Don't Worry, Be Negative: Set Theoretical Semantics for Stratified Refinement Types

ANONYMOUS AUTHOR(S)

I love negative occurrences so much!

1 Introduction

Refinement types [6] extend an existing type system by allowing logical predicates to constrain the values of a type. For example, the type $\{x\colon Int \mid x \geq 0\}$ refines the type of integers to include only non-negative values.

When refinement types are restricted to the so-called Liquid Types [9]—that is, when logical formulas are confined to a decidable fragment handled by an SMT solver—the system gains a high degree of automation. This enables automatic discharge of proof obligations and even refinement type inference.

Several languages, including Haskell [11], Rust [7], and Java [4], implement Liquid Types. These implementations have been used to verify a range of properties, from safe array indexing [14] to information flow security [8]. However, this SMT-based automation imposes a limitation: refinements must belong to a decidable logical fragment.

[12] and Ferrarini [3] address this limitation by extending the refinement logic to support arbitrary (terminating) expressions of the host programming language within refinements.

Yet, even with such extensions, refinement types fall short of supporting complex mechanization projects akin to those in dependently typed languages. Taking inspiration from dependent type theory, Liquid Haskell introduces *Data Propositions*, by refining the constructors of inductive data types. [3] further demonstrate that they can be used in place of indexed families and how to add dependent pattern matching to the system. [2] uses them to mechanize a proof of safety and preservation for a model of the refinement type system behind Liquid Haskell.

Still, one powerful mechanism from dependently typed languages remains out of reach: *large elimination*. Because refinement type systems only enrich the type language and not the term language this technique is unavailable unless the base language is itself dependently typed.

To work around this, [3] rely on data types with negative occurrences, in the style of stratified types as introduced by [5] present in Beluga. However, Ferrarini et al. does not provide a formal treatment of stratified types within the context of refinement types, and in particular they never prove consistency of the resulting system.

In this work, we close this gap. We show that both positive inductive families and stratified families can be expressed directly within a refinement type system, without extending the refinement logic beyond the quantifier-free fragment. Our encoding refines data constructors alone, making dependent and stratified types immediately available to existing refinement type systems.

2 Overview

As an illustrative example, we present the shallow embedding of the simply typed lambda calculus from [3] and compare it to the one in [10], written in Agda [1]. We show how stratified types naturally emerge when translating programs from dependently typed languages to *liquidly* typed ones.

2.1 Inductive types

We represent types of STLC using a simple inductive type, here there is no difference between the Agda and Liquid Haskell definitions, other than the syntax.

```
data Ty : Set where
  ι   : Ty
  Arr : Ty → Ty → Ty
```

(a) Object type system in Agda

```
data Ty
  = Iota
  | Arr Ty Ty
```

(b) Object type system in Liquid Haskell

```
data Term : Ctx → Ty → Set where
  app : ∀ {Γ σ τ} → Term Γ (Arr σ τ) → Term Γ σ → Term Γ τ
  lam : ∀ {Γ σ τ} → Term (σ :: Γ) τ → Term Γ (Arr σ τ)
  var : ∀ {Γ σ} → Ref σ Γ → Term Γ
```

Fig. 2. Object syntax in Agda

```
data Term where
  {-@ App :: σ:Ty -> τ:Ty -> γ:Ctx -> Ix Term (γ, (Arrow σ τ))
          -> Ix Term (γ, σ) -> Ix Term (γ, τ) @-}
  App :: Ctx -> Ty -> Ty -> Term -> Term -> Term
  {-@ Lam :: σ:Ty -> τ:Ty -> γ:Ctx -> Ix Term ((Cons σ γ), τ)
          -> Ix Term (γ, (Arrow σ τ))  @-}
  Lam :: Ctx -> Ty -> Ty -> Term -> Term -> Term
  {-@ Var :: σ:Ty -> γ:Ctx -> Ix Ref (σ, γ) -> Ix Term (γ, σ) @-}
  Var :: Ctx -> Ty -> Ref -> Term
```

Fig. 3. Object syntax in Liquid Haskell

```
Value : Ty → Set
Value ι         = Z
Value (Arr σ τ) = Value σ → Value τ
```

Fig. 4. Shallow embedding of STLC values in Agda

The real difference arises when we introduce indexed types, where we can parameterize the type constructor with values. We use this to encode well-typed syntax for STLC.

We omit the definitions of contexts `Ctx` and references in the context `Ref`.

