describes a function that applies its only input to itself. If we were to apply this function to itself, we would obtain the self-reproducing term $(\lambda x.x\ x)\ (\lambda x.x\ x)$, which does not normalize. The problem with these terms is that the variable x is used as a both function and its argument, which introduces a form of impredicativity.

Types allow us to reject the bad terms described above. In LF, types are allowed to depend on terms (hence we say that it is dependently-typed). A type depending on a term M corresponds to a property of M expressed in first-order logic (FOL) and classifies those terms that can be seen as proofs of that property. More generally, a type depending on terms $M_1, ..., M_n$ represents a relation between those terms and is inhabited by proofs of that relation. In other words, LF allows a Curry-Howard isomorphism with (intuitionistic) FOL, according to which types are propositions and terms are proofs. Types are defined by the following syntax:

$$\begin{array}{lll} \text{Atomic types} & P ::= & \mathbf{a}\ \vec{M} \\ \text{Canonical types} & A ::= & P \mid \Pi x{:}A_1.A_2 \end{array}$$

The atomic types $\mathbf{a}\ \vec{M}$ consist of a type constant $\mathbf{a}$ applied to a spine. Like for terms, type constants are stored in the signature and must be assigned a kind K at declaration. The spine $\vec{M}$ must be made of terms that match the corresponding types in K and we require

that all type families be fully applied. We will write only **a** for **a** `nil`.

Canonical types are either atomic or dependent function spaces $\Pi x{:}A_1.A_2$ from A_1 to A_2, where x may occur in A_2. We adopt the common convention of writing $A_1 \to A_2$ instead of $\Pi x{:}A_1.A_2$ when x does not occur in A_2, and we call $A_1 \to A_2$ a simple function space.

Next, the fact that types are allowed to depend on terms introduces the possibility that some types are ill-formed. For instance, if we have a type **even** expressing that a number is even, then **even** M is only meaningful when M is a number. To handle this, we need a notion of *kinds*. Intuitively, a kind classifies types based on the objects to which they can be applied, similarly to how types classify terms based on the arguments to which they can be applied. They are given by the following syntax:

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

The kind `type` classifies all the well-formed types, that is those with no unspecified dependencies. Every other kind has the form $\Pi x_1{:}A_1.\Pi x_2{:}A_2.\cdots\Pi x_n{:}A_n.\texttt{type}$ and classifies those types with dependencies on objects of types $A_1, A_2, ...A_n$. For our purposes, only unapplied type constants **a** can have a kind different than `type`.

Next, we need a way to keep track of bound variables as we traverse open terms, which we do using contexts. For now, we view contexts simply as lists of variables together with their types. We thus obtain the following syntax for contexts:

$$\text{Context} \quad \Gamma ::= \quad \cdot \mid \Gamma, x{:}A$$

where $\cdot$ is the empty context, and $\Gamma, x{:}A$ extends the context Γ with a fresh (i.e. new) variable x of type A.

An LF program consists of a sequence of declarations of either constant objects or atomic type families. When declared, every constant object must be assigned a type and every atomic type must be assigned a kind. Each declaration is stored in a globally accessible signature, denoted Sig (Σ is also commonly used, but we reserve it for later).

In our work, constant declarations are bound within type declarations. This is because constants are really constructors for atomic types, so they can only be defined while defining the atomic type. In this case, there is only one form of LF declaration for us, given by the following syntax :

$$\text{Declaration} \quad D ::= \quad (\text{LF } \mathbf{a} : K = \mathbf{c}_1 : A_1 \mid \mathbf{c}_2 : A_2 \mid ... \mid \mathbf{c}_n : A_n)$$

$$\text{Signature} \quad \text{Sig} ::= \quad \cdot \mid \text{Sig}, D$$

For simplicity, this thesis treats only of sequential declarations. That is, a declaration may only reference previously declared constants and types. Mutual definitions can be handled by including subordination into our judgments (Virga, 1999).

Example: Simply typed λ-calculus

We demonstrate the usefulness of LF as a specification language by encoding a simply-typed λ-calculus (STLC) with natural numbers. Every snippet of code needed in the mechanization is accompanied by an informal definition of what is encoded, so as to show how closely related the formal and informal versions are. We begin by encoding the types of STLC:

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

STLC Types $T ::=$

$\mathbb{N}$

$\mid T_1 \to T_2$

On the left, we see the syntax of an LF declaration of the type `tp`. It is defined by the two constructors `nat` and `arr`. The type of `nat` tells that this constructor is already a `tp`. On the other hand, the type of `arr` indicates that this constructor must be given two arguments of type `tp` in order to produce a new object of type `tp`. If we contrast this formal definition with the informal syntax of STLC types on the right, we see that $\mathbb{N}$ is indeed a type on its own, while the function types $T_1 \to T_2$ necessitate two previously constructed types T_1 and T_2. In other words, the formal and informal definitions of STLC types can clearly be seen to encode the same concept.

Next, we encode the terms of STLC:

```
LF tm : type =
| zero : tm
| succ : tm
| lam : tp → (tm → tm) → tm
| app : tm → tm → tm;
```

STLC Terms $e ::= x \mid 0 \mid \mathbf{S}\ e \mid \lambda x{:}T.e \mid e_1\ e_2$

Here, the correspondence between formal and informal encoding is a little less clear. In particular, the informal syntax includes variables x, but the formal definition does not have a constructor for variables. Instead, the fact that `tm` includes variables is hidden in the negative occurrence of `tm` in the type of the constructor `lam`.

We note that negative occurrences do not cause trouble in LF because of its weak function space. With strong functions spaces, the type `(tm → tm)` would be strictly larger than `tm`, which creates a paradox since each such function may be used to create a `tm` (implying that `tm` is at least as large as `(tm → tm)`). With LF's weak function spaces, the negative

occurrence of `tm` in `(tm → tm)` is simply represented as an LF variable of type `tm` that may occur within the positive occurrence of `tm`, so both types are countably infinite. This is also in accordance with the principles of HOAS, whereby variables of the OL are represented as variables of the meta-language (in this case, LF). HOAS then also permits the use of meta-language substitutions for representing OL substitutions through function applications. This provides us with the necessary tools to elegantly encode the typing judgment :

$\boxed{\Gamma \vdash M : A}$ – STLC typing

```
LF oft : tm → tp → type =

| t-zero : oft zero nat

| t-succ : oft e nat →
            oft (succ e) nat

| t-lam : ({x:tm} oft x T → oft (e x) T') →
           oft (lam T e) (arr T T')

| t-app : oft e1 (arr T' T) → oft e2 T' →
           oft (app e1 e2) T
```

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}}$$

$$\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \mathbf{S}\ e : \mathbb{N}}$$

$$\frac{\Gamma, x{:}T \vdash e : T'}{\Gamma \vdash \lambda x{:}T.e : T \to T'}$$

$$\frac{\Gamma \vdash e_1 : T' \to T \quad \Gamma \vdash e_2 : T'}{\Gamma \vdash e_1\ e_2 : T}$$

The most interesting aspect of this definition is the encoding of the rule for λ-abstractions. We use the curly braces `{x:tm}` to denote explicit universal quantification over `tm`. Notice in particular how we represent the informal context extension $\Gamma, x{:}T$ using the negative occurrences of `tm` and `oft x T`. Notice also how the dependency of the function body e on the variable x is represented using a function application in `oft (e x) T'`.

$\boxed{\Gamma \vdash H \Rightarrow A}$ – Synthesize type A for head H

$$\frac{(\mathbf{c}{:}A) \in \text{Sig}}{\Gamma \vdash \mathbf{c} \Rightarrow A} \qquad \frac{(x{:}A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}$$

$\boxed{\Gamma \vdash \vec{M} : A > P}$ – Apply A to $\vec{M}$, resulting in P

$$\frac{}{\Gamma \vdash \texttt{nil} : P > P} \qquad \frac{\Gamma \vdash M \Leftarrow A_1 \quad \Gamma \vdash \vec{M} : [M/x]A_2 > P}{\Gamma \vdash M; \vec{M} : \Pi x{:}A_1.A_2 > P}$$

$\boxed{\Gamma \vdash R \Rightarrow A}$ – Synthesize type A for neutral term A

$$\frac{\Gamma \vdash H \Rightarrow A' \quad \Gamma \vdash \vec{M} : A' > A}{\Gamma \vdash H\ \vec{M} \Rightarrow A}$$

$\boxed{\Gamma \vdash M \Leftarrow A}$ – Check M against A

$$\frac{\Gamma \vdash R \Rightarrow P}{\Gamma \vdash R \Leftarrow P} \qquad \frac{\Gamma, x{:}A_1 \vdash M \Leftarrow A_2}{\Gamma \vdash \lambda x.M \Leftarrow \Pi x{:}A_1.A_2}$$

Figure 2.1: LF typing rules

2.1.2 Judgments

Now that we have defined and exemplified the syntax of LF, we can discuss its judgments. We focus here on the typing judgments, although, for completeness, we also need a kinding judgment and a context formation judgment. We leave out kinding from this discussion as it is straightforward to define it, but we provide its definition in Appendix A. We will discuss context formation in depth later on, but for now we think of a context Γ as well-formed if it contains no duplicate variable and every type in Γ is well-formed (i.e. has kind `type`).

We use a bi-directional typing algorithm. This means that we have two main judgments:

type synthesis for neutral terms, denoted $\Gamma \vdash R \Rightarrow P$, and type checking for normal terms, denoted $\Gamma \vdash M \Leftarrow A$. Note that synthesis is restricted to atomic type, which enforces that terms are η-long. As the names suggest, the type A is an output of type synthesis judgment and an input of the type checking judgment. From a proof-theoretic perspective, the synthesis phase corresponds to using only elimination rules, while the checking phase uses only introduction rules. Here, we only have functions, so elimination is application and introduction is λ-abstraction, but the idea generalizes to richer type systems.

In addition, we need a type synthesis judgment for heads, denoted $\Gamma \vdash H \Rightarrow A$, and a type checking judgments for spines, denoted $\Gamma \vdash \vec{M} : A' > A$. Synthesis for heads is simply a lookup in the signature (for constants) or context (for variables). Type checking for spines takes in a spine $\vec{M} = M_1; ...; M_n$ and a type $A' = \Pi x_1{:}A_1 ... \Pi x_n{:}A_n.A''$, then validates that $\Gamma \vdash M_i \Leftarrow [M_1/x_1, ..., M_{i-1}/x_{i-1}]A_i$ for each i, and finally produces the output $A = [M_1/x_1, ..., M_n/x_n]A''$.

These judgments should be read bottom-up. This means that we start by a type-checking phase, during which we peel off any λ-abstraction that occurs and extend the context with the new variables. Once we reach a function application, we attempt to synthesize its type by first synthesizing the type of the head, and then verifying that each argument in the spine checks against the expected type. Once we have synthesized a type for the application, we compare it (syntactically) with the type that we were checking against. A type-checking derivation succeeds only when the comparison yields that the two types are indeed equal.

Generally, equality of dependent types reduces to equality between terms. Since our language only allows normal terms, equality of terms is simply syntactic equality (modulo

α-renaming).

Example: Comparing LF and OL derivations

To finish our presentation, let us see how an informal typing derivation in STLC compares to its formal LF derivation. Consider the STLC function $\lambda x{:}\mathbb{N}.\mathbf{S}\ x$ that increments a natural number. Clearly, this function has type $\mathbb{N} \to \mathbb{N}$, which we can indeed validate with the following proof:

$$\dfrac{\dfrac{\dfrac{(x{:}\mathbb{N}) \in (x{:}\mathbb{N})}{x{:}\mathbb{N} \vdash x : \mathbb{N}}}{x{:}\mathbb{N} \vdash (\mathbf{S}\ x) : \mathbb{N}}}{\vdash (\lambda x{:}\mathbb{N}.\mathbf{S}\ x) : (\mathbb{N} \to \mathbb{N})}$$

In contrast, the corresponding encoding of the function $\lambda x{:}\mathbb{N}.\mathbf{S}\ x$ is the LF object `lam nat (λx. succ x)`, and the type $\mathbb{N} \to \mathbb{N}$ translates to `arr nat nat`. We then expect the LF type `oft (lam nat (λx. succ x)) (arr nat nat)` to be inhabited by a proof similar to the above. It is not difficult to work out that the LF object `t-lam (λx.λt. t-succ t)` is the one we want, as demonstrated by the following LF derivation :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\texttt{t:oft x nat}) \in (\texttt{x:tm, t:oft x nat})}{\texttt{x:tm, t:oft x nat} \vdash \texttt{t : (oft x nat)}}}{\texttt{x:tm, t:oft x nat} \vdash \texttt{(t-succ t) : (oft (succ x) nat)}}}{\texttt{x:tm} \vdash \texttt{(}\lambda\texttt{t.t-succ t) : }\Pi\texttt{t:oft x nat.oft (lam nat (}\lambda\texttt{x.succ x)) (arr nat nat)}}}{\vdash \texttt{(}\lambda\texttt{x.}\lambda\texttt{t.t-succ t) : (}\Pi\texttt{x:tm.}\Pi\texttt{t:oft x nat.oft (lam nat (}\lambda\texttt{x.succ x)) (arr nat nat))}}}{\vdash \texttt{(t-lam (}\lambda\texttt{x.}\lambda\texttt{t. t-succ t)) : (oft (lam nat (}\lambda\texttt{x. succ x)) (arr nat nat))}}$$

Now, we can clearly see that the informal STLC typing derivation and its formal LF counterpart have roughly the same shape, although the formal proof has a few extra steps. Specifically, there are two extra steps needed to perform the context extension properly.

In particular, notice how we end up with the context `x:tm, t:oft x nat` to represent the informal assumption x:$\mathbb{N}$. This is inevitable since x:$\mathbb{N}$ introduces both the term variable x and the assumption that it has type $\mathbb{N}$.

Generally speaking, informal presentation of typed λ-calculi may allow various ways to extend a context with new bindings. These bindings can have complex structures involving several assumptions and constraints on how these assumptions are made. For instance, in a polymorphic language, we may have type variables of any kind, but only allow term variables to have a type of kind `type` (see Chapter 5 for more details).

The central idea behind our HOAS consists of representing OL variables as LF variables, so it is crucial that we represent the binders (and other components of the OL) correctly. The LF methodology for ensuring this is to prove adequacy theorems, which means establishing a *compositional bijection* between a rigorous, informal presentation of an OL and its formal encoding in LF (Harper et al., 1993). Here, bijection means that for every object of interest (terms, types, contexts, judgments, etc.) in the informal presentation, there corresponds an LF object in the formal encoding, and vice versa. Compositionality means that the bijection commutes with substitutions, or, intuitively, that applying an OL substitution to an OL term is the same as applying the corresponding LF substitution to the corresponding LF term. This compositionality criterion is what guarantees that our mechanization correctly represents the variables, contexts, and substitutions of the OL. In our example, we have given this compositional bijection informally by presenting the informal and formal representations side by side. While adequacy is an important aspect of mechanization, we omit further discussion of it, opting instead for these informal side-by-side comparisons. We refer to

Pfenning (1997); Harper and Licata (2007) for more details on adequacy.

In short, the correspondence between an OL context and the LF context representing it is not always obvious. Since our HOAS is based around the principle that OL variables can be represented as LF variables, it follows that OL contexts can be represented as LF contexts. However, as we have just seen, an LF context must have a particular structure in order to adequately encode an OL context. In order to properly reason about the meta-theoretic properties of an OL, it is imperative that this structure be preserved as we traverse terms under binders and extend the contexts. Therefore, we need additional tools to enforce the particular structure of contexts and facilitate their manipulation. To achieve this, we will extend LF to Contextual LF.

2.2 Contextual LF

Contextual LF extends LF with contextual types. Simply, a contextual type $\Gamma \vdash A$ consists of a type A together with a context Γ containing all of the free variables in A. An object of type $\Gamma \vdash A$ has the form $\Gamma \vdash M$ and must satisfy $\Gamma \vdash M \Leftarrow A$. The key advantage of this approach lies in the fact that $\Gamma \vdash \Pi x{:}A.A'$ is isomorphic to $\Gamma, x{:}A \vdash A'$, in the sense that one can transform any derivation of $\Gamma \vdash \Pi x{:}A.A'$ into a derivation of $\Gamma, x{:}A \vdash A'$ and vice versa. This allows us to express traversals under binders directly through context extensions.

In addition, we revise the notion of LF contexts to improve their ability to represent OL contexts. In particular, we add the notion of a *schema* to classify contexts based on their structures, thus enforcing the correct representation of OL binders. Moreover, we extend the language with an explicit substitution calculus that allows moving an object from one

context to another while preserving its meaningfulness.

2.2.1 Revision of contexts

As illustrated in our last example, representing the binding structures of an OL may require multiple LF variables. To ensure the adequacy of our representations, we must specify how to extend an LF context so that it actually corresponds to an OL context. To handle this, Felty et al. (2015a) suggests structuring contexts into lists of tuples of variables instead of flat lists of variables. With this additional structure, we can more accurately characterize the assumptions contained within a context. We achieve this by adding schemas to the language.

Schemas originated in the work of Schürmann (2000) as classifiers of LF contexts. They were first included to Twelf (Pfenning and Schürmann, 1999), a proof environment that implements LF. In this work, a schema element (or block schema in their terminology) is a parameterized record type, and a context schema is a collection of schema elements. Intuitively, the record of a schema element contains all the LF variables necessary to encode one OL assumption, and it is parameterized with the premises of the OL's context formation rules, in accordance with the judgment-as-types principle. Then, a context schema is the list of all possible ways to introduce an OL assumption.

Beluga has had a notion of schemas since its beginning (Pientka, 2008), although it differs from Schürmann (2000) in that schema elements are just types. However, the language presented by Pientka (2008) already contains tuples and it is simply-typed, so there is no need for parameterizing them. In this sense, the difference from Schürmann (2000) is superficial. Soon after, the dependently-typed formulation of Beluga by Pientka and Dunfield (2008)

returns to a notion of schema in line with Schürmann (2000). Since records are not valid LF types, variables with record types may only be accessed through projections.

Later formulations by Cave and Pientka (2012, 2013); Pientka and Abel (2015) restrict the records of schema elements to be single types. The removal of records simplifies the treatment of contexts since there are no more projections to handle. However, we know that this is insufficient to properly describe an OL context.

Here, we return again to a notion of schema with records. Moreover, we consider schemas with a generative flavour: a schema specifies how a context can be constructed. In this sense, we think of schemas as the atomic types of contexts, where schema elements correspond to constructors. An important difference with atomic types is that schema elements only tell us how to extend a context of the same schema. In a sense, every schema element can be thought of as having an implicit argument corresponding to the context that is being extended.

We now define contexts and schemas with the following syntax:

Blocks of declarations	B	$::= \quad \cdot \mid \Sigma x{:}A.B$
Schema elements	E	$::= \quad B \mid \Pi x{:}A.E$
Context schemas	G	$::= \quad \cdot \mid G + \mathbf{w}{:}E$
Contexts	Γ	$::= \quad \cdot \mid \psi \mid \Gamma, x{:}A \mid \Gamma, b{:}\mathbf{w} \cdot \vec{M}$
Heads	H	$::= \quad ... \mid b.i$

Blocks of declarations represent tuples of labelled assumptions, i.e. variables. The empty block $\cdot$ is not valid on its own, rather it is a syntactic device that indicates the end of a block. Empty blocks are not strictly necessary for the system to work, but facilitate the theoretical

development by providing a simple base case.

A schema element is a parameterized block of declarations. While blocks express specific instances of assumptions, schema elements encode the general requirements of a particular form of assumption. Note that every block of declarations is also a schema element that relies on no additional parameters.

A schema is given as a list of named schema elements, and we write $\mathbf{w}$ to indicate these names. Schema elements can only be defined within schema declarations, much like constants are defined within type declarations. Consequently, every schema element used in a mechanization must have a name. The empty schema $\cdot$ describes the collection of contexts that can be formed without ever adding assumptions, i.e. only the empty context. Like for blocks, the interest of an empty schema lies mainly in simplifying the meta-theory of our system, and so it could be omitted entirely.

Finally, contexts are now allowed to contain blocks of declarations, denoted $b{:}\mathbf{w} \cdot \vec{M}$. If $\mathbf{w} : \Pi(\overrightarrow{x{:}A}).B$, then $\mathbf{w} \cdot \vec{M}$ corresponds to the concrete block of declarations $[\vec{M}/\vec{x}]/B$. Since Σ-types are not proper LF types, block variables can only be used in LF objects through projections, denoted $b.i$. Intuitively, this projection corresponds to a single variable within the block, hence we extend our syntax of heads with $b.i$. We also need to add an inference rule to the type synthesis for head judgment:

$$\frac{(b : \mathbf{w} \cdot \vec{M}) \in \Gamma \quad \Delta; \Gamma \vdash \mathbf{w} \cdot \vec{M} > B \quad \Delta; \Gamma \vdash b : B \gg_1^i A}{\Delta; \Gamma \vdash b.i \Leftarrow A}$$

where the judgment $\Delta; \Gamma \vdash \mathbf{w} \cdot \vec{M} > B$ computes the concrete block of declarations B represented by $\mathbf{w} \cdot \vec{M}$, and the judgment $\Delta; \Gamma \vdash b : B \gg_1^i A$ extracts the type A of the i^{th} component of B. We omit the definition of these judgments here as they are straightforward,

but provide them in Appendix A.

In addition, we allow LF contexts to start with a context variable ψ. We stress that although ψ may occur within an LF context Γ, it is not itself an LF variable, but rather a Contextual LF variable. As such, ψ must be bound outside of Γ. To handle this issue, we use a meta-context Δ that is allowed to contain, among other things, our context variables. We will discuss meta-contexts in details when we formally introduce contextual types and contextual objects. For now, it suffices to know that Δ can contain assumptions $\psi{:}G$ that context variable ψ has schema G.

Note that, while arbitrary single assumptions $x{:}A$ may still appear within an LF context, the type A is not described by the syntax of schema elements. However, the type A is essentially the same as the block of declarations (and therefore schema element) $\Sigma x{:}A.\cdot$. In this sense, the single assumptions are not strictly necessary for mechanizations themselves. However, several of our judgments (in particular, block and schema element well-formedness) rely on adding single assumptions to LF contexts, so we at least need them for our current definition of the language.

Now, let us go over the schema-checking judgment $\Delta \vdash \Gamma : G$ (see Figure 2.2). First, the rule **SC-empty** tells us that the empty context inhabits every schema. The other base case is the rule **SC-var**, that tells us that a context variable has the schema that it was declared to have.

The rule **SC-ext** is used to validate schemas of contexts containing actual variables. It is restricted to block variables $b{:}\mathbf{w} \cdot \vec{M}$, where $\mathbf{w} : E$ is defined in the schema that we are checking against. To validate that a context extension is correct, we simply need to check

$\boxed{\Delta \vdash \Gamma : G}$ – LF context has schema G in meta-context Δ

$$\frac{\Delta \vdash G : \texttt{schema}}{\Delta \vdash \cdot : G}\ \textbf{SC-empty} \qquad \frac{(\psi : G) \in \Delta}{\Delta \vdash \psi : G}\ \textbf{SC-var}$$

$$\frac{\Delta \vdash \Gamma : G \quad (\mathbf{w}{:}E) \in G \quad \Delta; \Gamma \vdash \vec{M} : E > B}{\Delta \vdash (\Gamma, b{:}\mathbf{w} \cdot \vec{M}) : G}\ \textbf{SC-ext}$$

$\boxed{\Delta; \Gamma \vdash \vec{M} : E > B}$ – Instantiate schema element E with parameters $\vec{M}$, yielding B.

$$\frac{}{\Delta; \Gamma \vdash \texttt{nil} : B > B}\ \textbf{Inst-block} \qquad \frac{\Delta; \Gamma \vdash M \Leftarrow A \quad \Delta; \Gamma \vdash \vec{M} : [M/x]E > B}{\Delta; \Gamma \vdash (M; \vec{M}) : (\Pi x{:}A.E) > B}\ \textbf{Inst-pi}$$

Figure 2.2: Schema-checking rules

the terms of $\vec{M}$ against the type parameters dictated by the schema element E. We perform this verification with the auxiliary judgment $\Delta; \Gamma \vdash \vec{M} : E > B$, which incidentally generates the concrete block of declarations B. Although we do not need to generate the actual blocks for schema-checking, we will need to do so in order to derive the types of variables later on and we can reuse the same judgment. In practice, we would use two separate judgments to avoid type-checking $\vec{M}$ every time a block variable is used, which would significantly impact performance.

Example: Terms of STLC

When we defined the type `tm` of terms in our OL, we assigned the type `(tm → tm) → tm` to the constructor `lam`, and said that the negative occurrence of `tm` posed no problem as it would simply be represented as an LF variable in the ambient context. Now, we want to make the LF context explicit and use it to represent the OL context, so we need to specify a

schema that classifies those LF context that correspond to and OL context. In this setting, we only need a variable of type `tm`, so we define the schema as follows:

```
LF lam-ctx : schema =
| lc-var : block (x:tm);
```

This new explicit syntax of schema declarations is purposefully similar to the one of atomic type declarations. It differs only in two ways: First, the kind is replaced with `schema`, which we think of as analogous to the kind `type`. Second, the constructors are assigned schema elements instead of types. The keyword `block` indicates which fresh variables should be included in the new block of declarations.

Now, in the typing judgment `oft`, the rule for λ-abstraction is encoded through the constructor (`t-lam : ({x:tm} oft x A → oft (M x) B) → oft (lam A M) (arr A B)`), which contains negative occurrences of both `tm` and `oft`. In this case, adequately representing the contexts of STLC requires extending the LF context with two variables. Thus, our schema `lam-ctx`, despite accurately representing all terms of the OL, is ill-suited to representing also its typing derivations. We can solve this issue with a more sophisticated context schema:

```
LF stlc-ctx : schema =
| typ-var : some [A:tp] block (x:tm, t:oft x A);
```

The keyword `some` corresponds to the Π of schema elements, and all the parameters are introduced within the square brackets (in this case, there is only `A:tp`). Then, when provided with a particular `A`, `typ-var` produces the block of declarations `(x:tm, t:oft x A)` which corresponds to the negative occurrences in the type of `t-lam`. We denote the extension of

a context (Ψ : `stlc-ctx`) as (Ψ, `b:typ-var A`), and access the variables contained in the block with the projections (`b.1` to access `x` and `b.2` for `t`).

2.2.2 Substitutions

Next, we address how the new structure of contexts impacts substitutions. Since we are using a canonical forms presentation, we need to use hereditary substitutions as well (Watkins et al., 2002). This means that substitutions must perform certain reduction steps to ensure that the resulting object remains in normal form. Essentially, whenever a variable occurs in a head position and is substituted for a λ-abstraction, the hereditary substitution will apply β-reduction. For instance, $[(\lambda x.x)/x'](x'\ M)$ would ordinarily yield $(\lambda x.x)\ M$, which is not normal, but the hereditary substitution will further reduce this into M.

So far, every substitution that we have used included an explicit specification of the substituted variables. Our most recent one, $[(\lambda x.x)/x']$, substitutes only the variable x'. In what follows, we instead require that substitutions have the same shape as their context domain. Now, let us look at the syntax of substitutions:

$$\text{Substitutions} \quad \sigma ::= \quad \cdot \mid \mathtt{id}_\psi \mid s[\sigma] \mid \sigma, M \mid \sigma, \vec{M}$$

We distinguish five forms of substitutions and write $\Delta; \Gamma' \vdash \sigma : \Gamma$ to indicate that σ has domain Γ and co-domain Γ'. We will now discuss this judgment, whose formal definition is given in Figure 2.3.

The empty substitution $\cdot$ has the empty context as a domain and can have any other context as a co-domain. The substitution $\mathtt{id}_\psi$ is the identity for the context variable ψ, so it has ψ for a domain.

$\boxed{\Delta; \Gamma' \vdash \sigma : \Gamma}$ – σ is a well-formed substitution with domain Γ and co-domain Γ'.

$$\frac{\Delta \vdash \Gamma_1 : \mathtt{ctx}}{\Delta; \Gamma_1 \vdash \cdot : \cdot}\ \textbf{Subst-empty} \qquad \frac{(\psi : G) \in \Delta \quad \Delta \vdash \Gamma_1 : \mathtt{ctx}}{\Delta; \Gamma_1 \vdash \mathtt{id}_\psi : \psi}\ \textbf{Subst-id}$$

$$\frac{(s{:}\Gamma_1 \vdash \Gamma_2) \in \Delta \quad \Delta; \Gamma \vdash \sigma : \Gamma_1}{\Delta; \Gamma \vdash s[\sigma] : \Gamma_2}\ \textbf{Subst-var} \qquad \frac{\Delta; \Gamma_1 \vdash \sigma : \Gamma_2 \quad \Delta; \Gamma_1 \vdash M \Leftarrow [\sigma]A}{\Delta; \Gamma_1 \vdash (\sigma, M) : (\Gamma_2, x{:}A)}\ \textbf{Subst-tm}$$

$$\frac{\Delta; \Gamma_1 \vdash \sigma : \Gamma_2 \quad \Delta; \Gamma_2 \vdash \mathbf{w} \cdot \vec{M}' > D \quad \Delta; \Gamma_1 \vdash \vec{M} \Leftarrow [\sigma]D}{\Delta; \Gamma_1 \vdash (\sigma, \vec{M}) : (\Gamma_2, b{:}\mathbf{w} \cdot \vec{M}')}\ \textbf{Subst-spn}$$

Figure 2.3: Explicit substitution calculus

Substitution variables $s : \Gamma_1 \vdash \Gamma_2$ are located in the meta-context and have domain Γ_2 and co-domain Γ_1. These can only be used when paired with a delayed substitution σ, which we indicate by writing it on the right of s rather than on its left. The delayed substitution is necessary to ensure that substitution variables have the correct co-domain.

Next, (σ, M) has domain $(\Gamma, x{:}A)$ and co-domain Γ' provided that σ has domain Γ and codomain Γ', and $\Gamma' \vdash M \Leftarrow [\sigma]A$. Notice that the terms composing a substitution must be meaningful in its co-domain. These four forms were part of previous formulations of Beluga and have been thoroughly discussed in the past (see Pientka (2008); Cave and Pientka (2013) for instance).

Here, we add a fifth form of substitutions, $(\sigma, \vec{M})$, that allows users to extend a substitution with several objects at once, so that we can substitute blocks more easily. More precisely, the spine $\vec{M}$ (which we really think of more as an n-ary tuple in this case) consists

of terms that match against each variable in a block $b{:}\mathbf{w} \cdot \vec{M}'$. If we simply substitute $\vec{M}$ for b in an LF object, we can obtain expressions of the form $\vec{M}.i$ since b may only occur within projections. This poses a problem since $\vec{M}.i$ is not in normal form, which breaks the correct behaviour of hereditary substitutions. Fortunately, we can extend hereditary substitutions to compute the actual projections of the spine, simply by stating that:

$$[\vec{M}/b](b.i) = M_i \qquad \text{if } \vec{M} = M_1; ...; M_n \text{ and } 1 \leq i \leq n$$

$$[\vec{M}/b](b.i) \quad \text{fails} \qquad \text{otherwise}$$

For other objects, we define $[\vec{M}/b](M')$ via the usual congruence rules, with the base cases $[\vec{M}/b](x) = x$ and $[\vec{M}/b](\mathbf{c}) = \mathbf{c}$.

2.2.3 Contextual objects

As mentioned above, a contextual type $\Gamma \vdash A$ consists of a type A together with a context Γ containing all the free variables occurring in A. We also discussed how a contextual object $\Gamma \vdash M$ has type $\Gamma \vdash A$ if $\Gamma \vdash M \Leftarrow A$. If $x{:}A \in \Gamma$, then $\Gamma \vdash x \Leftarrow A$, so we have a contextual object $\Gamma \vdash x$ of type $\Gamma \vdash A$. However, we cannot say that x itself is a variable of type $\Gamma \vdash A$. Consequently, this view of contextual objects is limited by a lack of variables with contextual types. Similarly to context and substitution variables, the problem comes from the fact that variables of contextual types must exist outside of the LF context.

To address this issue, we simply allow our meta-contexts Δ to contain assumptions of the form $u{:}(\Gamma \vdash P)$. We call u a *meta-variable* and restrict its type to $\Gamma \vdash P$, that is contextual atomic LF types. This restriction is superficial since the types $\Gamma \vdash \Pi x{:}A.A'$ and $\Gamma, x{:}A \vdash A'$

are isomorphic. Since we want meta-variables to occur within LF objects, we must once again extend the syntax and type synthesis of heads:

$$\text{Heads} \quad H ::= \quad ... \mid u[\sigma] \qquad\qquad \frac{u{:}(\Gamma' \vdash P) \in \Delta \quad \Delta; \Gamma \vdash \sigma : \Gamma'}{\Delta; \Gamma \vdash u[\sigma] \Rightarrow [\sigma]P}$$

So, a meta-variable u may only occur in an LF object when it is paired with a substitution. This is necessary because u can denote an object meaningful in a different LF context from the one we are currently working in. Again, we write the substitution on the the right to indicate that it is a delayed computation. That is, once we substitute u with a proper contextual object $\Gamma' \vdash M$, we apply the substitution to bring M to context Γ. Note that, while we include meta-variables in the syntax of heads, the restriction that they have an atomic type means that we can only meaningfully apply it to the empty spine and convert it to normal term. Accordingly, we could equivalently specify $u[\sigma]$ as a normal term.

2.2.4 Meta-layer

We have now identified several kinds of variables that belong in the meta-context Δ. As we move to the computation-level, we wish to be able to quantify over each of these variables. In the current presentation, this would require the introduction of distinct function spaces for each kind of variables, leading to an unnecessarily verbose system. To address this issue, Cave and Pientka (2012) suggests unifying the different kinds of assumptions into what they call the *meta-layer*.

Simply put, we define a notion of meta-type that combines the classifiers of contexts, substitutions, and contextual objects. Similarly, we unify contexts, substitutions, and contextual objects into the notion of meta-object. This approach also simplifies the definitions

$\boxed{\Delta \vdash \mathcal{M} : \mathcal{A}}$ – Meta-object $\mathcal{M}$ has meta-type A

$$\frac{\Delta; \Gamma \vdash R \Leftarrow P}{\Delta \vdash (\hat{\Gamma}.R) : (\Gamma.P)}\ \textbf{MOft-tm} \qquad \frac{\Delta; \Gamma_1 \vdash \sigma : \Gamma_2}{\Delta \vdash (\hat{\Gamma}_1.\sigma) : (\Gamma_1.\Gamma_2)}\ \textbf{MOft-subst}$$

$$\frac{\Delta \vdash \Gamma : G \text{ (schema checking)}}{\Delta \vdash \Gamma : G \text{ (mtype checking)}}\ \textbf{MOft-ctx}$$

Figure 2.4: Meta-level typing

of meta-contexts and meta-substitutions, which we are finally ready to discuss. First, let us look at the syntax of the meta-layer:

Meta-object	$\mathcal{M}$ ::=	$\hat{\Gamma}.R \mid \hat{\Gamma}.\sigma \mid \Gamma$
Meta-type	$\mathcal{A}$ ::=	$\Gamma.P \mid \Gamma.\Gamma' \mid G$
Meta-variable	X ::=	$u \mid \psi$
Meta-context	Δ ::=	$\cdot \mid \Delta, X{:}\mathcal{A}$
Meta-substitution	θ ::=	$\cdot \mid \theta, \mathcal{M}$

We write $\hat{\Gamma}$ to denote the erasure of Γ, that is the list of variables obtained by removing all type annotations from Γ. The annotations are irrelevant when considering LF terms and substitutions since these do not rely on any type information. Moreover, we can recover the types of variables in an erased context by looking at the non-erased context in the meta-type. Notice however that we only consider a fully annotated context as a valid meta-object. This is necessary since we cannot recover the types of variables from a schema alone.

