Contextual (modal) refinement types

Antoine Gaulin

October 11, 2023

Contents

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 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.

We present Beluga and its extension with refinement types in chapters 3 and 4, which discuss the data-level and computation-level, respectively. The core of the extension consists of replacing the LF layer of Beluga with a variation on the LFR system of Lovas and

Pfenning (2010); Lovas (2010). We then lift the refinement relations to the computation-level through straightforward congruence rules and show that the extension is conservative, meaning that every well-sorted program of our extension is well-typed in conventional Beluga. However, this result only applies if we consider Beluga as a general-purpose language rather than a proof environment. This is because a Beluga proof is a recursive function that terminates on every inputs and refinements allow specifying more precise domains. Thus, the types that we obtain from conservativity can extend the domain of a function, leading to undefined behaviour on certain inputs. In this sense, the extension permits interpreting some partial recursive functions as proofs. While termination is an important part of our work, we focus here on defining the refinements and leave termination checking for future work.

Once the system is defined, we argue for its usefulness through several case studies, detailed in chapter 5. The most stricking example is a benchmark challenge taken from Felty et al. (2015). These benchmarks use relations between contexts as a way to specify lemmas and theorem in a reusable fashion. Our notion of refinement schema coincides nicely with some of these context relations and this leads to significant improvements in the formal statements of theorem.

Finally, we survey the litterature in chapter **??** and conclude in chapter 6.

Chapter 2

Preliminaries

This chapter aims to introduce the background information needed to understand our work. We present first a detailed overview of the structure of Beluga. Then, we discuss previous work on refinement types. In the process, we introduce several of the key components of our system. We also discuss the modifications that were made to previous work so as to facilitate a smooth integration of refinements to Beluga.

2.1 Beluga

Beluga 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 section 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 clearness 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.

2.1.1 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, through a

Curry-Howard correspondence, 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 classfiers 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. Additionally, canonical forms go hand-in-hand with bi-directional type checking, which allows us to get rid of type annotations in function abstractions. This means that types never appear directly within terms, which is compatible with an extrinsic view of typing. Extrinsic typing has limited impact here since LF (and the entire Beluga) has type uniqueness, but it will play an important role when we introduce refinement types later on.

Syntax

We also differ from Harper et al. (1993) by defining terms 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. A spine $\vec{M}$ is simply a list of normal terms. 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 an atomic type constant $\mathbf{a}$ applied to a spine. $\mathbf{a}$ must have been declared and assigned a kind K (see the discussion of declarations below). The spine $\vec{M}$ must be made of terms that match the corresponding type in K and we require that all type families be fully applied. We will write only $\mathbf{a}$ for $\mathbf{a}$ `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 depends 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 type constants $\mathbf{a}$ 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 :

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

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

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 occurence of `tm` in the type of the constructor `lam`.

We note that negative occurences do not cause trouble in LF because of its weak function space. Specifically, the negative occurence of `tm` in `(tm → tm)` is simply represented as an LF variable of type `tm` that may occur within the positive occurence of `tm`. This is 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 :

```
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
```

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

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

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

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

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

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. Notices in particular how we represent the informal context extension $\Gamma, x{:}T$ using the negative occurences 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'.

Judgments

Now that we have defined and examplified the syntax of LF, we can discuss its judgments. We focus here on the typing judgments, athough, 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.

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

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

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

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

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

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

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

Figure 2.1: LF typing rules

type synthesis for neutral terms, denoted $\Gamma \vdash R \Rightarrow A$, and type checking for normal terms, denoted $\Gamma \vdash M \Leftarrow A$. 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 spine 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 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 $\lambda x{:}\mathbb{N}.\mathbf{S}\ \mathbb{N}$ 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`.

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. Therefore, to properly represent OLs, we need a tool to enforce the particular structure of their contexts and facilitate their manipulation. To achieve this, we will extend LF to Contextual LF.

2.1.2 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 equivalent to $\Gamma, x{:}A \vdash A'$, and similarly for objects. 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.

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. (2015) 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 Schurmann (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 premisses 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 Schurmann (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 Schurmann (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 Schurmann (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	$B ::=$	$\cdot \mid \Sigma x{:}A.B$
Schema elements	$E ::=$	$B \mid \Pi x{:}A.E$
Context schemas	$G ::=$	$\cdot \mid G + \mathbf{w}{:}E$
Contexts	$\Gamma ::=$	$\cdot \mid \psi \mid \Gamma, x{:}A \mid \Gamma, b{:}\mathbf{w} \cdot \vec{M}$
Heads	$H ::=$	$... \mid b.i$

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 $b{:}\mathbf{w} \cdot \vec{M}$. 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 ψ: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) Σx:$A.\cdot$. 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 need them for the meta-theory at

$\boxed{\Delta \vdash \Gamma : G}$ – 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 D.

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

Figure 2.2: Schema-checking rules

least.

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 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 occurence 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. In 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)` contains negative occurences 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 sophisiticated 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 occurences 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).

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

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

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

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

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

Figure 2.3: Explicit substitution calculus

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.

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 substutition 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 $\Gamma, x{:}A$ 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 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 $[\vec{M}/b](\mathbf{c}) = \mathbf{c}$.

Contextual objects

As mentionned above, a contextual type $\Gamma \vdash A$ consists of a type A together with a context Γ containing all the free variables occuring 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 variables, the problem comes from the fact 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:

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

So, a meta-variable u may only occur in an LF object when it is paired with a substitution. We have to this 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 subsitution 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.

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

$\boxed{\Delta \vdash \mathcal{M} : \mathcal{A}}$ – 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

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

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 subsitutiton well-formedness judgments. We omit their definitions here, but provide them in Appendix A for completeness.

2.1.3 Computations

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.

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 section. 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. First, we do not consider recursion since it complicates the syntax of patterns significantly. Specifically, valid recursive calls have to

be specified as part of every pattern (although they can be inferred, so users don't need to provide them explicitly). Without recursion, patterns are just (boxed) contextual objects. Second, our typing rules do not require coverage for pattern matching. This distinction cements our view of Beluga as a general-purpose language, which will be crucial in establishing conservativity of our extension later on. We will discuss this further when we define the refinement system for Beluga in Chapter 4. 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 difficulties of full dependent types. The expressions $\texttt{fn}\ y{:}\tau \Rightarrow e$ 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 invariants have the form $\Pi\Delta_1.\Pi X_0 : \mathcal{A}_0.\tau_0$. Intuitively, the context Δ in a branch $\Delta; [\mathcal{M}] \Rightarrow e$ consists of all and only the free modal variables occuring within $\mathcal{M}$. Intuitively, a branch $\Delta; [\mathcal{M}] \Rightarrow e$ satisfies the invariant $\Pi X_0{:}\mathcal{A}_0.\tau_0$ if $\mathcal{M}$ has type $\mathcal{A}_0$ and e has type $[\![\mathcal{M}/X_0]\!]\tau_0$, where $\mathcal{M}$, e, and their types can depend on Δ.

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:

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

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

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

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

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

$$\frac{\tau = \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0 \quad \Delta \vdash \theta : \Delta_0 \quad \Delta \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}_0 \quad \Delta; \Xi \vdash b : \tau \text{ (for all } b \in \vec{b})}{\Delta; \Xi \vdash (\texttt{case}^{\tau}\ [\mathcal{M}]\ \texttt{of}\ \vec{b}) : [\![\rho, \mathcal{M}/X_0]\!]\tau_0} \textbf{ CT-case}$$

$\boxed{\Delta; \Xi \vdash b : \tau}$

$$\frac{\Delta_0 \vdash \mathcal{M}_0 : \mathcal{A}_0 \quad \Delta, \Delta_0 \vdash \mathcal{A} \doteq \mathcal{A}_0/(\theta, \Delta') \quad \Delta'; [\![\theta]\!]\Xi \vdash [\![\theta]\!]e : [\![\theta]\!]\tau_0}{\Delta; \Xi \vdash (\Delta_0; [\mathcal{M}_0] \mapsto e) : \Pi\Delta_1.\Pi X_0{:}\mathcal{A}_0.\tau_0} \textbf{ CT-branch}$$