In Haskell, we can achieve something similar with GADTs, but the indexes can only be types and not terms. Instead of relying on GADTs, Liquid Haskell uses *Data Propositions*. We introduce in the refinement logic a new uninterpreted function `prop` and define a type alias: `type Ix T e = {v:T | e = prop v}`. We use `prop` as a proxy to assign indexes to individual values.

By refining the types of the constructors, we simultaneously axiomatize and enforce the invariants on their arguments.

2.2 Large elimination

Now we would like to provide a shallow embedding for values of STLC. In languages based on dependent type theory we can freely mix terms and types, and through large elimination we can construct a function that assigns to each object type its corresponding meta type.

But we can't do the same with refinement types if we aren't refining a language that has dependent types already, expressions must also be well typed in the base language so that would

```
data Value where
  {-@ VIota :: Int -> Ix Value Iota @-}
  VIota :: Int -> Value
  {-@ VFun :: σ:Ty -> τ:Ty -> (Ix Value σ -> Ix Value τ)
           -> Ix Value (Arrow σ τ) @-}
  VFun :: Ty -> Ty -> (Value -> Value) -> Value
```

Fig. 5. Shallow embedding of STLC values in Liquid Haskell

require us to find a suitable type ?? in `value :: τ:Ty -> {v:?? | ...}` as the dependency in the refined function space can be only in the refinement annotation.

So we can't define a function and the only possibility is to define a new datatype, the idea is instead of representing the function `val` directly in Liquid Haskell, we can define a type that encodes the graph of the function.

But then why didn't we use the same definition in Agda? The answer is simple, because it would have been rejected, and the same holds for pretty much any proof assistant based on dependent type theory. The issue is that the definition of `Value` breaks the positivity condition enforced on constructors of inductive datatypes, in particular in `VFun`, `Value` appears directly to the left hand side of the arrow; this declaration was also rejected by Liquid Haskell; since positivity serves as a syntactic check to guarantee that type definitions are well-founded, even though it only approximates the underlying semantic notion of soundness.

In the same style of [5] we will argue that this definition like the one of `Value` where in all the self references the index is getting structurally smaller than the one in the return type of the constructor are also well-defined, it is important to stress that also the one that are in positive position must be smaller.

3 Set Theoretical Model of Refinement Types

In this section we present a set theoretical model for refinement types. Concretely, we present the syntax (Section 3.1) of a core language $\lambda_r$that extends the simply typed lambda calculus with refinement types, the type system and its set theoretical semantics (Section 3.2), and we show that our type system is relatively consistent (Section 3.3). In the next sections we will gradually extend $\lambda_r$to include recursive refinement types (Section 4) and inductive types (Section 5).

3.1 Syntax of Refinement Types

Types	$\tau ::=$	$Bool$	*Booleans*
	$\mid$	Nat	*Natural numbers*
	$\mid$	$Unit$	*Unit*
	$\mid$	$(x\colon \tau_x) \to \tau$	*Dependent function*
	$\mid$	$\{x\colon \tau \mid e\}$	*Refinement type*
Expressions	$e ::=$	x	*Variable*
	$\mid$	$e_1\ e_2$	*Application*
	$\mid$	$\lambda x\colon \tau.\ e$	*Lambda abstraction*
	$\mid$	$True \mid False$	*Booleans*
	$\mid$	$unit$	*Unit*
	$\mid$	$n \in \mathbb{N}$	*Natural numbers*
	$\mid$	$\textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_e$	*If-then-else*
	$\mid$	$e_l \circledcirc e_r \quad \circledcirc \in \{<, =, +\}$	*Basic operators*
Contexts	$\Gamma ::=$	$\emptyset$	*Empty context*
	$\mid$	$\Gamma,\ x\colon \tau$	*Variable binding*

Fig. 6. Syntax of λ_r

Figure 6 presents the syntax of λ_r, a simply typed lambda calculus extended with booleans, natural numbers and unit types, and refinement types. A refinement type $\{x\colon \tau \mid e\}$ is the type of all values of type τ that satisfy the predicate e where the variable x is bound to the value being checked. The function type $(x\colon \tau_x) \to \tau$ is a dependent function type where the refinements in the codomain type τ can depend on the argument x of type τ_x.

3.2 Typing Rules and Semantics

Figure 7 presents the typing rules of λ_r. Next, we present the set theoretical interpretation of these rules defined on well formed contexts, types and expressions.