The meta-level typing rule, defined in Figure 2.4, are simply conversion rules from the previously defined judgments. That is, we obtain $\Delta \vdash (\hat{\Gamma}.R) : \Gamma.P$ by invoking the LF

typing judgment $\Delta; \Gamma \vdash R \Leftarrow P$, that $\Delta \vdash \hat{\Gamma}.\sigma : \Gamma.\Gamma'$ by invoking the LF substitution correctness judgment $\Delta; \Gamma \vdash \sigma : \Gamma'$, and that $\Delta \vdash \Gamma : G$ by invoking the schema-checking judgment $\Delta \vdash \Gamma : G$. The same idea applies for the judgment $\Delta \vdash \mathcal{A} : \texttt{mtype}$ that validates meta-type well-formedness. As for meta-context and meta-substitutions well-formedness, we use judgments $\vdash \Delta : \texttt{mctx}$ and $\Delta \vdash \theta : \Delta'$ (respectively). These are defined similarly to the LF context and LF substitution well-formedness judgments. We omit their definitions here, but provide them in Appendix A for completeness.

2.3 Computation-level

Beluga is an ML-style functional programming language supporting pattern matching over contextual LF objects. It features an indexed function space, so that types are allowed to depend only on data-level objects. Contextual objects and types are embedded in the computation-level via a box modality. We note that, although we restrict our data-level to contextual LF here, Beluga is parametric in its specification language (Pientka and Abel, 2015). We choose contextual LF because it expresses OLs very nicely, as discussed earlier.

The central goal of Beluga is to provide a meta-language in which expressing OLs and their meta-theory is reasonably easy. An OL is specified in Contextual LF using HOAS, as described in the previous sections. Then, we write proofs about the OL as recursive Beluga programs. Crucially, a recursive function may only be seen as a proof if it is total, that is it terminates on every input. This being said, we also allow non-terminating functions, making Beluga into a general-purpose programming language rather than just a proof environment.

We present here a version of Beluga closely inspired by the one of Pientka and Abel

(2015), although it differs in two important ways, namely the absence of recursive functions and of a totality checker. We expect that recursive functions can be added without significant difficulties, but that requires several technical details that are irrelevant to the present work. Specifically, the valid recursive calls have to be specified as part of every pattern (although they can be inferred, so users do not need to provide them explicitly). Without recursion, patterns are essentially (boxed) contextual objects, which we can handle straightforwardly. Totality, on the other hand, presents fundamental challenges, which we will discuss in more details as we introduce the refinement system, in Chapter 3. For now, let us look at the syntax of Beluga:

$$
\begin{array}{lrl}
\text{Types} & \tau ::= & [\mathcal{A}] \mid \tau_1 \to \tau_2 \mid \Pi X{:}\mathcal{A}.\tau \\
\text{Contexts} & \Xi ::= & \cdot \mid \Xi, y{:}\tau \\
\text{Expressions} & e ::= & [\mathcal{M}] \mid \texttt{fn}\ y{:}\tau \Rightarrow e \mid e_1\ e_2 \mid \texttt{mlam}\ X{:}\mathcal{A} \Rightarrow e \mid e\ \mathcal{M} \\
 & & \quad \mid \texttt{let}\ [X] = e_1\ \texttt{in}\ e_2 \mid \texttt{case}^{\tau}\ [\mathcal{M}]\ \texttt{of}\ \vec{b} \\
\text{Branches} & b ::= & \Omega; [\mathcal{M}] \Rightarrow e
\end{array}
$$

We now discuss this syntax and the associated typing rules (see Figure 2.5) The lifting of meta-types and meta-objects to the computation level is achieved via a (contextual) box modality, which we denote using square brackets $[\mathcal{A}]$. The elimination form for the modality is given by the `let` expressions: an expression $e_1 : [\mathcal{A}]$ is unboxed as the meta-variable X, which may then be used in the expression e_2.

We distinguish two kinds of function spaces, the simple function space $\tau_1 \to \tau_2$ and the dependent function space $\Pi X{:}\mathcal{A}.\tau$. So, dependencies are restricted to objects from the index domain, which provides strong reasoning power over the index domain without all the

$\boxed{\Delta; \Xi \vdash e : \tau}$ – Expression e has type τ in context Γ and meta-context Δ

$$\frac{\Delta \vdash \Xi : \texttt{cctx} \quad (y{:}\tau) \in \Xi}{\Delta; \Xi \vdash y{:}\tau}\ \textbf{CT-var} \qquad \frac{\Delta \vdash \mathcal{M} : \mathcal{A} \quad \Delta \vdash \Xi : \texttt{cctx}}{\Delta; \Xi \vdash [\mathcal{M}] : [\mathcal{A}]}\ \textbf{CT-box}$$

$$\frac{\Delta; \Xi, y{:}\tau_1 \vdash e : \tau_2}{\Delta; \Xi \vdash \texttt{fn}\ y{:}\tau_1 \Rightarrow e : \tau_1 \to \tau_2}\ \textbf{CT-fn} \qquad \frac{\Delta; \Xi \vdash e_1 : \tau_2 \to \tau_1 \quad \Delta; \Xi \vdash e_2 : \tau_2}{\Delta; \Xi \vdash e_1\ e_2 : \tau_1}\ \textbf{CT-app}$$

$$\frac{\Delta, X{:}\mathcal{A}; \Xi \vdash e : \tau}{\Delta; \Xi \vdash \texttt{mlam}\ X{:}\mathcal{A} \Rightarrow e : \Pi X{:}\mathcal{A}.\tau}\ \textbf{CT-mlam} \qquad \frac{\Delta; \Xi \vdash e : \Pi X{:}\mathcal{A}.\tau \quad \Delta \vdash \mathcal{M} : \mathcal{A}}{\Delta; \Xi \vdash e\ \mathcal{M} : [\![\mathcal{M}/X]\!]\tau}\ \textbf{CT-mapp}$$

$$\frac{\Delta; \Xi \vdash e_1 : [\mathcal{A}] \quad \Delta, X{:}\mathcal{A}; \Xi \vdash e_2 : \tau}{\Delta; \Xi \vdash \texttt{let}\ [X] = e_1 \texttt{in}\ e_2 : \tau}\ \textbf{CT-let}$$

$$\frac{\begin{array}{ll}\tau = \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0 & \Delta \vdash \theta : \Delta_0 \\ \cdot;\cdot \vdash \tau : \texttt{ctype} & \Delta \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}_0 \qquad \Delta; \Xi \vdash^{[\![\theta]\!]\mathcal{A}_0} b_i : \tau \text{ (for all } b_i \in \vec{b})\end{array}}{\Delta; \Xi \vdash (\texttt{case}^{\tau}\ [\mathcal{M}]\ \texttt{of}\ \vec{b}) : [\![\theta, \mathcal{M}/X_0]\!]\tau_0}\ \textbf{CT-case}$$

$\boxed{\Delta; \Xi \vdash^{\mathcal{A}} b : \tau}$ – Branch b satisfies invariant τ when matching against an object of type $\mathcal{A}$

$$\frac{\begin{array}{lll} & \Delta' \vdash \theta'_i : \Delta_i & \\ \Delta_i \vdash \theta : \Delta_0 & \Delta' \vdash \theta' : \Delta & \\ \Delta_i \vdash \mathcal{M}_0 : [\![\theta]\!]\mathcal{A}_0 & \Delta' \vdash [\![\theta']\!]\mathcal{A} = [\![\theta'_i]\!][\![\theta_i]\!]\mathcal{A}_0 & \Delta'; [\![\theta']\!]\Xi \vdash [\![\theta', \theta'_i]\!]e_i : [\![\theta'_i]\!][\![\theta_i]\!]\tau_0\end{array}}{\Delta; \Xi \vdash^{\mathcal{A}} (\Delta_i; [\mathcal{M}_i] \mapsto e_i) : \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0}\ \textbf{CT-branch}$$

Figure 2.5: Beluga typing rules

difficulties of full dependent types. The expressions `fn` $y{:}\tau \Rightarrow e$ and $e_1\ e_2$ correspond to the introduction and elimination forms for simple function spaces, respectively. On the other hand, `mlam` $X{:}\mathcal{A} \Rightarrow e$ and $e\ \mathcal{M}$ correspond to the introduction and elimination forms for dependent function spaces, respectively.

The language also supports pattern matching on meta-objects through the use of `case` expressions. While we do not allow pattern matching on arbitrary expressions, any expression that has a box type can be matched against by first unboxing it with a `let` expression and then matching on the new variable. The type superscript τ in `case` expression corresponds to the invariant that must be satisfied by all the branches in $\vec{b}$. We require that pattern matching invariants have the form $\Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0$ and that they be closed. Intuitively, a branch $\Delta; [\mathcal{M}] \mapsto e$ satisfies this invariant if there is a meta-substitution θ from Δ_0 to Δ such that $\Delta \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}_0$ and $\Delta; \Xi \vdash e : [\![\theta, \mathcal{M}/X_0]\!]\tau_0$. Formally, we also need to verify that the type $\mathcal{A}_0$ of the object in a branch matches the type $\mathcal{A}$ of the object that we are pattern matching on (called the guard). We do this in the branch validation judgment, $\Delta; \Xi \vdash^{\mathcal{A}} b : \tau$, which is indexed by the type $\mathcal{A}$ of the guard. More specifically, the three premises $\Delta' \vdash \theta'_i : \Delta_i$, $\Delta' \vdash \theta' : \theta$, and $\Delta' \vdash [\![\theta']\!]\mathcal{A} = [\![\theta'_i]\!][\![\theta_i]\!]\mathcal{A}_0$, state together that $\mathcal{A}$ and $\mathcal{A}_0$ are unified in meta-context Δ'. Note that we do not here provide a procedure for computing the unifying meta-context and substitutions, so our typing judgment is not algorithmic. Pientka and Pfenning (2003); Pientka (2003) give a unification algorithm for CMTT, which can be adapted to Beluga in a straightforward manner. We omit its discussion from our presentation due to space considerations.

Example: Evaluation in STLC

Now that we have Beluga's computation-level at our disposal, we can finally start proving things. Since our current description of STLC does not include an evaluation semantics, there is not much to prove at the moment. So, let us start by specifying a small-step operational semantics for STLC:

```
LF step : tm → tm → type =
| s-succ : step M N →
           step (succ M) (succ N)
| s-beta : step (app (lam M) N) (M N)
| s-app1 : step M1 M2 →
           step (app M1 N) (app M2 N)
| s-app2 : step N1 N2 →
           step (app M N1) (app M N2);
```

$\boxed{M \longrightarrow N}$: M steps to N (single step)

$$\frac{M \longrightarrow N}{\mathbf{S}\ M \longrightarrow \mathbf{S}\ N}$$

$$\frac{}{(\lambda x.M)\ N \longrightarrow [N/x]M}$$

$$\frac{M_1 \longrightarrow M_2}{(M_1\ N) \longrightarrow (M_2\ N)}$$

$$\frac{N_1 \longrightarrow N_2}{(M\ N_1) \longrightarrow (M\ N_2)}$$

There are four possible ways to evaluate a term. The rules `s-succ`, `s-app1` and `s-app2` are simply congruences that propagate the evaluation of subterms into larger ones. All the real work happens in `s-beta`, which applies functions to their arguments. Notice how substitution in the OL (denoted as $[N/x]M$ in the informal rule) is represented as function application (denoted as `M N`) in the LF representation.

Now, the rule `s-succ` only allows stepping a term of the form `succ M`, and all of the remaining rules only allow stepping terms of the form `app M N`. In particular, there are no ways to step either `zero` or `lam M`. These are instances of *values* of STLC, and we see them as the endpoints of the evaluation process. We encode what it means to be a value as follows:

```
LF val : tm → type =
```

```
| v-zero : val zero

| v-succ : val M → val (succ M)

| v-lam : val (lam M);
```

And now we can prove that values do not step using a recursive Beluga function:

```
LF false : type =;

rec vds : (Γ : lam-ctx) [Γ ⊢ val M] → [Γ ⊢ step M M'] → [⊢ false] =

fn V, S => case V of

| [Γ ⊢ v-zero] => impossible S

| [Γ ⊢ v-succ V'] =>

    let [Γ ⊢ s-succ S'] = S in

    vds [Γ ⊢ V'] [Γ ⊢ S']

| [Γ ⊢ v-lam] => impossible S;
```

We start by declaring an empty type `false` so that we can express the intuitionistic negation (recall, $\neg A \triangleq (A \to \bot)$). The keyword `rec` introduces a new recursive function, called `vds` (for values do not step) in our case. The parenthesis around the context variable, `(Γ : lam-ctx)`, indicate implicit universal quantification over contexts of schema `lam-ctx`. Then, the function (i.e. proof) proceeds by case analysis (i.e. induction) on the proof that we have a value. The cases for `zero` and `lam` are quickly dismissed as impossible since no constructor of `step` allows such terms to step. So, the only case with actual work to do is the one for `succ`, where we essentially only need to defer to our inductive hypothesis.

Notice that we use here the schema `lam-ctx` and not `stlc-ctx`. This is because the

typing information is irrelevant to our purpose, so good design principles dictate that it should be omitted. However, this leads to significant challenges when we need to use the lemma for typed terms, i.e. those defined in a context of schema `stlc-ctx`. Simply put, the mismatching context schema prevents us from using the lemma since the inputs with `stlc-ctx` would not type-check against what `vds` expects. The easy solution is to copy the function and change the schema annotation on Γ to `stlc-ctx`, but this leads to duplication of code that should be avoidable since `vds` is perfectly safe to use even with an `stlc-ctx`. Felty et al. (2015a) discusses alternative solutions involving explicit relations between contexts of the different schemas. We will discuss these in more depth when presenting our case studies in Chapter 5.

2.3.1 User-defined computation-level types

Beluga supports (co)inductive and stratified computation-level types (Jacob-Rao et al., 2018). A stratified type has an inductive structure hidden within its dependencies: any negative occurrence of the type in some constructor must depend on a term that is structurally smaller than the constructor's output's dependency. These play a crucial role in representing proofs by logical relations (Cave and Pientka, 2018), such as normalization proofs. Inductive types are also important in expressing the aforementioned context relations. Coinductive types allow representing infinite data structures and reasoning about their properties.

An interesting advantage of refinement types is that they can represent several (but not all) useful inductive types. As such, they offer a conceptually simpler alternative that is less cumbersome to use in practice, as we will illustrate in Chapter 5. We omit further discussion

of (co)inductive and stratified types from the current presentation as their inclusion complicates the language significantly, and we wish to focus on refinement types. However, given the usefulness of logical relations in the theory of programming languages, including them into our extension with refinement should be a priority for future work.

Chapter 3

Datasort refinements for Beluga

Now that we have introduced Beluga in detail, we move on to discussing its extension with refinement types. As mentioned previously, our work uses a notion refinements known as datasorts, and is closely inspired by the LFR system of Lovas and Pfenning (2010). In essence, the refinement system for Beluga is obtained by replacing the LF specification layer by LFR. Accordingly, we begin the chapter with a review of LFR, which we modify in various ways to fit our needs. The technical contributions occur later, as we extend the refinement system to Contextual LFR, and eventually to Beluga's computation-level.

3.1 LFR

The LFR system (Lovas and Pfenning, 2010; Lovas, 2010) extends the Edinburgh logical framework LF (Harper et al., 1993) with datasort refinement types. The objects of LFR are exactly the same as in LF, that is, expressions of a λ-calculus with constants. The types of LFR are also the same as those of LF, but they play a different role, in a sense secondary to

sorts. Because sorts express more specific information than types, the properties represented by types become less interesting. Specifically, types merely introduce new syntax, thereby providing a kind of upper bound on the names that may occur within a term. Sorts, on the other hand, impose constraints on previously defined syntax, allowing users to isolate the fragment of object (of a given type) that satisfy these constraints. Freeman and Pfenning (1991) note the correspondence between fragments expressible as datasorts and properties recognizable by regular tree-automata. So, intuitively, datasorts are like regular expressions for arbitrary types (instead of just strings).

3.1.1 Types, sorts, and the refinement relation

Now, let us formally look at the refinement types of LFR. As mentioned above, the classifiers of our language are separated into two layers, types and sorts (or refinements), which are related by a refinement relation. We write $S \sqsubset A$ to indicate that sort S refines type A. Sorts are defined essentially in the same way as types, whose syntax we recall to highlight the resemblance:

	Type level	Refinement level
Atomic families	$P ::= \mathbf{a} \mid P\ M$	$Q ::= \mathbf{s} \mid Q\ M \mid P$
Canonical families	$A ::= P \mid \Pi x{:}A_1.A_2$	$S ::= Q \mid \Pi x{:}S_1.S_2$

The refinement relation ultimately boils down to what the user specifies. An atomic type family $\mathbf{a}$ is defined by its constructors and their types. An atomic sort family $\mathbf{s} \sqsubset \mathbf{a}$ is then defined by selecting a subset of the constructors of $\mathbf{a}$ and assigning them sorts that refine their previously specified types. In this sense, refinements offer a way to safely reuse

constructors. Finally, the relation is lifted to other types with simple congruence rules :

$$\frac{Q \sqsubset P}{Q\ M \sqsubset P\ M} \qquad\qquad \frac{S_1 \sqsubset A_1 \quad x{:}S_1 \vdash S_2 \sqsubset A_2}{\Pi x{:}S_1.S_2 \sqsubset \Pi x{:}A_1.A_2}$$

Intuitively, a refinement relation $S \sqsubset A$ holds if S and A have the same shape (including term dependencies) and every atom $\mathbf{s}$ occurring in S refines the corresponding atom $\mathbf{a}$ in A. Whether or not $\mathbf{s} \sqsubset \mathbf{a}$ can be determined by looking at the declaration of $\mathbf{s}$ in the signature. Since $\mathbf{s}$ can only refine a single $\mathbf{a}$, given any sort S, we can generate the unique type A such that $S \sqsubset A$ simply by traversing S and replacing any occurrence of an atom $\mathbf{s}$ by the atom $\mathbf{a}$ which it refines. Consequently, we can view the type A as an output of the judgment $S \sqsubset A$.

Our syntax also includes an embedding of atomic types into atomic sorts. This is not strictly necessary since it is possible to define a sort that corresponds to the entire type that it refines, simply by preserving all of the constructors and imposing no restrictions on them. However, the embedding will be helpful for the addition of subsorting to the language, where embedded types act as upper bounds for the subsorting relation. While we only allow the explicit embedding for atomic types, it can be extended to all types through a straightforward induction.

The other syntactic categories of the language are similarly duplicated at the refinement level. Each category is equipped with a refinement relation that is induced by the refinement for types. In pure LFR, this means that we have kind refinements and context refinements, but this duplication will keep happening as we reach the computation-level. We will discuss contexts in details in Section 3.2 and focus our attention on kinds for now. Kinds and their refinements (called *classes* by Lovas and Pfenning (2010)) are given by the following syntax:

	Type level	Refinement level
Kinds	$K ::= \texttt{type} \mid \Pi x{:}A.K$	$L ::= \texttt{sort} \mid \Pi x{:}S.L$

and the refinement relation is given by the following two rules :

$$\frac{}{\texttt{sort} \sqsubset \texttt{type}} \qquad\qquad \frac{S \sqsubset A \quad x{:}S \vdash L \sqsubset K}{\Pi x{:}S.L \sqsubset \Pi x{:}A.K}$$

Intuitively, the refinement relation $L \sqsubset K$ holds if L and K have the same shape and all the sorts in L refine the corresponding type K. Hence, kind refinement is simply a consequence of type refinement. In particular, since A is an output of $S \sqsubset A$ and `sort` only refines `type`, we can also view K as an output of $L \sqsubset K$. In fact, a similar principle applies to all of our refinement relations to come since they are all induced by type refinements.

Example: From judgments to algorithms, an evaluation strategy for STLC

When we discussed the stepping semantics for STLC earlier, we defined a non-deterministic judgment. In particular, there can be several ways to step a function application. For instance, we can choose to first evaluate on the right and then on the left, of vice versa. In practice, we need to fix an evaluation strategy since algorithms must be deterministic.

Here, we use refinements to extract those stepping derivations that follow a call-by-value (CBV) strategy. This means that we evaluate a function application by first reducing the left-hand side to a λ-expression, then we reduce the right-hand side to a value, and finally we apply β-reduction. We can impose these constraints by restricting some terms to values in the sorts of the constructors of `step`. But first, we specify values as a refinement of terms:

```
LFR val ⊏ tm : sort =

| zero : val
```

```
  | succ : val → val
  | lam : (tm → tm) → val;
```

As mentioned before, a sort is defined by picking some constructors of a type and assigning them sorts. Here, we pick all of the constructors except for `app` and we restrict the sort of `succ` to `val → val` since the successor of an arbitrary term may not be a value. The sort of `lam` is also changed so that the output is `val` instead of `tm`, and the argument being of sort `(val → tm)` enforces that functions can only be computed when they are applied to values. Note that the sort of `lam` is also changed so that the output is `val` instead of `tm`. However, since the input `(tm → tm)` is unrestricted, it remains true that all `lam`-expressions are classified by `val` [1]. Notice also that we use here the embedding of (atomic) types into (atomic) sorts and view `tm` as a sort rather than a type.

Now, we can define a CBV strategy as a refinement of `step`:

```
LFR cbv ⊏ step : tm → tm → sort =
  | s-succ : cbv M N → cbv (succ M) (succ N)
  | s-app1 : cbv M1 M2 → cbv (app M1 N) (app M2 N)
  | s-app2 : (V : val) cbv N1 N2 → cbv (app V N1) (app V N2)
  | s-beta : (V : val) cbv (app (lam M) V) (M V);
```

Here, we have mainly restricted `s-app2` and `s-beta`. We are only allowed to reduce the argument of a function application if the function is already a value, and we only allow uses

[1] It would also be possible to assign `lam : (val → tm) → val`, enforcing the fact that computations can only occur when functions are applied to values. However, justifying that all `lam`-expressions of STLC are valid `val` would then require the use of subsorting, which we have not yet introduced.

of s-beta if the argument is already a value. We impose these restrictions with the implicit quantification, denoted with round brackets (V : val). Here, the fact that V is an implicit argument is imposed by the types assigned to s-app2 and s-beta in the definition of step. Note that for s-app2, we only enforce that the left-hand side of an application has sort val, not that it is a lam. However, the typing rules of STLC enforce that the left-hand side of function applications have function types, and values of function types are all of the form lam M for some M.

While we are on the subject of evaluation, let us consider the multi-step semantics and how refinements may help us in expressing it. step only tells us how to take one step in the evaluation process, but we are usually interested in stepping terms until they become values. To express longer evaluation sequence, we define a reflexive transitive closure of the stepping relation:

```
LF mstep : tm → tm → type =

| m-refl : mstep M M

| m-trans : step M1 M2 → mstep M2 M3 →

            mstep M1 M3
```

$\boxed{M \longrightarrow^* N}$: M multi-steps to N

$$\frac{}{M \longrightarrow^* M}$$

$$\frac{M_1 \longrightarrow M_2 \quad M_2 \longrightarrow^* M_3}{M_1 \longrightarrow^* M_3}$$

The judgment $M \longrightarrow^* N$ expresses that M steps to N in 0 or more steps. We encode this in LF with the type mstep, defined by two constructors: m-refl takes care of the case for 0 steps, while m-trans allows adding one more step to a multi-step derivation. One issue with this judgment is that it characterizes the intermediate steps of an evaluation algorithm, but offers no way to validate that the evaluation is complete. This is because the reflexivity step, which acts as a stopping point, may be applied to any term, including those that could

be stepped further. We can address this issue using a refinement:

```
LFR eval ⊏ mstep : tm → val → sort =
| m-refl : eval V V
| m-trans : cbv M N → eval N V → eval M V;
```

Now, the refinement kind of the sort `eval` imposes that the second dependency be of sort `val`. Consequently, the reflexivity rule may only be used on values, and thus it provides a satisfying stopping point for the evaluation. Generally, the kind `tm → val → sort` ensures that `eval` extracts the fragment of `mstep` that corresponds to complete evaluation sequences.

3.1.2 Subsorting

One goal of refinement types is to provide a manageable form of subtyping. However, we want to maintain the fact that any well-formed expression has a unique type, and this fundamentally goes against having subtyping. Instead, we use the notion of *subsorting*, which is like subtyping, but at the level of sorts. Since we do allow well-formed expressions to have several different sorts, a subsorting relation can be meaningful, provided that any two subsorts refine the same type.

This thesis will not discuss subsorting too much since we are more interested in the refinement relation itself. However, it is a useful feature and necessary for some of our examples to go through, so we include it as part of the formal definition of the language. We present here a high-level overview of subsorting for LFR, and a formal definition may be found in appendices A (for data-level) and B (for computation-level). Much like refinement, the subsorting relation ultimately boils down to its behaviour in LFR.

The refinement relation for atomic families $Q \sqsubset P$ is similar to the notion of constructor subtyping (Barthe and Frade, 1999), according to which a subtyping relation $P_1 \leq P_2$ occurs between two inductive types when P_1 is defined by a subset of the constructors of P_2. As such, it is sensible to consider a notion of subsorting such that $Q \leq P$ whenever $Q \sqsubset P$. In particular, a subsumption rule is admissible for refinements of atomic families.

The similarity between subtyping and refinement does not carry to function spaces because the rule for establishing $\Pi x{:}S_1.S_2 \sqsubset \Pi x{:}A_1.A_2$ requires that $S_1 \sqsubset A_1$. This is in contrast with subtyping, which is contra-variant in the domain of function spaces, i.e. requires that $A_1 \leq A_1'$ to conclude that $\Pi A_1'.A_2' \leq \Pi A_1.A_2$. The contra-variance is essential in ensuring that subsumption remains sound on function spaces: A function defined over a large domain A is safe to use on objects from a smaller domain $A' \leq A$, but a function defined over the small domain A' may not be safe to use on all objects of type A. For this reason, we cannot generally consider refinement types to be a form of subtyping, although there is a natural subsorting relation that emerges from the refinement relation. This being said, subsumption is admissible for all LF types because the function spaces are weak, so there is no issue of ensuring totality. Once we reach the computation-level and include pattern matching, allowing subsumption does not always preserve coverage.

We also allow users to specify their own subsorting relation, but only for atomic sorts. Intuitively, the subsorting relation $\mathbf{s}_1 \leq \mathbf{s}_2 \sqsubset \mathbf{a}$ corresponds to the presence of a conversion rule from $\mathbf{s}_1$ to $\mathbf{s}_2$. We require that the subsort $\mathbf{s}_1$ be specified prior (or simultaneously) to its supersort $\mathbf{s}_2$. The collection of user-defined sorts $\mathbf{s}$ refining a specific type $\mathbf{a}$ may then be structured as a directed graph, where the $\mathbf{s}$ are vertices and there is an edge from $\mathbf{s}_1$ to $\mathbf{s}_2$

if there is a declaration that $\mathbf{s}_1 \leq \mathbf{s}_2 \sqsubset \mathbf{a}$. Currently, our system only allows one subsort to be given, meaning that the graph is always a tree. We expect no fundamental challenges to emerge from generalizing this structure to a directed acyclic graph.

Once the user has defined subsorting on atomic sorts, the relation is propagated through the rest of the language in two steps. First, we take the reflexive transitive closure of subsorting for atomic sorts. Then, we define subsorting for function spaces with the usual contra-variant rule. In this case, reflexivity and transitivity cannot be used directly for function spaces, although the two rules are easily shown to be admissible via inductive arguments. We only allow reflexivity and transitivity for atomic sorts because these rules are not syntax-directed, and therefore enlarge the proof space significantly. On the other hand, verifying that $\mathbf{s}_1 \leq \mathbf{s}_2 \sqsubset \mathbf{a}$ may be accomplished by a simple graph traversal.

For any type A, the set of sorts $S \sqsubset A$ forms a partially ordered set (poset). We know that $S \leq A \sqsubset A$ whenever $S \sqsubset A$, so the poset has a maximal element (commonly denoted $\top$). Moreover, we can always define an empty sort $\bot \sqsubset A$ such that $\bot \leq S \sqsubset A$ for any $S \sqsubset A$, so the poset can have a minimal element.

Example: Neutral and normal terms

We demonstrate the uses of subsorting by encoding neutral and normal terms of the untyped λ-calculus. We use the same type `tm` as before, which we recall encoded an untyped λ-calculus with natural numbers. A term is normal if none of its subterms can be evaluated further. This is similar to the notion of values discussed in the previous example, except that now we also need to ensure that the bodies of functions cannot be evaluated further.

To properly define normal terms, we must ensure that no λ-abstraction appears on the

left-hand side of a function application, as otherwise β-reduction could be performed. To achieve this, we need to separate term formation into two phases, neutral and normal. We start by constructing neutral terms from constants, variables, and function application. Then, we convert neutral terms into normal terms, at which point only λ-abstractions can be introduced. We encode this language of normal terms as follows:

```
LFR neutral ⊏ tm : sort =
| zero : neutral
| succ : neutral → neutral
| app : neutral → normal → neutral

and neutral ≤ normal ⊏ tm : sort =
| lam : (neutral → normal) → normal;
```

When defining `normal`, we specify that it extends `neutral`. This means that any neutral term may be interpreted as a normal term. Once we have a normal term, we can start binding its free variables with `lam`. However, the constructor `app` can no longer be used since it is not part of the definition of `normal`. Thus, these two sorts allow us to specify a more precise order in which the constructors may be used.

Note that the negative occurrence of `tm` in the type of `lam` is refined with the sort `neutral`. This means that a context used to construct normal terms should have its variables restricted to sort `neutral`. In this case, the schema `lam-ctx` discussed earlier no longer provides a good representation of OL contexts used for constructing normal terms. To handle this difficulty, we will need a notion of refinement schema, which will be introduced in the next chapter.

3.1.3 Changes to LFR

Our presentation of LFR differs from that of Lovas and Pfenning (2010); Lovas (2010) in several ways and describes a slightly different language. The changes that we made were motivated by two main reasons: because they facilitate a smooth integration of refinements into the setting of Beluga, or because we considered them as improvements on the original LFR. We will now discuss these changes in detail.

Our first modification concerns the embedding of types into sorts. Our syntax allows interpreting any atomic type as an atomic sort, so that $P \sqsubset P$ for any atomic type P. In contrast, Lovas and Pfenning (2010) have a special sort, called $\top$ (read top), such that $\top \sqsubset A$ for any type A. Consequently, there is no distinction between the $\top$-sort for type A and the one for a different type A'. This means that upon encountering $\top$ in a sorting derivation, the type that is refined cannot be directly inferred. In previous work, this issue was solved by performing type-checking first, so that we could know which type is refined by which $\top$. We consider this to be wasteful since sorting derivations have the same shape as their typing counterparts. Thus, performing typing and sorting results in doing much of the work twice. Moreover, there are many cases where $\top$ does not appear within a sort, so that the whole typing derivation serves no real purpose. Using an explicit embedding of types into sorts allows to switch to a typing procedure only when we encounter a type, which reduces the amount of redundant work. In this sense, we consider this modification as a strict improvement of Lovas and Pfenning (2010)'s work.

We note, however, that performing type-checking first offers other benefits. First, it

guarantees the conservativity of the extension with refinements, whereas we must prove it. Second, the fact that constants can have multiple declared sorts implies that there is sometimes a need to backtrack during the sort synthesis phase. This is because synthesis will only succeed when we find the correct sort declaration for a given constant, so we might need to perform synthesis several times, until the correct signature lookup occurs. As a consequence, sort-checking is more expansive than type-checking. In this case, typing errors can be detected faster by the type-checker than the sort-checker. In other words, our approach performs better only on well-typed (but possibly ill-sorted) programs.

Secondly, we distinguish sorting and typing contexts, whereas Lovas and Pfenning (2010) used mixed contexts that ascribe both sorts and types to each variable. That is, their contexts have the form $x_1 : S_1 \sqsubset A_1, ..., x_n : S_n \sqsubset A_n$, while we would have a refinement relation directly between the contexts, that is $(x_1 : S_1, ..., x_n : S_n) \sqsubset (x_1 : A_1, ..., x_n : A_n)$. The reason for this change is that it facilitates the transition to contextual refinements. We will discuss this issue in more depth when we introduce Contextual LFR, in the next section.

The last technical modifications that we made is the omission of intersection sorts. Lovas and Pfenning (2010) only allows constants to be given a sort once, but the whole idea of refinements only works if we can assign them multiple sorts. The standard way to tackle this issue is to allow users to declare multiple sorts at once through intersection sorts. That is, the sort $S_1 \wedge S_2$ classifies objects that inhabit both sorts S_1 and S_2. Here, we take instead a different approach and allow constants to appear several times within a signature. However, we also change the form of declaration so that each declared constructor appears inside a type or sort declaration, following Cave and Pientka (2018).

Note that the absence of intersections prevents variations on the possible uses of a constructor for a particular sort. That is, we cannot have $\mathbf{c} : \vec{S} \to \mathbf{s}$ and $\mathbf{c} : \vec{S}' \to \mathbf{s}$ since $\mathbf{c}$ can only have one sort generating an $\mathbf{s}$. The original system of Lovas and Pfenning (2010) does support such definitions since intersections can be used anywhere. Moreover, Lovas and Pfenning (2010) gives an example that uses intersections in this way, namely the encoding of the languages of the λ-cube (Barendregt, 1991). It would be difficult to reproduce this mechanization in our language without including intersection sorts. This being said, we expect that the addition of intersection sorts to our system would not affect the results presented here.

3.2 Contextual LFR

Having gained a better understanding of the core ideas behind datasort, let us start developing a notion of contextual refinements. This means extending LFR with a notion of schemas, and then defining Contextual LFR through addition contextual sorts. In the process, we will need to distinguish sorting contexts Ψ from typing contexts Γ, which are related through refinements. Then, a contextual sort $\Psi \vdash S$ refines a contextual type $\Gamma \vdash A$ if Ψ refines Γ and S refines A. Intuitively, contextual types characterize derivations, and contextual sorts can isolate those derivations whose HOAS trees have particular structures. In this sense, it is natural to interpret contextual refinement as a relation between judgments. This realization will then lead us towards defining our refinements as relations on judgments, thereby facilitating the transition from LFR refinements to Contextual LFR refinements.

As before, we will need a notion of meta-context during our discussion of Contextual LFR. This time however, it must contain sorting assumptions instead of typing assumptions. We will write Ω for a refinement-level meta-context, which we will discuss further at the end of this section.