Figure 2.5: Beluga typing rules

```
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 should be seen as the endpoints of the evaluation process. We can encode what it means to be 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 don't 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 `val-no-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 omited. 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 `val-no-step` 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 `val-no-step` is perfectly safe to use even with an `stlc-ctx`. Felty et al. (2015) 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.

2.1.4 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 occurence 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 aforementionned context relations.

We omit further discussion of (co)inductive and stratified types from the current presentation as their inclusion significantly complexifies the language 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.

2.2 Refinement types

Now that we have introduced Beluga in details, we move on to discussing previous work on refinement types. Our work is inspired by 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 ditched 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. Our work starts by replacing the LF layer of Beluga by LFR, and extending the refinements to the rest of the language. Before we can do this, we must introduce LFR in greater details.

2.2.1 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.

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. 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.

Types, sorts, and the refinement relation

Now, let us formally look at the refinement types of LFR. As mentionned 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 $\mathbf{s}$ occuring 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 occurence 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 $S \sqsubset A$.

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 3.1 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 derivation that follow a call-by-value (CBV) strategy. This means that we evaluate a function application by first reducing the right-hand side to a λ-expression, then we reduce the left-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 mentionned 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. 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 of `s-beta` if the argument is already a value. 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`.

2.2.2 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.

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 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 (i.e. subtyping at the level of sorts) 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 famously 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 de-

fined 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 would not always preserve coverage.

We also allow the user 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 $\mathbf{s}_2$.

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.

Allowing reflexivity and transitivity only for atomic sorts ensures that any uses of these rules can be validated with a simple procedures. Reflexivity is just syntactic equality of a constant and transitivity can be established by traversing the lattice of sorts refining a given type.

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 subterm 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 normal ≥ neutral ⊏ 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 occurence 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.

2.2.3 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 origial LFR. We will now discuss these changes in details.

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 recall also from our first example that every type can be redefined as a sort by the user. In this sense, it is redundant to embed types into sorts. However, its presence is necessary to provide the subsorting relation $Q \leq P \sqsubset P$.

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, the addition of intersection sorts poses no fundamental challenge.

2.2.4 Other forms of refinements

Our work is inspired by the datasort approach of refinement types, but other forms of refinements have been developped in the past. We briefly discuss here two alternative forms of refinement types that are particularly interesting.

Another important (and perhaps more common) approach is index refinements, first introduced by Xi (1998); Xi and Pfenning (1999) for the core language of Standard ML. They design 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. 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.

Jones and Ramsay (2021) used refinement types to validate termination in the presence of non-exhaustive pattern matching. Given that Beluga views terminating functions as proofs, the ability to validate it in the presence of non-exhaustive pattern matching is particularly useful for our purpose. Jones and Ramsay (2021)'s 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 of that of non-empty lists.

Chapter 3

Contextual LFR

This chapter presents the specification language, or data-level, of our extension of Beluga with refinement types. Conventional Beluga uses Contextual LF and our extension replaces that by a contextual variant of the LFR system developed by Lovas and Pfenning (2010); Lovas (2010). Contextual objects, sorts and types allow us to keep track of variables and their types as we traverse open terms. This is crucial in obtaining pattern matching on LF objects since we may need to match against functions. With contextual objects, we can extend our context and analyze the body of a function while keeping track of the variables that may occur within.

3.1 Contextual LFR

Next, we extend LFR with contextual modal types and sorts.

The main difference between LFR and Contextual LFR is that objects and classifiers are bundled with a context in which they are meaningful. This facilitates traversing open objects

in recursive functions since the "free" variables are kept in the context, i.e. as part of the object itself. In addition, substitutions are used to move between contexts while preserving the meaningfulness of objects and types.

3.1.1 Contexts and schemas

Syntax of LFR contexts and schemas :

	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	$\Gamma ::= \cdot \mid \Gamma, x{:}A \mid \Gamma, b{:}E \cdot \vec{M}$	$\Psi ::= \cdot \mid \Psi, x{:}S \mid \Psi.b{:}F \cdot \vec{M}$
Context schemas	$G ::= \cdot \mid G + E$	$H ::= \cdot \mid H + F$

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. For instance, a typing assumption (informally denoted by $x : A$) is characterized by the schema element $\Pi A : \texttt{tp}.\Sigma x{:}\texttt{tm}.\Sigma t{:}\texttt{oft}\ x\ A.\cdot$, while a particular instance of this assumption would be $\Sigma x{:}\texttt{tm}.\Sigma t{:}\texttt{oft}\ x\ A.\cdot$.

LFR contexts can contain two kinds of variables. Ordinary variables, denoted by x, stand for an arbitrary LFR object of the specified type. Block variables, denoted by b, stand for tuples of assumptions satisfying the specification of a schema element. In conventional

$\boxed{\Omega; \Psi \vdash F \sqsubset E}$ – 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

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

$\boxed{\Omega \vdash \Psi \sqsubset \Gamma}$ – 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

Beluga, block variables are directly assigned with a block of declaration instead of a schema element applied to some objects. Here, we require that these objects be specified explicitly, so that they can be recovered when pattern matching on a context.

The refinement relations for contexts and schemas are simple. 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.

3.1.2 Objects

Now that we have discussed the classifiers of contextual LFR, let us look at the objects that they classify. There are two kinds of interesting objects, namely terms and substitutions. Terms are classified by sorts (and types), while substitutions are classified by contexts. Only normal forms are allowed at the data-level, and this is enforced with a canonical form presentation (Watkins et al., 2002). The syntax is as follows :

$$
\begin{array}{lrl}
\text{Neutral term} & R ::= & \mathbf{c} \mid x \mid b.k \mid R\ M \\
\text{Spine} & \vec{M} ::= & \cdot \mid M; \vec{M} \\
\text{Normal term} & M ::= & R \mid u[\sigma] \mid \lambda x.M \\
\text{Substitution} & \sigma ::= & \cdot \mid \sigma, M \mid \sigma, \vec{M}
\end{array}
$$

The separation of terms into neutral and normal ensures that no β-reduction can be done. The typing rules (see Figure 3.2) will also guarantee that all terms are η-long. Substitutions consist of normal terms and spines of normal terms. The domain of a substitution is not specified in the substitution itself. Rather, a substitution has the same structure as its domain context. In particular, substitution extension with a single term will match context extensions with a single assumptions, while substitution extension with a spine will match context extension with a block of assumptions.

3.2 Meta-types and meta-objects

The meta-layer unifies the different kinds of objects and classifiers of the data-level into unique constructs. This facilitates function abstraction at the computation-level since oth-

$\boxed{\Omega; \Psi \vdash M \Leftarrow S \sqsubset A}$ – Check normal term M against sort S.

$$\frac{\Omega; \Psi \vdash R \Rightarrow S \sqsubset A}{\Omega; \Psi \vdash R \Leftarrow S \sqsubset A} \qquad \frac{\Omega; \Psi, x{:}S_1 \vdash M \Leftarrow S_2 \sqsubset A_2}{\Omega; \Psi \vdash \lambda x.M \Leftarrow \Pi x{:}S_1.S_2 \sqsubset \Pi x{:}A_1.A_2}$$

$\boxed{\Omega; \Psi \vdash R \Rightarrow S \sqsubset A}$ – Synthesize sort S for neutral term R.

$$\frac{(b : E \cdot \vec{M}) \in \Psi \quad E \cdot \vec{M} \rightsquigarrow C \quad C \gg_k S \sqsubset A}{\Omega; \Psi \vdash b.k \Rightarrow S \sqsubset A}$$

$$\frac{\Omega; \Psi \vdash R \Rightarrow \Pi x{:}S_1.S_2 \sqsubset \Pi x{:}A_1.A_2 \quad \Omega; \Psi \vdash M \Leftarrow S_1 \sqsubset A_1}{\Omega; \Psi \vdash R\ M \Rightarrow [M/x]S_2 \sqsubset [M/x]A_2}$$

$\boxed{\Omega; \Psi \vdash b : D \gg_i^k S}$ – Extract sort S for k^{th} projection of block b (i^{th} step)

$$\frac{}{\Omega; \Psi \vdash b : \Sigma x{:}S.D \gg_k^k S} \qquad \frac{\Omega; \Psi \vdash b : [b.i/x]D \gg_{i+1}^k S}{\Omega; \Psi \vdash b : \Sigma x{:}S'.D \gg_i^k S}$$

$\boxed{\Omega; \Psi_1 \vdash \sigma : \Psi_2 \sqsubset \Gamma_2}$ – σ is a well-formed substitution from Ψ_2 to Ψ_1

$$\frac{\Omega \vdash \Psi_1 \sqsubset \Gamma_1}{\Omega; \Psi_1 \vdash \cdot : \cdot \sqsubset \cdot} \qquad \frac{\Omega; \Psi_1 \vdash \sigma : \Psi_2 \sqsubset \Gamma_2 \quad \Omega; \Psi_1 \vdash M \Leftarrow S \sqsubset A}{\Omega; \Psi_1 \vdash (\sigma, M) : (\Psi_2, x{:}S) \sqsubset (\Gamma_2, x{:}A)}$$

$$\frac{\Omega; \Psi_1 \vdash \sigma : \Psi_2 \sqsubset \Gamma_2 \quad (F[\vec{M}_2] \rightsquigarrow C) \sqsubset (E[\vec{M}_2] \rightsquigarrow D) \quad \Omega; \Psi \vdash \vec{M}_1 \Leftarrow C \sqsubset D}{\Omega; \Psi_1 \vdash (\sigma, \vec{M}_1) : (\Psi_2, b{:}F[\vec{M}_2]) \sqsubset (\Gamma_2, b{:}E[\vec{M}_2])}$$

Figure 3.2: Bi-directional typing rules

erwise each kind of object would need a special kind of function space. As before, our classifiers are separated into types and refinement types. Moreover, since meta-objects include contexts, we naturally obtain a refinement relation for objects as well.

The syntax is as follows :

$\boxed{(\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})}$ – 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 3.3: Meta-level typing and sorting

	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	$\rho ::= \cdot \mid \rho, \mathcal{M}$	$\theta ::= \cdot \mid \theta, \mathcal{N}$

Contexts with hats ($\hat{\Gamma}, \hat{\Psi}$) are called *erased* and contain no type or sort information, so they consist only of variables. Erased contexts are sufficient in this setting since the LF neutral objects and LF substitution do not refer to any type or sort information present in the context. Note that if $\Psi \sqsubset \Gamma$, then $\hat{\Psi} = \hat{\Gamma}$. This means that if $\mathcal{N} \sqsubset \mathcal{M}$ are not just contexts, then $\mathcal{N} = \mathcal{M}$. Accordingly, refinements of meta-objects (and later, computation-level objects) only provides information when contexts are used as objects.

The refinement relation for meta-types is obtained by lifting the corresponding refinement relations (developed previously). Similarly, the refinement relation for meta-objects is obtained by lifting the refinement relation on LF contexts. For example, the natural

refinement rule for $\Psi.Q$ is the following :

$$\frac{\Omega \vdash \Psi \sqsubset \Gamma \quad \Omega; \Psi \vdash Q \sqsubset P}{\Omega \vdash \Psi.Q \sqsubset \Gamma.P}$$

In a sense, this raises the refinement relation to the level of LFR judgments. It is helpful to pursue this idea further by formulating our rules for the meta- and computation-level as a refinement relation between judgments. The above rule would then become :

$$\frac{\vdash \Omega \sqsubset \Delta \quad \Omega \vdash \Psi \sqsubset \Gamma \quad \Omega; \Psi \vdash Q \sqsubset P}{(\Omega \vdash \Psi.Q) \sqsubset (\Delta \vdash \Gamma.P)}$$

This judgment $(\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A})$ can then serve as both a type well-formedness and a refinement judgment. We can similarly unify the sorting and typing judgments (see Figure 3.3), the context well-formenedness and refinement judgments, and so on. In all of these judgments, we can consider the type-level part to be an output.

3.3 Conservativity of refinements

Theorem 3.3.1 (Conservativity for data-level)

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

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

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

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

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

12. *If* $(\Omega \vdash G) \sqsubset (\Delta \vdash F : \texttt{schema})$, *then* $\Delta \vdash F : \texttt{schema}$.

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

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

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

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{c}\mathcal{D}' \\ (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx})\end{array}}{(\Omega; \Psi \vdash \texttt{sort}) \sqsubset (\Delta; \Gamma \vdash \texttt{type} : \texttt{kind})}\ \textbf{KR-type}$$

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**

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

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

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

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

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

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

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

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

$$\mathcal{D}'$$

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

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

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

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

$$\mathcal{D}'$$

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

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

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

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

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

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

Then $\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A.K > \texttt{type}$ 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

$$\mathcal{D}_1 \qquad\qquad \mathcal{D}_2$$

$$\textbf{Case } \mathcal{D} = \frac{(\mathbf{c} : S \sqsubset A) \in \Sigma \qquad (\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx})}{(\Omega; \Psi \vdash \mathbf{c} \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash \mathbf{c} \Rightarrow A)} \textbf{ TRS-const}$$

We have $(\mathbf{c} : A) \in \Sigma$ by inversion on 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**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{ccc}\mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ (x{:}S) \in \Psi & (x{:}A) \in \Gamma & \{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx})\}\end{array}}{(\Omega; \Psi \vdash x \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash x \Rightarrow A)}\ \textbf{TRS-x}$$

We have $(x{:}A) \in \Gamma$ by assumption $\mathcal{D}_2$.

$\Big\{$ We have $\Delta \vdash \Gamma : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}_3$ $\Big\}$

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

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{ll}\mathcal{D}_1 : \ (b{:}\mathbf{w}[\vec{M}]) \in \Psi & \mathcal{D}_3 : \ (\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \mathbf{v}[\vec{M}] > C) \\ \mathcal{D}_2 : \ (b{:}\mathbf{v}[\vec{M}]) \in \Gamma & \mathcal{D}_4 : \ (\Omega; \Psi \vdash b : D \gg_1^k S) \sqsubset (\Delta; \Gamma \vdash b : C \gg_1^k A)\end{array}}{(\Omega; \Psi \vdash b.k \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash b.k \Rightarrow A)}\ \textbf{TRS-b}$$

We have $(b{:}\mathbf{v}[\vec{M}]) \in \Gamma$ by assumption $\mathcal{D}_2$

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

We have $\Delta; \Gamma \vdash b : C \gg_1^k A$ by inductive hypothesis on $\mathcal{D}_4$

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 two cases to consider :

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{c}\mathcal{D}' \\ \{(\Omega; \Psi \vdash D) \sqsubset (\Delta; \Gamma \vdash C : \mathtt{block})\}\end{array}}{(\Omega; \Gamma \vdash D[\mathtt{nil}] > D) \sqsubset (\Delta; \Gamma \vdash C[\mathtt{nil}] > C)}\ \textbf{R-Inst-nil}$$

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

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

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

We have $(\mathbf{v} = V) \in \Sigma$ 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**

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

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

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 :

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

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

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{c}\mathcal{D}' \\ (\Omega; \Psi \vdash b : [b.i/x]D \gg_{i+1}^k S) \sqsubset (\Delta; \Gamma \vdash b : [b.i/x]C \gg_{i+1}^k)\end{array}}{(\Omega; \Psi \vdash b : \Sigma x{:}S'.D \gg_i^k S) \sqsubset (\Delta; \Gamma \vdash b : \Sigma x{:}A'.C \gg_i^k A)} \textbf{ R-Ext-cont}$$

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

Then $\Delta; \Gamma \vdash b : \Sigma x{:}A'.C \gg_i^k A$ 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 :

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

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**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1 :\ (u : \Psi'.S) \in \Omega \qquad \mathcal{D}_3 \\ \mathcal{D}_2 :\ (u : \Gamma'.A) \in \Delta \qquad (\Omega; \Psi \vdash \sigma : \Psi') \sqsubset (\Delta; \Gamma \vdash \sigma : \Gamma')\end{array}}{(\Omega; \Psi \vdash u[\sigma] : [\sigma]S) \sqsubset (\Delta; \Gamma \vdash u[\sigma] : [\sigma]A)}\ \textbf{TRS-mvar}$$

We have $(u : \Gamma'.A) \in \Delta$ by assumption $\mathcal{D}_2$

We have $\Delta; \Gamma \vdash \sigma : \Gamma'$ by inductive hypothesis on $\mathcal{D}_3$

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 :

$$\mathcal{D}'$$

$$\textbf{Case } \mathcal{D} = \frac{\{(\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \texttt{type})\}}{(\Omega; \Psi \vdash \texttt{nil} : S > S) \sqsubset (\Delta; \Gamma \vdash \texttt{nil} : A > A)}\ \textbf{TRC-spn-nil}$$

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

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

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1 :\ (\Omega; \Psi \vdash M \Leftarrow S_1') \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A_1') \\ \mathcal{D}_2 :\ (\Omega; \Psi \vdash \vec{M} : [M/x]S_2' > S) \sqsubset (\Delta; \Gamma \vdash \vec{M} : [M/x]A_2' > A)\end{array}}{(\Omega; \Psi \vdash (M; \vec{M}) : \Pi x{:}S_1'.S_2' > S) \sqsubset (\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A_1'.A_2' > A)}\ \textbf{TRC-spn-cons}$$

We have $\Delta; \Gamma \vdash M \Leftarrow A_1'$ by inductive hypothesis on $\mathcal{D}_2$

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

Then $\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A_1'.A_2' > A$ 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 :

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

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

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

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

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 :

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We have $(\mathbf{v} = V : \texttt{world}) \in \Sigma$ by signature formation rules on $\mathcal{D}_2$

We have $\mathbf{v} \notin G$ by assumption $\mathcal{D}_4$

Then $\Delta \vdash G + \mathbf{v} : \texttt{schema}$ by rule **S-ext**

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

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

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

$$\mathcal{D}'$$

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

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

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

$$\mathcal{D}_1 \qquad\qquad \mathcal{D}_2$$

$$\textbf{Case } \mathcal{D} = \frac{(\psi : H) \in \Omega \qquad (\psi : G) \in \Delta}{(\Omega \vdash \psi : H) \sqsubset (\Delta \vdash \psi : G)}\ \textbf{SRC-var}$$

We have $(\psi : G) \in \Delta$ by assumption $\mathcal{D}_2$