3.2.1 Well-formed Contexts. The judgment $\vdash \Gamma$, at the top of Fig. 7, defines well-formed contexts. A context is well-formed either if it is empty (Ctx-Empty) or if it extends an already well-formed context with a new binding whose type is well-formed in the previous context, as types can depend on previous variables in the context (Ctx-Var).

The semantics of a context is the set of all variable assignments drawn from the semantics of their declared types. $\llbracket \vdash \Gamma \rrbracket : Set$ and we use the metavariable γ to range over elements of this set.

Concretelty, the semantics of well-formed contexts is defined inductively as follows:

$$\left\llbracket \frac{}{\vdash \emptyset} \right\rrbracket \triangleq \mathbb{1} \qquad\qquad \left\llbracket \frac{\vdash \Gamma \qquad \Gamma \vdash \tau}{\vdash \Gamma,\ x\colon \tau} \right\rrbracket \triangleq \gamma \in \llbracket \vdash \Gamma \rrbracket \times \llbracket \Gamma \vdash \tau \rrbracket\,(\gamma)$$

The empty context has no variables and thus corresponds to the trivial assignment, represented by the singleton $\mathbb{1}$. Extending a context introduces a new variable, and its semantics is given by the set-theoretic product of the semantics of the preceding context and the semantics of the new variable's type in that context. Because types may depend on earlier bindings, the latter semantics is parameterized by the semantics of the preceding context.

Well-Formed Contexts $\vdash \Gamma$

$$\frac{}{\vdash \emptyset}\ \textsc{Ctx-Empty} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau}{\vdash \Gamma,\ x\colon \tau}\ \textsc{Ctx-Var}$$

Well-Formed Types $\Gamma \vdash \tau$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathit{Bool}}\ \textsc{Wf-Bool} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathit{Nat}}\ \textsc{Wf-Nat} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathit{Unit}}\ \textsc{Wf-Unit}$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \tau_x \qquad \Gamma,\ x\colon \tau_x \vdash \tau}{\Gamma \vdash (x\colon \tau_x) \to \tau}\ \textsc{Wf-Fun} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau \qquad \Gamma,\ x\colon \tau \vdash r\colon \mathit{Bool}}{\Gamma \vdash \{x\colon \tau \mid r\}}\ \textsc{Wf-Ref}$$

Well-typed Terms $\Gamma \vdash e\colon \tau$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathit{True}\colon \mathit{Bool}}\ \textsc{T-True} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathit{False}\colon \mathit{Bool}}\ \textsc{T-False} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathit{unit}\colon \mathit{Unit}}\ \textsc{T-Unit}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash n\colon \mathit{Nat}}\ \textsc{T-Nat} \qquad \frac{\begin{matrix}\vdash \Gamma \qquad \Gamma \vdash \tau \qquad \Gamma \vdash e_g\colon \mathit{Bool} \\ \Gamma \vdash e_t\colon \tau \qquad \Gamma \vdash e_e\colon \tau\end{matrix}}{\Gamma \vdash \textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_e\colon \tau}\ \textsc{T-If} \qquad \frac{\begin{matrix}\vdash \Gamma \\ \Gamma \vdash e_l\colon \mathit{Nat} \qquad \Gamma \vdash e_r\colon \mathit{Nat}\end{matrix}}{\Gamma \vdash e_l < e_r\colon \mathit{Bool}}\ \textsc{T-Le}$$

$$\frac{\begin{matrix}\vdash \Gamma \qquad \Gamma \vdash \tau \\ \Gamma \vdash e_l\colon \tau \qquad \Gamma \vdash e_r\colon \tau\end{matrix}}{\Gamma \vdash e_l = e_r\colon \mathit{Bool}}\ \textsc{T-Eq}$$

Fig. 7. Typing Rules of λ_r

$$\begin{array}{ll}
[\![\vdash \Gamma]\!] : \mathit{Set} & \gamma \in [\![\vdash \Gamma]\!] \\
[\![\Gamma \vdash \tau]\!] : [\![\vdash \Gamma]\!] \longrightarrow \mathit{Set} & w \in [\![\Gamma \vdash \tau]\!]\,(\gamma) \\
[\![\Gamma \vdash e\colon \tau]\!] : \gamma \in [\![\vdash \Gamma]\!] \longrightarrow [\![\Gamma \vdash \tau]\!]\,(\gamma) &
\end{array}$$