3.2.1 Contexts and schemas

We begin by defining refinements for contexts and schemas. Similarly as before, we achieve our goal by duplicating the relevant syntactic categories, yielding the following:

	Type level	Refinement level
Blocks of declarations	$B ::= \cdot \mid \Sigma x{:}A.B$	$C ::= \cdot \mid \Sigma x{:}S.C$
Schema elements	$E ::= B \mid \Pi x{:}A.E$	$F ::= C \mid \Pi x{:}S.F$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x{:}A \mid \Gamma, b{:}\mathbf{w} \cdot \vec{M}$	$\Psi ::= \cdot \mid \Psi, x{:}S \mid \Psi, b{:}\mathbf{w} \cdot \vec{M}$
Context schemas	$G ::= \cdot \mid G + E$	$H ::= \cdot \mid H + F$

Each of these syntactic categories have the same interpretation as before. However, we think of the type level versions as intrinsic properties of objects, whereas their refinement counterparts are extrinsic. This distinction plays a particularly important role when it comes to schemas. Type schemas provide information on the structure of a context, such as the number of variables within a block and the dependencies between them. Of course, it also contains the typing information for all the variables, and since types are unique, every context also has (at most) a unique type schema. In contrast, refinement schemas provide flexibility in the interpretation of the variables by assigning them sorts. Since contexts are created as sequences of (schema element) names applied to argument, one particular context may be

$\boxed{\Omega; \Psi \vdash F \sqsubset E}$ – Refinement relation for schema elements

$$\frac{\Omega \vdash \Psi \sqsubset \Gamma}{\Omega; \Psi \vdash \cdot \sqsubset \cdot} \qquad \frac{\Omega; \Psi \vdash S \sqsubset A \quad \Omega; \Psi, x{:}S \vdash C \sqsubset B}{\Omega; \Psi \vdash \Sigma x{:}S.C \sqsubset \Sigma x{:}A.B} \qquad \frac{\Omega; \Psi \vdash S \sqsubset A \quad \Omega; \Psi, x{:}S \vdash C \sqsubset B}{\Omega; \Psi \vdash \Pi x{:}S.F \sqsubset \Pi x{:}A.E}$$

$\boxed{\Omega \vdash H \sqsubset G}$ – Refinement relation for context schemas

$$\frac{\vdash \Omega \sqsubset \Delta}{\Omega \vdash \cdot \sqsubset \cdot} \qquad \frac{\Omega \vdash H \sqsubset G \quad \mathbf{w} \notin H \quad (\mathbf{w}{:}E) \in G \quad \Omega; \cdot \vdash F \sqsubset E}{\Omega \vdash (H + \mathbf{w}{:}F) \sqsubset G}$$

$\boxed{\Omega \vdash \Psi \sqsubset \Gamma}$ – Refinement relation for contexts

$$\frac{\vdash \Omega \sqsubset \Gamma}{\Omega \vdash \cdot \sqsubset \cdot} \qquad \frac{\Omega \vdash \Psi \sqsubset \Gamma \quad \Omega; \Psi \vdash S \sqsubset A}{\Omega \vdash (\Psi, x{:}S) \sqsubset (\Gamma, x{:}A)} \qquad \frac{\Omega \vdash \Psi \sqsubset \Gamma \quad \Omega; \Psi \vdash F \sqsubset E}{\Omega \vdash (\Psi, b{:}F \cdot \vec{M}) \sqsubset (\Gamma, b{:}E \cdot \vec{M})}$$

Figure 3.1: Refinement relations for contexts and schemas

classified by several refinement schemas. This means that we can interpret variables locally, based on what properties we want them to have, all without affecting the shape of a context.

The refinement relations for contexts and schemas (provided in Figure 3.1) are simply inherited from type refinement. For schema elements, we just check one sort at a time, starting with the parameters and then the assumptions in the block. Similarly, contexts are checked one assumption at a time. For block assumptions, refinement requires the same parameters to be used in the refined schema element. The relation on schemas is similarly simple, but we take care not to allow duplicate schema elements in G (or in H for that matter). We do this mainly because duplicate elements serve no purpose in practice, but also to highlight the fact that multiple elements of H can refine the same element of G. As always, the type-level components of all the refinement relations are viewed as outputs of the judgments.

Refinement-level contexts are checked against refinement-level schemas in a way that closely resembles the analogous type-level judgment. This becomes evident when we place the rules of both judgments side-by-side:

Type-level schema-checking:

$$\frac{\Delta \vdash G : \texttt{schema}}{\Delta \vdash \cdot : G}$$

$$\frac{\Delta \vdash \Gamma : G \qquad (\mathbf{w}{:}E) \in G \qquad \Delta; \Gamma \vdash \vec{M} : E > B}{\Delta \vdash (\Gamma, b{:}\mathbf{w} \cdot \vec{M}) : G}$$

Refinement-level schema-checking:

$$\frac{\Omega \vdash H \sqsubset G}{\Omega \vdash \cdot : H}$$

$$\frac{\Omega \vdash \Psi : H \qquad (\mathbf{w}{:}F) \in H \qquad \Omega; \Psi \vdash \vec{M} : F > C}{\Omega \vdash (\Psi, b{:}\mathbf{w} \cdot \vec{M}) : H}$$

The most significant differences between those rules is in the premise of the case for an empty context: Instead of validating the well-formedness of a schema, the refinement-level rule calls a refinement judgment. This difference is superficial since the refinement relations are essentially the well-formedness judgments of the refinement level. It is then tempting to relate the new (refinement-level) rules and their associated judgments to the old (type-level) rules and judgments, through a refinement relation. An important aspect of our refinement relations is that the type-level part (i.e. everything that occurs on the right of $\sqsubset$) can be considered as an output of the judgments. Moreover, these outputs have a nice regularity. In particular, for any refinement-level context $\Psi : H$, we can find Γ and G such that $\Psi \sqsubset \Gamma$, $H \sqsubset G$, and $\Gamma : G$ (we prove this and several similar result in Chapter 4).

Example: Neutral variables

In the previous chapter, we saw how subsorting can be used to extract neutral and normal terms. We ended that example by saying that variables should be neutral terms. Now, we

can represent this using a refinement schema:

```
LFR neut-ctx ⊏ lam-ctx : schema =
| lc-var : block (x : neutral);
```

Here, we define a new schema `neut-ctx` as a refinement of our previously defined term formation schema `lam-ctx`. Just as we would define a sort, we must select some of the schema elements of `lam-ctx`, and provide a refinement of its previously specified type. Here, our only choice is to take `lc-var`, and we assign its only variable the sort `neutral`, which indeed refines `tm`.

An alternative approach using only types consists of defining "neutral" as a property of terms, that is as a dependent type `neutral' : tm → type`. In this scenario, we would define our neutral contexts with the following type schema:

```
LF neut-ctx' : schema =
| nc-var : block (x:tm, n:neutral' x);
```

The disadvantage here is that a context of schema `neut-ctx'` never has schema `lam-ctx` Moreover, we cannot easily go from a `neut-ctx'` to a `lam-ctx`, because there is no way to construct a proof that `neutral' x` for a variable `x` in the context, so there is no substitution. A lesser (yet still non-negligible) difficulty occurs in the other direction: there is always a weakening substitution from a `lam-ctx` to a `neut-ctx'`, but specifying the correct weakening substitution (i.e. the one that does not add more variables than needed) requires the use of computation-level inductive types.

In contrast, alternating between a `lam-ctx` and a `neut-ctx` can be achieved without transforming the context in any way. This is because the annotation to a block variable

`b:lc-var` does not specify whether we view the variable inside as a `tm` or a `neutral`. Generally speaking, any context that has a schema contains no explicit sort or type, and can therefore only be interpreted based on what the schema specifies. This being said, sorts express stronger properties than types, so moving from a `lam-ctx` to a `neut-ctx` is a form of strengthening. It is not clear that such strengthenings are always valid, so we must be careful in how we use it. Here, we take a highly restrictive stance and prevent the use of strengthening altogether. Dually, moving from a `neut-ctx` to a `lam-ctx` is a form of weakening, and can therefore safely be used at any point. We will see in Chapter 5 that the combination of weakenings and recursion allows us to recover at least some strengthening in a safe way.

3.2.2 Meta-types and meta-objects

Now that we have discussed all of the classifiers of LFR, we can move on to the unified setting of the meta-layer. It is in this transition that the need for refinements as a relation on judgments should become evident. First, let us recall the syntax of the meta-layer, which is now duplicated at the refinement level:

	Type level	Refinement level
Meta-types	$\mathcal{A} ::= \Gamma.P \mid \Gamma.\Gamma' \mid G$	$\mathcal{S} ::= \Psi.Q \mid \Psi.\Psi' \mid H$
Meta-objects	$\mathcal{M} ::= \hat{\Gamma}.R \mid \hat{\Gamma}.\sigma \mid \Gamma$	$\mathcal{N} ::= \hat{\Psi}.R \mid \hat{\Psi}.\sigma \mid \Psi$
Meta-contexts	$\Delta ::= \cdot \mid \Delta, X{:}\mathcal{A}$	$\Omega ::= \cdot \mid \Omega, X{:}\mathcal{S}$
Meta-substitutions	$\rho ::= \cdot \mid \rho, \mathcal{M}$	$\theta ::= \cdot \mid \theta, \mathcal{N}$

An interesting aspect of refinements at the meta-layer is that we must now consider

refinement-level objects. This comes from the separation of sorting and typing contexts, as well as the fact that contexts cannot be erased when viewed as objects of their own. Notice, however, that if $\Psi \sqsubset \Gamma$, then the only difference between the two contexts is in the annotations of single variables. In this case, we can conclude that $\hat{\Psi}$ and $\hat{\Gamma}$ are identical, since they no longer contain any annotation. Moreover, if Ψ has a schema, then, by definition, it can only contain blocks of assumptions, and no single assumption. In this case, Γ also contains no single assumption since the refinement relation between them ensures that Ψ and Γ have exactly the same shape. Thus, we can also conclude that Ψ and Γ are the same. In other words, in many (but not all) cases where $\mathcal{N} \sqsubset \mathcal{M}$, we actually have that $\mathcal{N}$ and $\mathcal{M}$ are syntactically equal. We note that this separation of meta-objects also induces a similar separation for meta-substitutions, since those are composed of meta-objects.

Now, the refinement relations at the meta-layer are, once again, inherited from previously defined refinement relations, and ultimately from LF type refinements. However, here the fact that we incorporate contexts into types and objects leads to some complications. Let us consider, for instance, how the refinement relation could be defined for a contextual type $\Gamma.A$, and how it compares to the analogous type formation rule:

$$\frac{\Delta; \Gamma \vdash P \Leftarrow \texttt{type}}{\Delta \vdash (\Gamma.P) : \texttt{mtype}} \qquad\qquad \frac{\Omega \vdash \Psi \sqsubset \Gamma \quad \Omega; \Psi \vdash Q \sqsubset P}{\Omega \vdash (\Psi.Q) \sqsubset (\Gamma.P)}$$

As mentioned previously, the refinement relations judgments are analogous to the well-formedness judgments of the type-level. Then, in the above, the two conclusions are analogous in this way, and the premises $\Delta; \Gamma \vdash P \Leftarrow \texttt{type}$ and $\Omega; \Psi \vdash Q \sqsubset P$ are also analogous in this way. However, our only way to obtain the context Γ is by adding an additional premise

$\Omega \vdash \Psi \sqsubset \Gamma$ to the refinement-level rule. For the type-formation rule, we do not have the analogous premise $\Delta \vdash \Gamma : \texttt{ctx}$ since that is enforced through the type formation judgment. In fact, the refinement judgment $\Omega; \Psi \vdash Q \sqsubset P$ does also enforce that $\Omega \vdash \Psi \sqsubset \Gamma$, but this information is not directly available without the extra premise.

Now, we have only discussed one rule, but similar issues occur in various other places, and propagate into the computation-level. Every time we need to add a premise simply to extract some information that is otherwise validated by our judgments, we slow down the sort-checking process. Our approach to remedy these difficulties is to consider refinements as a relation on the judgments themselves. This allows us to extract all the refinement information at once. This gives rise to a judgment of the form $(\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype})$, and similarly for other syntactic categories. For instance, instead of the above refinement rule for contextual types, we can use the following:

$$\frac{(\Omega; \Psi \vdash Q) \sqsubset (\Delta; \Gamma \vdash P \Leftarrow \texttt{type})}{(\Omega \vdash \Psi.Q) \sqsubset (\Delta \vdash \Gamma.P : \texttt{mtype})}$$

Note that for this approach to work as intended, we must go back through our definition of LFR and rewrite its judgments in the same style. We give these definitions in Appendix A.

Now, we have discussed how the refinement relations judgments are analogous to the well-formedness judgments of the type-level, and we have illustrated that these refinement relations may be best understood as relations on the judgments directly, rather than on syntactic objects. Interestingly, a similar idea can be applied to every other judgment in our language. We provide here (in Figure 3.2) the sorting rules for the meta-layer, written in this new style. We obtain, in this case, the judgment $(\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})$. What is most advantageous about this approach is that, in every refinement judgment, the type-level

$\boxed{(\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})}$ – Sorting and typing judgments

$$\frac{\vdash \Omega \sqsubset \Delta \quad \Omega \vdash \Psi \sqsubset \Gamma \quad \Omega; \Psi \vdash R \Leftarrow Q \sqsubset P}{(\Omega \vdash \hat{\Psi}.R : \Psi.Q) \sqsubset (\Delta \vdash \hat{\Gamma}.R : \Gamma.P)} \qquad \frac{\vdash \Omega \sqsubset \Delta \quad \Omega \vdash \Psi \sqsubset \Gamma \quad \Omega \vdash \Psi : H \sqsubset G}{(\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G)}$$

$$\frac{\vdash \Omega \sqsubset \Delta \quad \Omega \vdash \Psi_1 \sqsubset \Gamma_1 \quad \Omega \vdash \Psi_2 \sqsubset \Gamma_2 \quad \Omega; \Psi_1 \vdash \sigma : \Psi_2}{(\Omega \vdash \hat{\Psi}_1.\sigma : \Psi_1.\Psi_2) \sqsubset (\Delta \vdash \hat{\Gamma}_1.\sigma : \Gamma_1.\Gamma_2)}$$

Figure 3.2: Meta-level typing and sorting

part can still be considered as an output. Moreover, our design ensures that the type-level part of a refinement judgment coincides exactly with the analogous type-level judgment. In other words, the judgment $(\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})$ is a verification of the sorting judgment $\Omega \vdash \mathcal{N} : \mathcal{S}$ that additionally produces a typing derivation $\Delta \vdash \mathcal{M} : \mathcal{A}$.

3.3 Computation-level refinements

It remains only to extend our refinements to Beluga's computation-level. We achieve this in roughly the same way as before, that is by duplicating the syntax and rules of the type-level, and defining a refinement relation between the new syntax and the old one. The computation-level inherits a refinement relation on expressions (i.e. objects) from the meta-layer. Once again, each refinement relation ultimately boils down to refinements of LF types. Now, let us present the syntax of refinements more formally:

	Type level	Refinement level
Types	τ	$\zeta ::= [\mathcal{S}] \mid \zeta_1 \to \zeta_2 \mid \Pi X{:}\mathcal{S}.\zeta$
Contexts	Ξ	$\Phi ::= \cdot \mid \Phi, y{:}\zeta$
Expressions	e	$f ::= [\mathcal{N}] \mid \texttt{fn}\ y{:}\zeta \Rightarrow f \mid f_1\ f_2 \mid \texttt{mlam}\ X{:}\mathcal{S} \Rightarrow f \mid f\ \mathcal{N}$
		$\mid \texttt{let}\ [X] = f_1\ \texttt{in}\ f_2 \mid \texttt{case}^{\zeta}\ [\mathcal{N}]\ \texttt{of}\ \vec{c}$
Branches	b	$c ::= \Omega; [\mathcal{N}] \Rightarrow f$

We have omitted redefining the type-level syntax, which can be found either in Chapter 2 or Appendix B, although we recall the symbols used for each syntactic category. All the refinement relations are simply component-wise, and they are defined as relations between refinement-level and type-level well-formedness judgments, following what we did in the previous section. We will discuss here the sorting judgment (whose rules are given in Figure 3.3), but we leave out the others as they are straightforward. The complete definition of these judgments is located in Appendix B.

The most interesting aspects of the sorting judgments are the rules related to pattern matching. In particular, notice that **CTR-case** only requires that we check the branches on the refinement side of the relation. This is consistent with the idea that the type side is an output of the judgment, in which case all the type side branches are in fact generated. This means that the refinement relation on expressions is only possible when pattern matching expressions have the same number of branches on the refinement and type levels. Since a central component of datasorts is their ability to omit some of the constructors from type definitions, we expect that exhaustive pattern matching on the refinement-level contains fewer cases than on the type-level. As a consequence, the typing derivation that we obtain cannot

$\boxed{(\Omega; \Phi \vdash f : \zeta) \sqsubset (\Delta; \Xi \vdash e : \tau)}$ – Sorting and typing judgments for expressions

$$\frac{(\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx}) \quad (y{:}\zeta) \in \Phi \quad (y{:}\tau) \in \Xi}{(\Omega; \Phi \vdash y{:}\zeta) \sqsubset (\Delta; \Xi \vdash y{:}\tau)}\ \textbf{CTR-var}$$

$$\frac{(\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A}) \quad (\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx})}{(\Omega; \Phi \vdash [\mathcal{N}] : [\mathcal{S}]) \sqsubset (\Delta; \Xi \vdash [\mathcal{M}] : [\mathcal{A}])}\ \textbf{CTR-box}$$

$$\frac{(\Omega; \Phi, y{:}\zeta_1 \vdash f : \zeta_2) \sqsubset (\Delta; \Xi, y{:}\tau_1 \vdash e : \tau_2)}{(\Omega; \Phi \vdash \texttt{fn}\ y{:}\zeta_1 \Rightarrow f : \zeta_1 \to \zeta_2) \sqsubset (\Delta; \Xi \vdash \texttt{fn}\ y{:}\tau_1 \Rightarrow e : \tau_1 \to \tau_2)}\ \textbf{CTR-fn}$$

$$\frac{(\Omega; \Phi \vdash f_1 : \zeta_2 \to \zeta_1) \sqsubset (\Delta; \Xi \vdash e_1 : \tau_2 \to \tau_1) \quad (\Omega; \Phi \vdash f_2 : \zeta_2) \sqsubset (\Delta; \Xi \vdash e_2 : \tau_2)}{(\Omega; \Phi \vdash f_1\ f_2 : \zeta_1) \sqsubset (\Delta; \Xi \vdash e_1\ e_2 : \tau_1)}\ \textbf{CTR-app}$$

$$\frac{(\Omega, X{:}\mathcal{S}; \Phi \vdash f : \zeta) \sqsubset (\Delta, X{:}\mathcal{A}; \Xi \vdash e : \tau)}{(\Omega; \Phi \vdash \texttt{mlam}\ X{:}\mathcal{S} \Rightarrow f : \Pi X{:}\mathcal{S}.\zeta) \sqsubset (\Delta; \Xi \vdash \texttt{mlam}\ X{:}\mathcal{A} \Rightarrow e : \Pi X{:}\mathcal{A}.\tau)}\ \textbf{CTR-mlam}$$

$$\frac{(\Omega; \Phi \vdash f : \Pi X{:}\mathcal{S}.\zeta) \sqsubset (\Delta; \Xi \vdash e : \Pi X{:}\mathcal{A}.\tau) \quad (\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})}{(\Omega; \Phi \vdash f\ \mathcal{N} : [\![\mathcal{N}/X]\!]\zeta) \sqsubset (\Delta; \Xi \vdash e\ \mathcal{M} : [\![\mathcal{M}/X]\!]\tau)}\ \textbf{CTR-mapp}$$

$$\frac{(\Omega; \Phi \vdash f_1 : [\mathcal{S}]) \sqsubset (\Delta; \Xi \vdash e_1 : [\mathcal{A}]) \quad (\Omega, X{:}\mathcal{S}; \Phi \vdash f_2 : \zeta) \sqsubset (\Delta, X{:}\mathcal{A}; \Xi \vdash e_2 : \tau)}{(\Omega; \Phi \vdash \texttt{let}\ [X] = f_1\ \texttt{in}\ f_2 : \zeta) \sqsubset (\Delta; \Xi \vdash \texttt{let}\ [X] = e_1 \texttt{in}\ e_2 : \tau)}\ \textbf{CTR-let}$$

$$\frac{\begin{array}{ll}\tau = \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0 & (\Omega \vdash \rho : \Omega_0) \sqsubset (\Delta \vdash \theta : \Delta_0) \\ \zeta = \Pi\Omega_0.\Pi X_0{:}\mathcal{S}_0.\zeta_0 & (\Omega \vdash \mathcal{N} : [\![\rho]\!]\mathcal{S}_0) \sqsubset (\Delta \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}_0) \\ (\cdot;\cdot \vdash \zeta) \sqsubset (\cdot;\cdot \vdash \tau : \texttt{ctype}) & (\Omega; \Phi \vdash^{[\![\rho]\!]\mathcal{S}_0} c_i : \zeta) \sqsubset (\Delta; \Xi \vdash^{[\![\theta]\!]\mathcal{A}_0} b_i : \tau)\ (\text{for all } c_i \in \vec{c})\end{array}}{(\Omega; \Phi \vdash (\texttt{case}^{\zeta}\ [\mathcal{N}]\ \texttt{of}\ \vec{c}) : [\![\rho, \mathcal{N}/X_0]\!]\zeta_0) \sqsubset (\Delta; \Xi \vdash (\texttt{case}^{\tau}\ [\mathcal{M}]\ \texttt{of}\ \vec{b}) : [\![\theta, \mathcal{M}/X_0]\!]\tau_0)}\ \textbf{CTR-case}$$

$\boxed{(\Omega; \Phi \vdash^{\mathcal{S}} c : \zeta) \sqsubset (\Delta; \Xi \vdash^{\mathcal{A}} b : \tau)}$ – Branch c satisfies invariant ζ when matching against an object of sort $\mathcal{S}$

$$\frac{\begin{array}{l}(\Omega_i \vdash \rho : \Omega_0) \sqsubset (\Delta_i \vdash \theta : \Delta_0) \\ (\Omega_i \vdash \mathcal{N}_0 : [\![\rho]\!]\mathcal{S}_0) \sqsubset (\Delta_i \vdash \mathcal{M}_0 : [\![\theta]\!]\mathcal{A}_0) \\ (\Omega' \vdash \rho'_i : \Omega_i) \sqsubset \Delta' \vdash \theta'_i : \Delta_i \\ (\Omega' \vdash \rho' : \Omega) \sqsubset \Delta' \vdash \theta' : \Delta \\ (\Omega' \vdash [\![\rho']\!]\mathcal{S} = [\![\rho'_i]\!][\![\rho_i]\!]\mathcal{S}_0) \sqsubset \Delta' \vdash [\![\theta']\!]\mathcal{A} = [\![\theta'_i]\!][\![\theta_i]\!]\mathcal{A}_0 \\ (\Omega'; [\![\rho']\!]\Phi \vdash [\![\rho', \rho'_i]\!]f_i : [\![\rho'_i]\!][\![\rho_i]\!]\zeta_0) \sqsubset (\Delta'; [\![\theta']\!]\Xi \vdash [\![\theta', \theta'_i]\!]e_i : [\![\theta'_i]\!][\![\theta_i]\!]\tau_0)\end{array}}{(\Omega; \Phi \vdash^{\mathcal{S}} (\Omega_i; [\mathcal{N}_i] \mapsto f_i) : \Pi\Omega_0.\Pi X_0{:}\mathcal{S}_0.\zeta_0) \sqsubset (\Delta; \Xi \vdash^{\mathcal{A}} (\Delta_i; [\mathcal{M}_i] \mapsto e_i) : \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0)}\ \textbf{CTR-branch}$$

Figure 3.3: Sorting as a refinement of typing

ensure coverage, since for that we would need to generate the missing cases automatically. This is the reason why our rules do not enforce coverage, contrary to those of Pientka and Abel (2015). It would be possible to enforce coverage only at the refinement-level, although we leave this to future work.

The second part of validating pattern matching is the branch sorting judgment. Like the branch typing judgment, branch sorting is defined by a single rule whose purpose is to validate that the branch matches the specified invariant and can be unified with the guard. Once again, we obtain this rule by refining its type-level equivalent with the relevant sort-level judgments.

Example: A better proof that values do not step

Now that we have defined computation-level refinements, let us revisit our earlier proof that values do not step. In this chapter, we have seen how to define values as a refinement of terms instead of predicate. This simple modification allows us to improve the statement of the theorem. For convenience, let us remind ourselves of how exactly we stated it earlier:

```
rec vds : (Γ : lam-ctx) [Γ ⊢ val M] → [Γ ⊢ step M M'] → [⊢ false]
```

This statement is fairly close to how one might informally express the result, and it will be difficult to significantly improve upon it. By using the sort `val` instead of the type, we obtain the following:

```
rec vds-ref : (Γ : lam-ctx) {M : [Γ ⊢ val]} [Γ ⊢ step M M'] → [⊢ false]
```

On the surface, this seems roughly the same as before, with the exception that we use a dependent function space to bind `M` instead of a simple function space for `[Γ ⊢ val M]`

However, this is in part the result of us "cheating" our way out of elaboration by allowing silent implicit quantification. Under the hood, these will be expanded into the following:

```
rec vds : (Γ : lam-ctx) (M : [Γ ⊢ tm]) (M' : [Γ ⊢ tm]) [Γ ⊢ val M] →
          [Γ ⊢ step M M'] → [⊢ false]
rec vds-ref : (Γ : lam-ctx) {M : [Γ ⊢ val]} (M' : [Γ ⊢ tm])
              [Γ ⊢ step M M'] → [⊢ false]
```

So, we do in fact get rid of one of the function's arguments, thus simplifying the statement. While this does not appear immediately to the user, simplifying the elaboration process still improves the overall user experience, since the assistance provided by the compiler would be based on the elaborated types or sorts. Moreover, the simplicity of this example leaves little room for improvement. In Chapter 5, we will consider more complex mechanizations, allowing us to observe that refinements can, in fact, provide significant simplifications of theorem statements.

Chapter 4

Meta-theory

This chapter contains the proof that our extension is conservative. Our formulation of refinements as a relation on judgments provides an elegant formulation of the theorem. Essentially, if we write an arbitrary type-level judgment as $\mathcal{I}$ and an arbitrary refinement-level judgment as $\mathcal{J}$, then the theorem states that if the refinement judgment $\mathcal{J} \sqsubset \mathcal{I}$ holds, then the type-level judgment $\mathcal{I}$ is valid on its own. We will have one theorem statement for each particular judgment, and the various statements are proven simultaneously in two groups: First, the data-level judgments, and, second, the computation-level judgments. All of the proofs are based on the definitions found in the appendices.

Before we start the proof, we need to discuss two technical details that will play an important role, and which we have omitted until now since they are of no conceptual interest. Our presentation of refinements on judgments leads to a very simple proof, as virtually all cases are solved by straightforward inductions. In fact, the only exceptions are the base cases, which mainly consist of context lookups and signature lookups.

In the same spirit as other judgments, we write our (LF)context lookup as $(x{:}S \in \Psi) \sqsubset (x{:}A \in \Gamma)$, with the intention that the lookup $x{:}A \in \Gamma$ is an output. This is important since the synthesis rule will then need to produce the typing derivation correctly. This lookup judgment can be seen as a shorthand for the three premisses $\Psi \sqsubset \Gamma$, $x{:}S \in \Psi$, and $x{:}A \in \Gamma$. We need the refinement of contexts there since the conclusion of the rule needs to produce Γ, not just A. Note that, since $\Psi \sqsubset \Gamma$, the two contexts must have the exact same structure, and so that we can perform the two lookups simultaneously. A similar idea applies for lookups in meta-contexts and in computation-level contexts.

The situation is a bit more complex when it comes to signature lookups. We have designed our signature formation rules so that our results would follow easily. So, before moving on to the proof of conservativity, we need to briefly discuss signatures. Their complete definition appears in Appendix B.

A signature consists of a sequence of declarations. Our language currently supports six forms of declarations, including the type-level ones. They are given by the following syntax:

Signatures	$\text{Sig} ::=$	$\cdot \mid \text{Sig}, D$	
Declarations	$D ::=$	$\texttt{LF}\ \mathbf{a} : K = \mathbf{c}_1{:}A_1 \mid ... \mid \mathbf{c}_n{:}A_n$	LF type
		$\texttt{LFR}\ \mathbf{s} \sqsubset \mathbf{a} : L = \mathbf{c}_1{:}S_1 \mid ... \mid \mathbf{c}_n{:}S_n$	LFR sort
		$\texttt{LFR}\ \mathbf{s}_1 \leq \mathbf{s}_2 \sqsubset \mathbf{a} : L = \mathbf{c}_1{:}S_1 \mid ... \mid \mathbf{c}_n{:}S_n$	LFR supersort
		$\texttt{LF}\ \mathbf{g} : \texttt{schema} = \mathbf{w}_1{:}V_1 \mid ... \mid \mathbf{w}_n{:}V_n$	LF schema
		$\texttt{LFR}\ \mathbf{h} \sqsubset \mathbf{g} : \texttt{schema} = \mathbf{w}_1{:}W_1 \mid ... \mid \mathbf{w}_n{:}W_n$	LFR schema
		$\texttt{rec}\ \mathbf{f} : \zeta = f$	Recursive function

Signatures keep track of all the user-defined names and how to use them. The crucial

aspects of the various declarations, then, is which names are new and which ones are previously known. We have mentioned on several occasions that any type-level declaration can only introduce new names. On the other hand, refinement-level declarations must introduce exactly one name. Moreover, refinement-level declarations are limited in which other names they may use. Specifically, the names of their constructors must be selected from the collection of names of constructors of the type (or schema) which is refined. For instance, the rule for well-formedness of a sort declaration is the following:

$$\frac{\begin{array}{ll}\mathbf{s} \notin \Sigma & (\cdot;\cdot \vdash_\Sigma L) \sqsubset (\cdot;\cdot \vdash_\Sigma K : \mathtt{kind}) \\ (\mathtt{LF}\ \mathbf{a} : K = \mathbf{c}_1{:}A_1 \mid ... \mid \mathbf{c}_n{:}A_n) \in \Sigma & (\cdot;\cdot \vdash_{\Sigma,\mathbf{s}\sqsubset\mathbf{a}:L} S_{i_j}) \sqsubset (\cdot;\cdot \vdash_{\Sigma,\mathbf{s}\sqsubset\mathbf{a}:L} A_{i_j})\ \text{(for all j)}\end{array}}{\vdash_\Sigma (\mathtt{LFR}\ \mathbf{s} \sqsubset \mathbf{a} : L = \mathbf{c}_{i_1}{:}S_{i_1} \mid ... \mid \mathbf{c}_{i_k}{:}S_{i_k}) : \mathbf{decl}}$$

So, the premises ensure that the new name **s** has not been used, and that the remaining names are taken from the definition of **a**, as desired. This means that when we reach a base case where we need a signature lookup for **s**, then we know that indeed the type **a** exists, and we can produce it. The remaining rules have a similar form. We note that in supersort declarations, it is $\mathbf{s}_2$ that is introduced, and not $\mathbf{s}_1$.

One notable declaration is the one for recursive functions, which only has a refinement-level version. We have this declaration so that the name of a recursive function is available when we sort-check its actual definition. In this way, we avoid having recursion as part of the syntax of expression, which enforces that all recursive functions are defined at the top-level (i.e. there is no nested recursion).

4.1 Conservativity for data-level

Theorem 4.1 (Conservativity for data-level)**.**

1. If $(\Omega; \Psi \vdash L) \sqsubset (\Delta; \Gamma \vdash K : \mathtt{kind})$, then $\Delta; \Gamma \vdash K : \mathtt{kind}$.

2. If $(\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \mathtt{type})$, then $\Delta; \Gamma \vdash A \Leftarrow \mathtt{type}$.

3. If $(\Omega; \Psi \vdash \vec{M} : L > \mathtt{sort}) \sqsubset (\Delta; \Gamma \vdash \vec{M} : K > \mathtt{type})$, then $\Delta; \Gamma \vdash \vec{M} : K > \mathtt{type}$.

4. If $(\Omega; \Psi \vdash H \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash H \Rightarrow A)$, then $\Delta; \Gamma \vdash H \Rightarrow A$.

5. If $(\Omega; \Psi \vdash W[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash V[\vec{M}] > C)$, then $\Delta; \Gamma \vdash V[\vec{M}] > C$.

6. If $(\Omega; \Psi \vdash b : D \gg_i^k S) \sqsubset (\Delta; \Gamma \vdash b : C \gg_i^k A)$, then $\Delta; \Gamma \vdash b : C \gg_i^k A$.

7. If $(\Omega; \Psi \vdash R \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash R \Rightarrow A)$, then $\Delta; \Gamma \vdash R \Rightarrow A$.

8. If $(\Omega; \Psi \vdash \vec{M} : S' > S) \sqsubset (\Delta; \Gamma \vdash \vec{M} : A' > A)$, then $\Delta; \Gamma \vdash \vec{M} : A' > A$.

9. If $(\Omega; \Psi \vdash M \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A)$, then $\Delta; \Gamma \vdash M \Leftarrow A$.

10. If $(\Omega; \Psi \vdash D) \sqsubset (\Delta; \Gamma \vdash C : \mathtt{block})$, then $\Delta; \Gamma \vdash C : \mathtt{block}$.

11. If $(\Omega; \Psi \vdash W) \sqsubset (\Delta; \Gamma \vdash V : \mathtt{world})$, then $\Delta; \Gamma \vdash V : \mathtt{world}$.

12. If $(\Omega \vdash H) \sqsubset (\Delta \vdash G : \mathtt{schema})$, then $\Delta \vdash G : \mathtt{schema}$.

13. If $(\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G)$, then $\Delta \vdash \Gamma : G$.

14. If $(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx})$, then $\Delta \vdash \Gamma : \mathtt{ctx}$.

15. If $(\Omega; \Psi_1 \vdash \sigma : \Psi_2) \sqsubset (\Delta; \Gamma_1 \vdash \sigma : \Gamma_2)$, then $\Delta; \Gamma_1 \vdash \sigma : \Gamma_2$.

16. If $(\Omega; \Psi \vdash \vec{M} \Leftarrow D) \sqsubset (\Delta; \Gamma \vdash \vec{M} \Leftarrow C)$, then $\Delta; \Gamma \vdash \vec{M} \Leftarrow C$.

17. If $(\vdash \Omega) \sqsubset (\vdash \Delta : \texttt{mctx})$, then $\vdash \Delta : \texttt{mctx}$.

18. If $(\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype})$, then $\Delta \vdash \mathcal{A} : \texttt{mtype}$.

19. If $(\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})$, then $\Delta \vdash \mathcal{M} : \mathcal{A}$.

20. If $(\Omega_1 \vdash \rho : \Omega_2) \sqsubset (\Delta_1 \vdash \theta : \Delta_2)$, then $\Delta_1 \vdash \theta : \Delta_2$.

21. If $(\Omega; \Psi \vdash S_1 \leq S_2) \sqsubset (\Delta; \Gamma \vdash A)$, then $\Delta; \Gamma \vdash A \Leftarrow \texttt{type}$

22. If $(\Omega; \Psi \vdash D_1 \leq D_2) \sqsubset (\Delta; \Gamma \vdash C : \texttt{block})$, then $\Delta; \Gamma \vdash C : \texttt{block}$.

23. If $(\Omega; \Psi \vdash W_1 \leq W_2) \sqsubset (\Delta; \Gamma \vdash V : \texttt{world})$, then $\Delta; \Gamma \vdash V : \texttt{world}$.

24. If $(\Omega; \Psi \vdash H_1 \leq H_2) \sqsubset (\Delta; \Gamma \vdash G : \texttt{schema})$, then $\Delta; \Gamma \vdash G : \texttt{schema}$.

25. If $(\Omega \vdash \mathcal{S}_1 \leq \mathcal{S}_2) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype})$, then $\Delta \vdash \mathcal{A} : \texttt{mtype}$.

Proof.

We argue by simultaneous induction on the given derivation $\mathcal{D}$.