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

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{ll} \mathcal{D}_1 : & (\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G) \qquad \mathcal{D}_3 : \ \mathbf{w} \in H \\ \mathcal{D}_2 : & \left\{(\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \mathbf{v}[\vec{M}] > C)\right\} \qquad \mathcal{D}_4 : \ \mathbf{v} \in G \end{array}}{(\Omega \vdash (\Psi, b{:}\mathbf{w}[\vec{M}]) : H) \sqsubset (\Delta \vdash (\Gamma, b{:}\mathbf{v}[\vec{M}]) : G)}\ \textbf{SRC-ext}$$

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

$\left\{\text{We have } \Delta; \Gamma \vdash \mathbf{v}[\vec{M}] > C\right.$ by inductive hypothesis on $\mathcal{D}_2$

We have $\mathbf{v} \in G$ by assumption $\mathcal{D}_4$

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

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

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{c}\mathcal{D}' \\ (\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx})\end{array}}{(\Omega \vdash \cdot) \sqsubset (\Delta \vdash \cdot : \mathtt{ctx})}\ \textbf{CR-empty}$$

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

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

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{ccc}\mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ (\psi : H) \in \Omega & (\psi : G) \in \Delta & (\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx})\end{array}}{(\Omega \vdash \psi) \sqsubset (\Delta \vdash \psi : \mathtt{ctx})}\ \textbf{CR-var}$$

We have $(\psi : G) \in \Delta$ by assumption $\mathcal{D}_2$

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

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

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

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**

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

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

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

Then $\Delta \vdash (\Gamma, b{:}\mathbf{v}[\vec{M}]) : \mathtt{ctx}$ 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 :

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

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

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

Case $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 : \ (\psi : H) \in \Omega & \mathcal{D}_3 \\ \mathcal{D}_2 : \ (\psi : G) \in \Delta & \{(\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1 : \mathtt{ctx})\}\end{array}}{(\Omega; \Psi_1 \vdash \mathtt{id}_\psi : \psi) \sqsubset (\Delta; \Gamma_1 \vdash \mathtt{id}_\psi : \psi)}$ **SubstR-id**

We have $(\psi : G) \in \Delta$ by assumption $\mathcal{D}_2$

$\big\{$We have have $\Delta \vdash \Gamma_1 : \mathtt{ctx}$ by inductive hypothesis on $\mathcal{D}_3\big\}$

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

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

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

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

We have $\Delta; \Gamma_2 \vdash \mathbf{v}[\vec{M'}] > D$ 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 $\Delta; \Gamma_1 \vdash (\sigma, \vec{M}) : (\Gamma_2, b{:}\mathbf{v}[\vec{M'}])$ 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 :

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

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 :: (\Omega; \Psi \vdash M \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A) \\ \mathcal{D}_2 :: (\Omega; \Psi \vdash \vec{M} \Leftarrow [M/x]D) \sqsubset (\Delta; \Gamma \vdash \vec{M} \Leftarrow [M/x]C)\end{array}}{(\Omega; \Psi \vdash (M; \vec{M}) \Leftarrow \Sigma x{:}S.D) \sqsubset (\Delta; \Gamma \vdash (M; \vec{M}) \Leftarrow \Sigma x{:}A.C)}$ **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 $\mathcal{D} = \dfrac{\mathcal{D}_1 \quad \mathcal{D}_2}{(\vdash \Omega) \sqsubset (\vdash \Delta : \mathtt{mctx}) \qquad (\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \mathtt{mtype})}{(\vdash (\Omega, X{:}\mathcal{S}) \sqsubset (\vdash (\Delta, X{:}\mathcal{A}) : \mathtt{mctx})}$ **MCR-cons**

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

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

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

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

$$\textbf{Case } \mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ (\Omega; \Psi \vdash Q) \sqsubset (\Delta; \Gamma \vdash P \Leftarrow \texttt{type})\end{array}}{\big(\Omega \vdash (\Psi.Q)\big) \sqsubset \big(\Delta \vdash (\Gamma.P) : \texttt{mtype}\big)}\ \textbf{MTR-tp}$$

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

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

$$\textbf{Case } \mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ (\Omega \vdash H) \sqsubset (\Delta \vdash G : \texttt{schema})\end{array}}{(\Omega \vdash H) \sqsubset (\Delta \vdash G : \texttt{mtype})}\ \textbf{MTR-schema}$$

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

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

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

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

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**

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

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

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**

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

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**

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

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 :

$$\textbf{Case } \mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \{(\vdash \Omega_1) \sqsubset (\vdash \Delta_1 : \texttt{mctx})\}\end{array}}{(\Omega_1 \vdash \cdot : \cdot) \sqsubset (\Delta_1 \vdash \cdot : \cdot)}\ \textbf{MSubstR-nil}$$

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 :

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

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

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

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

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{c}\mathcal{D}' \\ (\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \texttt{type})\end{array}}{(\Omega; \Psi \vdash S \leq S) \sqsubset (\Delta; \Gamma \vdash A)}\ \textbf{Sub-refl}$$

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

$$\textbf{Case } \mathcal{D} = \frac{(\Omega; \Psi \vdash S_1 \leq S_2) \sqsubset (\Delta; \Gamma \vdash A) \quad (\Omega; \Psi \vdash S_2 \leq S_3) \sqsubset (\Delta; \Gamma \vdash A)}{(\Omega; \Psi \vdash S_1 \leq S_3) \sqsubset (\Delta; \Gamma \vdash A)}\ \textbf{Sub-trans}$$

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

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

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

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

:

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

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

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

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

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

:

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

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**

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

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

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 :

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

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

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

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

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

Then $\Delta \vdash G + \mathbf{w}{:}V : \texttt{schema}$ by rule **S-ext**

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 :

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

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**

Case $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ (\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1) & (\Omega \vdash \Psi_2) \sqsubset (\Delta \vdash \Gamma_2)\end{array}}{(\Omega \vdash \Psi_1.\Psi_2 \leq \Psi_1.\Psi_2) \sqsubset (\Delta \vdash \Gamma_1.\Gamma_2)}$ **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**

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ (\Omega \vdash H_1 \leq H_2) \sqsubset (\Delta \vdash G : \texttt{schema})\end{array}}{(\Omega \vdash H_1 \leq H_2) \sqsubset (\Delta \vdash G : \texttt{mtype})}$ **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**

■

Chapter 4

Computation-level

Beluga's computation-level is an ML-style functional programming language supporting pattern matching over contextual 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.

4.1 Computation-level refinements

In our extension, the computation-level is separated into a type layer and a refinement layer, just like the data-level. Since contextual objects can occur in computation-level expression, we maintain a refinement relation for expressions in addition to all other syntactic categories. Our presentation is inspired by the one of Pientka and Abel (2015), but differs in two important ways. First, we do not consider recursion since it complicates the syntax of patterns significantly. Specifically, valid recursive calls have to be specified as part of every pattern (although they can be inferred, so users don't need to provide them explicitly).

Without recursion, patterns are just (boxed) contextual objects. Second, our sorting (and typing) rules do not require coverage for pattern matching. The syntax of the computation-level is the following :

	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	$c ::= \Omega; [\mathcal{N}] \Rightarrow f$

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 $[\mathcal{S}]$. The elimination form for the modality is given by the `let` expressions : an expression $e_1 : [\mathcal{S}]$ 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 $\zeta_1 \to \zeta_2$ and the dependent function space $\Pi X{:}\mathcal{S}.\zeta$. So, dependencies are restricted to objects from the index domain, which provides strong reasoning power over the index domain without all the difficulties of full dependent types.

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 sort can be matched against by first unboxing it with a `let` expression and then matching on the new variable. The sort superscript ζ in `case` expression corresponds to the sort invariant that must be satisfied by all the branches in $\vec{c}$. We require that invariants

have the form $\Pi X_0 : \mathcal{S}_0.\zeta_0$. Intuitively, a branch $\Omega; [\mathcal{N}] \Rightarrow e$ satisfies the invariant $\Pi X_0{:}\mathcal{S}_0.\zeta_0$ if $\mathcal{N}$ has sort $\mathcal{S}_0$ and e has sort $[\![\mathcal{N}/X_0]\!]\zeta_0$, where $\mathcal{N}$, e, and their sorts can depend on Ω.

The judgments for the computation-level have a similar structure as those for contextual LFR. In particular, type-level and refinement-level judgments are performed simultaneously, with the type-level judgment seen as an output of the simultaneous judgment. Since the derivations produced on both sides of the refinement relation are almost exactly the same, we give the rules with only the refinement part. For instance, sorting and typing is expressed as $(\Omega; \Phi \vdash f : \zeta) \sqsubset (\Delta; \Xi \vdash e : \tau)$, but we define only $\Omega; \Phi \vdash f : \zeta$ for conciseness. We focus here on the rules related to pattern matching. The remaining rules are standard and can be found in the appendix. The rule for `case`-expressions is the following :

$$\frac{\zeta = \Pi\Omega_0.\Pi X_0{:}\mathcal{S}_0.\zeta_0 \quad \Omega \vdash \rho : \Omega_0 \quad \Omega \vdash \mathcal{N} : [\![\rho]\!]\mathcal{S}_0 \quad \Omega; \Phi \vdash c : \zeta \text{ (for all } c \in \vec{c})}{\Omega; \Phi \vdash (\texttt{case}^{\zeta}\ [\mathcal{N}]\ \texttt{of}\ \vec{c}) : [\![\rho, \mathcal{N}/X_0]\!]\zeta_0}$$

The important part of this rule is the last premise, which requires validating that every branch satisfies the given invariant. This is achieved with the judgment $\Omega; \Phi \vdash c : \zeta$ defined by the following rule :

$$\frac{\Omega_0 \vdash \mathcal{N}_0 : \mathcal{S}_0 \quad \Omega, \Omega_0 \vdash \mathcal{S} \doteq \mathcal{S}_0/(\rho, \Omega') \quad \Omega'; [\![\rho]\!]\Phi \vdash [\![\rho]\!]f : [\![\rho]\!]\zeta_0}{\Omega; \Phi \vdash (\Omega_0; [\mathcal{N}_0] \mapsto f) : \Pi\Omega_1.\Pi X_0{:}\mathcal{S}_0.\zeta_0}$$

Where the judgment $\Omega \vdash \mathcal{S} \doteq \mathcal{S}'/(\rho, \Omega')$ denotes (meta-type) unification. Intuitively, it means that $[\![\rho]\!]\mathcal{S}$ and $[\![\rho]\!]\mathcal{S}'$ are syntactically equal in Ω'. Note that since we allow term dependencies in sorts, the different branches of pattern matching may have different sorts as well.

4.2 Termination checking

In Beluga, a recursive program corresponds to a proof only if it terminates. Pientka and Abel (2015) established normalization for the fragment of Beluga where all pattern matching satisfies coverage (meaning that every possible case is represented by one of the patterns). Their presentation of Beluga is parametric in the index domain and assumes only that three conditions are satisfied: First, there is a unification algorithm for objects of the index domain; Second, there is a splitting algorithm for objects of the index domain; Last, there is a well-founded order for objects of the index domain. In addition, they explain how these three task can be achieved for contextual LF. In our extension, the index language is replaced with contextual LFR, whose objects follow the same structure as those of contextual LF. Consequently, the approach of Pientka and Abel (2015) can be adapted to our setting without significant difficulties.

This section briefly goes over each of the algorithms and how they need to be modified to fit our purpose. The key idea is the same as for the rest of the extension: we define refinements directly on the judgments defining the algorithms. In particular, the refinement rules closely mimic their type-level analogues, yielding straightforward conservativity results.

4.2.1 Unification

The main challenge of unification in the dependently-typed setting is to unify the terms on which the types that we want to unify depend.

4.2.2 Splitting

4.2.3 Order

4.3 Conservativity of extension

The conservativity results for the data-level (Theorem 3.3.1) carries over to the computation-level via straightforward inductions.

Theorem 4.3.1 (Conservativity for computation-level)

1. *If* $(\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \mathtt{cctx})$, *then* $\Delta \vdash \Xi : \mathtt{cctx}$.
2. *If* $(\Omega; \Phi \vdash \zeta) \sqsubset (\Delta; \Xi \vdash \tau : \mathtt{ctype})$, *then* $\Delta; \Xi \vdash \tau : \mathtt{ctype}$.
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 $\mathcal{D} :: (\Omega \vdash \Phi) \sqsubset (\Delta \vdash \Xi : \mathtt{cctx})$. There are two cases to consider :

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

We have $\vdash \Delta : \texttt{mctx}$ by Theorem 3.3.1 on $\mathcal{D}'$

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

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

We have $\Delta \vdash \Xi : \texttt{cctx}$ by Theorem 3.3.1 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 :

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

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

We have $\Delta \vdash \mathcal{A} : \texttt{mtype}$ by Theorem 3.3.1 on $\mathcal{D}_2$

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

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

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**

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

We have $\Delta \vdash \mathcal{A} : \texttt{mtype}$ by Theorem 3.3.1 on $\mathcal{D}_1$

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

Then $\Delta; \Xi \vdash \Pi X{:}\mathcal{A}.\tau : \texttt{ctype}$ 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

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

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

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

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

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

We have $\Delta \vdash \mathcal{M} : \mathcal{A}$ by Theorem 3.3.1 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**

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

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**

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

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**

$\mathcal{D}'$

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

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**

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

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 3.3.1 on $\mathcal{D}_2$

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

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

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**

$$\textbf{Case } \mathcal{D} = \frac{\begin{array}{cc}\zeta = \Pi X_0{:}\mathcal{S}_0.\zeta_0 & \mathcal{D}'' :\ (\Omega \vdash \mathcal{N} : \mathcal{S}_0) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A}_0) \\ \mathcal{D}' :\ \tau = \Pi X_0{:}\mathcal{A}_0.\tau_0 & \mathcal{D}_i :\ (\Omega; \Phi \vdash c_i : \zeta) \sqsubset (\Delta; \Xi \vdash b_i : \tau)\ \text{(for all } i)\end{array}}{(\Omega; \Phi \vdash (\texttt{case}^{\zeta}\ [\mathcal{N}]\ \texttt{of}\ \vec{c}) : [\![\mathcal{N}/X_0]\!]\zeta_0) \sqsubset (\Delta; \Xi \vdash (\texttt{case}^{\tau}\ [\mathcal{M}]\ \texttt{of}\ \vec{b}) : [\![\mathcal{M}/X_0]\!]\tau_0)}\ \textbf{CTR-case}$$

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

We have $\Delta \vdash \mathcal{M} : \mathcal{A}_0$ by Theorem 3.3.1 on $\mathcal{D}''$

We have $\Delta; \Xi \vdash b_i : \tau$ by inductive hypothesis on $\mathcal{D}_i$ (for each i)

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

$$\textbf{Case}\ \mathcal{D} = \frac{\begin{array}{ll}\mathcal{D}_1 & (\Omega_0 \vdash^{\ell} \mathcal{N}_0 : \mathcal{S}_0) \sqsubset (\Delta_0 \vdash^{\ell} \mathcal{M}_0 : \mathcal{A}_0) \\ \mathcal{D}_2 & (\Omega, \Omega_0; \Phi \vdash f : [\![\mathcal{N}_0/X_0]\!]\zeta_0) \sqsubset (\Delta, \Delta_0; \Xi \vdash e : [\![\mathcal{M}_0/X_0]\!]\tau_0)\end{array}}{(\Omega; \Phi \vdash (\Omega_0; [\mathcal{N}_0] \Rightarrow f) : \Pi X_0{:}\mathcal{S}_0.\zeta_0) \sqsubset (\Delta; \Xi \vdash (\Delta_0; [\mathcal{M}_0] \Rightarrow e) : \Pi X_0{:}\mathcal{A}_0.\tau_0)}\ \textbf{CTR-branch}$$

We have $\Delta_0 \vdash^{\ell} \mathcal{M}_0 : \mathcal{A}_0$ by lemma **??** on $\mathcal{D}_1$

We have $\Delta, \Delta_0; \Xi \vdash e : [\![\mathcal{M}_0/X_0]\!]\tau_0$ by inductive hypothesis on $\mathcal{D}_2$

Then $\Delta; \Xi \vdash (\Delta_0; [\mathcal{M}_0] \Rightarrow e) : \Pi X_0{:}\mathcal{A}_0.\tau_0$ 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 :

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

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

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

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

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

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

The aim of this chapter is to demonstrate the usefulness of refinement types. As mentionned previously, the main benefit of refinements is that they allow us to state theorems in a way that closely resemble their informal statements. In conventional settings, theorem statements can be very verbose due to type uniqueness. More precisely, the absence of subtyping means that expressing a subtype requires defining an additional type encoding it as well as a type encoding its embedding into the larger type. These embeddings must then appear within theorem statements whenever we want to use an object of the subtype as an object of the supertype.

5.1 Evaluation in untyped λ-calculus

Our first example consists of proving the determinacy of a particular evaluation strategy for λ-terms. We start by defining a declarative small-step semantic, meaning that it allows evaluating any subterm in any order. Then, we define a call-by-value evaluation algorithm

as a refinement of the declarative small-step semantic.

5.1.1 Definition of the language

Before we can talk about evaluation, we must define a language to evaluate. For simplicity, we will use an untyped λ-calculus with natural numbers.

Terms

It is defined as follows (left is Beluga code, right is informal specification) :

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

Terms $M ::=$		
x		Variables
$\mid 0$		Zero
$\mid \mathbf{S}\ M$		Successor
$\mid \lambda x.M$		Functions
$\mid M\ N$		Application

Note that variables are part of the informal syntax, but are not modelled by any constructor of `tm`. Instead, the negative occurence of `tm` in the constructor `lam` implies that variables of type `tm` may occur within other objects of type `tm`. This is in accordance with the HOAS principle stipulating that variables in the OL are represented as LF variables (or, more generally, as variables in the meta-language). Intuitively, we view the function space `(tm → tm)` as consisting of objects of type `tm` (possibly) containing a variable of type `tm`. As such, `(tm → tm)` is a weak function space (and so is every other LF function space). The advantage of this approach is that the size of `lam M` is larger than the size of `M`. This

allows us to recover an inductive view of types like `tm` that contain negative occurences in their definition.

Small-step semantics

Now that we have defined the terms of the language, we can move on to its (small-step) operational semantics. Conceptually, a step corresponds to a single action that could be performed by an evaluation algorithm.