Fig. 8. Semantics Definitions

3.2.2 Well-formed Refinement Types. The judgment $\Gamma \vdash \tau$ asserts that the type τ is well-formed in the context Γ. Because τ may contain free variables bound in Γ, its semantics is defined relative to an assignment for those variables. Consequently, the semantics of a well-formed type is a function from the semantics of the context to a set.

$$[\![\Gamma \vdash \tau]\!] : [\![\vdash \Gamma]\!] \longrightarrow \mathit{Set}$$

Basic types. Well-formedness of the basic types are checked by the rules Wf-Bool, Wf-Nat, and Wf-Unitrespectively stating that Booleans *Bool*, Unit *Unit* and Natural numbers *Nat* are well formed under any well-formed context. Their semantics is the one of their set theoretical counterparts.

$$\left[\!\left[\frac{\vdash \Gamma}{\Gamma \vdash \mathit{Bool}}\right]\!\right](\gamma) \triangleq \mathbb{2} \qquad \left[\!\left[\frac{\vdash \Gamma}{\Gamma \vdash \mathit{Nat}}\right]\!\right](\gamma) \triangleq \mathbb{N} \qquad \left[\!\left[\frac{\vdash \Gamma}{\Gamma \vdash \mathit{Unit}}\right]\!\right](\gamma) \triangleq \mathbb{1}$$

Where $\mathbb{2} = \{tt, f\!f\}$, $\mathbb{1} = \{\star\}$ and $\mathbb{N}$ is the set of natural numbers.

Dependent function. Rule Wf-Fun checks that a dependent function type is well-formed in context Γ when the domain type τ_x is well-formed in Γ, and the codomain type τ is well-formed in the extended context $\Gamma,\ x\colon \tau_x$. Its semantics is the set of functions that map each element of the semantic domain type to an element of the corresponding semantic codomain type. Because the codomain type may depend on the domain variable, the semantics of the codomain is evaluated in the extended semantic context.

$$\left[\!\left[\frac{\vdash \Gamma \qquad \Gamma \vdash \tau_x \qquad \Gamma,\ x\colon \tau_x \vdash \tau}{\Gamma \vdash (x\colon \tau_x) \to \tau}\right]\!\right](\gamma) \triangleq w \in [\![\Gamma \vdash \tau_x]\!]\,(\gamma) \to [\![\Gamma,\ x\colon \tau_x \vdash \tau]\!]\,(\gamma,\ w) \tag{1}$$

And the definition is well defined by Lemma A.1.

Refined type. Rule Wf-Ref checks that a refinement type is well formed in context Γ when the base type τ is well formed and the refinement expression r is a *Bool*-typed expression in the extended context $\Gamma,\ x\colon \tau$. The semantics is described by the collection of semantic values of the base type who also satisfy the predicate described by r. The semantics of $\Gamma,\ x\colon \tau \vdash r\colon \mathit{Bool}$ is the one of the function from the interpretation of the extended context $[\![\vdash \Gamma,\ x\colon \tau]\!]$ to $\mathbb{2}$.

$$\left[\!\left[\frac{\vdash \Gamma \qquad \Gamma \vdash \tau \qquad \Gamma,\ x\colon \tau \vdash r\colon \mathit{Bool}}{\Gamma \vdash \{x\colon \tau \mid r\}}\right]\!\right](\gamma) \triangleq \left\{ w \;\middle|\; \begin{array}{c} w \in [\![\Gamma \vdash \tau]\!]\,(\gamma) \\ [\![\Gamma,\ x\colon \tau \vdash r\colon \mathit{Bool}]\!]\,(\gamma,\ w) = tt \end{array} \right\} \tag{2}$$

And the definition is well defined by Theorem A.2.