1. We have $\mathcal{D} :: (\Omega; \Psi \vdash L) \sqsubset (\Delta; \Gamma \vdash K : \texttt{kind})$. There are two cases to consider

   Case $\mathcal{D} = \dfrac{\mathcal{D}' :: (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx})}{(\Omega; \Psi \vdash \texttt{sort}) \sqsubset (\Delta; \Gamma \vdash \texttt{type} : \texttt{kind})}$ **KR-sort**

   We have $\Delta \vdash \Gamma : \texttt{ctx}$ by inductive hypothesis on $\mathcal{D}'$

   Then $\Delta; \Gamma \vdash \texttt{type} : \texttt{kind}$ by rule **K-type**

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :: \ (\Omega; \Psi, x{:}S \vdash L) \sqsubset (\Delta; \Gamma, x{:}A \vdash K : \mathtt{kind})}{(\Omega; \Psi \vdash \Pi x{:}S.L) \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A.K : \mathtt{kind})} \ \textbf{KR-pi}$$

We have $\Delta; \Gamma, x{:}A \vdash K : \mathtt{kind}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta; \Gamma \vdash \Pi x{:}A.K : \mathtt{kind}$ by rule **K-pi**

2. We have $\mathcal{D} :: \ (\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \mathtt{type})$. There are two cases to consider :

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 : \ \big(\mathbf{s}{:}L) \sqsubset (\mathbf{a}{:}K)\big) \in \Sigma \\ \mathcal{D}_2 : \ (\Omega; \Psi \vdash \vec{M} : L > \mathtt{sort}) \sqsubset (\Delta; \Gamma \vdash \vec{M} : K > \mathtt{type})\end{array}}{(\Omega; \Psi \vdash \mathbf{s} \ \vec{M}) \sqsubset (\Delta; \Gamma \vdash \mathbf{a} \ \vec{M} \Leftarrow \mathtt{type})} \ \textbf{TR-atom}$$

We have $(\mathbf{a}{:}K) \in \Sigma$ by inversion on signature formation rules with $\mathcal{D}_1$

We have $\Delta; \Gamma \vdash \vec{M} : K > \mathtt{type}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash \mathbf{a} \ \vec{M} \Leftarrow \mathtt{type}$ by rule **T-atom**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 : \ (\Omega; \Psi \vdash S_1) \sqsubset (\Delta; \Gamma \vdash A_1 \Leftarrow \mathtt{type}) \\ \mathcal{D}_2 : \ (\Omega; \Psi, x{:}S_1 \vdash S_2) \sqsubset (\Delta; \Gamma, x{:}A_1 \vdash A_2 \Leftarrow \mathtt{type}\end{array}}{(\Omega; \Psi \vdash \Pi x{:}S_1.S_2) \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A_1.A_2 \Leftarrow \mathtt{type})} \ \textbf{TR-pi}$$

We have $\Delta; \Gamma \vdash A_1 \Leftarrow \mathtt{type}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma, x{:}A_1 \vdash A_2 \Leftarrow \mathtt{type}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash \Pi x{:}A_1.A_2 \Leftarrow \mathtt{type}$ by rule **T-pi**

3. We have $\mathcal{D} :: \ (\Omega; \Psi \vdash \vec{M} : L > \mathtt{sort}) \sqsubset (\Delta; \Gamma \vdash \vec{M} : K > \mathtt{type})$. There are two cases to consider :

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :: \ (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx})}{(\Omega; \Psi \vdash \mathtt{nil} : \mathtt{sort} > \mathtt{sort}) \sqsubset (\Delta; \Gamma \vdash \mathtt{nil} : \mathtt{type} > \mathtt{type})} \ \textbf{KR-spn-nil}$$

We have $\Delta \vdash \Gamma : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta; \Gamma \vdash \mathtt{nil} : \mathtt{type} > \mathtt{type}$ by rule **K-spn-nil**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :: (\Omega; \Psi \vdash M \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A) \\ \mathcal{D}_2 :: (\Omega; \Psi \vdash \vec{M} : [M/x]L > \mathtt{sort}) \sqsubset (\Delta; \Gamma \vdash \vec{M} : [M/x]K > \mathtt{type})\end{array}}{(\Omega; \Psi \vdash (M; \vec{M}) : \Pi x{:}S.L > \mathtt{sort}) \sqsubset (\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A.K > \mathtt{type})}\ \textbf{KR-spn-cons}$$

We have $\Delta; \Gamma \vdash M \Leftarrow A$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma \vdash \vec{M} : [M/x]K > \mathtt{type}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A.K > \mathtt{type}$ by rule **K-spn-cons**

4. We have $\mathcal{D} :: (\Omega; \Psi \vdash H \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash H \Rightarrow A)$. There are three cases to consider

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :: (\mathbf{c} : S \sqsubset A) \in \Sigma \\ \mathcal{D}_2 :: (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx})\end{array}}{(\Omega; \Psi \vdash \mathbf{c} \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash \mathbf{c} \Rightarrow A)}\ \textbf{TRS-const}$$

We have $(\mathbf{c} : A) \in \Sigma$ by signature formation rules on $\mathcal{D}_1$

We have $\Delta \vdash \Gamma : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash \mathbf{c} \Rightarrow A$ by rule **TS-cons**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :: ((x{:}S) \in \Psi) \sqsubset ((x{:}A) \in \Gamma) \\ \mathcal{D}_2 :: (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx})\end{array}}{(\Omega; \Psi \vdash x \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash x \Rightarrow A)}\ \textbf{TRS-x}$$

We have $(x{:}A) \in \Gamma$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta \vdash \Gamma : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash x \Rightarrow A$ by rule **TS-x**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :: ((b{:}\mathbf{w}[\vec{M}]) \in \Psi) \sqsubset ((b{:}\mathbf{w}[\vec{M}]) \in \Gamma) \\ \mathcal{D}_2 :: (\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \mathbf{w}[\vec{M}] > C) \\ \mathcal{D}_3 :: (\Omega; \Psi \vdash b : D \gg_1^k S) \sqsubset (\Delta; \Gamma \vdash b : C \gg_1^k A)\end{array}}{(\Omega; \Psi \vdash b.k \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash b.k \Rightarrow A)}\ \textbf{TRS-b}$$

We have $(b{:}\mathbf{w}[\vec{M}]) \in \Gamma$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma \vdash \mathbf{w}[\vec{M}] > C)$ by inductive hypothesis on $\mathcal{D}_2$

We have $\Delta; \Gamma \vdash b : C \gg_1^k A$ by inductive hypothesis on $\mathcal{D}_3$

Then $\Delta; \Gamma \vdash b.k \Rightarrow A$ by rule **TS-b**

5. We have $\mathcal{D} :: \ (\Omega; \Psi \vdash W[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash V[\vec{M}] > C)$. There are three cases to consider:

Case $\mathcal{D} = \dfrac{}{(\Omega; \Gamma \vdash D[\mathtt{nil}] > D) \sqsubset (\Delta; \Gamma \vdash C[\mathtt{nil}] > C)}$ **R-Inst-nil**

We have $\Delta; \Gamma \vdash C[\mathtt{nil}] > C$ by rule **Inst-nil**

Case $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: \ (\mathbf{w} : W \sqsubset V) \in \Sigma \\ \mathcal{D}_2 :: \ (\Omega; \Psi \vdash W[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash V[\vec{M}] > C)\end{array}}{(\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \mathbf{w}[\vec{M}] > C)}$ **R-Inst-const**

We have $(\mathbf{w}{:}V) \in \Sigma$ by signature formation rules and $\mathcal{D}_1$

We have $\Delta; \Gamma \vdash V[\vec{M}] > C$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash \mathbf{v}[\vec{M}] > c$ by rule **Inst-const**

Case $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: \ (\Omega; \Psi \vdash M \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A) \\ \mathcal{D}_2 :: \ (\Omega; \Psi \vdash ([M/x]W)[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash ([M/x]V)[\vec{M}] > C)\end{array}}{(\Omega; \Psi \vdash \Pi x{:}S.W[M; \vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A.V[M; \vec{M}]) > C}$ **R-Inst-pi**

We have $\Delta; \Gamma \vdash M \Leftarrow A$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma \vdash ([M/x]V)[\vec{M}] > C$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Psi \vdash \Pi x{:}A.V[M'\vec{M}] > C$ by rule **Inst-pi**

6. We have $\mathcal{D} :: \ (\Omega; \Psi \vdash b : D \gg_i^k S) \sqsubset (\Delta; \Gamma \vdash b : C \gg_i^k A)$. There are two cases to consider:

Case $\mathcal{D} = \dfrac{}{(\Omega; \Psi \vdash b : \Sigma x{:}S.D \gg_k^k S) \sqsubset (\Delta; \Gamma \vdash b : \Sigma x{:}A.C \gg_k^k A)}$ **R-Ext-stop**

We have $\Delta; \Gamma \vdash b : \Sigma x{:}A.C \gg_k^k A$ by rule **Ext-stop**.

Case $\mathcal{D} = \dfrac{\mathcal{D}' :: (\Omega; \Psi \vdash b : [b.i/x]D \gg^k_{i+1} S) \sqsubset (\Delta; \Gamma \vdash b : [b.i/x]C \gg^k_{i+1})}{(\Omega; \Psi \vdash b : \Sigma x{:}S'.D \gg^k_i S) \sqsubset (\Delta; \Gamma \vdash b : \Sigma x{:}A'.C \gg^k_i A)}$ **R-Ext-cont**

We have $\Delta; \Gamma \vdash b : [b.i/x]C \gg^k_{i+1}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta; \Gamma \vdash b : \Sigma x{:}A'.C \gg^k_i A$ by rule **Ext-cont**

7. We have $\mathcal{D} :: (\Omega; \Psi \vdash R \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash R \Rightarrow A)$. There are two cases to consider:

Case $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: (\Omega; \Psi \vdash H \Rightarrow S') \sqsubset (\Delta; \Gamma \vdash H \Rightarrow A') \\ \mathcal{D}_2 :: (\Omega; \Psi \vdash \vec{M} : S' > S) \sqsubset (\Delta; \Gamma \vdash \vec{M} : A' > A)\end{array}}{(\Omega; \Psi \vdash H\ \vec{M} \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash H\ \vec{M} \Rightarrow A)}$ **TRS-app**

We have $\Delta; \Gamma \vdash H \Rightarrow A'$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma \vdash \vec{M} : A' > A$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash H\ \vec{M} \Rightarrow A$ by rule **TS-app**

Case $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: ((u : \Psi'.S) \in \Omega) \sqsubset ((u : \Gamma'.A) \in \Delta) \\ \mathcal{D}_2 :: (\Omega; \Psi \vdash \sigma : \Psi') \sqsubset (\Delta; \Gamma \vdash \sigma : \Gamma')\end{array}}{(\Omega; \Psi \vdash u[\sigma] : [\sigma]S) \sqsubset (\Delta; \Gamma \vdash u[\sigma] : [\sigma]A)}$ **TRS-mvar**

We have $(u : \Gamma'.A) \in \Delta$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma \vdash \sigma : \Gamma'$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash u[\sigma] : [\sigma]A$ by rule **TS-mvar**

8. We have $(\Omega; \Psi \vdash \vec{M} : S' > S) \sqsubset (\Delta; \Gamma \vdash \vec{M} : A' > A)$. There are two cases to consider:

Case $\mathcal{D} = \dfrac{}{(\Omega; \Psi \vdash \mathtt{nil} : Q > Q) \sqsubset (\Delta; \Gamma \vdash \mathtt{nil} : P > P)}$ **TRC-spn-nil**

We have $\Delta; \Gamma \vdash \mathtt{nil} : P > P$ by rule **TC-spn-nil**

Case $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: (\Omega; \Psi \vdash M \Leftarrow S_1) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A_1) \\ \mathcal{D}_2 :: (\Omega; \Psi \vdash \vec{M} : [M/x]S_2 > Q) \sqsubset (\Delta; \Gamma \vdash \vec{M} : [M/x]A_2 > P)\end{array}}{(\Omega; \Psi \vdash (M; \vec{M}) : \Pi x{:}S_1.S_2 > Q) \sqsubset (\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A_1.A_2 > P)}$ **TRC-spn-cons**

We have $\Delta; \Gamma \vdash M \Leftarrow A_1$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma \vdash \vec{M} : [M/x]A_2 > P$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A_1.A_2 > P$ by rule **TC-spn-cons**

9. We have $\mathcal{D} :: (\Omega; \Psi \vdash M \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A)$. There are two cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 : (\Omega; \Psi \vdash R \Rightarrow Q') \sqsubset (\Delta; \Gamma \vdash R \Rightarrow P) \\ \mathcal{D}_2 : (\Omega; \Psi \vdash Q' \leq Q) \sqsubset (\Delta; \Gamma \vdash P)\end{array}}{(\Omega; \Psi \vdash R \Leftarrow Q) \sqsubset (\Delta; \Gamma \vdash R \Leftarrow Q)}\ \textbf{TRC-conv}$$

We have $\Delta; \Gamma \vdash R \Rightarrow P$ by inductive hypothesis on $\mathcal{D}_1$

Then $\Delta; \Gamma \vdash R \Leftarrow P$ by rule **TC-conv**

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' : (\Omega; \Psi, x{:}S \vdash M \Leftarrow S') \sqsubset (\Delta; \Gamma, x{:}A \vdash M \Leftarrow A')}{(\Omega; \Psi \vdash \lambda x.M \Leftarrow \Pi x{:}S.S') \sqsubset (\Delta; \Gamma \vdash \lambda x.M \Leftarrow \Pi x{:}A.A')}\ \textbf{TRC-lam}$$

We have $\Delta; \Gamma, x{:}A \vdash M \Leftarrow A'$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta; \Gamma \vdash \lambda x.M \Leftarrow \Pi x{:}A.A'$ by rule **TC-lam**

10. We have $(\Omega; \Psi \vdash D) \sqsubset (\Delta; \Gamma \vdash C : \texttt{block})$. There are two cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' : (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx})}{(\Omega; \Psi \vdash \cdot) \sqsubset (\Delta; \Gamma \vdash \cdot : \texttt{block})}\ \textbf{BR-empty}$$

We have $\Delta \vdash \Gamma : \texttt{ctx}$ by inductive hypothesis on $\mathcal{D}'$

We have $\Delta \vdash \cdot : \texttt{block}$ by rule **B-empty**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 : (\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \texttt{type}) \\ \mathcal{D}_2 : (\Omega; \Psi, x{:}S \vdash D) \sqsubset (\Delta; \Gamma, x{:}A \vdash C : \texttt{block})\end{array}}{(\Omega; \Psi \vdash \Sigma x{:}S.D) \sqsubset (\Delta; \Gamma \vdash \Sigma x{:}A.C : \texttt{block})}\ \textbf{BR-sigma}$$

We $\Delta; \Gamma \vdash A \Leftarrow \texttt{type}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma, x{:}A \vdash C : \texttt{block}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash \Sigma x{:}A.C : \texttt{block}$ by rule **B-sigma**

11. We have $\mathcal{D} ::\ (\Omega; \Psi \vdash W) \sqsubset (\Delta; \Gamma \vdash V : \mathtt{world})$. There are two cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :\ (\Omega; \Psi \vdash D) \sqsubset (\Delta; \Gamma \vdash C : \mathtt{block})}{(\Omega; \Psi \vdash D) \sqsubset (\Delta; \Gamma \vdash C : \mathtt{world})}\ \textbf{WR-conv}$$

We have $\Delta; \Gamma \vdash C : \mathtt{block}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta; \Gamma \vdash C : \mathtt{world}$ by rule **W-conv**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \mathtt{type}) \\ \mathcal{D}_2 :\ (\Omega; \Psi, x{:}S \vdash W) \sqsubset (\Delta; \Gamma, x{:}A \vdash V : \mathtt{world})\end{array}}{(\Omega; \Psi \vdash \Pi x{:}S.W) \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A.V : \mathtt{world})}\ \textbf{WR-pi}$$

We have $\Delta; \Gamma \vdash A \Leftarrow \mathtt{type}$ by induction hypothesis on $\mathcal{D}_2$

We have $\Delta; \Gamma, x{:}A \vdash V : \mathtt{world}$ by induction hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash \Pi x{:}A.V : \mathtt{world}$ by rule **W-pi**

12. We have $\mathcal{D} ::\ (\Omega \vdash G) \sqsubset (\Delta \vdash F : \mathtt{schema})$. There are three cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :\ (\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx})}{(\Omega \vdash \cdot) \sqsubset (\Delta \vdash \cdot : \mathtt{schema})}\ \textbf{SR-empty}$$

We have $\vdash \Delta : \mathtt{mctx}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta \vdash \cdot : \mathtt{schema}$ by rule **S-empty**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{ll}\mathcal{D}_1 :\ (\mathbf{w}{:}V) \in G & \mathcal{D}_3 :\ (\Omega \vdash H) \sqsubset (\Delta \vdash G : \mathtt{schema}) \\ \mathcal{D}_2 :\ \mathbf{w} \notin H & \mathcal{D}_4 :\ (\Omega; \cdot \vdash W) \sqsubset (\Delta; \cdot \vdash V : \mathtt{world})\end{array}}{(\Omega \vdash H + \mathbf{w}{:}W) \sqsubset (\Delta \vdash G : \mathtt{schema})}\ \textbf{SR-ext}$$

We have $\Delta \vdash G : \mathtt{schema}$ by inductive hypothesis on $\mathcal{D}_3$

13. We have $(\Omega \vdash \Psi : G) \sqsubset (\Delta \vdash \Gamma : F)$. There are three cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :\ (\Omega \vdash H) \sqsubset (\Delta \vdash G : \mathtt{schema})}{(\Omega \vdash \cdot : H) \sqsubset (\Delta \vdash \cdot : G)}\ \textbf{SRC-empty}$$

We have $\Delta \vdash G : \mathtt{schema}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta \vdash \cdot : G$ by rule **SC-empty**

Case $\mathcal{D} = \dfrac{\mathcal{D}' :: ((\psi : H) \in \Omega) \sqsubset ((\psi : G) \in \Delta)}{(\Omega \vdash \psi : H) \sqsubset (\Delta \vdash \psi : G)}$ **SRC-var**

We have $(\psi : G) \in \Delta$ by assumption $\mathcal{D}'$

Then $\Delta \vdash \psi : G$ by rule **SC-var**

Case $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: ((\mathbf{w}{:}W) \in H) \sqsubset ((\mathbf{w}{:}V) \in G) \\ \mathcal{D}_2 :: (\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G) \\ \mathcal{D}_3 :: (\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \mathbf{w}[\vec{M}] > C)\end{array}}{(\Omega \vdash (\Psi, b{:}\mathbf{w}[\vec{M}]) : H) \sqsubset (\Delta \vdash (\Gamma, b{:}\mathbf{w}[\vec{M}]) : G)}$ **SRC-ext**

We have $(\mathbf{w}{:}V) \in G$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta \vdash \Gamma : G$ by inductive hypothesis on $\mathcal{D}_2$

We have $\Delta; \Gamma \vdash \mathbf{w}[\vec{M}] > C$ by inductive hypothesis on $\mathcal{D}_3$

Then $\Delta \vdash (\Gamma, b{:}\mathbf{w}[\vec{M}]) : G$ by rule **SC-ext**

14. We have $(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx})$. There are four cases to consider:

Case $\mathcal{D} = \dfrac{\mathcal{D}' :: (\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx})}{(\Omega \vdash \cdot) \sqsubset (\Delta \vdash \cdot : \mathtt{ctx})}$ **CR-empty**

We have $\vdash \Delta : \mathtt{mctx}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta \vdash \cdot : \mathtt{ctx}$ by rule **C-empty**

Case $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: ((\psi : H) \in \Omega) \sqsubset ((\psi : G) \in \Delta) \\ \mathcal{D}_2 :: (\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx})\end{array}}{(\Omega \vdash \psi) \sqsubset (\Delta \vdash \psi : \mathtt{ctx})}$ **CR-var**

We have $(\psi : G) \in \Delta$ by assumption $\mathcal{D}_1$

We have $\vdash \Delta : \mathtt{mctx}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta \vdash \psi : \mathtt{ctx}$ by rule **C-var**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx}) \\ \mathcal{D}_2 :\ (\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \mathtt{type})\end{array}}{(\Omega \vdash \Psi, x{:}S) \sqsubset (\Delta \vdash (\Gamma, x{:}A) : \mathtt{ctx})}\ \textbf{CR-cons-x}$$

We have $\Delta \vdash \Gamma : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma \vdash A \Leftarrow \mathtt{type}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta \vdash (\Gamma, x{:}A) : \mathtt{ctx}$ by rule **C-cons-x**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx}) \\ \mathcal{D}_2 :\ (\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > C)\end{array}}{(\Omega \vdash \Psi, b{:}\mathbf{w}[\vec{M}]) \sqsubset (\Delta \vdash (\Gamma, b{:}\mathbf{w}[\vec{M}]) : \mathtt{ctx})}\ \textbf{CR-cons-b}$$

We have $\Delta \vdash \Gamma : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > C$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta \vdash (\Gamma, b{:}\mathbf{w}[\vec{M}]) : \mathtt{ctx}$ by rule **C-cons-b**

15. We have $(\Omega; \Psi_1 \vdash \sigma : \Psi_2) \sqsubset (\Delta; \Gamma_1 \vdash \sigma : \Gamma_2)$. There are four cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :\ (\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1 : \mathtt{ctx})}{(\Omega; \Psi_1 \vdash \cdot : \cdot) \sqsubset (\Delta; \Gamma_1 \vdash \cdot : \cdot)}\ \textbf{SubstR-empty}$$

We have $\Delta \vdash \Gamma_1 : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta; \Gamma_1 \vdash \cdot : \cdot$ by rule **Subst-empty**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ ((\psi : H) \in \Omega) \sqsubset ((\psi : G) \in \Delta) \\ \mathcal{D}_2 :\ (\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1 : \mathtt{ctx})\end{array}}{(\Omega; \Psi_1 \vdash \mathtt{id}_\psi : \psi) \sqsubset (\Delta; \Gamma_1 \vdash \mathtt{id}_\psi : \psi)}\ \textbf{SubstR-id}$$

We have $(\psi : G) \in \Delta$ by assumption $\mathcal{D}_1$

We have have $\Delta \vdash \Gamma_1 : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma_1 \vdash \mathtt{id}_\psi : \psi$ by rule **Subst-id**

$$\textbf{Case } \mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: (\Omega; \Psi_1 \vdash \sigma : \Psi_2) \sqsubset (\Delta; \Gamma_1 \vdash \sigma : \Gamma_2) \\ \mathcal{D}_2 :: (\Omega; \Psi_1 \vdash M \Leftarrow [\sigma]S) \sqsubset (\Delta; \Gamma_1 \vdash M \Leftarrow [\sigma]A)\end{array}}{\big(\Omega; \Psi_1 \vdash (\sigma, M) : (\Psi_2, x{:}S)\big) \sqsubset \big(\Delta; \Gamma_1 \vdash (\sigma, M) : (\Gamma_2, x{:}A)\big)}\ \textbf{SubstR-tm}$$

We have $\Delta; \Gamma_1 \vdash \sigma : \Gamma_2$ by inductive hypothesis on D_2

We have $\Delta; \Gamma_1 \vdash M \Leftarrow [\sigma]A$ by inductive hypothesis on D_2

Then $\Delta; \Gamma_1 \vdash (\sigma, M) : (\Gamma_2, x{:}A)$ by rule **Subst-tm**

$$\textbf{Case } \mathcal{D} = \dfrac{\begin{array}{ll}\mathcal{D}_1 & (\Omega; \Psi_1 \vdash \sigma : \Psi_2) \sqsubset (\Delta; \Gamma_1 \vdash \sigma : \Gamma_2) \\ \mathcal{D}_2 & (\Omega; \Psi_2 \vdash \mathbf{w}[\vec{M}'] > D) \sqsubset (\Delta; \Gamma_2 \vdash \mathbf{w}[\vec{M}'] > C) \\ \mathcal{D}_3 & (\Omega; \Psi_1 \vdash \vec{M} \Leftarrow [\sigma]D) \sqsubset (\Delta; \Gamma_1 \vdash \vec{M} \Leftarrow [\sigma]C)\end{array}}{(\Omega; \Psi_1 \vdash (\sigma, \vec{M}) : (\Psi_2, b{:}\mathbf{w}[\vec{M}'])) \sqsubset \big(\Delta; \Gamma_1 \vdash (\sigma, \vec{M}) : (\Gamma_2, b{:}\mathbf{w}[\vec{M}'])\big)}\ \textbf{SubstR-spn}$$

We have $\Delta; \Gamma_1 \vdash \sigma : \Gamma_2$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma_2 \vdash \mathbf{w}[\vec{M}'] > D$ by inductive hypothesis on $\mathcal{D}_2$

We have $\Delta; \Gamma_1 \vdash \vec{M} \Leftarrow [\sigma]D$ by inductive hypothesis on $\mathcal{D}_3$

Then $\Delta; \Gamma_1 \vdash (\sigma, \vec{M}) : (\Gamma_2, b{:}\mathbf{w}[\vec{M}'])$ by rule **Subst-spn**

16. We have $\mathcal{D} :: (\Omega; \Psi \vdash \vec{M} \Leftarrow D) \sqsubset (\Delta; \Gamma \vdash \vec{M} \Leftarrow C)$. There are two cases to consider:

$$\textbf{Case } \mathcal{D} = \dfrac{\mathcal{D}' : (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx})}{(\Omega; \Psi \vdash \mathtt{nil} \Leftarrow \cdot) \sqsubset (\Delta; \Gamma \vdash \mathtt{nil} \Leftarrow \cdot)}\ \textbf{ChkR-spn-nil}$$

We have $\Delta \vdash \Gamma : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}'$

We have $\Delta; \Gamma \vdash \mathtt{nil} \Leftarrow \cdot$ by rule **Chk-spn-nil**

$$\textbf{Case } \mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: (\Omega; \Psi \vdash M \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A) \\ \mathcal{D}_2 :: (\Omega; \Psi \vdash \vec{M} \Leftarrow [M/x]D) \sqsubset (\Delta; \Gamma \vdash \vec{M} \Leftarrow [M/x]C)\end{array}}{(\Omega; \Psi \vdash (M; \vec{M}) \Leftarrow \Sigma x{:}S.D) \sqsubset (\Delta; \Gamma \vdash (M; \vec{M}) \Leftarrow \Sigma x{:}A.C)}\ \textbf{ChkR-spn-sigma}$$

We have $\Delta; \Gamma \vdash M \Leftarrow A$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma \vdash \vec{M} \Leftarrow [M/x]C$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash (M; \vec{M}) \Leftarrow \Sigma x{:}A.C$ by rule **Chk-spn-sigma**

17. We have $\mathcal{D} ::\ (\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx})$. There are two cases to consider:

Case $\mathcal{D} = \dfrac{}{(\vdash \cdot) \sqsubset (\vdash \cdot : \mathtt{mctx})}$ **MCR-nil**

We have $\vdash \cdot : \mathtt{mctx}$ by rule **MC-nil**

Case $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :\ (\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx}) \\ \mathcal{D}_2 :\ (\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \mathtt{mtype})\end{array}}{(\vdash (\Omega, X{:}\mathcal{S}) \sqsubset (\vdash (\Delta, X{:}\mathcal{A}) : \mathtt{mctx})}$ **MCR-cons**

We have $\vdash \Delta : \mathtt{mctx}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta \vdash \mathcal{A} : \mathtt{mtype}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\vdash (\Delta, X{:}\mathcal{A}) : \mathtt{mctx}$ by rule **MC-cons**

18. We have $\mathcal{D} ::\ (\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \mathtt{mtype})$. There are three cases to consider:

Case $\mathcal{D} = \dfrac{\mathcal{D}' :\ (\Omega; \Psi \vdash Q) \sqsubset (\Delta; \Gamma \vdash P \Leftarrow \mathtt{type})}{\big(\Omega \vdash (\Psi.Q)\big) \sqsubset \big(\Delta \vdash (\Gamma.P) : \mathtt{mtype}\big)}$ **MTR-tp**

We have $\Delta; \Gamma \vdash P \Leftarrow \mathtt{type}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta \vdash (\Gamma.P) : \mathtt{mtype}$ by rule **MT-tp**

Case $\mathcal{D} = \dfrac{\mathcal{D}' :\ (\Omega \vdash H) \sqsubset (\Delta \vdash G : \mathtt{schema})}{(\Omega \vdash H) \sqsubset (\Delta \vdash G : \mathtt{mtype})}$ **MTR-schema**

We have $\Delta \vdash G : \mathtt{schema}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta \vdash G : \mathtt{mtype}$ by rule **MT-schema**

Case $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1 : \mathtt{ctx}) \\ \mathcal{D}_2 :\ (\Omega \vdash \Psi_2) \sqsubset (\Delta \vdash \Gamma_2 : \mathtt{ctx})\end{array}}{\big(\Omega \vdash (\Psi_1.\Psi_2)\big) \sqsubset \big(\Delta \vdash (\Gamma_1.\Gamma_2) : \mathtt{mtype}\big)}$ **MTR-subst**

We have $\Delta \vdash \Gamma_1 : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta \vdash \Gamma_2 : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta \vdash (\Gamma_1.\Gamma_2) : \mathtt{mtype}$ by rule **MT-subst**

19. We have $\mathcal{D} ::\ (\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})$. There are three cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' ::\ (\Omega; \Psi \vdash R \Leftarrow Q) \sqsubset (\Delta; \Gamma \vdash R \Leftarrow P)}{\big(\Omega \vdash (\hat{\Psi}.R) : (\Psi.Q)\big) \sqsubset \big(\Delta \vdash (\hat{\Gamma}.R) : (\Gamma.P)\big)}\ \textbf{MOftR-tm}$$

We have $\Delta; \Gamma \vdash R \Leftarrow P$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta \vdash (\hat{\Gamma}.R) : (\Gamma.P)$ by rule **MOft-tm**

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' ::\ (\Omega; \Psi_1 \vdash \sigma : \Psi_2) \sqsubset (\Delta; \Gamma_1 \vdash \sigma : \Gamma_2)}{\big(\Omega \vdash (\hat{\Psi}_1.\sigma) : (\Psi_1.\Psi_2)\big) \sqsubset \big(\Delta \vdash (\hat{\Gamma}_1.\sigma) : (\Gamma_1.\Gamma_2)\big)}\ \textbf{MOftR-subst}$$

We have $\Delta; \Gamma_1 \vdash \sigma : \Gamma_2$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta \vdash (\hat{\Gamma}_1.\sigma) : (\Gamma_1.\Gamma_2)$ by rule **MOft-subst**

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' ::\ (\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G)\ \text{(schema checking)}}{(\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G)\ \text{(msort checking)}}\ \textbf{MOftR-ctx}$$

We have $\Delta \vdash \Gamma : G$ (schema checking) by inductive hypothesis on $\mathcal{D}'$

Then $\Delta \vdash \Gamma : G$ (meta-type checking) by rule **MOft-ctx**

20. We have $\mathcal{D} ::\ (\Omega_1 \vdash \rho : \Omega_2) \sqsubset (\Delta_1 \vdash \theta : \Delta_2)$. There are two cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' ::\ (\vdash \Omega_1) \sqsubset (\vdash \Delta_1 : \texttt{mctx})}{(\Omega_1 \vdash \cdot : \cdot) \sqsubset (\Delta_1 \vdash \cdot : \cdot)}\ \textbf{MSubstR-nil}$$

We have $\vdash \Delta_1 : \texttt{mctx}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta_1 \vdash \cdot : \cdot$ by rule **MSubst-nil**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 ::\ (\Omega_1 \vdash \rho : \Omega_2) \sqsubset (\Delta_1 \vdash \theta : \Delta_2) \\ \mathcal{D}_2 ::\ (\Omega_1 \vdash \mathcal{N} : [\![\rho]\!]\mathcal{S}) \sqsubset (\Delta_1 \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A})\end{array}}{\big(\Omega_1 \vdash (\rho, \mathcal{N}) : (\Omega_2, X{:}\mathcal{S})\big) \sqsubset \big(\Delta_1 \vdash (\theta, \mathcal{M}) : (\Delta_2, X{:}\mathcal{A})\big)}\ \textbf{MSubstR-cons}$$

We have $\Delta_1 \vdash \theta : \Delta_2$ by inductive hypothesis on $\mathcal{D}_2$

We have $\Delta_1 \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta_1 \vdash (\theta, \mathcal{M}) : (\Delta_2, X{:}\mathcal{A})$ by rule **MSubst-cons**

21. We have $\mathcal{D} :\ (\Omega; \Psi \vdash S_1 \leq S_2) \sqsubset (\Delta; \Gamma \vdash A)$. There are four cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :: (\mathrm{LFR}\ \mathbf{s}_1 \leq \mathbf{s}_2 \sqsubset \mathbf{a} : L) \in \Sigma \\ \mathcal{D}_2 :: (\Omega; \Psi \vdash \vec{M} : L > \mathtt{sort}) \sqsubset (\Delta; \Gamma \vdash \vec{M} : K > \mathtt{type})\end{array}}{(\Omega; \Psi \vdash \mathbf{s}_1\ \vec{M} \leq \mathbf{s}_2\ \vec{M}) \sqsubset (\Delta; \Gamma \vdash \mathbf{a}\ \vec{M})}\ \textbf{Sub-atom}$$

We have $(\mathrm{LF}\ \mathbf{a} : K) \in \Sigma$ by signature formation rules

We have $\Delta; \Gamma \vdash \vec{M} : K > \mathtt{type}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta \vdash \mathbf{a}\ \vec{M} \Leftarrow \mathtt{type}$ by rule **T-atom**

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D} :: (\Omega; \Psi \vdash Q) \sqsubset (\Delta; \Gamma \vdash P \Leftarrow \mathtt{type})}{(\Omega; \Psi \vdash Q \leq Q) \sqsubset (\Delta; \Gamma \vdash P)}\ \textbf{Sub-refl}$$

We have $\Delta; \Gamma \vdash P \Leftarrow \mathtt{type}$ by inductive hypothesis on $\mathcal{D}'$

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :: (\Omega; \Psi \vdash Q_1 \leq Q_2) \sqsubset (\Delta; \Gamma \vdash P) \\ \mathcal{D}_2 :: (\Omega; \Psi \vdash Q_2 \leq Q_3) \sqsubset (\Delta; \Gamma \vdash P)\end{array}}{(\Omega; \Psi \vdash Q_1 \leq Q_3) \sqsubset (\Delta; \Gamma \vdash P)}\ \textbf{Sub-trans}$$

We have $\Delta; \Gamma \vdash P \Leftarrow \mathtt{type}$ by inductive hypothesis on $\mathcal{D}_1$

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :: (\Omega; \Psi \vdash S_2 \leq S_1) \sqsubset (\Delta; \Gamma \vdash A) \\ \mathcal{D}_2 :: (\Omega; \Psi, x{:}S_2 \vdash S_1' \leq S_2') \sqsubset (\Delta; \Gamma, x{:}A \vdash A')\end{array}}{(\Omega; \Psi \vdash \Pi x{:}S_1.S_1' \leq \Pi x{:}S_2.S_2') \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A.A')}\ \textbf{Sub-pi}$$

We have $\Delta; \Gamma \vdash A \Leftarrow \mathtt{type}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma, x{:}A \vdash A' \Leftarrow \mathtt{type}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash \Pi x{:}A.A' \Leftarrow \mathtt{type}$ by rule **T-pi**