```
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-eta : step M (lam (λx.app M x))
| s-app1 : step M1 M2 →
           step (app M1 N) (app M2 N)
| s-app2 : step N1 N2 →
           step (app M N1) (app M N2)
| s-lam : ({x : tm} step (M x) (N x))
          → step (lam M) (lam N);
```

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

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

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

$$\overline{M \longrightarrow \lambda x.M\ x}$$

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

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

$$\frac{x \vdash M \longrightarrow N}{(\lambda x.M) \longrightarrow (\lambda x.N)}$$

d

d

d

```
LF mstep : tm → tm → type =

| m-refl : mstep M M

| m-step : step M M' → mstep M' N

            → mstep M N;
```

$\boxed{M \longrightarrow^* N}$: M steps to N (multiple steps)

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

$$\frac{M \longrightarrow M' \qquad M' \longrightarrow^* N}{M \longrightarrow^* N}$$

5.1.2 Extracting an evaluation strategy

According to our small-step semantics, a given term may step in several different ways. As such, it does not define a concrete evaluation algorithm, but rather a declarative notion of evaluation. In practice, we need to precisely specify an order in which the rules are to be used. For instance, the CBV strategy is to fully evaluate the arguments passed to a function before applying β-reductions. We might also decide to omit some of the rules even though they are meaningful evaluation steps. For instance, the extensionality rule `s-lam` is helpful in the dependently-typed setting since we need term equality for type-cheking, but it is typically omited in the simply-typed setting.

Our goal now is to extract from `step` and `mstep` a deterministic evaluation strategy. First, we need to specify the stopping points of evaluation, that is values. Here, we consider all functions to be values, in addition to zero and the successors of values. We represent this using a refinement :

d

d

d

```
LFR val ⊏ tm : sort =
| zero : val
| succ : val → val
| lam : (val → tm) → val;
```

Value $V ::=$		
	x	Variables
	$\mid 0$	Zero
	$\mid \mathbf{S}\ V$	Successor
	$\mid \lambda x.M$	Functions

Next, we define a call-by-value semantics that evaluates function calls by first evaluating the function, then the argument, and finally applies β-reduction. We also propagate evaluation under successor, but avoid all uses of η-expansions or extensionality. The resulting strategy is then expressed as follows :

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

$\boxed{M \longrightarrow_{cbv} N}$: single CBV step

$$\frac{M \longrightarrow_{cbv} N}{\mathbf{S}\ M \longrightarrow_{cbv} \mathbf{S}\ N}$$

$$\overline{(\lambda x.M)\ V \longrightarrow_{cbv} [V/x]M}$$

$$\frac{M_1 \longrightarrow_{cbv} M_2}{(M_1\ N) \longrightarrow_{cbv} (M_2\ N)}$$

$$\frac{N_1 \longrightarrow_{cbv} N_2}{(V\ N_1) \longrightarrow_{cbv} (V\ N_2)}$$

d

d

d

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

$\boxed{M \longrightarrow^*_{cbv} N}$: CBV evaluation

$$\frac{}{V \longrightarrow^*_{cbv} V}$$

$$\frac{M \longrightarrow_{cbv} M' \qquad M' \longrightarrow^*_{cbv} N}{M \longrightarrow^*_{cbv} V}$$

5.1.3 Determinacy of evaluation

The main result of this section is that if evaluation succeeds, then it produces a unique value. The fact that evaluation produces values is already known from the kind of `eval`, namely `tm → val → sort`.

Values don't step

Lemma 1 *If V is a value, then $V \longrightarrow_{cbv} M$ is impossible.*

In order to state this formally, we need a notion of negation. Beluga is an intuitionistic system, so negation is defined as $\neg A \triangleq (A \to \bot)$, where $\bot$ is an empty type (representing falsehood). We can specify this by defining an atomic LF type with no constructors :