3.2.3 Well-typed Terms. We use the judgment $\Gamma \vdash e\colon \tau$ to say that an expression e has type τ in context Γ. Their semantics are set theoretic functions going from the interpretation of contexts to the interpretation of the types in those contexts.

$$[\![\Gamma \vdash e\colon \tau]\!] : \gamma \in [\![\vdash \Gamma]\!] \to [\![\Gamma \vdash \tau]\!]\,(\gamma)$$

Basic Terms. Literal values are well typed under any well-formed context, as stated by rules T-True, T-False, T-Unit, and T-Nat. The semantics is the set theoretic counterpart.

$$\left[\!\left[\frac{\vdash \Gamma}{\Gamma \vdash \mathit{True}\colon \mathit{Bool}}\right]\!\right](\gamma) \triangleq tt \in [\![\Gamma \vdash \mathit{Bool}]\!]\,(\gamma) \qquad \left[\!\left[\frac{\vdash \Gamma}{\Gamma \vdash \mathit{False}\colon \mathit{Bool}}\right]\!\right](\gamma) \triangleq f\!f \in [\![\Gamma \vdash \mathit{Bool}]\!]\,(\gamma)$$

$$\left[\!\left[\frac{\vdash \Gamma}{\Gamma \vdash \mathit{unit}\colon \mathit{Unit}}\right]\!\right](\gamma) \triangleq \star \in [\![\Gamma \vdash \mathit{Unit}]\!]\,(\gamma) \qquad \left[\!\left[\frac{\vdash \Gamma}{\Gamma \vdash n\colon \mathit{Nat}}\right]\!\right](\gamma) \triangleq n \in [\![\Gamma \vdash \mathit{Nat}]\!]\,(\gamma)$$

We also type the following basic operations: conditional expressions (if-then-else), less-than comparison on natural numbers, and equality. Rule T-If states that the if-then-else expression is well typed when the guard has type *Bool* and both branches have the same type τ, resulting in an expression of type τ. Rule T-Le renders the less-than operation well typed when both operands have type *Nat*, producing an expression of type *Bool*. Rule T-Eq states that the equality operation is well typed when both operands have the same type τ, yielding an expression of type *Bool*. In particular, since τ can be arbitrary, equality may be applied to values of types for which equality is not computable, such as function types. While this causes no semantic issue, since set-theoretic equality is always defined, it is problematic for implementations, as evaluating expressions such as **if** $f = g$ **then** ... **else** ... does not make sense when f and g are functions. In practice, this is resolved by disallowing such comparisons in the main body and executable part of the program, while permitting them only within refinement expressions.

NV: HERE

The semantics of the if-then-else expression is given by case analysis on the semantics of the guard expression as $[\![Bool]\!](\gamma) = \mathbb{2} = \{tt, ff\}$ we can distinguish the two possible cases and assign the semantics of the corresponding branch.

AF: Should we add dependent pattern matching here?

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma \vdash \tau \qquad \Gamma \vdash e_g \colon Bool \quad \Gamma \vdash e_t \colon \tau \qquad \Gamma \vdash e_e \colon \tau}{\Gamma \vdash \textbf{if } e_g \textbf{ then } e_t \textbf{ else } e_e \colon \tau}\right]\!\!\right](\gamma) \triangleq \left(\begin{cases} [\![\Gamma \vdash e_t \colon \tau]\!](\gamma) & \text{if } [\![\Gamma \vdash e_g \colon Bool]\!](\gamma) = tt \\ [\![\Gamma \vdash e_e \colon \tau]\!](\gamma) & \text{if } [\![\Gamma \vdash e_g \colon Bool]\!](\gamma) = ff \end{cases}\right) \in [\![\Gamma \vdash \tau]\!](\gamma) \tag{3}$$

And the definition is well defined by Theorem A.3.
In the same fashion the semantics of the less-than operator on natural numbers is given by the set theoretical less-than operator on the set-theoretical natural numbers.

$$\left[\!\!\left[\frac{\vdash \Gamma \quad \Gamma \vdash e_l \colon Nat \qquad \Gamma \vdash e_r \colon Nat}{\Gamma \vdash e_l < e_r \colon Bool}\right]\!\!\right](\gamma) \triangleq \left(\begin{cases} tt & \text{if } [\![\Gamma \vdash e_l \colon Nat]\!](\gamma) < [\![\Gamma \vdash e_r \colon Nat]\!](\gamma) \\ ff & \text{if } [\![\Gamma \vdash e_l \colon Nat]\!](\gamma) \geq [\![\Gamma \vdash e_r \colon Nat]\!](\gamma) \end{cases}\right) \in [\![\Gamma \vdash Bool]\!](\gamma) \tag{4}$$

The definition is well defined by Theorem A.4.
Finally, the semantics of the equality operator is given by set-theoretic equality. For functions, this corresponds to the notion of exact equality defined in [3]: two functions are equal only if they are both defined on the same domain and return equal values everywhere. In particular, if one function is defined on a strictly smaller domain than the other, they are not considered equal, as this would violate the substitution principle and lead to inconsistencies.