22. We have $\mathcal{D} :: (\Omega; \Psi \vdash D_1 \leq D_2) \sqsubset (\Delta; \Gamma \vdash C : \mathtt{block})$. There are two cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :: (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma)}{(\Omega; \Psi \vdash \cdot \leq \cdot) \sqsubset (\Delta; \Psi \vdash \cdot)}\ \textbf{Sub-Bnil}$$

We have $\Delta \vdash \Gamma : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta; \Gamma \vdash \cdot : \mathtt{block}$ by rule **B-empty**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1 :\ (\Omega; \Psi \vdash S_1 \leq S_2) \sqsubset (\Delta; \Gamma \vdash A) \\ \mathcal{D}_2 :\ (\Omega; \Psi, x{:}S_1 \vdash D_1 \leq D_2) \sqsubset (\Delta; \Gamma \vdash C)\end{array}}{(\Omega; \Psi \vdash \Sigma x{:}S_1.D_1 \leq \Sigma x{:}S_2.D_2) \sqsubset (\Delta; \Gamma \vdash \Sigma x{:}A.C)}\ \textbf{Sub-sigma}$$

We have $\Delta; \Gamma \vdash A \Leftarrow \texttt{type}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma, x{:}A \vdash C : \texttt{block}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash \Sigma x{:}A.C : \texttt{block}$ by rule **B-sigma**

23. We have $\mathcal{D} :\ (\Omega; \Psi \vdash W_1 \leq W_2) \sqsubset (\Delta; \Gamma \vdash V : \texttt{world})$. There are two cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :\ (\Omega; \Psi \vdash D_1 \leq D_2) \sqsubset (\Delta; \Psi \vdash C : \texttt{block})}{(\Omega; \Psi \vdash D_1 \leq D_2) \sqsubset (\Delta; \Psi \vdash C : \texttt{world})}\ \textbf{SubW-conv}$$

We have $\Delta; \Psi \vdash C : \texttt{block}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta; \Psi \vdash C : \texttt{world}$ by rule **W-conv**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1 :\ (\Omega; \Psi \vdash S_2 \leq S_1) \sqsubset (\Delta; \Gamma \vdash A : \texttt{type}) \\ \mathcal{D}_2 :\ (\Omega; \Psi, x{:}S_2 \vdash W_1 \leq W_2) \sqsubset (\Delta; \Gamma \vdash V : \texttt{world})\end{array}}{(\Omega; \Psi \vdash \Pi x{:}S_1.W_1 \leq \Pi x{:}S_2.W_2) \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A.V : \texttt{world})}\ \textbf{SubW-pi}$$

We have $\Delta; \Gamma \vdash A \Leftarrow \texttt{type}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Gamma, x{:}A \vdash V : \texttt{world}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Gamma \vdash \Pi x{:}A.V : \texttt{world}$ by rule **W-pi**

24. We have $\mathcal{D} :\ (\Omega; \Psi \vdash H_1 \leq H_2) \sqsubset (\Delta; \Gamma \vdash G : \texttt{schema})$. There are two cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :\ (\Omega \vdash H) \sqsubset (\Delta \vdash G : \texttt{schema})}{(\Omega \vdash \cdot \leq H) \sqsubset (\Delta \vdash G)}\ \textbf{SubS-nil}$$

We have $\Delta \vdash G : \texttt{schema}$ by inductive hypothesis on $\mathcal{D}'$

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{ll}\mathcal{D}_1 :: (\mathbf{w}{:}V \in G) & \mathcal{D}_3 :: (\Omega \vdash H_1 \leq H_2) \sqsubset (\Delta \vdash G : \texttt{schema}) \\ \mathcal{D}_2 :: (\mathbf{w} \notin H_i) & \mathcal{D}_4 :: (\Omega; \cdot \vdash W_1 \leq W_2) \sqsubset (\Delta; \cdot \vdash V : \texttt{world})\end{array}}{(\Omega \vdash H_1 + \mathbf{w}{:}W_1 \leq H_2 + \mathbf{w}{:}W_2) \sqsubset (\Delta \vdash G)}\ \textbf{SubS-sum}$$

We have $\Delta \vdash G : \texttt{schema}$ by inductive hypothesis on $\mathcal{D}_1$

25. We have $\mathcal{D} :: (\Omega \vdash \mathcal{S}_1 \leq \mathcal{S}_2) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype})$. There are three cases to consider:

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :: (\Omega; \Psi \vdash S_1 \leq S_2) \sqsubset (\Delta; \Gamma \vdash A)}{(\Omega \vdash \Psi.S_1 \leq \Psi.S_2) \sqsubset (\Delta \vdash \Gamma.A)}\ \textbf{SubM-tp}$$

We have $\Delta; \Gamma \vdash A \Leftarrow \texttt{type}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta \vdash \Gamma.A : \texttt{mtype}$ by rule **MT-tp**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :: (\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1) \\ \mathcal{D}_2 :: (\Omega \vdash \Psi_2) \sqsubset (\Delta \vdash \Gamma_2)\end{array}}{(\Omega \vdash \Psi_1.\Psi_2 \leq \Psi_1.\Psi_2) \sqsubset (\Delta \vdash \Gamma_1.\Gamma_2)}\ \textbf{SubM-subst}$$

We have $\Delta \vdash \Gamma_1 : \texttt{ctx}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta \vdash \Gamma_2 : \texttt{ctx}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta \vdash \Gamma_1.\Gamma_2 : \texttt{mtype}$ by rule **MT-subst**

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :: (\Omega \vdash H_1 \leq H_2) \sqsubset (\Delta \vdash G : \texttt{schema})}{(\Omega \vdash H_1 \leq H_2) \sqsubset (\Delta \vdash G : \texttt{mtype})}\ \textbf{SubM-schema}$$

We have $\Delta \vdash G : \texttt{schema}$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta \vdash G : \texttt{mtype}$ by rule **MT-schema**

■

4.2 Conservativity for computation-level

Theorem 4.2 (Conservativity for computation-level)**.**

1. If $(\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx})$, then $\Delta \vdash \Xi : \texttt{cctx}$.

2. If $(\Omega; \Phi \vdash \zeta) \sqsubset (\Delta; \Xi \vdash \tau : \texttt{ctype})$, then $\Delta; \Xi \vdash \tau : \texttt{ctype}$.

3. If $(\Omega; \Phi \vdash f : \zeta) \sqsubset (\Delta; \Xi \vdash e : \tau)$, then $\Delta; \Xi \vdash e : \tau$.

4. If $(\Omega; \Phi \vdash c : \zeta) \sqsubset (\Delta; \Xi \vdash b : \tau)$, then $\Delta; \Xi \vdash b : \tau$.

5. If $(\Omega; \Phi \vdash \zeta_1 \leq \zeta_2) \sqsubset (\Delta; \Xi \vdash \tau)$, then $\Delta; \Xi \vdash \tau$.

Proof.

By simultaneous induction on the given derivation $\mathcal{D}$.

1. We have $\mathcal{D} :: (\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx})$. There are two cases to consider:

   Case $\mathcal{D} = \dfrac{\mathcal{D}' :: (\vdash \Omega) \sqsubset (\vdash \Delta : \texttt{mctx})}{(\Omega \vdash \cdot) \sqsubset (\Delta \vdash \cdot : \texttt{cctx})}$ **CCR-nil**

   We have $\vdash \Delta : \texttt{mctx}$ by Theorem 4.1 on $\mathcal{D}'$

   Then $\Delta \vdash \cdot : \texttt{cctx}$ by rule **CC-nil**

   Case $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: (\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx}) \\ \mathcal{D}_2 :: (\Omega; \Phi \vdash \zeta) \sqsubset (\Delta; \Xi \vdash \tau : \texttt{ctype})\end{array}}{(\Omega \vdash \Phi, y{:}\zeta) \sqsubset (\Delta \vdash (\Xi, y{:}\tau) : \texttt{cctx})}$ **CCR-cons**

   We have $\Delta \vdash \Xi : \texttt{cctx}$ by inductive hypothesis on $\mathcal{D}_1$

   We have $\Delta; \Xi \vdash \tau : \texttt{ctype}$ by inductive hypothesis on $\mathcal{D}_2$

   Then $\Delta \vdash (\Xi, y{:}\tau) : \texttt{cctx}$ by rule **CC-cons**

2. We have $\mathcal{D} :: (\Omega; \Phi \vdash \zeta) \sqsubset (\Delta; \Xi \vdash \tau : \texttt{ctype})$. There are three cases to consider:

   Case $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: (\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx}) \\ \mathcal{D}_2 :: (\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype})\end{array}}{(\Omega; \Phi \vdash [\mathcal{S}]) \sqsubset (\Delta; \Xi \vdash [\mathcal{A}] : \texttt{ctype})}$ **CTR-meta**

We have $\Delta \vdash \Xi : \mathtt{cctx}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta \vdash \mathcal{A} : \mathtt{mtype}$ by Theorem 4.1 on $\mathcal{D}_2$

Then $\Delta; \Xi \vdash [\mathcal{A}] : \mathtt{ctype}$ by rule **CR-meta**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :: (\Omega; \Phi \vdash \zeta_1) \sqsubset (\Delta; \Xi \vdash \tau_1 : \mathtt{ctype}) \\ \mathcal{D}_2 :: (\Omega; \Phi \vdash \zeta_2) \sqsubset (\Delta; \Xi \vdash \tau_2 : \mathtt{ctype})\end{array}}{(\Omega; \Phi \vdash \zeta_1 \to \zeta_2) \sqsubset \Delta; \Xi \vdash \tau_1 \to \tau_2 : \mathtt{ctype})}\ \textbf{CTR-arr}$$

We have $\Delta; \Xi \vdash \tau_1 : \mathtt{ctype}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Xi \vdash \tau_2 : \mathtt{ctype}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Xi \vdash \tau_1 \to \tau_2 : \mathtt{ctype}$ by rule **CT-arr**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :: (\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \mathtt{mtype}) \\ \mathcal{D}_2 :: (\Omega, X{:}\mathcal{S}; \Phi \vdash \zeta) \sqsubset (\Delta, X{:}\mathcal{A}; \Xi \vdash \tau : \mathtt{ctype})\end{array}}{(\Omega; \Phi \vdash \Pi X{:}\mathcal{S}.\zeta) \sqsubset (\Delta; \Xi \vdash \Pi X{:}\mathcal{A}.\tau : \mathtt{ctype})}\ \textbf{CTR-pi}$$

We have $\Delta \vdash \mathcal{A} : \mathtt{mtype}$ by Theorem 4.1 on $\mathcal{D}_1$

We have $\Delta, X{:}\mathcal{A}; \Xi \vdash \tau : \mathtt{ctype}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Xi \vdash \Pi X{:}\mathcal{A}.\tau : \mathtt{ctype}$ by rule **CT-pi**

3. We have $\mathcal{D} :: (\Omega; \Phi \vdash f : \zeta) \sqsubset (\Delta; \Xi \vdash e : \tau)$. There eight cases to consider

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :: (\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \mathtt{cctx}) \\ \mathcal{D}_2 :: ((y{:}\zeta) \in \Phi) \sqsubset ((y{:}\tau) \in \Xi)\end{array}}{(\Omega; \Phi \vdash y : \zeta) \sqsubset (\Delta; \Xi \vdash y : \tau)}\ \textbf{CTR-var}$$

We have $\Delta \vdash \Xi : \mathtt{cctx}$ by inductive hypothesis on $\mathcal{D}_1$

We have $(y{:}\tau) \in \Xi$ by assumption $\mathcal{D}_2$

Then $\Delta \vdash y : \tau$ by rule **CT-var**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A}) \\ \mathcal{D}_2 :\ (\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx})\end{array}}{(\Omega; \Phi \vdash [\mathcal{N}] : [\mathcal{S}]) \sqsubset (\Delta; \Xi \vdash [\mathcal{M}] : [\mathcal{A}])}\ \textbf{CTR-box}$$

We have $\Delta \vdash \mathcal{M} : \mathcal{A}$ by Theorem 4.1 on $\mathcal{D}_1$

We have $\Delta \vdash \Xi : \texttt{cctx}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Xi \vdash [\mathcal{M}] : [\mathcal{A}]$ by rule **CT-box**

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :\ (\Omega; \Phi, y{:}\zeta_1 \vdash f : \zeta_2) \sqsubset (\Delta; \Xi, y{:}\tau_1 \vdash e : \tau_2)}{(\Omega; \Phi \vdash (\texttt{fn}\ y{:}\zeta_1 \Rightarrow f) : \zeta_1 \to \zeta_2) \sqsubset (\Delta; \Xi \vdash (\texttt{fn}\ y{:}\tau_1 \Rightarrow e) : \tau_1 \to \tau_2)}\ \textbf{CTR-fn}$$

We have $\Delta; \Xi, y{:}\tau_1 \vdash e : \tau_2$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta; \Xi \vdash (\texttt{fn}\ y{:}\tau_1 \Rightarrow e) : \tau_1 \to \tau_2$ by rule **CT-fn**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega; \Phi \vdash f_1 : \zeta_2 \to \zeta_1) \sqsubset (\Delta; \Xi \vdash e_1 : \tau_2 \to \tau_1) \\ \mathcal{D}_2 :\ (\Omega; \Phi \vdash f_2 : \zeta_2) \sqsubset (\Delta; \Xi \vdash e_2 : \tau_2)\end{array}}{(\Omega; \Phi \vdash f_1\ f_2 : \zeta_1) \sqsubset (\Delta; \Xi \vdash e_1\ e_2 : \tau_1)}\ \textbf{CTR-app}$$

We have $\Delta; \Xi \vdash e_1 : \tau_2 \to \tau_1$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Xi \vdash e_2 : \tau_2$ by inductive hypothesis on $\mathcal{D}_2$

Then $(\Delta; \Xi \vdash e_1\ e_2 : \tau_1)$ by rule **CT-app**

$$\textbf{Case } \mathcal{D} = \frac{\mathcal{D}' :\ (\Omega, X{:}\mathcal{S}; \Phi \vdash f : \zeta) \sqsubset (\Delta, X{:}\mathcal{A}; \Xi \vdash e : \tau)}{(\Omega; \Phi \vdash (\texttt{mlam}\ X{:}\mathcal{S} \Rightarrow f) : \Pi X{:}\mathcal{S}.\zeta) \sqsubset (\Delta; \Xi \vdash (\texttt{mlam}\ X{:}\mathcal{A} \Rightarrow e) : \Pi X{:}\mathcal{A}.\tau)}\ \textbf{CTR-mlam}$$

We have $\Delta, X{:}\mathcal{A}; \Xi \vdash e : \tau$ by inductive hypothesis on $\mathcal{D}'$

Then $\Delta; \Xi \vdash (\texttt{mlam}\ X{:}\mathcal{A} \Rightarrow e) : \Pi X{:}\mathcal{A}.\tau$ by rule **CT-mlam**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega; \Phi \vdash f : \Pi X{:}\mathcal{S}.\zeta) \sqsubset (\Delta; \Xi \vdash e : \Pi X{:}\mathcal{A}.\tau) \\ \mathcal{D}_2 :\ (\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})\end{array}}{(\Omega; \Phi \vdash f\ \mathcal{N} : [\![\mathcal{N}/X]\!]\zeta) \sqsubset (\Delta; \Xi \vdash e\ \mathcal{M} : [\![\mathcal{M}/X]\!]\tau)}\ \textbf{CTR-mapp}$$

We have $\Delta; \Xi \vdash e : \Pi X{:}\mathcal{A}.\tau$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta \vdash \mathcal{M} : \mathcal{A}$ by Theorem 4.1 on $\mathcal{D}_2$

Then $\Delta; \Xi \vdash e\ [\mathcal{M}] : [\![\mathcal{M}/X]\!]\tau$ by rule **CT-mapp**

$$\textbf{Case}\ \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega; \Phi \vdash f_1 : [\mathcal{S}]) \sqsubset (\Delta; \Xi \vdash e_1 : [\mathcal{A}]) \\ \mathcal{D}_2 :\ (\Omega, X{:}\mathcal{S}; \Phi \vdash f_2 : \zeta) \sqsubset (\Delta, X{:}\mathcal{A}; \Xi \vdash e_2 : \tau)\end{array}}{(\Omega; \Phi \vdash (\texttt{let}\ [X] = f_1\ \texttt{in}\ f_2) : \zeta) \sqsubset (\Delta; \Xi \vdash (\texttt{let}\ [X] = e_1\ \texttt{in}\ e_2) : \tau)}\ \textbf{CTR-let}$$

We have $\Delta; \Xi \vdash e_1 : [\mathcal{A}]$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta, X{:}\mathcal{A}; \Xi \vdash e_2 : \tau$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Xi \vdash (\texttt{let}\ [X] = e_1\ \texttt{in}\ e_2) : \tau$ by rule **CT-let**

$$\textbf{Case}\ \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ \tau = \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0 \\ \mathcal{D}_2 :\ \zeta = \Pi\Omega_0.\Pi X_0{:}\mathcal{S}_0.\zeta_0 \\ \mathcal{D}_3 :\ (\cdot; \cdot \vdash \zeta) \sqsubset (\cdot; \cdot \vdash \tau : \texttt{ctype}) \\ \mathcal{D}_4 :\ (\Omega \vdash \rho : \Omega_0) \sqsubset (\Delta \vdash \theta : \Delta_0) \\ \mathcal{D}_5 :\ (\Omega \vdash \mathcal{N} : [\![\rho]\!]\mathcal{S}_0) \sqsubset (\Delta \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}_0) \\ \mathcal{D}_6 :\ (\Omega; \Phi \vdash^{[\![\rho]\!]\mathcal{S}_0} c_i : \zeta) \sqsubset (\Delta; \Xi \vdash^{[\![\theta]\!]\mathcal{A}_0} b_i : \tau)\ (\text{for all } c_i \in \vec{c})\end{array}}{(\Omega; \Phi \vdash (\texttt{case}^{\zeta}\ [\mathcal{N}]\ \texttt{of}\ \vec{c}) : [\![\rho, \mathcal{N}/X_0]\!]\zeta_0) \sqsubset (\Delta; \Xi \vdash (\texttt{case}^{\tau}\ [\mathcal{M}]\ \texttt{of}\ \vec{b}) : [\![\theta, \mathcal{M}/X_0]\!]\tau_0)}\ \textbf{CTR-case}$$

We have $\tau = \Pi X_0{:}\mathcal{A}_0.\tau_0$ by assumption $\mathcal{D}_1$

We have $\cdot; \cdot \vdash \tau : \texttt{ctype}$ by inductive hypothesis on $\mathcal{D}_3$

We have $\Delta \vdash \theta : \Delta_0$ by Theorem 4.1 on $\mathcal{D}_4$

We have $\Delta \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}_0$ by inductive hypothesis on $\mathcal{D}_5$

We have $\Delta; \Xi \vdash^{[\![\theta]\!]\mathcal{A}_0} b_i : \tau)$ (for all $c_i \in \vec{c}$ by inductive hypothesis on $\mathcal{D}_6$

Then $\Delta; \Xi \vdash (\texttt{case}^{\tau}\ [\mathcal{M}]\ \texttt{of}\ \vec{b}) : [\![\theta, \mathcal{M}/X_0]\!]\tau_0$ by rule **CT-case**

4. We have $\mathcal{D} ::\ (\Omega; \Phi \vdash c : \zeta) \sqsubset (\Delta; \Xi \vdash b : \tau)$. There is only one case to consider:

$$\textbf{Case}\ \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega_i \vdash \rho : \Omega_0) \sqsubset (\Delta_i \vdash \theta : \Delta_0) \\ \mathcal{D}_2 :\ (\Omega_i \vdash \mathcal{N}_0 : [\![\rho]\!]\mathcal{S}_0) \sqsubset (\Delta_i \vdash \mathcal{M}_0 : [\![\theta]\!]\mathcal{A}_0) \\ \mathcal{D}_3 :\ (\Omega' \vdash \rho'_i : \Omega_i) \sqsubset (\Delta' \vdash \theta'_i : \Delta_i) \\ \mathcal{D}_4 :\ (\Omega' \vdash \rho' : \Omega) \sqsubset (\Delta' \vdash \theta' : \Delta) \\ \mathcal{D}_5 :\ (\Omega' \vdash [\![\rho']\!]\mathcal{S} = [\![\rho'_i]\!][\![\rho_i]\!]\mathcal{S}_0) \sqsubset (\Delta' \vdash [\![\theta']\!]\mathcal{A} = [\![\theta'_i]\!][\![\theta_i]\!]\mathcal{A}_0) \\ \mathcal{D}_6 :\ (\Omega'; [\![\rho']\!]\Phi \vdash [\![\rho', \rho'_i]\!]f_i : [\![\rho'_i]\!][\![\rho_i]\!]\zeta_0) \sqsubset (\Delta'; [\![\theta']\!]\Xi \vdash [\![\theta', \theta'_i]\!]e_i : [\![\theta'_i]\!][\![\theta_i]\!]\tau_0)\end{array}}{(\Omega; \Phi \vdash^{\mathcal{S}} (\Omega_i; [\mathcal{N}_i] \mapsto f_i) : \Pi\Omega_0.\Pi X_0{:}\mathcal{S}_0.\zeta_0) \sqsubset (\Delta; \Xi \vdash^{\mathcal{A}} (\Delta_i; [\mathcal{M}_i] \mapsto e_i) : \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0)}\ \textbf{CTR-branch}$$

We have $\Delta_i \vdash \theta : \Delta_0$ by Theorem 4.1 on $\mathcal{D}_1$

We have $\Delta_i \vdash \mathcal{M}_0 : [\![\theta]\!]\mathcal{A}_0$ by Theorem 4.1 on $\mathcal{D}_2$

We have $\Delta' \vdash \theta'_i : \Delta_i$ by Theorem 4.1 on $\mathcal{D}_3$

We have $\Delta' \vdash \theta' : \Delta$ by Theorem 4.1 on $\mathcal{D}_4$

We have $\Delta' \vdash [\![\theta']\!]\mathcal{A} = [\![\theta'_i]\!][\![\theta_i]\!]\mathcal{A}_0$ by Theorem 4.1 on $\mathcal{D}_5$

We have $\Delta'; [\![\theta']\!]\Xi \vdash [\![\theta', \theta'_i]\!]e_i : [\![\theta'_i]\!][\![\theta_i]\!]\tau_0$ by inductive hypothesis on $\mathcal{D}_6$

Then $\Delta; \Xi \vdash^{\mathcal{A}} (\Delta_i; [\mathcal{M}_i] \mapsto e_i) : \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0$ by rule **CT-branch**

5. We have $\mathcal{D} :\ (\Omega; \Phi \vdash \zeta_1 \leq \zeta_2) \sqsubset (\Delta; \Xi \vdash \tau)$. There are three cases to consdier :

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega \vdash \mathcal{S}_1 \leq \mathcal{S}_2) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype}) \\ \mathcal{D}_2 :\ (\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx})\end{array}}{(\Omega; \Phi \vdash [\mathcal{S}_1] \leq [\mathcal{S}_2]) \sqsubset (\Delta; \Xi \vdash [\mathcal{A}])}\ \textbf{SubC-meta}$$

We have $\Delta \vdash \mathcal{A} : \texttt{mtype}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta \vdash \Xi : \texttt{cctx}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Xi \vdash [\mathcal{A}] : \texttt{ctype}$ by rule **CT-meta**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega; \Phi \vdash \zeta_2 \leq \zeta_1) \sqsubset (\Delta; \Xi \vdash \tau) \\ \mathcal{D}_2 :\ (\Omega; \Phi \vdash \zeta'_1 \leq \zeta'_2) \sqsubset (\Delta; \Xi \vdash \tau')\end{array}}{(\Omega; \Phi \vdash \zeta_1 \to \zeta'_1 \leq \zeta_2 \to \zeta'_2) \sqsubset (\Delta; \Xi \vdash \tau \to \tau')}\ \textbf{SubC-arr}$$

We have $\Delta; \Xi \vdash \tau_1 : \texttt{ctype}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Xi \vdash \tau_2 : \texttt{ctype}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Xi \vdash \tau_1 \to \tau_2 : \texttt{ctype}$ by rule **CT-arr**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{l}\mathcal{D}_1 :\ (\Omega \vdash \mathcal{S}_2 \leq \mathcal{S}_1) \sqsubset (\Delta \vdash \mathcal{A}) \\ \mathcal{D}_2 :\ (\Omega, u{:}\mathcal{S}_2; \Phi \vdash \zeta_1 \leq \zeta_2) \sqsubset (\Delta, u{:}\mathcal{A}; \Xi \vdash \tau)\end{array}}{(\Omega; \Phi \vdash \Pi u{:}\mathcal{S}_1.\zeta_1 \leq \Pi u{:}\mathcal{S}_2.\zeta_2) \sqsubset (\Delta; \Xi \vdash \Pi u{:}\mathcal{A}.\tau)}\ \textbf{SubC-pi}$$

We have $\Delta \vdash \mathcal{A} : \texttt{mtype}$ by inductive hypothesis on $\mathcal{D}_1$

We have $\Delta; \Xi \vdash \tau : \texttt{ctype}$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Xi \vdash \Pi u{:}\mathcal{A}.\tau : \texttt{ctype}$ by rule **CT-pi**

■

Chapter 5

Case studies

In previous chapters, we have illustrated the key ideas behind Beluga and its refinement system by mechanizing a simple property of the STLC, namely that values do not step. However, to really demonstrate the power of the language, we must consider more complex mechanizations. In this chapter, we will look at two additional examples. First, we will mechanize a polymorphic λ-calculus, showing that the design principles discussed earlier can scale up to richer OLs. Then, we will tackle one of the benchmarks presented by Felty et al. (2015a), which provides a more striking example of the benefits of refinements for stating theorems.

5.1 Polymorphic λ-calculus

Our first case study is a mechanization of System F^{ω}, based on the formulation of Abel (2009) (although we omit term equality here). The reason behind the choice of F^{ω} is that we need an OL in which type formation is non-trivial to properly explain one of the problems

solved by our new formulation of contexts. Specifically, the fact that contexts are extended by explicitly applying schema elements to parameters allows us to refer to these parameters, even when they do not appear within the corresponding block of declaration.

5.1.1 Encoding of F^{ω}

We begin by encoding F^{ω} using LF types. The language consists of kinds, types, and terms. Contrary to STLC, F^{ω} contains type-level computations, hence we need kinds to classify those. Additionally, the language requires type variables (of any kind) in order to support polymorphism. All type variables, including those that occur in binding, have a kind annotation, and, similarly, term variables have a type variables. Accordingly, kinds may occur in types, and both types and kinds may occur in terms. This means that we must define kinds first, then types, and finally terms. So, let us start with the syntax of kinds:

```
LF kd : type =
| star : kd
| karr : kd → kd → kd;
```

Kinds $K, L ::=$

$\star$

$| K \to L$

The kind `star` represent base types, while `karr` correspond to type-level function spaces. Next, we look at types:

```
LF tp : type =
| tarr : tp → tp → tp
| tall : kd → (tp → tp) → tp
| tlam : kd → (tp → tp) → tp
| tapp : tp → tp → tp;
```

Types $A, B ::=$ α

$| A \to B$

$| \forall \alpha{:}K.A$

$| \lambda \alpha{:}K.A$

$| A\ B$

Here, tarr is our usual function space, and tall is the polymorphic function space. Type-level computations are introduced using tlam, and eliminated with tapp. Finally, we can define the terms of F^{ω}:

```
LF tm : type =
| lam : tp → (tm → tm) → tm
| app : tm → tm → tm
| Lam : kd → (tp → tm) → tm
| App : tm → tp → tm;
```

Terms $M, N ::= x \mid \lambda x{:}A.M \mid M_1\ M_2 \mid \Lambda\alpha{:}K.M \mid M\ A$

Here, lam and app are the usual function introduction and elimination forms, respectively, while Lam and App are the polymorphic function introduction and elimination forms, respectively. Note that, as for STLC, variables are not modelled through explicit constructors (this applies to both type and term variables). Now, let us look at the kinding judgment:

```
LF ofk : tp → kd → type =

| o-tarr : ofk A star → ofk B star →
          ofk (tarr A B) star
| o-tall : ({α:tp} ofk α K → ofk (A α) star) →
          ofk (tall K A) star
| o-tlam : ({α:tp} ofk α K → ofk (A α) L) →
          ofk (tlam K A) (karr K L)
| o-tapp : ofk A (karr K L) → ofk B K →
          ofk (tapp A B) L;
```

$\boxed{\Gamma \vdash A : K}$ – Kinding

$$\frac{(\alpha{:}K) \in \Gamma}{\Gamma \vdash \alpha : K}$$

$$\frac{\Gamma \vdash A : \star \quad \Gamma \vdash B : \star}{\Gamma \vdash A \to B : \star}$$

$$\frac{\Gamma, \alpha{:}K \vdash A : \star}{\Gamma \vdash \forall\alpha{:}K.A : \star}$$

$$\frac{\Gamma, \alpha{:}K \vdash A : L}{\Gamma \vdash \lambda\alpha{:}K.A : K \to L}$$

$$\frac{\Gamma \vdash A : K \to L \quad \Gamma \vdash B : K}{\Gamma \vdash A\ B : L}$$

Notice that arrow ($\rightarrow$) and universal ($\forall$) types are limited to kind $\star$. Consequently, the only valid type-level computations are variables, `tlam`, and `tapp`.

Next, we want to define the typing judgment. In the presence of type-level computations, equality between types is no longer a trivial syntactic check. Therefore, we must first define a type equality judgment. Following Abel (2009), we use the kind-directed $\beta\eta$-equality:

```
LF teq : tp → tp → kd → type =
| t-beta : ({α : tp} ofk αK → ofk (A α) L) →
           ofk B K →
           teq (tapp (tlam K A) B) (A B) L
| t-eta : ofk A (karr K L) →
          teq A (tlam K (λα. tapp A α)) (karr K L)
  ⋮
```

$\boxed{\Gamma \vdash A \equiv B : K}$ – Type equality at kind K

$$\frac{\Gamma, \alpha{:}K \vdash A : L \quad \Gamma \vdash B : K}{\Gamma \vdash (\lambda\alpha{:}K.A)\ B \equiv [B/\alpha]A : L}$$

$$\frac{\Gamma \vdash A : K \rightarrow L}{\Gamma \vdash A \equiv (\lambda\alpha{:}K.A\ \alpha) : K \rightarrow L}$$

$\vdots$

For conciseness, we omit here the congruence rules that are needed to propagate type equality across the entire type system. In particular, there are subtle issues that occur when dealing with reflexivity and variables, which we will discuss in more details in our second case study. Now, we can finally define our typing judgment:

```
LF oft : tm → tp → type =

| o-lam : ({x:tm} oft x A → oft (M x) B) →
          oft (lam A M) (tarr A B)
| o-app : oft M (tarr A B) → oft N A →
          oft (app M N) B
| o-Lam : ({α:tp} ofk α K → oft (M α) (A α)) →
          oft (Lam K M) (tall K A)
| o-App : oft M (tall K A) → ofk B K →
          oft (App M B) (A B)
| o-teq : oft M A → teq A B star →
          oft M B;
```

$\boxed{\Gamma \vdash M : A}$ – Typing

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

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\frac{\Gamma, \alpha{:}K \vdash M : A}{\Gamma \vdash \Lambda\alpha{:}K.M : \forall\alpha{:}K.A}$$

$$\frac{\Gamma \vdash M : \forall\alpha{:}K.A \quad \Gamma \vdash B : K}{\Gamma \vdash M\ B : [B/\alpha]A}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A \equiv B : \star}{\Gamma \vdash M : B}$$

Notice, now, that the typing judgment ensures that any term must have a type that is equal to either an arrow or a universal, i.e. a type of kind $\star$, except for type variables. By using a suitable schema, we can ensure that any term variable is also assigned a type of kind $\star$, and thus that any well-typed term has a base type. We encode the schema as follows:

```
LF ctx : schema =

| tp-var : some [K:kd]
            block (α : tp, k : ofk α K)
| tm-var : some [A:tp, W:ofk A star]
            block (x : tm, t : oft x A);
```

$\boxed{\vdash \Gamma}$ – Γ is a well-formed context

$$\frac{\vdash \Gamma \quad \Gamma \vdash K}{\vdash \Gamma, \alpha{:}K}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash A : \star}{\vdash \Gamma, x{:}A}$$

Notice here that forming a `tm-var` assumption requires a parameter `W:ofk A star` which

appears nowhere in the block. The new notation for contexts that we suggest in this work is crucial to properly handling this kind of scenario appropriately. Consider, for instance, the following result stating that if $M : A$, then $A : \star$:

```
rec subderiv : (Γ : ctx) [Γ ⊢ oft M A] → [Γ ⊢ ofk A star] =
fn d => case d of
| [Γ, b:tm-var A W ⊢ b.2] => [Γ, b:tm-var A W ⊢ W]
    ⋮
```

Using the previous notation (Pientka and Dunfield, 2010) for context extensions, we would instead have only the pattern `[Γ, b:block(x:tm, t:oft x A)]`, from which it is impossible to recover the desired derivation. We first encountered this issue when attempting to mechanize Stone and Harper (2006)'s proof of normalization for Standard ML's module calculus. Solving that problem was the original motivation behind the present work (although, in time, it became a fairly small aspect of it). Unfortunately, we still only have a partial solution: although the variable case of the subderivation lemma can now be solved, other cases remain difficult to prove in our setting. Consider, in particular, the case of functions, which we would like to (but cannot) solve roughly as follows:

```
| [Γ ⊢ o-lam (λx.λt.D)] =>
    let [Γ, b:tm-var A W1 ⊢ W2] =
        subderiv [Γ, b:tm-var A W1 ⊢ D[..,b.1,b.2]] in
    [Γ ⊢ o-tarr W1 W2]
```

There are two main problems with this. First, the derivation `W2` may, in principle, depend on `b`, so we cannot use it in context Γ in the last line. This issue can be solved by prov-

ing a strengthening lemma since kinding derivations can never depend on terms or typing derivations (i.e. anything in a `tm-var` assumption).

The second, and more important, problem is that we do not know from the premise that the derivation `W1` actually exists. We can infer what `A` is since `D` depends on the variable `t`, which is known to have type `oft x A`. But, there is nothing there to allow us to retrieve a derivation of `ofk A star`. It is possible to work around this issue by including the missing assumption as part of `o-lam`'s type, but this is unsatisfactory since the informal formulation of the judgment does not include this assumption.

A more satisfactory solution could come from allowing full-fledged schema elements to appear in negative positions of type definitions. In this setting, the type of `o-lam` would become `({b:tm-var A W} oft (M b.1) B) → oft (lam M) (arr A B)`, which would then allow us to easily recover all the information that we need. For this approach to work, we would also need to simultaneously define judgments and the contexts in which they are meaningful. This being said, we have not yet investigated this idea, so we cannot assert, at this time, that it is a viable option.

5.1.2 Bi-directional typing

Next, let us briefly look at how one may extract some interesting fragments of F^ω using refinements. In Chapter 3, we briefly discussed how neutral and normal terms could be defined as refinements of terms in the STLC. Here, we revisit this example by extracting a bi-directional typing algorithm as a refinement of F^ω's typing judgment. For simplicity, we will focus only on typing, although one could also specify a bi-directional kinding judgment

in a similar way.