```
LF false : type =;

rec val-no-step : {V : [⊢ val]} [⊢ cbv V M] → [⊢ false] =
mlam V => fn s => case [⊢ V] of
| [⊢ zero] => impossible s
| [⊢ succ V] => let [⊢ s-succ S] = s in
                val-no-step [⊢ V] [⊢ S]
```

```
| [⊢ lam (λx. M)] => impossible s;
```

Determinacy of cbv

Lemma 2 *If* $M \longrightarrow_{cbv} N_1$ *and* $M \longrightarrow_{cbv} N_2$*, then* $N_1 = N_2$*.*

Where equality $N_1 = N_2$ is just reflexivity. We encode this equality as a Beluga type :

```
LF eq : tm → tm → type =
| eq-refl : eq M M;
```

And now we can state and prove the lemma as follows :

```
rec cbv-det : [⊢ cbv M N1] → [⊢ cbv M N2] → [⊢ eq N1 N2] =
fn s1, s2 => case s1 of
| [⊢ s-succ S1] => let [⊢ s-succ S2] = s2 in
                   cbv-det [⊢ S1] [⊢ S2]
| [⊢ s-beta] => (case s2 of
     | [⊢ s-beta] => [⊢ eq-refl]
     | [⊢ s-app1 S2] => impossible (val-no-step [⊢ _] [⊢ S2]
     | [⊢ s-app2 S2] => impossible (val-no-step [⊢ _] [⊢ S2])
| [⊢ s-app1 S1] => (case s2 of
     | [⊢ s-beta] => impossible (val-no-step [⊢ _] [⊢ S1]
     | [⊢ s-app1 S2] => cbv-det [⊢ S1] [⊢ S2]
     | [⊢ s-app2 S2] => impossible (val-no-step [⊢ _] [⊢ S1])
| [⊢ s-app2 S1] => (case s2 of
```

```
      | [⊢ s-beta] => impossible (val-no-step [⊢ _] [⊢ S1]
      | [⊢ s-app1 S2] => impossible (val-no-step [⊢ _] [⊢ S2]
      | [⊢ s-app2 S2] => cbv-det [⊢ S1] [⊢ S2]);
```

Determinacy of `eval`

Theorem 1 *If* $M \longrightarrow^*_{cbv} V_1$ *and* $M \longrightarrow^*_{cbv} V_2$, *then* $V_1 = V_2$.

```
rec eval-det : [⊢ eval M V1] → [⊢ eval M V2] → [⊢ eq V1 V2] =
fn s1, s2 => case s1 of
| [⊢ m-refl] => (case s2 of
      | [⊢ m-refl] => [⊢ eq-refl]
      | [⊢ m-step S2 _] => impossible (val-no-step [⊢ _] [⊢ S2])
| [⊢ m-step S1 E1] => (case s2 of
      | [⊢ m-refl] => impossible (val-no-step [⊢ _] [⊢ S1])
      | [⊢ m-step S2 E2] => let [⊢ eq-refl] = cbv-det [⊢ S1] [⊢ S2] in
                            eval-det [⊢ E1] [⊢ E2]);
```

5.1.4 Improvements on the conventional proof

The first difference between our refinement solution and a conventional solution lies in the encoding of values. Without refinements, we need to define `val` as a type. Since objects can only have one type, this requires introducing three new constructors, say `v-zero` `v-succ`, and `v-lam`, to be used in place of `zero`, `succ`, and `lam`, respectively. As a consequence, we

cannot directly view an object of type val as having type tm. To remedy this situation, we need an explicit embedding of values into terms, which can be defined as follows :

```
LF val-to-tm : val → tm → type =
| vt-zero : val-to-tm v-zero zero
| vt-succ : val-to-tm V M → val-to-tm (succ V) (succ M)
| vt-lam : val-to-tm (lam M) (lam M)
```

In the evaluation strategy, this embedding is then needed explicitly wherever we previously used an implicit quantification over values. For instance, the rule s-beta is given type `val-to-tm V N → cbv (app (lam M) N) (M N)`. In this example, we never need to consider a CBV evaluation as an arbitrary small-step evaluation, so there is no need to define an explicit embedding of cbv into step.

```
LF eval' : tm → val → type =
| m-refl : val-to-tm V M → eval' M V
| m-step : cbv M N → eval N V → eval M V;
```

The statements of theorems is similarly affected by the need to embed values into terms explicitly. This means that we abstract over more types, so the resulting proofs (functions) need to handle additional parameters. Due to the simplicity of our current example, the impact is small (see **??** for a more significant one). In fact, it is really only visible in the elaborated version of the statements. This is evident if we look at the first lemma (that values don't step) :

```
rec val-no-step' : [⊢ val-to-tm V M] → [⊢ cbv M N] → [⊢ false]
```

On the surface, this statement is very similar to its version using refinements : the only difference is that we have the embedding as the first parameter instead of the value itself. However, here we implicitly quantify over `V`, `M`, and `N`, while the refinement solution only implicitly quantifies over `N`. This means that the compiler has less work to do to fill in the gaps in the proof.

Although simple, this first example illustrates several of the key ideas behind refinements. In particular, we observed how sorts are defined by imposing further constraints on the constructors of atomic types, providing a natural subset interpretation. However, it concerned only closed objects, but Beluga excels at reasoning about and under binders.

5.1.5 Normal terms

The result about determinacy of evaluation can be extended to a semantics that evaluates under binders. To achieve this, we must specialize our notion of values to normal terms. This means that not all functions are considered end-points of evaluation, but only those that contain no redeces.

```
LFR neut ⊏ tm : sort =
| zero : neut
| succ : neut → neut
| app : neut → norm → neut

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

Neutral term $R ::= x \mid 0 \mid \mathbf{S}\ R \mid R\ M$

Normal term $N ::= R \mid \lambda x.N$

```
LFR n-step ⊏ step : tm → tm → sort =

| s-succ : n-step M M'

           → n-step (succ M) (succ M')

| s-beta : (N : norm)

           n-step (app (lam M) N) (M N)

| s-app1 : n-step M1 M2

           → n-step (app M1 M') (app M2 M')

| s-app2 : (N : val) n-step M1 M2

           → n-step (app N M1) (app N M2)

| s-lam : ({x:neut} n-step (M1 x) (M2 x))

          → n-step (lam M1) (lam M2);
```

$\boxed{M \longrightarrow_n M'}$: single step

$$\frac{M \longrightarrow_n M'}{\mathbf{S}\ M \longrightarrow_n \mathbf{S}\ M'}$$

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

$$\frac{M_1 \longrightarrow_n M_2}{(M_1\ M') \longrightarrow_n (M_2\ M')}$$

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

$$\frac{x \vdash M \longrightarrow_n M'}{(\lambda x.M) \longrightarrow_n (\lambda x.M')}$$

```
LFR n-eval ⊏ mstep : tm → norm → sort =

| m-refl : n-eval N N

| m-step : cbv M M' → n-eval M' N

           → n-eval M N;
```

$\boxed{M \longrightarrow^*_{cbv} N}$: CBV evaluation

$$\frac{}{V \longrightarrow^*_{cbv} V}$$

$$\frac{M \longrightarrow_{cbv} M' \qquad M' \longrightarrow^*_{cbv} N}{M \longrightarrow^*_{cbv} V}$$

5.2 Bi-directional type-checking

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

  LF tp : type =
  | nat : tp
  | eq : tp → tm → tm → tp
  | pi : tp → (tm → tp) → tp;

  LF kd : type =
  | typ : kd
  | kpi : tp → (tm → kd) → kd;
```

```
  LF oft : tm → tp → type

  | t-zero : oft zero nat

  | t-succ : oft M nat

             → oft (succ M) nat

  | t-refl : oft M A

             → oft (refl M) (eq A M M)

  | t-lam : ({x:tm} oft x A → oft (M x) (B x))

            → oft (lam M) (pi A B)

  | t-app : oft M (pi A B) → oft N A

            → oft (app M N) (B N);


  LF ofk : tp → kd → type

  | k-nat : ofk nat typ

  | k-eq : ofk A typ → oft M A

           → ofk (eq A M M) typ

  | k-pi : ofk A typ → ({x:tm} oft x A →

ofk (B x) typ)

           → ofk (pi A B) typ;
```

```
  LF wf-kd : kd → type =
  | wf-typ : wf-kd typ
  | wf-pi : ofk A typ → ({x:tm} oft x A →
wf-kd (L x))
            → wf-kd (kpi A L);

  schema xtG =
  | xtW : some [A : tp, W : ofk A typ] block (x:tm, t:oft x A);

  LFR neut ⊏ tm : sort =
  | app : neut → norm → neut


  and neut ≤ norm ⊏ tm : sort =
  | zero : norm
  | succ : norm → norm
  | refl : norm → norm
  | lam : (neut → norm) → norm;
```

```
LFR atom ⊏ tp : sort =

| nat : atom

| eq : canon → norm → norm → atom


and atom ≤ canon ⊏ tp : sort =

| pi : canon → (neut → canon) → canon

LFR synth ⊏ oft : neut → canon → sort =

| t-app : synth M (pi A B) → check N A

          → synth (app M N) (B N)


and synth ≤ check ⊏ oft : norm → canon → sort =

| t-zero : check zero nat

| t-succ : check M nat

           → check (succ M) nat

| t-refl : check M A

           → check (refl M) (eq A M M)

| t-lam : ({x:neut} synth x A → check (M x) (B x))

          → check (lam M) (pi A B)

schema neutG =

| xtW : some [A : tp, W : ofk A typ] block (x:neut, t:synth x A);
```

```
rec subderiv : (Γ : xtG) [Γ ⊢ oft M A] → [Γ ⊢ ofk A typ] =

fn d => case d of

| [Γ ⊢ #b.2] => let [Γ1, b:xtW A W, Γ2] = [Γ] in

                    [Γ1, b:xtW A W, Γ2 ⊢ W[..]]

| [Γ ⊢ t-zero] => [Γ ⊢ k-nat]

| [Γ ⊢ t-succ _] => [Γ ⊢ k-nat]

| [Γ ⊢ t-refl D] => let [Γ ⊢ K] = subderiv [Γ ⊢ D] in

                       [Γ ⊢ k-eq K D]

| [Γ ⊢ t-lam (λx.λt. D)] =>

|
```

5.3 Equality in polymorphic λ-calculus

Felty et al. (2015) proposed a series of benchmarks aimed at comparing the capabilities of different proof environment at reasoning about binders. Our next case study, proving the equivalence of algorithmic and declarative equalities in the polymorphic λ-calculus, is one of these benchmarks. The challenge here is that we must handle different kinds of binders simultaneously. This is due to the fact that the OL contains both type variables α and term variables x.