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma \vdash \tau \quad \Gamma \vdash e_l \colon \tau \qquad \Gamma \vdash e_r \colon \tau}{\Gamma \vdash e_l = e_r \colon Bool}\right]\!\!\right](\gamma) \triangleq \left(\begin{cases} tt & \text{if } [\![\Gamma \vdash e_l \colon \tau]\!](\gamma) = [\![\Gamma \vdash e_r \colon \tau]\!](\gamma) \\ ff & \text{if } [\![\Gamma \vdash e_l \colon \tau]\!](\gamma) \neq [\![\Gamma \vdash e_r \colon \tau]\!](\gamma) \end{cases}\right) \in [\![\Gamma \vdash Bool]\!](\gamma) \tag{5}$$

And the definition is well defined by Theorem A.5.

3.2.4 Variables. A variable reference to a variable x is well typed of type τ when the binding $x \colon \tau$ is in the context Γ.

$$\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ x \colon \tau,\ \dots}{\Gamma \vdash x \colon \tau}\ \textsc{T-Var}$$

And the semantics is the lookup of the variable assignment in the semantic context.

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ x \colon \tau,\ \dots}{\Gamma \vdash x \colon \tau}\right]\!\!\right](\delta,\ w,\ \dots) \triangleq w \in [\![\Gamma \vdash \tau]\!](\delta,\ x \colon \tau,\ \dots) \tag{6}$$

Which is well defined by Theorem A.6.

3.2.5 Function application. Function applications are well typed when the function expression is of a dependent function type and the argument expression is of the domain type, since the codomain type can depend on the argument, we need to substitute the argument expression for the variable in the codomain type.

$$\frac{\vdash \Gamma \qquad \Gamma \vdash e_1 \colon (x \colon \tau_x) \to \tau \qquad \Gamma \vdash e_2 \colon \tau_x}{\Gamma \vdash e_1\ e_2 \colon \tau[x/e_2]}\ \textsc{T-App}$$

And the semantics is given by the application of the set theoretic function.

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma \vdash e_1 \colon (x \colon \tau_x) \to \tau \qquad \Gamma \vdash e_2 \colon \tau_x}{\Gamma \vdash e_1\ e_2 \colon \tau[x/e_2]}\right]\!\!\right](\gamma)$$
$$\triangleq ([\![\Gamma \vdash e_1 \colon (x \colon \tau_x) \to \tau]\!](\gamma)\ \ [\![\Gamma \vdash e_2 \colon \tau_x]\!](\gamma)) \in [\![\Gamma \vdash \tau[x/e_2]]\!](\gamma) \tag{7}$$

Which is well defined by Theorem A.7.

3.2.6 Lambda abstraction. Lambda abstractions are are well typed when the body is well typed in the context extended with the argument variable binding.

$$\frac{\vdash \Gamma \qquad \Gamma,\ x \colon \tau_x \vdash e \colon \tau \qquad \Gamma \vdash \tau_x \qquad \Gamma,\ x \colon \tau_x \vdash \tau}{\Gamma \vdash \lambda x \colon \tau_x.\ e \colon (x \colon \tau_x) \to \tau}\ \textsc{T-Lam}$$

And the semantics is the one of set-theoretic functions.

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma,\ x \colon \tau_x \vdash e \colon \tau \qquad \Gamma \vdash \tau_x \qquad \Gamma,\ x \colon \tau_x \vdash \tau}{\Gamma \vdash \lambda x \colon \tau_x.\ e \colon (x \colon \tau_x) \to \tau}\right]\!\!\right](\gamma)$$
$$\triangleq (w \in [\![\Gamma \vdash \tau_x]\!](\gamma) \mapsto [\![\Gamma,\ x \colon \tau_x \vdash e \colon \tau]\!](\gamma,\ w)) \in [\![\Gamma \vdash (x \colon \tau_x) \to \tau]\!](\gamma) \tag{8}$$

And the definition is well defined by Theorem A.8.

3.2.7 Refinement reflection. We can always construct the singleton refinement type that semantically contains only one element, syntactically it can be expressed in many ways as multiple expressions can have identical semantics. We use the reflection rule to express that an expression of type τ can be given the more precise type of the refinement type that contains only the values of type τ that are equal to that expression.

$$\frac{\vdash \Gamma \qquad \Gamma \vdash e \colon \tau}{\Gamma \vdash e \colon\ \{x \colon \tau \mid x = e\}}\ \textsc{T-Refl}$$

Semantically the interpretation is straightforward the identity operation, as the semantics of e is already in the semantics of the refinement type by definition.