The first step in defining the bi-directional type checker is to extract neutral and normal terms. We achieve this elegantly thanks to subsorting, just like we did for STLC:

```
LFR neut ⊏ tm : sort =
| app : neut → norm → neut
| App : neut → tp → neut

and neut ≤ norm ⊏ tm : sort =
| lam : tp → (neut → norm) → norm
| Lam : kd → (tp → norm) → norm;
```

Neutral $R ::= x \mid R\ N \mid R\ A$

Normal $N ::= \lambda x{:}A.N \mid \Lambda\alpha{:}K.N$

Now, a bi-directional type checker has two phases: type synthesis for neutral terms, and type checking for normal terms, with a conversion from synthesis to checking. So, we can again use subsorting to obtain the conversion. Moreover, here we can use kind refinements to enforce that only neutral terms are allowed in synthesis, and only normal terms in checking. We obtain the following encoding:

```
LFR synth ⊏ oft : neut → tp → sort =

| o-app : synth R (arr A B) →
          check N A →
          synth (app R N) B
| o-App : synth R (tall K A) →
          ofk B K →
          synth (App R B) (A B)
```

$\boxed{\Gamma \vdash R \Rightarrow A}$ – Synthesis

$$\frac{(x{:}A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}$$

$$\frac{\Gamma \vdash R \Rightarrow A \to B \quad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash R\ N \Rightarrow B}$$

$$\frac{\Gamma \vdash R \Rightarrow \forall \alpha{:}K.A \quad \Gamma \vdash B : K}{\Gamma \vdash R\ B \Rightarrow [B/\alpha]A}$$

```
and synth ≤ check ⊏ oft : norm → tp → sort =

| o-lam : ({x:neut} synth x A → check (N x) B) →
           check (lam A N) (arr A B)
| o-Lam : ({α:tp} ofk α K → check (N α) (A α)) →
           check (Lam K N) (tall K A)
| o-teq : synth M A → teq A B →
           check M B;
```

$\boxed{\Gamma \vdash N \Leftarrow A}$ – Checking

$$\frac{\Gamma \vdash R \Rightarrow A}{\Gamma \vdash R \Leftarrow A}$$

$$\frac{\Gamma, x{:}A \vdash N \Leftarrow B}{\Gamma \vdash \lambda x{:}A.N \Leftarrow A \to B}$$

$$\frac{\Gamma, \alpha{:}K \vdash N \Leftarrow A}{\Gamma \vdash \Lambda \alpha{:}K.N \Leftarrow \forall \alpha{:}K.A}$$

Notice that the sort of o-teq also offers a conversion from synthesis to checking, so the subsorting is not strictly necessary for this example. This being said, using subsorting for conversion removes the need to check equality of a type with itself. Finally, we incorporate the fact that variables are neutral and subject to synthesis as part of our context schemas:

```
LFR bdt-ctx ⊏ ctx : schema =
```

```
| tp-var : some [K:kd] block (α:tp, k:ofk α K)
| tm-var : some [A:tp, W:ofk A star] block (x:neut, t:synth x A);
```

And this concludes the encoding of a bi-directional typing judgment as a refinement of the ordinary typing judgment. Now, our conservativity results imply that a subsumption property from bi-directional derivations to ordinary derivations is admissible. That is, if Γ:`bdt-ctx` and `M`:`[`Γ $\vdash$ `check M A]`, then Γ:`ctx` and `M`:`[`Γ $\vdash$ `oft M A]`. This means that we can apply our lemma `subderiv` to `M` and obtain a proof that `[`Γ $\vdash$ `ofk A star]`. In other words, the properties proven about ordinary typing derivations still hold of bi-directional typing derivations, hence refinements provide a form of proof reuse.

5.2 Equality

For our second example, we tackle one of the benchmarks presented by Felty et al. (2015a), showing the equivalence between two forms of equalities in the untyped λ-calculus. Several solutions to this benchmark were discussed in a follow-up paper (Felty et al., 2015b). This provides us with an opportunity to compare our mechanization with refinements to previous attempts with types only, and to observe more significant improvements than we previously obtained in our simple example that values do not step.

5.2.1 Overview of the benchmark

Lemmas play a crucial role in structuring complex proofs, both formally and informally. To allow flexible usage, a lemma's statement should make no assumption about the setting in

which it may be invoked. A particular lemma may come up in the proofs of several theorems, each of which makes slightly different and stronger assumptions than those required by the lemma. In our previous case study, we saw a subderivation property that holds of all typing derivations, and discussed how it also holds of bi-directional derivations. There, the assumption of being a bi-directional typing derivation was stronger than just being a typing derivation, and this was encompassed by our refinement subsumption. One reason why this worked is that the contexts of both typing disciplines have the same structure, but this may not always be the case. For instance, a lemma concerning all untyped terms should be applicable regardless of the typing discipline, but our refinements would not be helpful since untyped contexts have a different structure than typed contexts.

Felty et al. (2015a) suggests two solutions for this problem, along with several benchmarks aimed at comparing the two approaches (and their solutions using different proof environments). The generalized context approach (G version) consists of stating every lemma and theorem with the same context schema. Consequently, all the assumptions needed for some proof need to be carried across all proofs, thus failing at making only minimal assumptions. In contrast, the context relation approach (R version) uses minimal schemas in statements of lemmas and theorems, but requires the top-level theorem to manipulate several related contexts, so that it can call the relevant lemmas on the correct context. Despite being conceptually more pleasing, R version proofs tend to be more complicated than G version proofs. Even for the relatively simple benchmarks proposed by Felty et al. (2015a), we observe a significant blow-up in the size of types required to express theorems. This is only due to the fact that multiple contexts and relations between them need to be maintained and passed

as arguments to recursive calls. The increased size impacts the readability of mechanized proofs by drowning the important parts of a type in a sea of technical details. We can only expect this to get worse as we attempt to mechanize more and more complex proofs, so we conclude that the context relation approach does not easily scale.

Now, we will illustrate the benefits of refinement types by revisiting one of the benchmark proposed in Felty et al. (2015a), namely the equivalence of algorithmic and declarative equalities in the untyped λ-calculus. Algorithmic equality traverses terms recursively to establish that they are syntactically equal. Declarative equality does the same thing, but also includes rules encoding the axioms of an equivalence relation (reflexivity, symmetry, and transitivity). As such, we can represent algorithmic equality as a refinement of declarative equality. This will allow us to encode the context relation as a refinement relation, so that it does not need to be carried explicitly in the theorem. We end up with a proof that matches the simplicity of a G version, but still offers the flexibility of the R version.

5.2.2 Solution using refinement types

Let us now present a new solution using refinements, so that we can later discuss how it improves on previous approaches. The first step of the mechanization is to encode the untyped λ-calculus as a type:

```
LF tm : type =
| lam : (tm → tm) → tm
| app : tm → tm → tm;
```

Terms $M, N ::= x \mid \lambda x.M \mid M\ N$

There is nothing special about this definition that has not been discussed earlier, so we

move on to encoding the judgments for equalities. We start with declarative equality, which is represented as a type:

```
LF deq : tm → tm → type =

| e-lam : ({x:tm} deq x x → deq (M x) (N x)) →
           deq (lam M) (lam N)

| e-app : deq M1 N1 →
           deq M2 N2 →
           deq (app M1 M2) (app N1 N2)

| e-refl : deq M M

| e-sym : deq M N →
           deq N M

| e-trans : deq M1 M2 →
             deq M2 M3 →
             deq M1 M3;
```

$\boxed{\Gamma \vdash M \equiv N}$ – Declarative

$$\frac{x \in \Gamma}{\Gamma \vdash x \equiv x}$$

$$\frac{\Gamma, x \vdash M \equiv N}{\Gamma \vdash \lambda x.M \equiv \lambda x.N}$$

$$\frac{\Gamma \vdash M_1 \equiv M_2 \quad \Gamma \vdash N_1 \equiv N_2}{\Gamma \vdash M_1\ N_1 \equiv M_2\ N_2}$$

$$\frac{}{\Gamma \vdash M \equiv M}$$

$$\frac{\Gamma \vdash N \equiv M}{\Gamma \vdash M \equiv N}$$

$$\frac{\Gamma \vdash M_1 \equiv M_2 \quad \Gamma \vdash M_2 \equiv M_3}{\Gamma \vdash M_1 \equiv M_3}$$

So, we have a rule that says that variables are equal to themselves, two congruence rules to propagate equality through λ's and applications, and the three rules representing the axioms of equivalence relations, reflexivity, symmetry, and transitivity. Note that the variable rule is redundant due to the presence of reflexivity, so it is technically not needed for this judgment's definition. We follow Felty et al. (2015a) in maintaining the variable rule in order to facilitate the relation with algorithmic equality, which we will soon discuss. But first, let us encode the contexts required to express proofs of declarative equality:

```
LF xdG : schema =
| eq-var : block (x:tm, e:deq x x);
```

Contexts consists of term variables together with the assumption that they are declaratively equal to themselves. As usual, this coincides precisely with the negative occurrences that we see in the definition of `deq`, so we know that proofs of `deq` can be constructed in a context of schema `xdG`, as desired.

Now, algorithmic equality is just declarative equality without the three constructors encoding the axioms of equivalence relations. As such, it is best defined as a refinement of `deq` rather than as a separate atomic type:

```
LFR aeq ⊏ deq : tm → tm → sort =
| e-lam : ({x:tm} aeq x x → aeq (M x) (N x)) →
          aeq (lam M) (lam N)
| e-app : aeq M1 N1 →
          aeq M2 N2 →
          aeq (app M1 M2) (app N1 N2);
```

$\boxed{\Gamma \vdash M \Leftrightarrow N}$ – Algorithmic

$$\frac{x \in \Gamma}{\Gamma \vdash x \Leftrightarrow x}$$

$$\frac{\Gamma, x \vdash M \Leftrightarrow N}{\Gamma \vdash \lambda x.M \Leftrightarrow \lambda x.N}$$

$$\frac{\Gamma \vdash M_1 \Leftrightarrow M_2 \quad \Gamma \vdash N_1 \Leftrightarrow N_2}{\Gamma \vdash M_1\ N_1 \Leftrightarrow M_2\ N_2}$$

Here, the variable rule is crucial for the judgment since it provides the only base case. We note that a general reflexivity rule poses a conceptual problem when designing an algorithm to decide equality since it is not immediately clear that two terms are syntactically the same. That is, reflexivity concludes equality of arbitrary large terms in a single step, which is unrealistic. On the other hand, variables are the smallest terms (along with constants), so it is sensible to think that we can use reflexivity in one step for variables.

One advantage of representing algorithmic equality as a refinement of declarative equality is that we can exploit the subsumption associated to refinements to obtain soundness of algorithmic equality for free. If we had used only types, this would need to be proven explicitly, and, while this is not particularly difficult, it can be tedious to formally prove results that would informally be dismissed as trivial by definition. In fact, this soundness result is so obvious that Felty et al. (2015a) does not even mention it as a requirement for establishing equivalence between the two equalities, focusing instead on completeness.

Finally, we need contexts in which to perform our algorithmic equality. The solution, of course, is to refine declarative equality contexts:

```
LFR xaG ⊏ xdG : schema =
| eq-var : block (x:tm, e:aeq x x);
```

We use the same context schemas as suggested in Felty et al. (2015a), but we relate them using the refinement relation instead of an explicitly defined inductive relation on contexts. The refinement relation is then exploited in the proof of completeness to alternate between the two schemas. The proof relies on three lemmas that each reproduces one of the missing rules. We omit here their proofs as they are straightforward, and mention only their assigned sort (reflexivity, symmetry, and transitivity, respectively):

```
aeq-ref : (Ψ : xaG) {M : [Ψ ⊢ tm]} [Ψ ⊢ aeq M M];
aeq-sym : (Ψ : xaG) [Ψ ⊢ aeq M N] → [Ψ ⊢ aeq N M];
aeq-tra : (Ψ : xaG) [Ψ ⊢ aeq M1 M2] → [Ψ ⊢ aeq M2 M3] → [Ψ ⊢ aeq M1 M3];
```

Finally, we must prove completeness. To achieve this, we need be able to alternate between interpreting a context as an `xaG` and a `xdG`. We know from our conservativity result

(Theorem 4.1) that whenever we have a context Ψ : xaG, we can find Γ : xdG with $\Psi \sqsubset \Gamma$. Moreover, as discussed in Chapter 3, we know that Ψ and Γ are, in fact, the exact same context since Ψ has a schema, and therefore does not contain any type annotations (or, rather, they are hidden within the schema). So, we introduce the notation $\Psi^\top$ to indicate that we want to treat Ψ as having schema xdG instead of xaG. With this, we can state and prove our completeness theorem as follows:

```
rec ceq : (Ψ : xaG) [Ψ⊤ ⊢ deq M N] → [Ψ ⊢ aeq M N] =
fn d => case d of
  | [Ψ⊤ ⊢ #b.2] =>
        [Ψ ⊢ #b.2]
  | [Ψ⊤ ⊢ e-lam (λx.λe. D)] =>
        let [Ψ, b:eq-var ⊢ E] = ceq [Ψ⊤, b:eq-var ⊢ D[.., b.1, b.2]] in
        [Ψ ⊢ e-lam (λx.λe. E[..,<x;e>])]
  | [Ψ⊤ ⊢ e-app D1 D2] =>
        let [Ψ ⊢ E1] = ceq [Ψ⊤ ⊢ D1] in
        let [Ψ ⊢ E2] = ceq [Ψ⊤ ⊢ D2] in
        [Ψ ⊢ e-app E1 E2]
  | [Ψ⊤ ⊢ e-refl M] =>
        aeq-ref [Ψ ⊢ M]
  | [Ψ⊤ ⊢ e-sym D] =>
         let [Ψ ⊢ E] = ceq [Ψ⊤ ⊢ D] in
         aeq-sym [Ψ ⊢ E]
```

```
| [Ψ⊤ ⊢ e-trans D1 D2] =>
      let [Ψ ⊢ E1] = ceq [Ψ⊤ ⊢ D1] in
      let [Ψ ⊢ E2] = ceq [Ψ⊤ ⊢ D2] in
      aeq-tra [Ψ ⊢ E1] [Ψ ⊢ E2];
```

So, we prove the result by induction on the given proof of declarative equality, with six cases to consider. The first case, for variables, crucially exploits the good behaviour of refinements. It is essentially an identity case, in the sense that we merely output the same variable that was given as input. However, the interpretation differs because, in the input, we considered it as an element of $\Psi^\top$, which has schema `xdG`, but the output is an element of Ψ, which instead has schema `xaG`. Since the sort information is entirely located within the schema, and not the context itself, we know that the input has sort `deq #b.1 #b.1`, while the output has sort `aeq #b.1 #b.1`, as desired. Note also that we can be certain that `#b` appears in both contexts since they are the same, only interpreted slightly differently.

The next two cases correspond to our congruence rules. Since those appear in both judgments, we need only unpack the information, apply an inductive hypothesis, and repack the results. The case for `e-lam` illustrates well the fact that the contexts Ψ and $\Psi^\top$ must be syntactically equal: both contexts are extended with the same assumption `b:eq-var`. The notation `<x;e>` appearing in the substitution on the last line of this case is the concrete syntax for substitutions including n-ary tuples (or spines). It is needed here because we have a derivation `E` that makes sense in context Ψ`, b:eq-var`, but `e-lam` expects a derivation in context Ψ`, x:tm, e:aeq x x`.

The last three cases correspond to the rules encoding equivalence relations. These are

handled in a similar way to the congruence rules, except that we call the relevant lemma (instead of an inference rule) after applying the inductive hypothesis. Note that the derivation that we obtain from the recursive call is in context Ψ, so it correctly matches the sort expected by the lemma.

5.2.3 Comparison with previous solutions

Several solutions[1] have already been proposed for this benchmark (Felty et al., 2015b). This section will discuss how we have improved on previous work, focusing on the previous Beluga solution. The key takeaway is that we require fewer parameters to be passed as inputs to the function, and fewer types to encode these parameters. Moreover, the additional types are computation-level inductive types, so our solution, in addition to being simpler, relies on (arguably) less heavy machinery. Let us start by looking at the statement of the completeness result in Beluga, which provides a good example of an unreadable type:

```
ceq: Ctx_xaR [Γ] [Ψ] [Ψ ⊢ $σ] →
     Ctx_xdR [Γ] [Φ] [Φ ⊢ $ρ] →
     Ctx_adR [Ψ] [Φ] →
     Eq  [Γ] [Ψ] [Ψ ⊢ $σ] [Γ ⊢ M] [Ψ ⊢ Ma] →
     Eq  [Γ] [Ψ] [Ψ ⊢ $σ] [Γ ⊢ N] [Ψ ⊢ Na] →
     Eq' [Γ] [Φ] [Φ ⊢ $ρ] [Γ ⊢ M] [Φ ⊢ Md] →
     Eq' [Γ] [Φ] [Φ ⊢ $ρ] [Γ ⊢ N] [Φ ⊢ Nd] →
     [Φ ⊢ deq Md Nd] → [Ψ ⊢ aeq Ma Na]
```

[1] The complete solutions can be found at `https://github.com/pientka/ORBI`.

The first three parameters correspond to context relations, where Γ contains only `tm` variables, and Ψ and Φ additionally contain `aeq` and `deq` variables, respectively. The first relation, `Ctx_xaR [Γ] [Ψ] [Ψ ⊢ $σ]`, expresses that σ is a weakening substitution that adds to each block `(x:tm)` an assumption of type `aeq x x`. The relation `Ctx_xaR [Γ] [Φ] [Φ ⊢ $ρ ]`, is similar, except that it weakens with `deq x x` assumptions. Finally, the third relation, `Ctx_adR [Ψ] [Φ]`, essentially expresses our refinement relation $\Psi \sqsubset \Phi$.

The type `Eq` expresses that applying the substitution $\$\sigma$ to `M` results in `Ma`. We cannot directly use the same object since the variables of Γ are different from those of Ψ, so that `M` is not meaningful in context Γ. The type `Eq'` is similar, but relates objects in Γ to objects in Φ instead. Each of these seven arguments serve only as a setup for the last line to express the desired property.

In the interest of fair comparison, we observe that this result is slightly stronger than what we have proven using refinements. The `tm`'s compared, `M` and `N`, are meaningful in the context containing only `tm` assumptions, and then weakened to the larger contexts using `Eq` and `Eq'`. Our version only specifies that the `tm`'s are meaningful in both the `xaG` and the `xdG` contexts. To obtain, in conventional Beluga, the same property that we proved, we would only need the context relation `Ctx_adR` and we could use an equality predicate to directly relate terms in Ψ to terms in Φ (although there would be no substitution between the two contexts). Since our extension does not currently support definitions of computation-level types, we cannot yet reproduce the exact same result. However, we expect that a slightly richer form of refinements would allow us express it without the use of any user-defined computation-level types (see 6.2 for a discussion).

Chapter 6

Conclusion

We included a form of datasort refinement types (Freeman and Pfenning, 1991) to Beluga (Pientka and Dunfield, 2010), a proof environment specialized in proving meta-theoretic properties of programming languages and other similar formal systems. The key advantage that Beluga offers is its handling of contexts as first-class objects, which allows mechanized OLs to inherit the good properties of LF substitutions for free. Through the addition of refinements, in particular refinement schemas, we improved upon this advantage by expanding the precision of classifiers.

In the process, our formulation of the language went through several iterations, which led us to describe refinement as a relation between judgments, rather than between the various objects which compose them. We identify two main benefits to this approach. First, it becomes clear that the refinement relation may be interpreted as a function, meaning that every sort refines a unique type, and that said type may be derived from the shape of the sort. Second, it significantly simplifies the proof of conservativity, which would otherwise

rely on several lemmas (rather than be self contained). We note also that our formulation demonstrates the simplicity of extending a language with refinements by highlighting the close similarities between the new judgments and those that existed before.

Finally, we illustrated the benefits of refinements for Beluga through examples. In particular, we revisited one of the benchmarks proposed by Felty et al. (2015a), which is concerned with relations between contexts. Our notion of refinement schema provides an elegant representation of some of the relations, which we used to significantly simplify the statement of the main theorem. This is mainly caused by the fact that several predicates can be expressed without the use of additional parameters. Thus, refinements improve the clarity of mechanizations, bringing the formal and informal theorem statements closer together.

6.1 Discussion

This work has been guided by two distinct, yet intertwined, guiding lines. In the beginning, we were concerned with correctly representing contexts, particularly in the setting of HOAS. We revised our understanding of schemas so as to view them in a constructive way, similar to atomic types. Specifically, we now view schema elements as analogous to constructors, but for contexts. Then, much like atomic types provide the recipes for constructing particular terms, schemas provide the recipes for constructing particular contexts. This modification allows us to correctly interpret schemas as formal representations of context formation judgments of OLs, and thus maintains the "judgments-as-types" principles that Beluga inherits from LF. At first, the change seemed simple enough that it would have few ramifications: it merely solved a minor, technical bug in Beluga's representation of context.

The focus of our research then moved towards more conceptual challenges related to contexts in HOAS representations. Felty et al. (2015a) discusses how different judgments can rely on slightly different context structures. Since one OL usually involves several judgments, we may need several schemas to satisfyingly state all the meta-theoretic properties that we care about. In many cases, these various schemas are related, or rather the contexts inhabiting them are related, although not always in a clear way. Through multiple benchmarks, Felty et al. (2015a) investigates these issues, and provides a solution for a modular statement of lemmas and theorems, namely context relations. Unfortunately, context relations seriously complicate the statements of theorems, which obscures the correspondence between formal and informal theorems. This being said, the context relations solution is conceptually pleasing since it indicates a possible path towards scalable formal proofs. As such, it deserves further investigation, hence we decided to tackle its challenges with refinement types, our second guiding line.

While datasort refinements were previously used mainly as a way to provide subtyping and intersection types to a language, our work focuses on the refinement relation itself. In particular, we discussed how refinements could allow validating the correctness of proofs containing non-exhaustive pattern matching. As such, refinement types can help a proof environment reproduce the informal method of omitting irrelevant possibilities, which crucially relies on knowing what a function will be called on. Unfortunately, we could not, in this work, finalize the design of a coverage checker, so the interpretation of programs as proofs remains incomplete.

Our notion of refinement differs from previous datasort systems (Freeman and Pfenning,

1991; Lovas and Pfenning, 2010) in that we do not consider sort-checking as phase that occurs after type-checking. Rather, we view type-checking as a by-product of sort-checking, in the sense that the sort-checking algorithm can generate a valid typing derivation. We achieved this by formulating our refinement relations between judgments themselves, rather than on the various syntactic objects that compose them. Our view is that refinements are best understood as a means of safely reusing constructors that were uniquely introduced through type definitions.

Through this extension, we studied the novel notion of refinement schemas. These express more precise information about contexts, much like refinement types express more precise properties about objects. In particular, refinement schemas are useful to deal with a special kind of context relations, namely those when contexts are related by refinements. We demonstrated this by revisiting one of the benchmarks suggested by Felty et al. (2015a), in which our solution was significantly simpler than the previous, types-only, Beluga solution. Our investigation of refinement schemas further reinforced the correctness of our reformulation of contexts and schemas. By viewing schema elements as constructors of contexts, we can consider the refinement relation on schemas as a way to reuse the same syntax, much like we interpret atomic sorts. In particular, this means that we can now view a single context in multiple different ways, which eliminates the need for certain context relations.

A key objective that motivated the present work is the development of a practical proof environment, in the sense that most of a user's efforts should go into the conceptual aspects of a proof, rather than the technical implementation details imposed by the underlying type theory. Refinement types play an interesting role in this endeavour. When included in an al-

ready practical system, such as Standard ML (Freeman and Pfenning, 1991; Davies, 2005), they increase the power of the type theory, allowing it to provide stronger security guarantees without significantly affecting the usability. On the other hand, by adding refinements to an already expressive language, such as LF (Lovas and Pfenning, 2010) or Beluga, we instead improve the usability without sacrificing any type theoretic power. This makes them a useful tool in virtually any language. Moreover, the wealth of work on datasort refinements hints that they can scale well to very rich type theories. With the formulation suggested in this thesis, it seems that they would also be relatively easy to add in most systems: basically, we only duplicated everything in the language and added a few rules to handle sort declarations in the signature formation judgment.

In Beluga, every computation-level expression has a unique type, but they can have many sorts in our extension. Pfenning (2008) discusses this phenomenon and how it relates intrinsic and extrinsic views of typing, respectively. That is, types may be viewed as an intrinsic, syntactic property of expressions, whereas sorts represent properties that are externally asserted. The fact that our judgments produce typing derivation then shows that sorts can only express properties about objects that are intrinsically meaningful. Moreover, it tells us that that the unique intrinsic property of a program is hidden within any extrinsic property that applies to that program.

Conceptually, an extrinsic view of typing is satisfying. This is especially true because terms evaluate in the same way no matter the particular typing discipline enforced. However, this view relies on the possibility to define terms independently of types (or sorts), which proves difficult when we consider contexts as objects. This is because we cannot discard the

type information of the variables in a context without losing information on its structure, especially in the case of blocks. Our new formulation of contexts and schemas using worlds provides a partial solution to this problem: a context only inhabits a schema if all of its variables are constructed from its schema elements. Since a schema element is only a name, contexts are expressed as lists of names applied to terms. Consequently, contexts do not directly contain any type (or sort) information.

On the other hand, intrinsic typing offers the benefit of filtering out ill-defined terms from the start. Under this view, we recognize that our underlying model of computation, the (untyped) λ-calculus, describes many programs that are simply meaningless. Thus, adopting a strict typing discipline that removes these programs from the language will only save us the headache of fixing errors. However, the intrinsic type of a program should be uniquely determined from the shape of the program alone, so this view is only compatible with languages in which type uniqueness holds. In practice, this means that code cannot easily be reused in slightly different settings, since the type would have to change.

Previous versions of Beluga have opted for the intrinsic typing approach. In our setting of datasort refinements, we settle instead for a compromise between the two views of typing: types correspond to intrinsic properties, and sorts to extrinsic properties. In doing so, we can leverage the advantages of both approaches: types provide strong guarantees that programs perform only meaningful computations, while sorts allow us to express multiple precise statements on the behaviour of programs. However, we also inherit some of the inconveniences from both approaches: types no longer ensure that programs always perform computations because non-exhaustive pattern matching is allowed, and sort-checking is more

expensive than type-checking because constructors have multiple sorts, so we have to make the right choices in signature lookups (or backup and try again).

Handling contexts and substitutions is at the source of many difficulties in the field of programming languages. As mentioned in Chapter 5, this work originated in an attempt at mechanizing the meta-theory of Standard ML, which failed due to subtle errors in Beluga's understanding of contexts. Previous mechanizations of Standard ML also had a hard time dealing with contexts: Lee et al. (2007) reports that about half of their Twelf (Pfenning and Schürmann, 1999) mechanization pertains to proving (hereditary) substitution properties, and Rossberg et al. (2010) notes that most of the lemmas in their Coq (Dowek et al., 1993) mechanization are devoted to the context and substitution infrastructure. These proofs tend to be long and tedious, which may discourage language designers from formalizing their systems. The main advantage of HOAS is that it relegates much of these tedious lemmas to the underlying meta-language, so the user obtains them for free. Thus, we can expect a Beluga mechanization to be much shorter than previous attempts.

The language discussed in the present thesis went through several stages of formulation, mainly because of difficulties related to contexts. In particular, we started with contexts where variables are assigned both a sort and a type, following Lovas and Pfenning (2010). This proved inadequate when dealing with contextual refinements, where it was more meaningful to fully separate sorting and typing contexts, as we did now. However, this meant that several premises establishing refinement relations were needed in some rules, which eventually led us to formulate refinements directly on judgments. Originally, we only intended this formulation to validate all the necessary refinement relations at the same time, but once we

had this formulation, it became a lot clearer that we could view the refinement relation as a function (i.e. that types could be generated). In other words, handling these small, subtle problems with contexts led to one of the central contributions of our work.

Our formulation of Beluga also differs from previous ones (Pientka, 2008; Cave and Pientka, 2013; Pientka and Abel, 2015) in its treatment of substitutions. Like them, we ensure that substitutions have the same shape as their domain context, thus eliminating the need to explicitly specify said domain. However, since our contexts are structured into blocks, substitutions needed to also have that form. This is why we allow n-ary tuples in substitutions. Once again, this is a design decision that was based on how we understand contexts. We note that both the structure of contexts into blocks and the idea that substitutions should have the same shape as their domain were previously discussed, yet, to our knowledge, the idea that substitutions be structured into blocks as well had not been previously explored.

6.2 Future work

We end this discussion by looking at some of the possible avenues for future work. A most simple direction is to carry on the extension of refinements to the remaining features of Beluga. In particular, we expect that there should be no significant challenges with extending our refinement relations to computation-level inductive types, especially given that datasorts were originally designed for inductive types. Moreover, we expect that refinements for computation-level inductive types would increase the expressive power of Beluga type. This is because these types cannot be dependencies of other types, so we cannot express

predicates about objects of computation-level inductive types. As pointed out by Lovas and Pfenning (2010), datasorts can be understood as (proof irrelevant) predicates on the type which they refine.

Beluga's computation-level also supports coinductive types and index-stratified types, neither of which has ever been studied through the lens of refinements (to our knowledge). (Co)inductive and stratified are defined together in Jacob-Rao et al. (2018), so it would be straightforward to extend our computation-level judgments with the additional rules that they have. All of these computation-level types allow us to mechanize additional proof methods, such as logical relations. Given the importance of such proofs in programming language theory, supporting them would certainly improve our system.

Another immediately useful addition to the present work would be the development of a refinement-level totality checker. The type-level totality checker was developed in Pientka and Abel (2015) and is essential in identifying which functions can be understood as proofs. There are three important components needed by the totality checker: First, a unification procedure, second, a way to split on LF objects, that is, identify the possible patterns representing that object, and third, a well-founded order on objects of the data-level. For the first point, we note that the main challenge of unification is unifying terms. Since we have not changed the language of terms, we expect that the procedure of Pientka (2003) can be adapted to our purpose. Likewise, the third point is already solved by using the same order that Pientka and Abel (2015) suggested, which is still applicable to us since the terms of LF and LFR are the same. This leaves us, then, only with the task of designing a splitting algorithm. The algorithm suggested by Pientka and Abel (2015) proceeds essentially by

brute-force, generating all the possible patterns based on the definition of the atomic LF type of the guard. Due to the close similarity between type definitions and sort definitions, it should again be possible to adapt their algorithm to our setting. However, complications arise in the presence of subsorting since generating all the patterns that can yield an object of a given atomic sort Q also requires generating the patterns for all of its subsorts. This will be particularly difficult in the case where a particular constructor is allowed by both Q and some of its subsorts, as there would then be multiple cases associated to the same constructor. A simple solution for this problem would be to use explicit coercions whenever subsorting occurs, and to associate a special pattern to these coercions. In this way, we can differentiate the different possible uses of a constructor, thus resolving the issue. We recall that refinements were originally intended to solve this non-exhaustive pattern matching problem (Freeman and Pfenning, 1991), so it is really not surprising that we can apply them for this purpose here as well.

The other important theme of this work is the structure of contexts and of relations between them, which can also benefit from further investigations. We have discussed in Chapter 5 that refinement schemas could be used for representing some of the context relations described by Felty et al. (2015a). However, our current notion of refinements can represent only a small subset of these context relations. In particular, the limitation that a refined schema element must have exactly the same shape as what it refines prevents describing relations between contexts that do not have the exact same structure. In our current setting, a refinement between contexts, say $\Psi \sqsubset \Gamma$, indicates that Ψ is "stronger" than Γ, in the sense that the assumptions contained in Ψ entail those contained in Γ. In this case, there

are more proofs possible in context Ψ than there are in context Γ. Keeping in line with this idea, it is interesting to consider how the refinement relation may be meaningfully extended to allow Ψ to contain more variables than Γ. Felty et al. (2015a) already discussed such context relations, so there are several known examples for which this extended refinement relation could be applied.

Bibliography

Abel, A. (2009). Typed applicative structures and normalization by evaluation for system F^{ω}. In Grädel, E. and Kahle, R., editors, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, volume 5771 of *Lecture Notes in Computer Science*, pages 40–54. Springer.

Barendregt, H. (1991). Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154.

Barthe, G. and Frade, M. J. (1999). Constructor subtyping. In Swierstra, S. D., editor, *Programming Languages and Systems*, pages 109–127, Berlin, Heidelberg. Springer Berlin Heidelberg.

Cave, A. and Pientka, B. (2012). Programming with binders and indexed data-types. In Field, J. and Hicks, M., editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 413–424. ACM.

Cave, A. and Pientka, B. (2013). First-class substitutions in contextual type theory. In

Momigliano, A., Pientka, B., and Pollack, R., editors, *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-languages: Theory & Practice, LFMTP 2013, Boston, Massachusetts, USA, September 23, 2013*, pages 15–24. ACM.

Cave, A. and Pientka, B. (2018). Mechanizing proofs with logical relations - Kripke-style. *Math. Struct. Comput. Sci.*, 28(9):1606–1638.

Davies, R. (2005). *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, USA, USA. AAI3168521.

Dowek, G., Felty, A., Herbelin, H., Huet, G., Murthy, C., Parent, C., Paulin-Mohring, C., and Werner, B. (1993). The coq proof assistant user's guide version 5.8. Technical report, INRIA. Technical Report 154, INRIA, May 1993.

Dunfield, J. (2007). *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University. CMU-CS-07-129.

Felty, A. P., Momigliano, A., and Pientka, B. (2015a). The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 1-A common infrastructure for benchmarks. *CoRR*, abs/1503.06095.

Felty, A. P., Momigliano, A., and Pientka, B. (2015b). The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2-A survey. *J. Autom. Reason.*, 55(4):307–372.

Freeman, T. (1994). *Refinement Types for ML*. PhD thesis, Carnegie Mellon University, USA, USA. UMI Order No. GAX94-19722.

Freeman, T. S. and Pfenning, F. (1991). Refinement types for ML. In Wise, D. S., editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277. ACM.

Gaulin, A. and Pientka, B. (2023). Contextual refinement types. In *Proceedings of the Eighteenth Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2023, Rome, Italy, 2nd July 2023*, EPTCS.

Gécseg, F. and Steinby, M. (2015). Tree automata. *CoRR*, abs/1509.06233.

Harper, R., Honsell, F., and Plotkin, G. D. (1993). A framework for defining logics. *J. ACM*, 40(1):143–184.

Harper, R. and Licata, D. R. (2007). Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673.

Jacob-Rao, R., Pientka, B., and Thibodeau, D. (2018). Index-stratified types. In Kirchner, H., editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 19:1–19:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Jones, E. and Ramsay, S. (2021). Intensional datatype refinement: With application to scalable verification of pattern-match safety. *Proc. ACM Program. Lang.*, 5(POPL).

Lee, D. K., Crary, K., and Harper, R. (2007). Towards a mechanized metatheory of standard ML. In Hofmann, M. and Felleisen, M., editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 173–184. ACM.

Lehmann, N., Geller, A. T., Vazou, N., and Jhala, R. (2023). Flux: Liquid types for Rust. *Proc. ACM Program. Lang.*, 7(PLDI):1533–1557.

Lovas, W. (2010). *Refinement Types for Logical Frameworks*. PhD thesis, Carnegie Mellon University, USA.

Lovas, W. and Pfenning, F. (2010). Refinement types for logical frameworks and their interpretation as proof irrelevance. *Log. Methods Comput. Sci.*, 6(4).

Nanevski, A., Pfenning, F., and Pientka, B. (2008). Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3):23:1–23:49.

Pfenning, F. (1997). *Computation and Deduction*. Cambridge University Press. In preparation. Draft from April 1997 available electronically.