```
LF tp : type =
| arr : tp → tp → tp
| all : (tp → tp) → tp;
```

Type	$A, B ::=$	α	Variables
		$\mid A \to B$	Simple functions
		$\mid \forall \alpha.A$	Polymorphic functions

```
LF tm : type =                    Term  M, N ::=   x        Variables
| lam : (tm → tm) → tm                            | λx.M    Simple functions
| app : tm → tm → tm                              | M N     Simple application
| tlam : (tp → tm) → tm                           | Λα.M    Polymorphic functions
| tapp : tm → tp → tm;                            | M A     Polymorphic application

LF oft : tm → tp → type =
| of-lam : ({x:tm} oft x A → oft (M x) B)
            → oft (lam M) (arr A B)
| of-app : oft M (arr A B) → oft N A
            → oft (app M N) B
| of-tlam : ({α:tp} oft (M α) (A α))
             → oft (tlam M) (all A)
| of-tapp : oft M (all B) → {A:tp}
             → oft (tapp M A) (B A);

LF atp : tp → tp → type =
| at-arr : atp A1 A2 → atp B1 B2
            → atp (arr A1 B1) (arr A2 B2)
| at-all : ({α:tp} atp α  α→ atp (A α) (B α))
            → atp (tlam A) (tlam B);

LF deq : tm → tm → type =
```

```
| d-lam : ({x:tm} deq x x → deq (M x) (N x))
          → deq (lam M) (lam N)
| d-app : deq M1 M2 → deq N1 N2
          → deq (app M1 N1) (app M2 N2)
| d-tlam : ({α:tp} atp α α→ deq (M α) (N α))
           → deq (tlam M) (tlam N)
| d-tapp : deq M N → atp A B
           → deq (tapp M A) (tapp N B)
| d-ref : deq M M
| d-sym : deq M N
          → deq N M
| d-tra : deq M1 M2 → deq M2 M3
          → deq M1 M3;

LFR aeq ⊏ deq : tm → tm → sort =
| d-lam : ({x:tm} aeq x x → aeq (M x) (N x))
          → aeq (lam M) (lam N)
| d-app : aeq M1 M2 → aeq N1 N2
          → aeq (app M1 N1) (app M2 N2)
| d-tlam : ({α:tp} atp α α→ aeq (M α) (N α))
           → aeq (tlam M) (tlam N)
| d-tapp : aeq M N → atp A B
```

```
            → aeq (tapp M A) (tapp N B);

schema aeqG =

| α W : block (α:tp, at:atp α  α)

| aeqW : block (x:tm, ae:aeq x x);