Pfenning, F. (2008). Church and Curry: Combining intrinsic and extrinsic typing. In C.Benzmüller, C.Brown, J.Siekmann, and R.Statman, editors, *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday*, Studies in Logic 17, pages 303–338. College Publications.

Pfenning, F. and Elliott, C. (1988). Higher-order abstract syntax. In *Proceedings of the ACM*

SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88, page 199–208, New York, NY, USA. Association for Computing Machinery.

Pfenning, F. and Schürmann, C. (1999). System description: Twelf — a meta-logical framework for deductive systems. In *Automated Deduction — CADE-16*, pages 202–206, Berlin, Heidelberg. Springer Berlin Heidelberg.

Pientka, B. (2003). *Tabled higher-order logic programming.* PhD thesis, Carnegie Mellon University, USA, Pittsburgh, Pa.

Pientka, B. (2008). A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In Necula, G. C. and Wadler, P., editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 371–382. ACM.

Pientka, B. and Abel, A. (2015). Well-founded recursion over contextual objects. In Altenkirch, T., editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 273–287. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Pientka, B. and Dunfield, J. (2008). Programming with proofs and explicit contexts. In Antoy, S. and Albert, E., editors, *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pages 163–173. ACM.

Pientka, B. and Dunfield, J. (2010). Beluga: A framework for programming and reasoning

with deductive systems (system description). In Giesl, J. and Hähnle, R., editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21. Springer.

Pientka, B. and Pfenning, F. (2003). Optimizing higher-order pattern unification. In Baader, F., editor, *Automated Deduction – CADE-19*, pages 473–487, Berlin, Heidelberg. Springer Berlin Heidelberg.

Reynolds, J. C. (1997). *Design of the Programming Language Forsythe*, pages 173–233. Birkhäuser Boston, Boston, MA.

Rondon, P. M., Kawaguchi, M., and Jhala, R. (2010). Low-level liquid types. In Hermenegildo, M. V. and Palsberg, J., editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 131–144. ACM.

Rondon, P. M., Kawaguci, M., and Jhala, R. (2008). Liquid types. *SIGPLAN Not.*, 43(6):159–169.

Rossberg, A., Russo, C. V., and Dreyer, D. (2010). F-ing modules. In Kennedy, A. and Benton, N., editors, *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010*, pages 89–102. ACM.

Schürmann, C. E. (2000). *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, USA, USA. AAI9986626.

Stone, C. A. and Harper, R. (2006). Extensional equivalence and singleton types. *ACM Trans. Comput. Log.*, 7(4):676–722.

Vazou, N. (2016). *Liquid Haskell: Haskell as a Theorem Prover*. PhD thesis, University of California, San Diego, USA.

Virga, R. (1999). *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, USA. AAI9950039.

Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. (2002). A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University.

Xi, H. (1998). *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, USA, USA. AAI9918624.

Xi, H. and Pfenning, F. (1999). Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 214–227, New York, NY, USA. Association for Computing Machinery.

Appendix A

Definition of Contextual LFR

A.1 Syntax

A.1.1 Syntax of CLFR objects

Heads	$H ::=$	$\mathbf{c} \mid x \mid b.k$
Spines	$\vec{M} ::=$	$\mathtt{nil} \mid M; \vec{M}$
Neutral terms	$R ::=$	$H\ \vec{M} \mid u[\sigma]$
Normal terms	$M ::=$	$R \mid \lambda x.M$
Substitutions	$\sigma ::=$	$\cdot \mid \mathtt{id}_\psi \mid \sigma, \vec{M}$

A.1.2 Syntax of CLFR classifiers

Syntactic category	Type level	Refinement level
Kinds	$K ::= \texttt{type} \mid \Pi x{:}A.K$	$L ::= \texttt{sort} \mid \Pi x{::}S.L$
Atomic families	$P ::= \mathbf{a} \cdot \vec{M}$	$Q ::= \mathbf{s} \cdot \vec{M}$
Canonical families	$A ::= P \mid \Pi x{:}A_1.A_2$	$S ::= Q \mid \Pi x{::}S_1.S_2$
Blocks	$C ::= A \mid \Sigma x{:}A.C$	$D ::= S \mid \Sigma x{::}S.D$
Worlds	$V ::= C \mid \Pi x{:}A.V$	$W ::= D \mid \Pi x{::}S.W$
Contexts	$\Gamma ::= \cdot \mid \psi \mid \Gamma, x{:}A \mid \Gamma, b{:}(E \cdot \vec{M})$	$\Psi ::= \cdot \mid \psi \mid \Psi, x{::}S \mid \Psi, b{::}(F \cdot \vec{M})$
Schemas	$G ::= \cdot \mid G + \mathbf{v}$	$H ::= \cdot \mid H + \mathbf{w}$

A.1.3 Syntax of meta-layer

Syntactic category	Type level	Refinement level
Meta-objects	$\mathcal{M} ::= \hat{\Gamma}.R \mid \hat{\Gamma}.\sigma \mid \Gamma$	$\mathcal{N} ::= \hat{\Psi}.R \mid \hat{\Psi}.\sigma \mid \Psi$
Meta-types	$\mathcal{A} ::= \Gamma.P \mid \Gamma_1.\Gamma_2 \mid G$	$\mathcal{S} ::= \Psi.Q \mid \Psi_1.\Psi_2 \mid H$
Meta-contexts	$\Delta ::= \cdot \mid \Delta, X{:}\mathcal{A}$	$\Omega ::= \cdot \mid \Omega, X{:}\mathcal{S}$
Meta-substitutions	$\theta ::= \cdot \mid \theta, \mathcal{M}$	$\rho ::= \cdot \mid \rho, \mathcal{N}$
Meta-variables	$X ::= u \mid \psi$	

A.2 Type-level judgments

A.2.1 Type formation

$\boxed{\Delta;\Gamma \vdash K : \texttt{kind}}$ – K is a well-formed kind.

$$\frac{\Delta \vdash \Gamma\ \texttt{ctx}}{\Delta;\Gamma \vdash \texttt{type} : \texttt{kind}}\ \textbf{K-type} \qquad \frac{\Delta;\Gamma, x{:}A \vdash K : \texttt{kind}}{\Delta;\Psi \vdash \Pi x{:}A.K : \texttt{kind}}\ \textbf{K-pi}$$

$\boxed{\Delta;\Gamma \vdash A \Leftarrow \texttt{type}}$ – Canonical type family A is well-formed.

$$\frac{(\mathbf{a}{:}K) \in \Sigma \qquad \Delta;\Gamma \vdash \vec{M} : K > \texttt{type}}{\Delta;\Gamma \vdash \mathbf{a}\ \vec{M} \Leftarrow \texttt{type}}\ \textbf{T-atom}$$

$$\frac{\Delta;\Gamma \vdash A_1 : \texttt{type} \qquad \Delta;\Gamma, x{:}A_1 \vdash A_2 \Leftarrow \texttt{type}}{\Delta;\Gamma \vdash \Pi x{:}A_1.A_2 \Leftarrow \texttt{type}}\ \textbf{T-pi}$$

$\boxed{\Delta;\Gamma \vdash \vec{M} : K > \texttt{type}}$ – Check spine $\vec{M}$ against K with target $\texttt{type}$.

$$\frac{\Delta \vdash \Gamma\ \texttt{ctx}}{\Delta;\Gamma \vdash \texttt{nil} : \texttt{type} > \texttt{type}}\ \textbf{K-spn-nil}$$

$$\frac{\Delta;\Gamma \vdash M \Leftarrow A \qquad \Delta;\Gamma \vdash \vec{M} : [M/x]K > \texttt{type}}{\Delta;\Gamma \vdash (M;\vec{M}) : \Pi x{:}A.K > \texttt{type}}\ \textbf{K-spn-cons}$$

A.2.2 Typing

$\boxed{\Delta;\Gamma \vdash H \Rightarrow A}$ – Synthesize type A for head H

$$\frac{(\mathbf{c}{:}A) \in \Sigma \quad \Delta \vdash \Gamma\ \texttt{ctx}}{\Delta;\Gamma \vdash \mathbf{c} \Rightarrow A}\ \textbf{TS-const} \qquad \frac{(x{:}A \in \Gamma) \quad \Delta \vdash \Gamma\ \texttt{ctx}}{\Delta;\Gamma \vdash x \Rightarrow A}\ \textbf{TS-x}$$

$$\frac{(b : \mathbf{w}[\vec{M}]) \in \Gamma \quad \Delta;\Gamma \vdash \mathbf{w}[\vec{M}] > C \quad \Delta;\Gamma \vdash b : C \gg_1^k A}{\Delta;\Gamma \vdash b.k \Rightarrow A}\ \textbf{TS-b}$$

$\boxed{\Delta; \Gamma \vdash V[\vec{M}] > C}$ – Instantiate block C for element V applied to $\vec{M}$

$$\frac{}{\Delta; \Gamma \vdash C[\mathtt{nil}] > C}\ \textbf{Inst-nil} \qquad \frac{(\mathbf{w}{:}V) \in \Sigma \quad \Delta; \Gamma \vdash V[\vec{M}] > C}{\Delta; \Gamma \vdash \mathbf{w}[\vec{M}] > C}\ \textbf{Inst-const}$$

$$\frac{\Delta; \Gamma \vdash M \Leftarrow A \quad \Delta; \Gamma \vdash ([M/x]V)[\vec{M}] > C}{\Delta; \Gamma \vdash \Pi x{:}A.V[M; \vec{M}] > C}\ \textbf{Inst-pi}$$

$\boxed{\Delta; \Gamma \vdash b : C \gg_i^k A}$ – Extract type A for k^{th} projection of block variable b

$$\frac{}{\Delta; \Gamma \vdash b : \Sigma x{:}A.C \gg_k^k A}\ \textbf{Ext-stop} \qquad \frac{\Delta; \Gamma \vdash b : [b.i/x]C \gg_{i+1}^k A}{\Delta; \Gamma \vdash b : \Sigma x{:}A'.C \gg_i^k A}\ \textbf{Ext-cont}$$

$\boxed{\Delta; \Gamma \vdash R \Rightarrow A}$ – Synthesize type A for neutral term R

$$\frac{\Delta; \Gamma \vdash H \Rightarrow A' \quad \Delta; \Gamma \vdash \vec{M} : A' > A}{\Delta; \Gamma \vdash H\ \vec{M} \Rightarrow A}\ \textbf{TS-app}$$

$$\frac{(u : \Gamma'.A) \in \Delta \quad \Delta; \Gamma \vdash \sigma : \Gamma'}{\Delta; \Gamma \vdash u[\sigma] : [\sigma]A}\ \textbf{TS-mvar}$$

$\boxed{\Delta; \Gamma \vdash \vec{M} : A > P}$ – Apply type A to spine $\vec{M}$ to obtain type P

$$\frac{}{\Delta; \Gamma \vdash \mathtt{nil} : P > P}\ \textbf{TC-spn-nil}$$

$$\frac{\Delta; \Gamma \vdash M \Leftarrow A_1 \quad \Delta; \Gamma \vdash \vec{M} : [M/x]A_2 > P}{\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A_1.A_2 > P}\ \textbf{TC-spn-cons}$$

$\boxed{\Delta; \Gamma \vdash M \Leftarrow A}$ – Check normal term M against type A.

$$\frac{\Delta; \Gamma \vdash R \Rightarrow P}{\Delta; \Gamma \vdash R \Leftarrow P}\ \textbf{TC-conv} \qquad \frac{\Delta; \Gamma, x{:}A \vdash M \Leftarrow A'}{\Delta; \Gamma \vdash \lambda x.M \Leftarrow \Pi x{:}A.A'}\ \textbf{TC-lam}$$

A.2.3 Schemas

$\boxed{\Delta;\Gamma \vdash C : \texttt{block}}$ – Block of declarations C is well-formed

$$\frac{\Delta \vdash \Gamma : \texttt{ctx}}{\Delta;\Gamma \vdash \cdot : \texttt{block}}\ \textbf{B-empty} \qquad \frac{\Delta;\Gamma \vdash A \Leftarrow \texttt{type} \quad \Delta;\Gamma, x{:}A \vdash C : \texttt{block}}{\Delta;\Gamma \vdash \Sigma x{:}A.C : \texttt{block}}\ \textbf{B-sigma}$$

$\boxed{\Delta;\Gamma \vdash V : \texttt{world}}$ – World V is well-formed

$$\frac{\Delta;\Gamma \vdash C : \texttt{block}}{\Delta;\Gamma \vdash C : \texttt{world}}\ \textbf{W-conv} \qquad \frac{\Delta;\Gamma \vdash A \Leftarrow \texttt{type} \quad \Delta;\Gamma, x{:}A \vdash V : \texttt{world}}{\Delta;\Gamma \vdash \Pi x{:}A.V : \texttt{world}}\ \textbf{W-pi}$$

$\boxed{\Delta \vdash G : \texttt{schema}}$ – Schema G is well-formed

$$\frac{\vdash \Delta : \texttt{mctx}}{\Delta \vdash \cdot : \texttt{schema}}\ \textbf{S-empty} \qquad \frac{\Delta \vdash G : \texttt{schema} \quad \mathbf{w} \notin \Sigma \quad \Delta;\cdot \vdash V : \texttt{world}}{\Delta \vdash G + \mathbf{w}{:}V : \texttt{schema}}\ \textbf{S-ext}$$

$\boxed{\Delta \vdash \Gamma : G}$ – LF context Γ has schema G in meta-context Δ

$$\frac{\Delta \vdash G : \texttt{schema}}{\Delta \vdash \cdot : G}\ \textbf{SC-empty} \qquad \frac{(\psi : G) \in \Delta}{\Delta \vdash \psi : G}\ \textbf{SC-var}$$

$$\frac{\Delta \vdash \Gamma : G \quad (\mathbf{w}{:}V) \in G \quad \Delta;\Gamma \vdash \mathbf{w}[\vec{M}] > C}{\Delta \vdash (\Gamma, b{:}\mathbf{w}[\vec{M}]) : G}\ \textbf{SC-ext}$$

A.2.4 Contexts

$\boxed{\Delta \vdash \Gamma : \texttt{ctx}}$ – LF context Γ is well-formed in meta-context Δ

$$\frac{\vdash \Delta : \texttt{mctx}}{\Delta \vdash \cdot : \texttt{ctx}}\ \textbf{C-empty} \qquad \frac{(\psi : G) \in \Delta \quad \vdash \Delta : \texttt{mctx}}{\Delta \vdash \psi : \texttt{ctx}}\ \textbf{C-var}$$

$$\frac{\Delta \vdash \Gamma : \texttt{ctx} \quad \Delta;\Gamma \vdash A \Leftarrow \texttt{type}}{\Delta \vdash (\Gamma, x{:}A) : \texttt{ctx}}\ \textbf{C-cons-x} \qquad \frac{\Delta \vdash \Gamma : \texttt{ctx} \quad \Delta;\Gamma \vdash \mathbf{w}[\vec{M}] > C}{\Delta \vdash (\Gamma, b{:}\mathbf{w}[\vec{M}]) : \texttt{ctx}}\ \textbf{C-cons-b}$$

$\boxed{\Delta;\Gamma_1 \vdash \sigma : \Gamma_2}$ – σ is a well-formed subsitution from Γ_2 to Γ_1

$$\frac{\Delta \vdash \Gamma_1 : \texttt{ctx}}{\Delta;\Gamma_1 \vdash \cdot : \cdot}\ \textbf{Subst-empty} \qquad \frac{(\psi : G) \in \Delta \quad \Delta \vdash \Gamma_1 : \texttt{ctx}}{\Delta;\Gamma_1 \vdash \texttt{id}_\psi : \psi}\ \textbf{Subst-id}$$

$$\frac{\Delta;\Gamma_1 \vdash \sigma : \Gamma_2 \quad \Delta;\Gamma_1 \vdash M \Leftarrow [\sigma]A}{\Delta;\Gamma_1 \vdash (\sigma, M) : (\Gamma_2, x{:}A)}\ \textbf{Subst-tm}$$

$$\frac{\Delta;\Gamma_1 \vdash \sigma : \Gamma_2 \quad \Delta;\Gamma_2 \vdash \mathbf{w}[\vec{M}'] > C \quad \Delta;\Gamma_1 \vdash \vec{M} \Leftarrow [\sigma]C}{\Delta;\Gamma_1 \vdash (\sigma, \vec{M}) : (\Gamma_2, b{:}\mathbf{w}[\vec{M}'])}\ \textbf{Subst-spn}$$

$\boxed{\Delta;\Gamma \vdash \vec{M} \Leftarrow C}$ – n-ary tuple $\vec{M}$ checks against block of declarations C

$$\frac{\Delta \vdash \Gamma : \mathtt{ctx}}{\Delta;\Gamma \vdash \mathtt{nil} \Leftarrow \cdot}\ \textbf{Chk-spn-nil} \qquad \frac{\Delta;\Gamma \vdash M \Leftarrow A \quad \Delta;\Gamma \vdash \vec{M} \Leftarrow [M/x]C}{\Delta;\Gamma \vdash (M; \vec{M}) \Leftarrow \Sigma x{:}A.C}\ \textbf{Chk-spn-sigma}$$

A.2.5 Meta-layer

$\boxed{\vdash \Delta : \mathtt{mctx}}$ – Δ is a well-formed meta-context

$$\frac{}{\vdash \cdot : \mathtt{mctx}}\ \textbf{MC-nil} \qquad \frac{\vdash \Delta : \mathtt{mctx} \quad \Delta \vdash \mathcal{A} : \mathtt{mtype}}{\vdash (\Delta, X{:}\mathcal{A}) : \mathtt{mctx}}\ \textbf{MC-cons}$$

$\boxed{\Delta \vdash \mathcal{A} : \mathtt{mtype}}$ – $\mathcal{A}$ is a well-formed meta-type in meta-context Δ

$$\frac{\Delta;\Gamma \vdash P \Leftarrow \mathtt{type}}{\Delta \vdash (\Gamma.P) : \mathtt{mtype}}\ \textbf{MT-tp} \qquad \frac{\Delta \vdash G : \mathtt{schema}}{\Delta \vdash G : \mathtt{mtype}}\ \textbf{MT-schema}$$

$$\frac{\Delta \vdash \Gamma_1 : \mathtt{ctx} \quad \Delta \vdash \Gamma_2 : \mathtt{ctx}}{\Delta \vdash (\Gamma_1.\Gamma_2) : \mathtt{mtype}}\ \textbf{MT-subst}$$

$\boxed{\Delta \vdash \mathcal{M} : \mathcal{A}}$ – Meta-objects $\mathcal{M}$ has meta-type $\mathcal{A}$

$$\frac{\Delta;\Gamma \vdash R \Leftarrow P}{\Delta \vdash (\hat{\Gamma}.R) : (\Gamma.P)}\ \textbf{MOft-tm} \qquad \frac{\Delta;\Gamma_1 \vdash \sigma : \Gamma_2}{\Delta \vdash (\hat{\Gamma}_1.\sigma) : (\Gamma_1.\Gamma_2)}\ \textbf{MOft-subst}$$

$$\frac{\Delta \vdash \Gamma : G \text{ (schema checking)}}{\Delta \vdash \Gamma : G \text{ (mtype checking)}}\ \textbf{MOft-ctx}$$

$\boxed{\Delta_1 \vdash \theta : \Delta_2}$ – θ is a well-formed meta-substitution from Δ_2 to Δ_1

$$\frac{\vdash \Delta_1 : \mathtt{mctx}}{\Delta_1 \vdash \cdot : \cdot}\ \textbf{MSubst-nil} \qquad \frac{\Delta_1 \vdash \theta : \Delta_2 \quad \Delta_1 \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}}{\Delta_1 \vdash (\theta, \mathcal{M}) : (\Delta_2, X{:}\mathcal{A})}\ \textbf{MSubst-cons}$$

A.3 Refinement-level judgments

A.3.1 Sort formation

$\boxed{(\Omega; \Psi \vdash L) \sqsubset (\Delta; \Gamma \vdash K : \texttt{kind})}$ – L is a kind refinement of K

$$\frac{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx})}{(\Omega; \Psi \vdash \texttt{sort}) \sqsubset (\Delta; \Gamma \vdash \texttt{type} : \texttt{kind})}\ \textbf{KR-sort}$$

$$\frac{(\Omega; \Psi, x{:}S \vdash L) \sqsubset (\Delta; \Gamma, x{:}A \vdash K : \texttt{kind})}{(\Omega; \Psi \vdash \Pi x{:}S.L) \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A.K : \texttt{kind})}\ \textbf{KR-pi}$$

$\boxed{(\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \texttt{type})}$ – Sort S refines type A

$$\frac{(\mathbf{s}{:}L \sqsubset \mathbf{a}{:}K) \in \Sigma \quad (\Omega; \Psi \vdash \vec{M} : L > \texttt{sort}) \sqsubset (\Delta; \Gamma \vdash \vec{M} : K > \texttt{type})}{(\Omega; \Psi \vdash \mathbf{s}\ \vec{M}) \sqsubset (\Delta; \Gamma \vdash \mathbf{a}\ \vec{M} \Leftarrow \texttt{type})}\ \textbf{TR-atom}$$

$$\frac{\begin{array}{l}(\Omega; \Psi \vdash S_1) \sqsubset (\Delta; \Gamma \vdash A_1 \Leftarrow \texttt{type}) \\ (\Omega; \Psi, x{:}S_1 \vdash S_2) \sqsubset (\Delta; \Gamma, x{:}A_1 \vdash A_2 \Leftarrow \texttt{type})\end{array}}{(\Omega; \Psi \vdash \Pi x{:}S_1.S_2) \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A_1.A_2 \Leftarrow \texttt{type})}\ \textbf{TR-pi}$$

$\boxed{(\Omega; \Psi \vdash \vec{M} : L > \texttt{sort}) \sqsubset (\Delta; \Gamma \vdash \vec{M} : K > \texttt{type})}$ – Check $\vec{M}$ against L with target $\texttt{sort}$

$$\frac{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx})}{(\Omega; \Psi \vdash \texttt{nil} : \texttt{sort} > \texttt{sort}) \sqsubset (\Delta; \Gamma \vdash \texttt{nil} : \texttt{type} > \texttt{type})}\ \textbf{KR-spn-nil}$$

$$\frac{\begin{array}{c}(\Omega; \Psi \vdash M \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A) \\ (\Omega; \Psi \vdash \vec{M} : [M/x]L > \texttt{sort}) \sqsubset (\Delta; \Gamma \vdash \vec{M} : [M/x]K > \texttt{type})\end{array}}{(\Omega; \Psi \vdash (M; \vec{M}) : \Pi x{:}S.L > \texttt{sort}) \sqsubset (\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A.K > \texttt{type})}\ \textbf{KR-spn-cons}$$

A.3.2 Sorting

$\boxed{(\Omega; \Psi \vdash H \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash H \Rightarrow A)}$ – Synthesize sort S for head H

$$\frac{(\mathbf{c} :: S \sqsubset A) \in \Sigma \quad (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx})}{(\Omega; \Psi \vdash \mathbf{c} \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash \mathbf{c} \Rightarrow A)}\ \textbf{TRS-const}$$

$$\frac{((x{:}S) \in \Psi) \sqsubset ((x{:}A) \in \Gamma) \quad (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx})}{(\Omega; \Psi \vdash x \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash x \Rightarrow A)}\ \textbf{TRS-x}$$

$$\frac{\begin{array}{l}((b{:}\mathbf{w}[\vec{M}]) \in \Psi) \sqsubset ((b{:}\mathbf{w}[\vec{M}]) \in \Gamma) \\ (\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \mathbf{w}[\vec{M}] > C) \\ (\Omega; \Psi \vdash b : D \gg_1^k S) \sqsubset (\Delta; \Gamma \vdash b : C \gg_1^k A)\end{array}}{(\Omega; \Psi \vdash b.k \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash b.k \Rightarrow A)}\ \textbf{TRS-b}$$

$\boxed{(\Omega; \Psi \vdash W[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash V[\vec{M}] > C)}$ – Instantiate block D for element $W\,\vec{M}$

$$\frac{}{(\Omega; \Psi \vdash D[\mathtt{nil}] > D) \sqsubset (\Delta; \Gamma \vdash C[\mathtt{nil}] > C)}\ \textbf{R-Inst-nil}$$

$$\frac{(\mathbf{w} : W \sqsubset V) \in \Sigma \quad (\Omega; \Psi \vdash W[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash V[\vec{M}] > C)}{(\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \mathbf{v}[\vec{M}] > C)}\ \textbf{R-Inst-const}$$

$$\frac{(\Omega; \Psi \vdash M \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A) (\Omega; \Psi \vdash ([M/x]W)[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash ([M/x]V)[\vec{M}] > C)}{(\Omega; \Psi \vdash \Pi x{:}S.W[M; \vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A.V[M; \vec{M}] > C)}\ \textbf{R-Inst-pi}$$

$\boxed{(\Omega; \Psi \vdash b : D \gg_i^k S) \sqsubset (\Delta; \Gamma \vdash b : C \gg_i^k A)}$ – Extract sort S for k^{th} projection of b

$$\frac{}{(\Omega; \Psi \vdash b : \Sigma x{:}S.D \gg_k^k S) \sqsubset (\Delta; \Gamma \vdash b : \Sigma x{:}A.C \gg_k^k A)}\ \textbf{R-Ext-stop}$$

$$\frac{(\Omega; \Psi \vdash b : [b.i/x]D \gg_{i+1}^k S) \sqsubset (\Delta; \Gamma \vdash b : [b.i/x]C \gg_{i+1}^k A)}{(\Omega; \Psi \vdash b : \Sigma x{:}S'.D \gg_i^k S) \sqsubset (\Delta; \Gamma \vdash b : \Sigma x{:}A'.C \gg_i^k A)}\ \textbf{R-Ext-cont}$$

$\boxed{(\Omega; \Psi \vdash R \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash R \Rightarrow A)}$ – Synthesize sort S for neutral term R

$$\frac{\begin{array}{l}(\Omega; \Psi \vdash H \Rightarrow S') \sqsubset (\Delta; \Gamma \vdash H \Rightarrow A') \\ (\Omega; \Psi \vdash \vec{M} : S' > S) \sqsubset (\Delta; \Gamma \vdash \vec{M} : A' > A)\end{array}}{(\Omega; \Psi \vdash H\ \vec{M} \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash H\ \vec{M} \Rightarrow A)}\ \textbf{TRS-app}$$

$$\frac{(u : \Psi'.S) \in \Omega \quad (u : \Gamma'.A) \in \Delta \quad (\Omega; \Psi \vdash \sigma : \Psi') \sqsubset (\Delta; \Gamma \vdash \sigma : \Gamma')}{(\Omega; \Psi \vdash u[\sigma] : [\sigma]S) \sqsubset (\Delta; \Gamma \vdash u[\sigma] : [\sigma]A)}\ \textbf{TRS-mvar}$$

$\boxed{(\Omega; \Psi \vdash \vec{M} : S > Q) \sqsubset (\Delta; \Gamma \vdash \vec{M} : A > P)}$ – Apply sort S to $\vec{M}$ to obtain sort Q

$$\frac{}{(\Omega; \Psi \vdash \mathtt{nil} : Q > Q) \sqsubset (\Delta; \Gamma \vdash \mathtt{nil} : P > P)}\ \textbf{TRC-spn-nil}$$

$$\frac{\begin{array}{l}(\Omega; \Psi \vdash M \Leftarrow S_1) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A_1) \\ (\Omega; \Psi \vdash \vec{M} : [M/x]S_2 > Q) \sqsubset (\Delta; \Gamma \vdash \vec{M} : [M/x]A_2 > P)\end{array}}{(\Omega; \Psi \vdash (M; \vec{M}) : \Pi x{:}S_1.S_2 > Q) \sqsubset (\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A_1.A_2 > P)}\ \textbf{TRC-spn-cons}$$

$\boxed{(\Omega; \Psi \vdash M \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A)}$ – Check normal term M against sort S.

$$\frac{(\Omega; \Psi \vdash R \Rightarrow Q') \sqsubset (\Delta; \Gamma \vdash R \Rightarrow P) \quad (\Omega; \Psi \vdash Q' \leq Q) \sqsubset (\Delta; \Gamma \vdash P)}{(\Omega; \Psi \vdash R \Leftarrow Q) \sqsubset (\Delta; \Gamma \vdash R \Leftarrow P)}\ \textbf{TRC-conv}$$

$$\frac{(\Omega; \Psi, x{:}S \vdash M \Leftarrow S') \sqsubset (\Delta; \Gamma, x{:}A \vdash M \Leftarrow A')}{(\Omega; \Psi \vdash \lambda x.M \Leftarrow \Pi x{:}S.S') \sqsubset (\Delta; \Gamma \vdash \lambda x.M \Leftarrow \Pi x{:}A.A')}\ \textbf{TRC-lam}$$

A.3.3 Schemas

$\boxed{(\Omega; \Psi \vdash D) \sqsubset (\Delta; \Gamma \vdash C : \texttt{block})}$ – Block of declarations D refines C

$$\frac{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx})}{(\Omega; \Psi \vdash \cdot) \sqsubset (\Delta; \Gamma \vdash \cdot : \texttt{block})}\ \textbf{BR-empty}$$

$$\frac{(\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \texttt{type}) \quad (\Omega; \Psi, x{:}S \vdash D) \sqsubset (\Delta; \Gamma, x{:}A \vdash C : \texttt{block})}{(\Omega; \Psi \vdash \Sigma x{:}S.D) \sqsubset (\Delta; \Gamma \vdash \Sigma x{:}A.C : \texttt{block})}\ \textbf{BR-sigma}$$

$\boxed{(\Omega; \Psi \vdash W) \sqsubset (\Delta; \Gamma \vdash V : \texttt{world})}$ – World W refines V

$$\frac{(\Omega; \Psi \vdash D) \sqsubset (\Delta; \Gamma \vdash C : \texttt{block})}{(\Omega; \Psi \vdash D) \sqsubset (\Delta; \Gamma \vdash C : \texttt{world})}\ \textbf{WR-conv}$$

$$\frac{(\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \texttt{type}) \quad (\Omega; \Psi, x{:}S \vdash W) \sqsubset (\Delta; \Gamma, x{:}A \vdash V : \texttt{world})}{(\Omega; \Psi \vdash \Pi x{:}S.W) \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A.V : \texttt{world})}\ \textbf{WR-pi}$$

$\boxed{(\Omega \vdash H) \sqsubset (\Delta \vdash G : \texttt{schema})}$ – Schema H refines G

$$\frac{(\vdash \Omega) \sqsubset (\vdash \Delta : \texttt{mctx})}{(\Omega \vdash \cdot) \sqsubset (\Delta \vdash \cdot : \texttt{schema})}\ \textbf{SR-empty}$$

$$\frac{\begin{array}{ll}(\mathbf{w}{:}V) \in G & (\Omega \vdash H) \sqsubset (\Delta \vdash G : \texttt{schema}) \\ \mathbf{w} \notin H & (\Omega; \cdot \vdash W) \sqsubset (\Delta; \cdot \vdash V : \texttt{world})\end{array}}{(\Omega \vdash H + \mathbf{w}{:}W) \sqsubset (\Delta \vdash G : \texttt{schema})}\ \textbf{SR-ext}$$

$\boxed{(\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G)}$ – Context Ψ has schema H

$$\frac{(\Omega \vdash H) \sqsubset (\Delta \vdash G : \mathtt{schema})}{(\Omega \vdash \cdot : H) \sqsubset (\Delta \vdash \cdot : G)}\ \textbf{SRC-empty} \qquad \frac{((\psi : H) \in \Omega) \sqsubset ((\psi : G) \in \Delta)}{(\Omega \vdash \psi : H) \sqsubset (\Delta \vdash \psi : G)}\ \textbf{SRC-var}$$

$$\frac{\begin{array}{l}((\mathbf{w}{:}W) \in H) \sqsubset ((\mathbf{w}{:}V) \in G) \\ (\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G) \qquad (\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \mathbf{w}[\vec{M}] > C)\end{array}}{(\Omega \vdash (\Psi, b{:}\mathbf{w}[\vec{M}]) : H) \sqsubset (\Delta \vdash (\Gamma, b{:}\mathbf{v}[\vec{M}]) : G)}\ \textbf{SRC-ext}$$

A.3.4 Contexts

$\boxed{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx})}$ – LFR context Ψ refines Γ

$$\frac{(\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx})}{(\Omega \vdash \cdot) \sqsubset (\Delta \vdash \cdot : \mathtt{ctx})}\ \textbf{CR-empty}$$

$$\frac{((\psi : H) \in \Omega) \sqsubset ((\psi : G) \in \Delta) \quad (\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx})}{(\Omega \vdash \psi) \sqsubset (\Delta \vdash \psi : \mathtt{ctx})}\ \textbf{CR-var}$$

$$\frac{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx}) \quad (\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \mathtt{type})}{(\Omega \vdash \Psi, x{:}S) \sqsubset (\Delta \vdash (\Gamma, x{:}A) : \mathtt{ctx})}\ \textbf{CR-cons-x}$$

$$\frac{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx}) \quad (\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > C)}{(\Omega \vdash \Psi, b{:}\mathbf{w}[\vec{M}]) \sqsubset (\Delta \vdash (\Gamma, b{:}\mathbf{w}[\vec{M}]) : \mathtt{ctx})}\ \textbf{CR-cons-b}$$

$\boxed{(\Omega; \Psi_1 \vdash \sigma : \Psi_2) \sqsubset (\Delta; \Gamma_1 \vdash \sigma : \Gamma_2)}$ – σ is a well-formed subsitution from Ψ_2 to Ψ_1

$$\frac{(\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1 : \mathtt{ctx})}{(\Omega; \Psi_1 \vdash \cdot : \cdot) \sqsubset (\Delta; \Gamma_1 \vdash \cdot : \cdot)}\ \textbf{SubstR-empty}$$

$$\frac{((\psi : H) \in \Omega) \sqsubset ((\psi : G) \in \Delta) \quad (\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1 : \mathtt{ctx})}{(\Omega; \Psi_1 \vdash \mathtt{id}_\psi : \psi) \sqsubset (\Delta; \Gamma_1 \vdash \mathtt{id}_\psi : \psi)}\ \textbf{SubstR-id}$$

$$\frac{\begin{array}{l}(\Omega; \Psi_1 \vdash \sigma : \Psi_2) \sqsubset (\Delta; \Gamma_1 \vdash \sigma : \Gamma_2) \\ (\Omega; \Psi_1 \vdash M \Leftarrow [\sigma]S) \sqsubset (\Delta; \Gamma_1 \vdash M \Leftarrow [\sigma]A)\end{array}}{\big(\Omega; \Psi_1 \vdash (\sigma, M) : (\Psi_2, x{:}S)\big) \sqsubset \big(\Delta; \Gamma_1 \vdash (\sigma, M) : (\Gamma_2, x{:}A)\big)}\ \textbf{SubstR-tm}$$

$$\frac{\begin{array}{l}(\Omega; \Psi_1 \vdash \sigma : \Psi_2) \sqsubset (\Delta; \Gamma_1 \vdash \sigma : \Gamma_2) \\ (\Omega; \Psi_2 \vdash \mathbf{w}[\vec{M}'] > D) \sqsubset (\Delta; \Gamma_2 \vdash \mathbf{w}[\vec{M}'] > C) \\ (\Omega; \Psi_1 \vdash \vec{M} \Leftarrow [\sigma]D) \sqsubset (\Delta; \Gamma_1 \vdash \vec{M} \Leftarrow [\sigma]C)\end{array}}{\big(\Omega; \Psi_1 \vdash (\sigma, \vec{M}) : (\Psi_2, b{:}\mathbf{w}[\vec{M}'])\big) \sqsubset \big(\Delta; \Gamma_1 \vdash (\sigma, \vec{M}) : (\Gamma_2, b{:}\mathbf{w}[\vec{M}'])\big)}\ \textbf{SubstR-spn}$$

$\boxed{(\Omega; \Psi \vdash \vec{M} \Leftarrow D) \sqsubset (\Delta; \Gamma \vdash \vec{M} \Leftarrow C)}$ – n-ary tuple $\vec{M}$ checks against block C

$$\frac{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx})}{(\Omega; \Psi \vdash \mathtt{nil} \Leftarrow \cdot) \sqsubset (\Delta; \Gamma \vdash \mathtt{nil} \Leftarrow \cdot)}\ \textbf{ChkR-spn-nil}$$

$$\frac{\begin{array}{l}(\Omega; \Psi \vdash M \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A) \\ (\Omega; \Psi \vdash \vec{M} \Leftarrow [M/x]D) \sqsubset (\Delta; \Gamma \vdash \vec{M} \Leftarrow [M/x]C)\end{array}}{(\Omega; \Psi \vdash (M; \vec{M}) \Leftarrow \Sigma x{:}S.D) \sqsubset (\Delta; \Gamma \vdash (M; \vec{M}) \Leftarrow \Sigma x{:}A.C)}\ \textbf{ChkR-spn-sigma}$$

A.3.5 Meta-layer

$\boxed{(\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx})}$ – Ω refines Δ

$$\frac{}{(\vdash \cdot) \sqsubset (\vdash \cdot : \mathtt{mctx})}\ \textbf{MCR-nil}$$

$$\frac{(\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx}) \quad (\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \mathtt{mtype})}{(\vdash (\Omega, X{:}\mathcal{S}) \sqsubset (\vdash (\Delta, X{:}\mathcal{A}) : \mathtt{mctx})}\ \textbf{MCR-cons}$$

$\boxed{(\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \mathtt{mtype})}$ – $\mathcal{S}$ refines $\mathcal{A}$ / is a well-formed meta-sort

$$\frac{(\Omega; \Psi \vdash Q) \sqsubset (\Delta; \Gamma \vdash P \Leftarrow \mathtt{type})}{\big(\Omega \vdash (\Psi.Q)\big) \sqsubset \big(\Delta \vdash (\Gamma.P) : \mathtt{mtype}\big)}\ \textbf{MTR-tp}$$

$$\frac{(\Omega \vdash H) \sqsubset (\Delta \vdash G : \mathtt{schema})}{(\Omega \vdash H) \sqsubset (\Delta \vdash G : \mathtt{mtype})}\ \textbf{MTR-schema}$$

$$\frac{(\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1 : \mathtt{ctx}) \quad (\Omega \vdash \Psi_2) \sqsubset (\Delta \vdash \Gamma_2 : \mathtt{ctx})}{\big(\Omega \vdash (\Psi_1.\Psi_2)\big) \sqsubset \big(\Delta \vdash (\Gamma_1.\Gamma_2) : \mathtt{mtype}\big)}\ \textbf{MTR-subst}$$

$\boxed{(\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})}$ – Meta-objects $\mathcal{N}$ has meta-sort $\mathcal{S}$

$$\frac{(\Omega; \Psi \vdash R \Leftarrow Q) \sqsubset (\Delta; \Gamma \vdash R \Leftarrow P)}{\big(\Omega \vdash (\hat{\Psi}.R) : (\Psi.Q)\big) \sqsubset \big(\Delta \vdash (\hat{\Gamma}.R) : (\Gamma.P)\big)}\ \textbf{MOftR-tm}$$

$$\frac{(\Omega; \Psi_1 \vdash \sigma : \Psi_2) \sqsubset (\Delta; \Gamma_1 \vdash \sigma : \Gamma_2)}{\big(\Omega \vdash (\hat{\Psi}_1.\sigma) : (\Psi_1.\Psi_2)\big) \sqsubset \big(\Delta \vdash (\hat{\Gamma}_1.\sigma) : (\Gamma_1.\Gamma_2)\big)}\ \textbf{MOftR-subst}$$

$$\frac{(\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G)\ \text{(schema checking)}}{(\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G)\ \text{(msort checking)}}\ \textbf{MOftR-ctx}$$

$\boxed{(\Omega_1 \vdash \rho : \Omega_2) \sqsubset (\Delta_1 \vdash \theta : \Delta_2)}$ – ρ is a meta-substitution refinement of θ from Ω_2 to Ω_1

$$\frac{(\vdash \Omega_1) \sqsubset (\vdash \Delta_1 : \texttt{mctx})}{(\Omega_1 \vdash \cdot : \cdot) \sqsubset (\Delta_1 \vdash \cdot : \cdot)}\ \textbf{MSubstR-nil}$$

$$\frac{(\Omega_1 \vdash \rho : \Omega_2) \sqsubset (\Delta_1 \vdash \theta : \Delta_2) \quad (\Omega_1 \vdash \mathcal{N} : [\![\rho]\!]\mathcal{S}) \sqsubset (\Delta_1 \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A})}{\big(\Omega_1 \vdash (\rho, \mathcal{N}) : (\Omega_2, X{:}\mathcal{S})\big) \sqsubset \big(\Delta_1 \vdash (\theta, \mathcal{M}) : (\Delta_2, X{:}\mathcal{A})\big)}\ \textbf{MSubstR-cons}$$

A.3.6 Subsorting rules

For LFR sorts $\boxed{(\Omega; \Psi \vdash S_1 \leq S_2) \sqsubset (\Delta; \Gamma \vdash A)}$ – S_1 is a subsort of S_2

$$\frac{(\text{LFR } \mathbf{s}_1 \leq \mathbf{s}_2 \sqsubset \mathbf{a} : L) \in \Sigma \quad (\Omega; \Psi \vdash \vec{M} : L > \texttt{sort}) \sqsubset (\Delta; \Gamma \vdash \vec{M} : K > \texttt{type})}{(\Omega; \Psi \vdash \mathbf{s}_1\ \vec{M} \leq \mathbf{s}_2\ \vec{M}) \sqsubset (\Delta; \Gamma \vdash \mathbf{a}\ \vec{M})}\ \textbf{Sub-atom}$$

$$\frac{(\Omega; \Psi \vdash Q) \sqsubset (\Delta; \Gamma \vdash P)}{(\Omega; \Psi \vdash Q \leq Q) \sqsubset (\Delta; \Gamma \vdash P)}\ \textbf{Sub-refl}$$

$$\frac{(\Omega; \Psi \vdash S_1 \leq S_2) \sqsubset (\Delta; \Gamma \vdash A) \quad (\Omega; \Psi \vdash S_2 \leq S_3) \sqsubset (\Delta; \Gamma \vdash A)}{(\Omega; \Psi \vdash S_1 \leq S_3) \sqsubset (\Delta; \Gamma \vdash A)}\ \textbf{Sub-trans}$$

$$\frac{(\Omega; \Psi \vdash S_2 \leq S_1) \sqsubset (\Delta; \Gamma \vdash A) \quad (\Omega; \Psi, x{:}S_2 \vdash S_1' \leq S_2') \sqsubset (\Delta; \Gamma, x{:}A \vdash A')}{(\Omega; \Psi \vdash \Pi x{:}S_1.S_1' \leq \Pi x{:}S_2.S_2') \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A.A')}\ \textbf{Sub-pi}$$

For blocks $\boxed{(\Omega; \Psi \vdash D_1 \leq D_2) \sqsubset (\Delta; \Gamma \vdash C)}$ – D_1 is a sub-block of D_2

$$\frac{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma)}{(\Omega; \Psi \vdash \cdot \leq \cdot) \sqsubset (\Delta; \Psi \vdash \cdot)}\ \textbf{Sub-Bnil}$$

$$\frac{(\Omega; \Psi \vdash S_1 \leq S_2) \sqsubset (\Delta; \Gamma \vdash A) \quad (\Omega; \Psi, x{:}S_1 \vdash D_1 \leq D_2) \sqsubset (\Delta; \Gamma \vdash C)}{(\Omega; \Psi \vdash \Sigma x{:}S_1.D_1 \leq \Sigma x{:}S_2.D_2) \sqsubset (\Delta; \Gamma \vdash \Sigma x{:}A.C)}\ \textbf{Sub-sigma}$$

For worlds $\boxed{(\Omega; \Psi \vdash W_1 \leq W_2) \sqsubset (\Delta; \Gamma \vdash V : \texttt{world})}$ – W_1 is a sub-world of W_2

$$\frac{(\Omega; \Psi \vdash D_1 \leq D_2) \sqsubset (\Delta; \Psi \vdash C : \texttt{block})}{(\Omega; \Psi \vdash D_1 \leq D_2) \sqsubset (\Delta; \Psi \vdash C : \texttt{world})}\ \textbf{SubW-conv}$$

$$\frac{(\Omega; \Psi \vdash S_2 \leq S_1) \sqsubset (\Delta; \Gamma \vdash A) \quad (\Omega; \Psi, x{:}S_2 \vdash S_1' \leq S_2') \sqsubset (\Delta; \Gamma \vdash A')}{(\Omega; \Psi \vdash \Pi x{:}S_1.S_1' \leq \Pi x{:}S_2.S_2') \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A.A')}\ \textbf{SubW-pi}$$

For schemas $\boxed{(\Omega; \Psi \vdash H_1 \leq H_2) \sqsubset (\Delta; \Gamma \vdash G)}$ – H_1 is a sub-schema of H_2

$$\frac{(\Omega \vdash H) \sqsubset (\Delta \vdash G : \texttt{schema})}{(\Omega \vdash \cdot \leq H) \sqsubset (\Delta \vdash G)}\ \textbf{SubS-nil}$$

$$\frac{\begin{array}{ll}(\mathbf{w}{:}V \in G) & (\Omega \vdash H_1 \leq H_2) \sqsubset (\Delta \vdash G : \texttt{schema}) \\ (\mathbf{w} \notin H_i) & (\Omega; \Psi \vdash W_1 \leq W_2) \sqsubset (\Delta; \Gamma \vdash V : \texttt{world})\end{array}}{(\Omega \vdash H_1 + \mathbf{w}{:}W_1 \leq H_2 + \mathbf{w}{:}W_2) \sqsubset (\Delta; \Gamma \vdash G)}\ \textbf{SubS-sum}$$

For meta-sorts $\boxed{(\Omega \vdash \mathcal{S}_1 \leq \mathcal{S}_2) \sqsubset (\Delta \vdash A)}$ – $\mathcal{S}_1$ is a sub-meta-sort of $\mathcal{S}_2$

$$\frac{(\Omega; \Psi \vdash S_1 \leq S_2) \sqsubset (\Delta; \Gamma \vdash A)}{(\Omega \vdash \Psi.S_1 \leq \Psi.S_2) \sqsubset (\Delta \vdash \Gamma.A)}\ \textbf{SubM-tp}$$

$$\frac{(\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1) \quad (\Omega \vdash \Psi_2) \sqsubset (\Delta \vdash \Gamma_2)}{(\Omega \vdash \Psi_1.\Psi_2 \leq \Psi_1.\Psi_2) \sqsubset (\Delta \vdash \Gamma_1.\Gamma_2)}\ \textbf{SubM-subst}$$

$$\frac{(\Omega \vdash H_1 \leq H_2) \sqsubset (\Delta \vdash G : \texttt{schema})}{(\Omega \vdash H_1 \leq H_2) \sqsubset (\Delta \vdash G : \texttt{mtype})}\ \textbf{SubM-schema}$$

Appendix B

Definition of Beluga

B.1 Syntax

Category	Type level	Refinement level
Types	$\tau ::= [\mathcal{A}] \mid \tau_1 \to \tau_2 \mid \Pi X{:}\mathcal{A}.\tau$	$\zeta ::= [\mathcal{S}] \mid \zeta_1 \to \zeta_2 \mid \Pi X{::}\mathcal{S}.\zeta$
Expressions	$e ::= y \mid [\mathcal{M}] \mid \texttt{fn}\ y{:}\tau \Rightarrow e \mid e_1\ e_2$	$f ::= y \mid [\mathcal{N}] \mid \texttt{fn}\ y{::}\zeta \Rightarrow f \mid f_1\ f_2$
	$\mid \texttt{mlam}\ X{:}\mathcal{A} \Rightarrow e \mid e\ \mathcal{M}$	$\mid \texttt{mlam}\ X{::}\mathcal{S} \Rightarrow f \mid f\ \mathcal{M}$
	$\mid \texttt{let}\ [X] = e_1\ \texttt{in}\ e_2$	$\mid \texttt{let}\ [X] = f_1\ \texttt{in}\ f_2$
	$\mid \texttt{case}^{\tau}\ [\mathcal{M}]\ \texttt{of}\ \vec{b}$	$\mid \texttt{case}^{\zeta}\ [\mathcal{N}]\ \texttt{of}\ \vec{c}$
Branches	$b ::= \Delta; [\mathcal{M}] \Rightarrow e$	$c ::= \Phi; [\mathcal{N}] \Rightarrow f$
Contexts	$\Omega ::= \cdot \mid \Omega, y{:}\tau$	$\Xi ::= \cdot \mid \Xi, y{::}\zeta$

B.2 Type-level judgments

$\boxed{\Delta \vdash \Xi : \texttt{cctx}}$ – Computation context formation

$$\frac{\vdash \Delta : \texttt{mctx}}{\Delta \vdash \cdot : \texttt{cctx}}\ \textbf{CC-nil} \qquad \frac{\Delta \vdash \Xi : \texttt{cctx} \quad \Delta; \Xi \vdash \tau : \texttt{ctype}}{\Delta \vdash \Xi, y{:}\tau : \texttt{cctx}}\ \textbf{CC-cons}$$

$\boxed{\Delta; \Xi \vdash \tau : \texttt{ctype}}$ – Type formation

$$\frac{\Delta \vdash \Xi : \texttt{cctx} \quad \Delta \vdash \mathcal{A} : \texttt{ctype}}{\Delta; \Xi \vdash [\mathcal{A}] : \texttt{ctype}}\ \textbf{CT-meta} \qquad \frac{\Delta; \Xi \vdash \tau_1 : \texttt{ctype} \quad \Delta; \Xi \vdash \tau_2 : \texttt{ctype}}{\Delta; \Xi \vdash \tau_1 \to \tau_2 : \texttt{ctype}}\ \textbf{CT-arr}$$

$$\frac{\Delta \vdash \mathcal{A} : \texttt{mtype} \quad \Delta, X{:}\mathcal{A}; \Xi \vdash \tau : \texttt{ctype}}{\Delta; \Xi \vdash \Pi X{:}\mathcal{A}.\tau : \texttt{ctype}}\ \textbf{CT-pi}$$

$\boxed{\Delta; \Xi \vdash e : \tau}$ – Expression e has type τ

$$\frac{\Delta \vdash \Xi : \texttt{cctx} \quad (y{:}\tau) \in \Xi}{\Delta; \Xi \vdash y{:}\tau}\ \textbf{CT-var} \qquad \frac{\Delta \vdash \mathcal{M} : \mathcal{A} \quad \Delta \vdash \Xi : \texttt{cctx}}{\Delta; \Xi \vdash [\mathcal{M}] : [\mathcal{A}]}\ \textbf{CT-box}$$

$$\frac{\Delta; \Xi, y{:}\tau_1 \vdash e : \tau_2}{\Delta; \Xi \vdash \texttt{fn}\ y{:}\tau_1 \Rightarrow e : \tau_1 \to \tau_2}\ \textbf{CT-fn} \qquad \frac{\Delta; \Xi \vdash e_1 : \tau_2 \to \tau_1 \quad \Delta; \Xi \vdash e_2 : \tau_2}{\Delta; \Xi \vdash e_1\ e_2 : \tau_1}\ \textbf{CT-app}$$

$$\frac{\Delta, X{:}\mathcal{A}; \Xi \vdash e : \tau}{\Delta; \Xi \vdash \texttt{mlam}\ X{:}\mathcal{A} \Rightarrow e : \Pi X{:}\mathcal{A}.\tau}\ \textbf{CT-mlam} \qquad \frac{\Delta; \Xi \vdash e : \Pi X{:}\mathcal{A}.\tau \quad \Delta \vdash \mathcal{M} : \mathcal{A}}{\Delta; \Xi \vdash e\ \mathcal{M} : [\![\mathcal{M}/X]\!]\tau}\ \textbf{CT-mapp}$$

$$\frac{\Delta; \Xi \vdash e_1 : [\mathcal{A}] \quad \Delta, X{:}\mathcal{A}; \Xi \vdash e_2 : \tau}{\Delta; \Xi \vdash \texttt{let}\ [X] = e_1\ \texttt{in}\ e_2 : \tau}\ \textbf{CT-let}$$

$$\frac{\begin{array}{lll}\tau = \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0 & \Delta \vdash \theta : \Delta_0 & \\ \cdot;\cdot \vdash \tau : \texttt{ctype} & \Delta \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}_0 & \Delta; \Xi \vdash^{[\![\theta]\!]\mathcal{A}_0} b_i : \tau \text{ (for all } b_i \in \vec{b})\end{array}}{\Delta; \Xi \vdash (\texttt{case}^{\tau}\ [\mathcal{M}]\ \texttt{of}\ \vec{b}) : [\![\theta, \mathcal{M}/X_0]\!]\tau_0}\ \textbf{CT-case}$$

$\boxed{\Delta; \Xi \vdash^{\mathcal{A}} b : \tau}$ – Branch b matches invariant τ

$$\frac{\begin{array}{lll} & \Delta' \vdash \theta_i' : \Delta_i & \\ \Delta_i \vdash \theta : \Delta_0 & \Delta' \vdash \theta' : \Delta & \\ \Delta_i \vdash \mathcal{M}_0 : [\![\theta]\!]\mathcal{A}_0 & \Delta' \vdash [\![\theta']\!]\mathcal{A} = [\![\theta_i']\!][\![\theta_i]\!]\mathcal{A}_0 & \Delta'; [\![\theta']\!]\Xi \vdash [\![\theta', \theta_i']\!]e_i : [\![\theta_i']\!][\![\theta_i]\!]\tau_0\end{array}}{\Delta; \Xi \vdash^{\mathcal{A}} (\Delta_i; [\mathcal{M}_i] \mapsto e_i) : \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0}\ \textbf{CT-branch}$$

B.3 Refinement-level judgments

$\boxed{(\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx})}$ – Computation context formation

$$\frac{(\vdash \Omega) \sqsubset (\vdash \Delta : \texttt{mctx})}{(\Omega \vdash \cdot) \sqsubset (\Delta \vdash \cdot : \texttt{cctx})}\ \textbf{CCR-nil}$$

$$\frac{(\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx}) \quad (\Omega; \Phi \vdash \zeta) \sqsubset (\Delta; \Xi \vdash \tau : \texttt{ctype})}{(\Omega \vdash \Phi, y{:}\zeta) \sqsubset (\Delta \vdash \Xi, y{:}\tau : \texttt{cctx})}\ \textbf{CCR-cons}$$

$\boxed{(\Omega; \Phi \vdash \zeta) \sqsubset (\Delta; \Xi \vdash \tau : \texttt{ctype})}$ – Sort/type formation

$$\frac{(\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx}) \quad (\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype})}{(\Omega; \Phi \vdash [\mathcal{S}]) \sqsubset (\Delta; \Xi \vdash [\mathcal{A}] : \texttt{ctype})}\ \textbf{CTR-meta}$$

$$\frac{(\Omega; \Phi \vdash \zeta_1) \sqsubset (\Delta; \Xi \vdash \tau_1 : \texttt{ctype}) \quad (\Omega; \Xi \vdash \zeta_2) \sqsubset (\Delta; \Xi \vdash \tau_2 : \texttt{ctype})}{(\Omega; \Phi \vdash \zeta_1 \to \zeta_2) \sqsubset (\Delta; \Xi \vdash \tau_1 \to \tau_2 : \texttt{ctype})}\ \textbf{CTR-arr}$$

$$\frac{(\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype}) \quad (\Omega, X{:}\mathcal{S}; \Phi \vdash \zeta) \sqsubset (\Delta, X{:}\mathcal{A}; \Xi \vdash \tau : \texttt{ctype})}{(\Omega; \Phi \vdash \Pi X{:}\mathcal{S}.\zeta) \sqsubset (\Delta; \Xi \vdash \Pi X{:}\mathcal{A}.\tau : \texttt{ctype})}\ \textbf{CTR-pi}$$

$\boxed{(\Omega; \Phi \vdash f : \zeta) \sqsubset (\Delta; \Xi \vdash e : \tau)}$ – Sorting and typing judgments for expressions

$$\frac{(\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx}) \quad (y{:}\zeta) \in \Phi \quad (y{:}\tau) \in \Xi}{(\Omega; \Phi \vdash y{:}\zeta) \sqsubset (\Delta; \Xi \vdash y{:}\tau)}\ \textbf{CTR-var}$$

$$\frac{(\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A}) \quad (\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx})}{(\Omega; \Phi \vdash [\mathcal{N}] : [\mathcal{S}]) \sqsubset (\Delta; \Xi \vdash [\mathcal{M}] : [\mathcal{A}])}\ \textbf{CTR-box}$$

$$\frac{(\Omega; \Phi, y{:}\zeta_1 \vdash f : \zeta_2) \sqsubset (\Delta; \Xi, y{:}\tau_1 \vdash e : \tau_2)}{(\Omega; \Phi \vdash \texttt{fn}\ y{:}\zeta_1 \Rightarrow f : \zeta_1 \to \zeta_2) \sqsubset (\Delta; \Xi \vdash \texttt{fn}\ y{:}\tau_1 \Rightarrow e : \tau_1 \to \tau_2)}\ \textbf{CTR-fn}$$

$$\frac{(\Omega; \Phi \vdash f_1 : \zeta_2 \to \zeta_1) \sqsubset (\Delta; \Xi \vdash e_1 : \tau_2 \to \tau_1) \quad (\Omega; \Phi \vdash f_2 : \zeta_2) \sqsubset (\Delta; \Xi \vdash e_2 : \tau_2)}{(\Omega; \Phi \vdash f_1\ f_2 : \zeta_1) \sqsubset (\Delta; \Xi \vdash e_1\ e_2 : \tau_1)}\ \textbf{CTR-app}$$

$$\frac{(\Omega, X{:}\mathcal{S}; \Phi \vdash f : \zeta) \sqsubset (\Delta, X{:}\mathcal{A}; \Xi \vdash e : \tau)}{(\Omega; \Phi \vdash \texttt{mlam}\ X{:}\mathcal{S} \Rightarrow f : \Pi X{:}\mathcal{S}.\zeta) \sqsubset (\Delta; \Xi \vdash \texttt{mlam}\ X{:}\mathcal{A} \Rightarrow e : \Pi X{:}\mathcal{A}.\tau)}\ \textbf{CTR-mlam}$$

$$\frac{(\Omega; \Phi \vdash f : \Pi X{:}\mathcal{S}.\zeta) \sqsubset (\Delta; \Xi \vdash e : \Pi X{:}\mathcal{A}.\tau) \quad (\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})}{(\Omega; \Phi \vdash f\ \mathcal{N} : [\![\mathcal{N}/X]\!]\zeta) \sqsubset (\Delta; \Xi \vdash e\ \mathcal{M} : [\![\mathcal{M}/X]\!]\tau)}\ \textbf{CTR-mapp}$$

$$\frac{(\Omega; \Phi \vdash f_1 : [\mathcal{S}]) \sqsubset (\Delta; \Xi \vdash e_1 : [\mathcal{A}]) \quad (\Omega, X{:}\mathcal{S}; \Phi \vdash f_2 : \zeta) \sqsubset (\Delta, X{:}\mathcal{A}; \Xi \vdash e_2 : \tau)}{(\Omega; \Phi \vdash \texttt{let}\ [X] = f_1\ \texttt{in}\ f_2 : \zeta) \sqsubset (\Delta; \Xi \vdash \texttt{let}\ [X] = e_1\ \texttt{in}\ e_2 : \tau)}\ \textbf{CTR-let}$$

$$\frac{\begin{array}{ll}\tau = \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0 & (\Omega \vdash \rho : \Omega_0) \sqsubset (\Delta \vdash \theta : \Delta_0) \\ \zeta = \Pi\Omega_0.\Pi X_0{:}\mathcal{S}_0.\zeta_0 & (\Omega \vdash \mathcal{N} : [\![\rho]\!]\mathcal{S}_0) \sqsubset (\Delta \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}_0) \\ (\cdot;\cdot \vdash \zeta) \sqsubset (\cdot;\cdot \vdash \tau : \texttt{ctype}) & (\Omega; \Phi \vdash^{[\![\rho]\!]\mathcal{S}_0} c_i : \zeta) \sqsubset (\Delta; \Xi \vdash^{[\![\theta]\!]\mathcal{A}_0} b_i : \tau)\ \text{(for all } c_i \in \vec{c})\end{array}}{(\Omega; \Phi \vdash (\texttt{case}^{\zeta}\ [\mathcal{N}]\ \texttt{of}\ \vec{c}) : [\![\rho, \mathcal{N}/X_0]\!]\zeta_0) \sqsubset (\Delta; \Xi \vdash (\texttt{case}^{\tau}\ [\mathcal{M}]\ \texttt{of}\ \vec{b}) : [\![\theta, \mathcal{M}/X_0]\!]\tau_0)}\ \textbf{CTR-case}$$

$\boxed{(\Omega; \Phi \vdash^{\mathcal{S}} c : \zeta) \sqsubset (\Delta; \Xi \vdash^{\mathcal{A}} b : \tau)}$ – Branch c matches invariant ζ

$$\frac{\begin{array}{l}(\Omega_i \vdash \rho : \Omega_0) \sqsubset (\Delta_i \vdash \theta : \Delta_0) \\ (\Omega_i \vdash \mathcal{N}_0 : [\![\rho]\!]\mathcal{S}_0) \sqsubset (\Delta_i \vdash \mathcal{M}_0 : [\![\theta]\!]\mathcal{A}_0) \\ (\Omega' \vdash \rho'_i : \Omega_i) \sqsubset (\Delta' \vdash \theta'_i : \Delta_i) \\ (\Omega' \vdash \rho' : \Omega) \sqsubset (\Delta' \vdash \theta' : \Delta) \\ (\Omega' \vdash [\![\rho']\!]\mathcal{S} = [\![\rho'_i]\!][\![\rho_i]\!]\mathcal{S}_0) \sqsubset (\Delta' \vdash [\![\theta']\!]\mathcal{A} = [\![\theta'_i]\!][\![\theta_i]\!]\mathcal{A}_0) \\ (\Omega'; [\![\rho']\!]\Phi \vdash [\![\rho', \rho'_i]\!]f_i : [\![\rho'_i]\!][\![\rho_i]\!]\zeta_0) \sqsubset (\Delta'; [\![\theta']\!]\Xi \vdash [\![\theta', \theta'_i]\!]e_i : [\![\theta'_i]\!][\![\theta_i]\!]\tau_0)\end{array}}{(\Omega; \Phi \vdash^{\mathcal{S}} (\Omega_i; [\mathcal{N}_i] \mapsto f_i) : \Pi\Omega_0.\Pi X_0{:}\mathcal{S}_0.\zeta_0) \sqsubset (\Delta; \Xi \vdash^{\mathcal{A}} (\Delta_i; [\mathcal{M}_i] \mapsto e_i) : \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0)}\ \textbf{CTR-branch}$$

$\boxed{(\Omega; \Phi \vdash \zeta_1 \leq \zeta_2) \sqsubset (\Delta; \Xi \vdash \tau)}$ – ζ_1 is a computation-level sub-sort of ζ_2

$$\frac{(\Omega \vdash \mathcal{S}_1 \leq \mathcal{S}_2) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype}) \quad (\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \texttt{cctx})}{(\Omega; \Phi \vdash [\mathcal{S}_1] \leq [\mathcal{S}_2]) \sqsubset (\Delta; \Xi \vdash [\mathcal{A}])}\ \textbf{SubC-meta}$$

$$\frac{(\Omega; \Phi \vdash \zeta_2 \leq \zeta_1) \sqsubset (\Delta; \Xi \vdash \tau) \quad (\Omega; \Phi \vdash \zeta'_1 \leq \zeta'_2) \sqsubset (\Delta; \Xi \vdash \tau')}{(\Omega; \Phi \vdash \zeta_1 \to \zeta'_1 \leq \zeta_2 \to \zeta'_2) \sqsubset (\Delta; \Xi \vdash \tau \to \tau')}\ \textbf{SubC-arr}$$

$$\frac{(\Omega \vdash \mathcal{S}_2 \leq \mathcal{S}_1) \sqsubset (\Delta \vdash \mathcal{A}) \quad (\Omega, u{:}\mathcal{S}_2; \Phi \vdash \zeta_1 \leq \zeta_2) \sqsubset (\Delta, u{:}\mathcal{A}; \Xi \vdash \tau)}{(\Omega; \Phi \vdash \Pi u{:}\mathcal{S}_1.\zeta_1 \leq \Pi u{:}\mathcal{S}_2.\zeta_2) \sqsubset (\Delta; \Xi \vdash \Pi u{:}\mathcal{A}.\tau)}\ \textbf{SubC-pi}$$

B.4 Signatures

Signatures	$\Sigma ::=$	$\cdot \mid \Sigma, \mathcal{D}$	
Declarations	$\mathcal{D} ::=$	$\mathtt{LF}\ \mathbf{a} : K = \mathbf{c}_1{:}A_1 \mid ... \mid \mathbf{c}_n{:}A_n$	LF type
		$\mathtt{LFR}\ \mathbf{s} \sqsubset \mathbf{a} : L = \mathbf{c}_1{:}S_1 \mid ... \mid \mathbf{c}_n{:}S_n$	LFR sort
		$\mathtt{LFR}\ \mathbf{s}_1 \leq \mathbf{s}_2 \sqsubset \mathbf{a} : L = \mathbf{c}_1{:}S_1 \mid ... \mid \mathbf{c}_n{:}S_n$	LFR subsort
		$\mathtt{schema}\ \mathbf{g} = \mathbf{w}_1{:}V_1 \mid ... \mid \mathbf{w}_n{:}V_n$	LF schema
		$\mathtt{schema}\ \mathbf{h} \sqsubset \mathbf{g} = \mathbf{w}_1{:}W_1 \mid ... \mid \mathbf{w}_n{:}W_n$	LFR schema
		$\mathtt{rec}\ \mathbf{f} : \zeta = f$	Recursive function

$\boxed{\vdash \Sigma : \mathbf{sig}}$ – Σ is a well-formed signature

$$\frac{}{\vdash \cdot : \mathbf{sig}} \qquad \frac{\vdash \Sigma : \mathbf{sig} \quad \vdash_\Sigma \mathcal{D} : \mathbf{decl}}{\vdash (\Sigma, \mathcal{D}) : \mathbf{sig}}$$

$\boxed{\vdash_\Sigma \mathcal{D} : \mathbf{decl}}$ – $\mathcal{D}$ is a well-formed declarature in signature Σ

$$\frac{\mathbf{a} \notin \Sigma \quad \mathbf{c}_1, ..., \mathbf{c}_n \notin \Sigma \quad \cdot;\cdot \vdash_\Sigma K : \mathtt{kind} \quad \cdot;\cdot \vdash_{\Sigma,\mathbf{a}:K} A_i \Leftarrow \mathtt{type}\ \text{(for all i)}}{\vdash_\Sigma (\mathtt{LF}\ \mathbf{a} : K = \mathbf{c}_1{:}A_1 \mid ... \mid \mathbf{c}_n{:}A_n) : \mathbf{decl}}$$

$$\frac{\begin{array}{ll}\mathbf{s} \notin \Sigma & (\cdot;\cdot \vdash_\Sigma L) \sqsubset (\cdot;\cdot \vdash_\Sigma K : \mathtt{kind}) \\ (\mathtt{LF}\ \mathbf{a} : K = \mathbf{c}_1{:}A_1 \mid ... \mid \mathbf{c}_n{:}A_n) \in \Sigma & (\cdot;\cdot \vdash_{\Sigma,\mathbf{s}\sqsubset\mathbf{a}:L} S_{i_j}) \sqsubset (\cdot;\cdot \vdash_{\Sigma,\mathbf{s}\sqsubset\mathbf{a}:L} A_{i_j})\ \text{(for all j)}\end{array}}{\vdash_\Sigma (\mathtt{LFR}\ \mathbf{s} \sqsubset \mathbf{a} : L = \mathbf{c}_{i_1}{:}S_{i_1} \mid ... \mid \mathbf{c}_{i_k}{:}S_{i_k}) : \mathbf{decl}}$$

$$\frac{\begin{array}{ll}\mathbf{s}_2 \notin \Sigma & \\ (\mathtt{LFR}\ \mathbf{s} \sqsubset \mathbf{a} : L) \in \Sigma & (\cdot;\cdot \vdash_\Sigma L) \sqsubset (\cdot;\cdot \vdash_\Sigma K : \mathtt{kind}) \\ (\mathtt{LF}\ \mathbf{a} : K = \mathbf{c}_1{:}A_1 \mid ... \mid \mathbf{c}_n{:}A_n) \in \Sigma & (\cdot;\cdot \vdash_{\Sigma,\mathbf{s}\sqsubset\mathbf{a}:L} S_{i_j}) \sqsubset (\cdot;\cdot \vdash_{\Sigma,\mathbf{s}\sqsubset\mathbf{a}:L} A_{i_j})\ \text{(for all j)}\end{array}}{\vdash_\Sigma (\mathtt{LFR}\ \mathbf{s}_1 \leq \mathbf{s}_2 \sqsubset \mathbf{a} : L = \mathbf{c}_{i_1}{:}S_{i_1} \mid ... \mid \mathbf{c}_{i_k}{:}S_{i_k}) : \mathbf{decl}}$$

$$\frac{\mathbf{g} \notin \Sigma \quad \mathbf{w}_1, ...\mathbf{w}_n \notin \Sigma \quad \cdot;\cdot \vdash_\sigma V_i : \texttt{world} \text{ (for all i)}}{\vdash_\Sigma (\texttt{schema } \mathbf{g} = \mathbf{w}_1{:}V_1 \mid ... \mid \mathbf{w}_n{:}V_n) : \mathbf{decl}}$$

$$\frac{\mathbf{h} \notin \Sigma \quad (\texttt{schema } \mathbf{g} = \mathbf{w}_1{:}V_1 \mid ... \mid \mathbf{w}_n{:}V_n) \in \Sigma \quad (\cdot;\cdot \vdash_\Sigma W_{i_j}) \sqsubset (\cdot;\cdot \vdash_\Sigma V_{i_j} : \texttt{world}) \text{ (for all j)}}{\vdash_\Sigma (\texttt{schema } \mathbf{h} \sqsubset \mathbf{g} = \mathbf{w}_{i_1}{:}W_{i_1} \mid ... \mid \mathbf{w}_{i_k}{:}W_{i_k}) : \mathbf{decl}}$$

$$\frac{\mathbf{f} \notin \Sigma \quad (\cdot;\cdot \vdash_{\Sigma,\texttt{rec } \mathbf{f}:\zeta} f : \zeta) \sqsubset (\cdot;\cdot \vdash_{\Sigma,\texttt{rec } \mathbf{f}:\zeta} e : \tau)}{\vdash_\Sigma (\texttt{rec } \mathbf{f} : \zeta = f) : \mathbf{decl}}$$