schema atpG ⊏ aeqG =

| α W : block (α:tp, at:atp α  α);
```

Chapter 6

Conclusion

We have developed an extension of Beluga with datasort refinement types. While datasort refinements are mainly used to provide subtyping and intersection types to a language, our work focuses on the refinement relation itself. In particular, we discussed how refinements 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.

Through this extension, we studied the notion of refinement schemas. They allow extracting more precise information about contexts, much like refinements extract more precise properties (than types) 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.

6.1 Discussion

Handling contexts is at the source of many difficulties in the field of programming languages. **?** notes that –insert large percertage– of mechanizing meta-theory consisted of proving properties of contexts and substitutions. Contextual types offer an elegant solution to manage these issues with HOAS, but there remains challenges, such as those explored by Felty et al. (2015). We have shown that schema refinements provide an adequate solution to some of the problems surrounding context relation. However, there remains several interesting context relations not encompassed by our refinement relations.

The challenges of handling contexts were also manifest in the development of the current work. In Lovas and Pfenning (2010)'s LFR, the variables in contexts are both sorted and typed, that is contexts have the form $x_1 : S_1 \sqsubset A_1, ..., x_n : S_n \sqsubset A_n$. We wanted to keep a clear separation between sorts and types at all levels of Beluga, and the presence of contextual sorts and types required separating contexts into sorting and typing. However, contexts are also meta-objects in Beluga, so this separation induced a further separation and refinement relation on meta-objects and computation-level expressions.

The fact that there were now several refinement relations to consider simultaneously led to their formulation over judgments. This formulation was instrumental in obtaining a straightforward proof of conservativity (with previous attempts at formulating the extension requiring several lemma).

In Beluga, every computation-level expression has a unique type, but can have many sorts. **?** 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.

Conceptually, and extrinsic view of terms 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 G if all of its variables are constructed from the worlds of G. These worlds are referred to as constants applied to terms. In this case, schemas are analogous to atomic types in that they specify constructors with which a context may be created to fit that schema.

Felty et al. (2015) raise the important issue that contexts are not simply flat lists of variables, as is so often depicted on paper. They suggest using instead the slightly richer structure of lists of parameterized tuples, which we represent using schemas. This simple improvement already provides several benefits, so one might naturally consider a notion of schema that permits a more general structure. For instance, allowing contexts made out of other contexts would immediately yield a representation of those contexts separated into zones, such as our $\Delta; \Gamma$. If we further allow a notion of dependent schema (that is a family of schemas indexed by objects of a type), we can also represent an infinite hierarchy of levels,

as in layered modal type theory (**?**).

6.2 Future work

Our current definition of refinement schemas is limited by the fact that our meta-theory requires sorts to refine only one type. In particular, we can only establish that $H \sqsubset G$ when every element of G is refined by some element of H. However, if $H \sqsubset G$, then it would also make sense to conclude $H \sqsubset G+\mathbf{v}$ for any element $\mathbf{v}$. In that case, H would express the more precise property of not containing any assumption of the form $\mathbf{v}$. Likewise, the refinement relations on blocks and schema elements could be extended in interesting ways. At the moment, refinement is only possible between blocks of the same size and that are submitted to the same number of constraints. Allowing a large and heavily constrained block to refine a small and lightly constrained block would offer suitable encodings of several more context relations. Similarly, refinements of contexts can be generalized to express conditions in which strengthening can be performed safely. Our early experiments suggest that generalizing the refinement relation in this way would allow the representation of several more context relations. In fact, all the relations suggested by Felty et al. (2015) can be represented with minor changes to refinements. Consequently, our next goal is to provide a more flexible form of refinements and to modify our proof of conservativity so that it no longer relies on type uniqueness for refinements.

An important limitation of refinement type systems is that refinements cannot be further refined. Instead, they have to be related with a subsorting relation, which is similar to subtyping except that it relates sorts instead of types. As discussed previously, refine-

ment and subtyping are fundamentally different relations. This issue was also raised by **?**, who designed a system in which multiple layers of refinements are allowed. Although their work was for a simply-typed setting, we expect that a similar approach would work well in Beluga.

Our work currently extends the core of Beluga, but ignores several of the more advanced features. In particular, (co)inductive and stratified types (Jacob-Rao et al., 2018) play an important role in representing proofs by logical relations (Cave and Pientka, 2018). We expect that our current system can be extended in a straightoforward way, simply by defining sorting rules that mimic the typing rules.

Bibliography

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.

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. (2015). The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 1-A common infrastructure for benchmarks. *CoRR*, abs/1503.06095.

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.

Gécseg, F. and Steinby, M. (2015). Tree automata.

Harper, R., Honsell, F., and Plotkin, G. D. (1993). A framework for defining logics. *J. ACM*, 40(1):143–184.

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).

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. (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. (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.

Reynolds, J. C. (1997). *Design of the Programming Language Forsythe*, pages 173–233. Birkhäuser Boston, Boston, MA.

Rondon, P. M., Kawaguci, M., and Jhala, R. (2008). Liquid types. *SIGPLAN Not.*, 43(6):159–169.

Schurmann, C. E. (2000). *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, USA, USA. AAI9986626.

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.

$$\frac{(\mathbf{a}{:}K) \in \Sigma \qquad \Delta;\Gamma \vdash \vec{M} : K > \texttt{type}}{\Delta;\Gamma \vdash \mathbf{a}\ \vec{M} \Leftarrow \texttt{type}}\ \textbf{T-atom}$$

$$\frac{\Delta;\Gamma \vdash A_1 : \texttt{type} \qquad \Delta;\Gamma, x{:}A \vdash A_2 \Leftarrow \texttt{type}}{\Delta;\Gamma \vdash \Pi A_1.A_2 \Leftarrow \texttt{type}}\ \textbf{T-pi}$$

$\boxed{\Delta;\Gamma \vdash \vec{M} : K > \texttt{type}}$ – 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

$$\frac{(\mathbf{c}{:}A) \in \Sigma \quad \Delta \vdash \Gamma\ \texttt{ctx}}{\Delta;\Gamma \vdash \mathbf{c} \Rightarrow A}\ \textbf{TS-const} \qquad \frac{(x{:}A \in \Gamma) \quad \Delta \vdash \Gamma\ \texttt{ctx}}{\Delta;\Gamma \vdash x \Rightarrow A}\ \textbf{TS-x}$$

$$\frac{(b : \mathbf{v}[\vec{M}]) \in \Gamma \quad \Delta;\Gamma \vdash \mathbf{v}[\vec{M}] > C \quad \Delta;\Gamma \vdash b : C \gg_1^k A}{\Delta;\Gamma \vdash b.k \Rightarrow A}\ \textbf{TS-b}$$

$\boxed{\Delta; \Gamma \vdash V[\vec{M}] > C}$ – Instantiate block C for element V applied to $\vec{M}$

$$\frac{}{\Delta; \Gamma \vdash C[\mathtt{nil}] > C}\ \textbf{Inst-nil} \qquad \frac{(\mathbf{v} = V) \in \Sigma \quad \Delta; \Gamma \vdash V[\vec{M}] > C}{\Delta; \Gamma \vdash \mathbf{v}[\vec{M}] > C}\ \textbf{Inst-const}$$

$$\frac{\{\Delta; \Gamma \vdash M \Leftarrow A\} \quad \Delta; \Gamma \vdash ([M/x]V)[\vec{M}] > C}{\Delta; \Gamma \vdash \Pi x{:}A.V[M; \vec{M}] > C}\ \textbf{Inst-pi}$$

$\boxed{\Delta; \Gamma \vdash b : D \gg_i^k A}$ – 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

$$\frac{\Delta; \Gamma \vdash H \Rightarrow A' \quad \Delta; \Gamma \vdash \vec{M} : A' > A}{\Delta; \Gamma \vdash H\ \vec{M} \Rightarrow A}\ \textbf{TS-app}$$

$$\frac{(u : \Gamma'.A) \in \Delta \quad \Delta; \Gamma \vdash \sigma : \Gamma'}{\Delta; \Gamma \vdash u[\sigma] : [\sigma]A}\ \textbf{TS-mvar}$$

$\boxed{\Delta; \Gamma \vdash \vec{M} : A' > A}$ – Apply type A' to spine $\vec{M}$ to obtain type A

$$\frac{\{\Delta; \Gamma \vdash A \Leftarrow \mathtt{type}\}}{\Delta; \Gamma \vdash \mathtt{nil} : A > A}\ \textbf{TC-spn-nil}$$

$$\frac{\Delta; \Gamma \vdash M \Leftarrow A_1' \quad \Delta; \Gamma \vdash \vec{M} : [M/x]A_2' > A}{\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A_1'.A_2' > A}\ \textbf{TC-spn-cons}$$

$\boxed{\Delta; \Gamma \vdash M \Leftarrow A}$ – Check normal term M against type A.

$$\frac{\Delta; \Gamma \vdash R \Rightarrow A}{\Delta; \Gamma \vdash R \Leftarrow A}\ \textbf{TC-conv} \qquad \frac{\Delta; \Gamma, x{:}A \vdash M \Leftarrow A'}{\Delta; \Gamma \vdash \lambda x.M \Leftarrow \Pi x{:}A.A'}\ \textbf{TC-lam}$$

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

$$\frac{\vdash \Delta : \texttt{mctx}}{\Delta \vdash \cdot : \texttt{schema}}\ \textbf{S-empty} \qquad \frac{\Delta \vdash G : \texttt{schema} \quad (\mathbf{v} = V : \texttt{world}) \in \Sigma \quad \mathbf{v} \notin G}{\Delta \vdash G + \mathbf{v} : \texttt{schema}}\ \textbf{S-ext}$$

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

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

$$\frac{\Delta \vdash \Gamma : G \quad \mathbf{v} \in G \quad \left\{\Delta;\Gamma \vdash \mathbf{v}[\vec{M}] > D\right\}}{\Delta \vdash (\Gamma, b{:}\mathbf{v}[\vec{M}]) : G}\ \textbf{SC-ext}$$

A.2.4 Contexts

$\boxed{\Delta \vdash \Gamma : \texttt{ctx}}$ – LF context Γ is well-formed in meta-context Δ

$$\frac{\vdash \Delta : \texttt{mctx}}{\Delta \vdash \cdot : \texttt{ctx}}\ \textbf{C-empty} \qquad \frac{(\psi : G) \in \Delta \quad \vdash \Delta : \texttt{mctx}}{\Delta \vdash \psi : \texttt{ctx}}\ \textbf{C-var}$$

$$\frac{\Delta \vdash \Gamma : \texttt{ctx} \quad \Delta;\Gamma \vdash A \Leftarrow \texttt{type}}{\Delta \vdash (\Gamma, x{:}A) : \texttt{ctx}}\ \textbf{C-cons-x} \qquad \frac{\Delta \vdash \Gamma : \texttt{ctx} \quad \Delta;\Gamma \vdash \mathbf{v}[\vec{M}] > C}{\Delta \vdash (\Gamma, b{:}\mathbf{v}[\vec{M}]) : \texttt{ctx}}\ \textbf{C-cons-b}$$

$\boxed{\Delta; \Gamma_1 \vdash \sigma : \Gamma_2}$ – σ is a well-formed subsitution from Γ_2 to Γ_1

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

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

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

$\boxed{\Delta; \Gamma \vdash \vec{M} \Leftarrow D}$ – n-ary tuple $\vec{M}$ checks against block of declarations D

$$\frac{\{\Delta \vdash \Gamma : \mathtt{ctx}\}}{\Delta; \Gamma \vdash \mathtt{nil} \Leftarrow \cdot}\ \textbf{Chk-spn-nil} \qquad \frac{\Delta; \Gamma \vdash M \Leftarrow A \quad \Delta; \Gamma \vdash \vec{M} \Leftarrow [M/x]D}{\Delta; \Gamma \vdash (M; \vec{M}) \Leftarrow \Sigma x{:}A.D}\ \textbf{Chk-spn-sigma}$$

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

$$\frac{\{\vdash \Delta_1 : \texttt{mctx}\}}{\Delta_1 \vdash \cdot : \cdot} \textbf{ MSubst-nil} \qquad \frac{\Delta_1 \vdash \theta : \Delta_2 \quad \Delta_1 \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}}{\Delta_1 \vdash (\theta, \mathcal{M}) : (\Delta_2, X{:}\mathcal{A})} \textbf{ MSubst-cons}$$

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

$$\frac{(\mathbf{c} :: S \sqsubset A) \in \Sigma \quad \{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx})\}}{(\Omega; \Psi \vdash \mathbf{c} \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash \mathbf{c} \Rightarrow A)}\ \textbf{TRS-const}$$

$$\frac{(x{:}S) \in \Psi \quad (x{:}A) \in \Gamma \quad \{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx})\}}{(\Omega; \Psi \vdash x \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash x \Rightarrow A)}\ \textbf{TRS-x}$$

$$\frac{\begin{array}{ll}(b{:}\mathbf{w}[\vec{M}]) \in \Psi & (\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \mathbf{v}[\vec{M}] > C) \\ (b{:}\mathbf{v}[\vec{M}]) \in \Gamma & (\Omega; \Psi \vdash b : D \gg_1^k S) \sqsubset (\Delta; \Gamma \vdash b : C \gg_1^k A)\end{array}}{(\Omega; \Psi \vdash b.k \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash b.k \Rightarrow A)}\ \textbf{TRS-b}$$

$\boxed{(\Omega; \Psi \vdash W[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash V[\vec{M}] > C)}$ – Instantiate block D for element $W\,\vec{M}$

$$\frac{\{(\Omega; \Psi \vdash D) \sqsubset (\Delta; \Gamma \vdash C : \texttt{block})\}}{(\Omega; \Psi \vdash D[\texttt{nil}] > D) \sqsubset (\Delta; \Gamma \vdash C[\texttt{nil}] > C)}\ \textbf{R-Inst-nil}$$

$$\frac{((\mathbf{w} = W) \sqsubset (\mathbf{v} = V)) \in \Sigma \quad (\Omega; \Psi \vdash W[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash V[\vec{M}] > C)}{(\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \mathbf{v}[\vec{M}] > C)}\ \textbf{R-Inst-const}$$

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

$\boxed{(\Omega; \Psi \vdash b : D \gg_i^k S) \sqsubset (\Delta; \Gamma \vdash b : C \gg_i^k A)}$ – 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

$$\frac{\begin{array}{l}(\Omega; \Psi \vdash H \Rightarrow S') \sqsubset (\Delta; \Gamma \vdash H \Rightarrow A') \\ (\Omega; \Psi \vdash \vec{M} : S' > S) \sqsubset (\Delta; \Gamma \vdash \vec{M} : A' > A)\end{array}}{(\Omega; \Psi \vdash H\ \vec{M} \Rightarrow S) \sqsubset (\Delta; \Gamma \vdash H\ \vec{M} \Rightarrow A)}\ \textbf{TRS-app}$$

$$\frac{(u : \Psi'.S) \in \Omega \quad (u : \Gamma'.A) \in \Delta \quad (\Omega; \Psi \vdash \sigma : \Psi') \sqsubset (\Delta; \Gamma \vdash \sigma : \Gamma')}{(\Omega; \Psi \vdash u[\sigma] : [\sigma]S) \sqsubset (\Delta; \Gamma \vdash u[\sigma] : [\sigma]A)}\ \textbf{TRS-mvar}$$

$\boxed{(\Omega; \Psi \vdash \vec{M} : S' > S) \sqsubset (\Delta; \Gamma \vdash \vec{M} : A' > A)}$ – Apply sort S' to $\vec{M}$ to obtain sort S

$$\frac{\{(\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \texttt{type})\}}{(\Omega; \Psi \vdash \texttt{nil} : S > S) \sqsubset (\Delta; \Gamma \vdash \texttt{nil} : A > A)}\ \textbf{TRC-spn-nil}$$

$$\frac{\begin{array}{c}(\Omega; \Psi \vdash M \Leftarrow S_1') \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A_1') \\ (\Omega; \Psi \vdash \vec{M} : [M/x]S_2' > S) \sqsubset (\Delta; \Gamma \vdash \vec{M} : [M/x]A_2' > A)\end{array}}{(\Omega; \Psi \vdash (M; \vec{M}) : \Pi x{:}S_1'.S_2' > S) \sqsubset (\Delta; \Gamma \vdash (M; \vec{M}) : \Pi x{:}A_1'.A_2' > A)}\ \textbf{TRC-spn-cons}$$

$\boxed{(\Omega; \Psi \vdash M \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash M \Leftarrow A)}$ – Check normal term M against sort S.

$$\frac{(\Omega; \Psi \vdash R \Rightarrow S') \sqsubset (\Delta; \Gamma \vdash R \Rightarrow A) \quad (\Omega; \Psi \vdash S' \leq S) \sqsubset (\Delta; \Gamma \vdash A)}{(\Omega; \Psi \vdash R \Leftarrow S) \sqsubset (\Delta; \Gamma \vdash R \Leftarrow A)}\ \textbf{TRC-conv}$$

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

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

$$\frac{(\vdash \Omega) \sqsubset (\vdash \Delta : \texttt{mctx})}{(\Omega \vdash \cdot) \sqsubset (\Delta \vdash \cdot : \texttt{schema})}\ \textbf{SR-empty}$$

$$\frac{\begin{array}{ll}\mathbf{w} \notin H & (\Omega \vdash H) \sqsubset (\Delta \vdash G : \texttt{schema}) \\ \mathbf{v} \notin G & ((\mathbf{w} = W) \sqsubset (\mathbf{v} = V : \texttt{world})) \in \Sigma\end{array}}{(\Omega \vdash H + \mathbf{w}) \sqsubset (\Delta \vdash G + \mathbf{v} : \texttt{schema})}\ \textbf{SR-ext}$$

$$\frac{\begin{array}{ll}\mathbf{w} \notin H & (\Omega \vdash H) \sqsubset (\Delta \vdash G : \texttt{schema}) \\ \mathbf{v} \in G & ((\mathbf{w} = W) \sqsubset (\mathbf{v} = V : \texttt{world})) \in \Sigma\end{array}}{(\Omega \vdash H + \mathbf{w}) \sqsubset (\Delta \vdash G + \mathbf{v} : \texttt{schema})}\ \textbf{SR-ext-dup}$$

$\boxed{(\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G)}$ – Context Ψ has schema H

$$\frac{(\Omega \vdash H) \sqsubset (\Delta \vdash G : \texttt{schema})}{(\Omega \vdash \cdot : H) \sqsubset (\Delta \vdash \cdot : G)}\ \textbf{SRC-empty} \qquad \frac{(\psi : H) \in \Omega \quad (\psi : G) \in \Delta}{(\Omega \vdash \psi : H) \sqsubset (\Delta \vdash \psi : G)}\ \textbf{SRC-var}$$

$$\frac{\begin{array}{ll}\mathbf{w} \in H & (\Omega \vdash \Psi : H) \sqsubset (\Delta \vdash \Gamma : G) \\ \mathbf{v} \in G & \left\{(\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Delta; \Gamma \vdash \mathbf{v}[\vec{M}] > C)\right\}\end{array}}{(\Omega \vdash (\Psi, b{:}\mathbf{w}[\vec{M}]) : H) \sqsubset (\Delta \vdash (\Gamma, b{:}\mathbf{v}[\vec{M}]) : G)}\ \textbf{SRC-ext}$$

A.3.4 Contexts

$\boxed{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx})}$ – LFR context Ψ refines Γ

$$\frac{(\vdash \Omega) \sqsubset (\vdash \Delta : \texttt{mctx})}{(\Omega \vdash \cdot) \sqsubset (\Delta \vdash \cdot : \texttt{ctx})}\ \textbf{CR-empty}$$

$$\frac{(\psi : H) \in \Omega \quad (\psi : G) \in \Delta \quad (\vdash \Omega) \sqsubset (\vdash \Delta : \texttt{mctx})}{(\Omega \vdash \psi) \sqsubset (\Delta \vdash \psi : \texttt{ctx})}\ \textbf{CR-var}$$

$$\frac{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \texttt{ctx}) \quad (\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A \Leftarrow \texttt{type})}{(\Omega \vdash \Psi, x{:}S) \sqsubset (\Delta \vdash (\Gamma, x{:}A) : \texttt{ctx})}\ \textbf{CR-cons-x}$$

$$\frac{(\Omega \vdash \Psi) \sqsubset (\Delta \vdash \Gamma : \mathtt{ctx}) \quad (\Omega; \Psi \vdash \mathbf{w}[\vec{M}] > D) \sqsubset (\Omega; \Psi \vdash \mathrm{v}[\vec{M}] > C)}{(\Omega \vdash \Psi, b{:}\mathbf{w}[\vec{M}]) \sqsubset (\Delta \vdash (\Gamma, b{:}\mathbf{v}[\vec{M}]) : \mathtt{ctx})}\ \textbf{CR-cons-b}$$

$\boxed{(\Omega; \Psi_1 \vdash \sigma : \Psi_2) \sqsubset (\Delta; \Gamma_1 \vdash \sigma : \Gamma_2)}$ – σ is a well-formed subsitution from Ψ_2 to Ψ_1

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

$$\frac{(\psi : H) \in \Omega \quad (\psi : G) \in \Delta \quad \{(\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1 : \mathtt{ctx})\}}{(\Omega; \Psi_1 \vdash \mathtt{id}_\psi : \psi) \sqsubset (\Delta; \Gamma_1 \vdash \mathtt{id}_\psi : \psi)}\ \textbf{SubstR-id}$$

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

$$\frac{\begin{array}{l}(\Omega; \Psi_1 \vdash \sigma : \Psi_2) \sqsubset (\Delta; \Gamma_1 \vdash \sigma : \Gamma_2) \\ (\Omega; \Psi_2 \vdash \mathbf{w}[\vec{M}'] > C) \sqsubset (\Delta; \Gamma_2 \vdash \mathbf{v}[\vec{M}'] > D) \\ (\Omega; \Psi_1 \vdash \vec{M} \Leftarrow [\sigma]C) \sqsubset (\Delta; \Gamma_1 \vdash \vec{M} \Leftarrow [\sigma]D)\end{array}}{\big(\Omega; \Psi_1 \vdash (\sigma, \vec{M}) : (\Psi_2, b{:}\mathbf{w}[\vec{M}'])\big) \sqsubset \big(\Delta; \Gamma_1 \vdash (\sigma, \vec{M}) : (\Gamma_2, b{:}\mathbf{v}[\vec{M}'])\big)}\ \textbf{SubstR-spn}$$

$\boxed{(\Omega; \Psi \vdash \vec{M} \Leftarrow C) \sqsubset (\Delta; \Gamma \vdash \vec{M} \Leftarrow D)}$ – 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

$$\frac{}{(\vdash \cdot) \sqsubset (\vdash \cdot : \mathtt{mctx})}\ \textbf{MCR-nil}$$

$$\frac{(\vdash \Omega) \sqsubset (\vdash \Delta : \texttt{mctx}) \quad (\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype})}{(\vdash (\Omega, X{:}\mathcal{S}) \sqsubset (\vdash (\Delta, X{:}\mathcal{A}) : \texttt{mctx})} \textbf{ MCR-cons}$$

$\boxed{(\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype})}$ – $\mathcal{S}$ refines $\mathcal{A}$ / is a well-formed meta-sort

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

$$\frac{(\Omega \vdash H) \sqsubset (\Delta \vdash G : \texttt{schema})}{(\Omega \vdash H) \sqsubset (\Delta \vdash G : \texttt{mtype})} \textbf{ MTR-schema}$$

$$\frac{(\Omega \vdash \Psi_1) \sqsubset (\Delta \vdash \Gamma_1 : \texttt{ctx}) \quad (\Omega \vdash \Psi_2) \sqsubset (\Delta \vdash \Gamma_2 : \texttt{ctx})}{\big(\Omega \vdash (\Psi_1.\Psi_2)\big) \sqsubset \big(\Delta \vdash (\Gamma_1.\Gamma_2) : \texttt{mtype}\big)} \textbf{ MTR-subst}$$

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

$$\frac{(\text{LFR } \mathbf{s}_1 \leq \mathbf{s}_2 \sqsubset \mathbf{a} : L) \in \Sigma \quad (\Omega; \Psi \vdash \vec{M} : L > \texttt{sort}) \sqsubset (\Delta; \Gamma \vdash \vec{M} : K > \texttt{type})}{(\Omega; \Psi \vdash \mathbf{s}_1 \leq \mathbf{s}_2) \sqsubset (\Delta; \Gamma \vdash \mathbf{a})}\ \textbf{Sub-atom}$$

$$\frac{(\Omega; \Psi \vdash S) \sqsubset (\Delta; \Gamma \vdash A)}{(\Omega; \Psi \vdash S \leq S) \sqsubset (\Delta; \Gamma \vdash A)}\ \textbf{Sub-refl}$$

$$\frac{(\Omega; \Psi \vdash S_1 \leq S_2) \sqsubset (\Delta; \Gamma \vdash A) \quad (\Omega; \Psi \vdash S_2 \leq S_3) \sqsubset (\Delta; \Gamma \vdash A)}{(\Omega; \Psi \vdash S_1 \leq S_3) \sqsubset (\Delta; \Gamma \vdash A)}\ \textbf{Sub-trans}$$

$$\frac{(\Omega; \Psi \vdash S_2 \leq S_1) \sqsubset (\Delta; \Gamma \vdash A) \quad (\Omega; \Psi, x{:}S_2 \vdash S_1' \leq S_2') \sqsubset (\Delta; \Gamma, x{:}A \vdash A')}{(\Omega; \Psi \vdash \Pi x{:}S_1.S_1' \leq \Pi x{:}S_2.S_2') \sqsubset (\Delta; \Gamma \vdash \Pi x{:}A.A'}\ \textbf{Sub-pi}$$

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 $\boxed{(\Omega; \Psi \vdash W_1 \leq W_2) \sqsubset (\Delta; \Gamma \vdash V)}$ – 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

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

$$\frac{(\mathbf{w} \notin G) \quad (\Omega \vdash H_1 \leq H_2) \sqsubset (\Delta \vdash G : \texttt{schema}) \quad (\Omega; \Psi \vdash W_1 \leq W_2) \sqsubset (\Delta; \Gamma \vdash V : \texttt{world})}{(\Omega \vdash H_1 + \mathbf{w}{:}W_1 \leq H_2 + \mathbf{w}{:}W_2) \sqsubset (\Delta; \Gamma \vdash G + \mathbf{w}{:}V)} \textbf{ SubS-sum}$$

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

$$\frac{\Delta \vdash \mathcal{A} : \texttt{ctype} \quad \Delta \vdash \Xi : \texttt{cctx}}{\Delta; \Xi \vdash [\mathcal{A}] : \texttt{ctype}}\ \textbf{CT-meta} \quad \frac{\Delta; \Xi \vdash \tau_1 : \texttt{ctype} \quad \Delta; \Xi \vdash \tau_2 : \texttt{ctype}}{\Delta; \Xi \vdash \tau_1 \to \tau_2 : \texttt{ctype}}\ \textbf{CT-arr}$$

$$\frac{\Delta \vdash \mathcal{A} : \texttt{mtype} \quad \Delta, X{:}\mathcal{A}; \Xi \vdash \tau : \texttt{ctype}}{\Delta; \Xi \vdash \Pi X{:}\mathcal{A}.\tau : \texttt{ctype}}\ \textbf{CT-pi}$$

$\boxed{\Delta; \Xi \vdash e : \tau}$ – Expression e has type τ

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

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

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

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

$$\frac{\tau = \Pi\Delta_0.\Pi X_0{:}\mathcal{A}_0.\tau_0 \quad \Delta \vdash \theta : \Delta_0 \quad \Delta \vdash \mathcal{M} : [\![\theta]\!]\mathcal{A}_0 \quad \Delta; \Xi \vdash b : \tau\ (\text{for all } b \in \vec{b})}{\Delta; \Xi \vdash (\texttt{case}^{\tau}\ [\mathcal{M}]\ \texttt{of}\ \vec{b}) : [\![\rho, \mathcal{M}/X_0]\!]\tau_0}\ \textbf{CT-case}$$

$\boxed{\Delta; \Xi \vdash b : \tau}$ – Branch b matches invariant τ

$$\frac{\Delta_0 \vdash \mathcal{M}_0 : \mathcal{A}_0 \quad \Delta, \Delta_0 \vdash \mathcal{A} \doteq \mathcal{A}_0/(\theta, \Delta') \quad \Delta'; [\![\theta]\!]\Xi \vdash [\![\theta]\!]e : [\![\theta]\!]\tau_0}{\Delta; \Xi \vdash (\Delta_0; [\mathcal{M}_0] \mapsto e) : \Pi\Delta_1.\Pi X_0{:}\mathcal{A}_0.\tau_0}\ \textbf{CT-branch}$$

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

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

$$\frac{(\Omega; \Phi \vdash \zeta_1) \sqsubset (\Delta; \Xi \vdash \tau_1 : \texttt{ctype}) \quad (\Omega; \Xi \vdash \zeta_2) \sqsubset (\Delta; \Xi \vdash \tau_2 : \texttt{ctype})}{(\Omega; \Phi \vdash \zeta_1 \to \zeta_2) \sqsubset (\Delta; \Xi \vdash \tau_1 \to \tau_2 : \texttt{ctype})}\ \textbf{CTR-arr}$$

$$\frac{(\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A} : \texttt{mtype}) \quad (\Omega, X{:}\mathcal{S}; \Phi \vdash \zeta) \sqsubset (\Delta, X{:}\mathcal{A}; \Xi \vdash \tau : \texttt{ctype})}{(\Omega; \Phi \vdash \Pi X{:}\mathcal{S}.\zeta) \sqsubset (\Delta; \Xi \vdash \Pi X{:}\mathcal{A}.\tau : \texttt{ctype})}\ \textbf{CTR-pi}$$

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

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

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

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

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

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

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

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

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

$\boxed{(\Omega;\Phi \vdash c : \zeta) \sqsubset (\Delta;\Xi \vdash b : \tau)}$ – Branch c/b matches invariant ζ/τ

$$\frac{\begin{array}{l}(\Omega_0 \vdash^{\ell} \mathcal{N}_0 : \mathcal{S}_0) \sqsubset (\Delta_0 \vdash^{\ell} \mathcal{M}_0 : \mathcal{A}_0) \\ (\Omega, \Omega_0;\Phi \vdash f : [\![\mathcal{N}_0/X_0]\!]\zeta_0) \sqsubset (\Delta, \Delta_0;\Xi \vdash e : [\![\mathcal{M}_0/X_0]\!]\tau_0)\end{array}}{(\Omega;\Phi \vdash (\Omega_0; [\mathcal{N}_0] \Rightarrow f) : \Pi X_0{:}\mathcal{S}_0.\zeta_0) \sqsubset (\Delta;\Xi \vdash (\Delta_0; [\mathcal{M}_0] \Rightarrow e) : \Pi X_0{:}\mathcal{A}_0.\tau_0)}\ \textbf{CTR-branch}$$

$\boxed{(\Omega;\Phi \vdash \zeta_1 \leq \zeta_2) \sqsubset (\Delta;\Xi \vdash \tau)}$ – 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
		$\mathtt{schema}\ \mathbf{g} = \mathbf{w}_1{:}V_1 \mid ... \mid \mathbf{w}_n{:}V_n$	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

$$\frac{\mathbf{a} \notin \Sigma \quad \mathbf{c}_1, ..., \mathbf{c}_n \notin \Sigma \quad \cdot;\cdot \vdash_\Sigma K : \mathtt{kind} \quad \cdot;\cdot \vdash_{\Sigma,\mathbf{a}:K} A_i \Leftarrow \mathtt{type}\ \text{(for all i)}}{\vdash_\Sigma (\mathtt{LF}\ \mathbf{a} : K = \mathbf{c}_1{:}A_1 \mid ... \mid \mathbf{c}_n{:}A_n) : \mathbf{decl}}$$

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

$$\frac{\mathbf{g} \notin \Sigma \quad \mathbf{w}_1, ...\mathbf{w}_n \notin \Sigma \quad \cdot;\cdot \vdash_\sigma V_i : \mathtt{world}\ \text{(for all i)}}{\vdash_\Sigma (\mathtt{schema}\ \mathbf{g} = \mathbf{w}_1{:}V_1 \mid ... \mid \mathbf{w}_n{:}V_n) : \mathbf{decl}}$$

$$\frac{\mathbf{h} \notin \Sigma \quad (\mathtt{schema}\ \mathbf{g} = \mathbf{w}_1{:}V_1 \mid ... \mid \mathbf{w}_n{:}V_n) \in \Sigma \quad (\cdot;\cdot \vdash_\Sigma W_{i_j}) \sqsubset (\cdot;\cdot \vdash_\Sigma V_{i_j} : \mathtt{world})\ \text{(for all j)}}{\vdash_\Sigma (\mathtt{schema}\ \mathbf{h} \sqsubset \mathbf{g} = \mathbf{w}_{i_1}{:}W_{i_1} \mid ... \mid \mathbf{w}_{i_k}{:}W_{i_k}) : \mathbf{decl}}$$

$$\frac{\mathbf{f} \notin \Sigma \quad (\cdot;\cdot \vdash_{\Sigma,\mathtt{rec}\ \mathbf{f}:\zeta} f : \zeta) \sqsubset (\cdot;\cdot \vdash_{\Sigma,\mathtt{rec}\ \mathbf{f}:\zeta} e : \tau)}{\vdash_\Sigma (\mathtt{rec}\ \mathbf{f} : \zeta = f) : \mathbf{decl}}$$