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma \vdash e \colon \tau}{\Gamma \vdash e \colon\ \{x \colon \tau \mid x = e\}}\right]\!\!\right](\gamma) \triangleq [\![\Gamma \vdash e \colon \tau]\!](\gamma) \in [\![\Gamma \vdash \{x \colon \tau \mid x = e\}]\!](\gamma) \tag{9}$$

And it is well defined by Theorem A.9.

3.2.8 Subtyping coercion. If we know that a type τ_2 is a subtype of τ_1 in context Γ, then any expression of type τ_2 can be given the type τ_1.

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \tau_2 \preceq \tau_1 \qquad \Gamma \vdash e \colon \tau_2}{\Gamma \vdash e \colon \tau_1}\ \textsc{T-Cast}$$

Semantically like the reflection rule, the interpretation is the identity operation, as the semantics of e is in the semantics of τ_1 by the fact that τ_2 is a subtype of τ_1 and in our model subtyping is interpreted as set inclusion.

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma \vdash \tau_2 \preceq \tau_1 \qquad \Gamma \vdash e\colon \tau_2}{\Gamma \vdash e\colon \tau_1}\right]\!\!\right](\gamma) \triangleq ([\![\Gamma \vdash e\colon \tau_2]\!](\gamma)) \in [\![\Gamma \vdash \tau_1]\!](\gamma) \tag{10}$$

And it is well defined by Theorem A.10.

3.2.9 Subtyping. We use the judgment $\Gamma \vdash \tau_1 \preceq \tau_1$ to say that the type τ_1 is a subtype of the type τ_2 in context Γ their semantics is a whiteness to the fact that the semantic interpretation of τ_1 is a subset of the one of τ_2. In addition we have an extra judgment $\Gamma \models e$ that represents information obtained trough the SMT solver and we assume that it is semantic respecting *i.e.*:

Assumption 1 (Entailment soundness).

$$\Gamma \models e \implies \forall \gamma \in [\![\vdash \Gamma]\!]\ [\![\Gamma \vdash e\colon Bool]\!](\gamma) = tt$$

Theorem 3.1 (Semantic subtyping). *If* $\Gamma \vdash \tau_1 \preceq \tau_2$ *then* $\forall \gamma \in [\![\vdash \Gamma]\!]$ $[\![\Gamma \vdash \tau_1]\!](\gamma) \subseteq [\![\Gamma \vdash \tau_2]\!](\gamma)$

Subtyping can happen at the refinement level and is witnessed by an implication between the two refinement predicates:

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \tau_1 \preceq \tau_2 \qquad \Gamma,\ x\colon \tau_1 \models e_1 \implies e_2[y/x]}{\Gamma \vdash \{x\colon \tau_1 \mid e_1\} \preceq \{y\colon \tau_2 \mid e_2\}}\ \textsc{Sub-Base}$$

3.2.10 Refinement unrolling. We can unroll a nested refinement as an if-then-else, the rule can go in both directions, here we show need only one direction.

$$\frac{\vdash \Gamma}{\Gamma \vdash \{x\colon\ \{y\colon \tau \mid e_i\} \mid e_o\} \preceq \left\{x\colon \tau \;\middle|\; \begin{array}{l}\textbf{if } e_i[y/x]\\ \textbf{then } e_o\\ \textbf{else } False\end{array}\right\}}\ \textsc{Sub-Flat}$$

3.2.11 Function subtyping. The subtyping rule internalizes the co(ntra)variance of set theoretic functions.

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \tau_y \preceq \tau_x \qquad \Gamma,\ y\colon \tau_y \vdash \tau_1[x/y] \preceq \tau_2}{\Gamma \vdash (x\colon \tau_x) \longrightarrow \tau_1 \preceq (y\colon \tau_y) \longrightarrow \tau_2}\ \textsc{Sub-Arr}$$

3.3 Relative consistency

Since we have given a set theoretic semantics to our type system, and interpreted the judgment $\Gamma \vdash e\colon \tau$ as function of type $\gamma \in [\![\vdash \Gamma]\!] \longrightarrow [\![\Gamma \vdash \tau]\!](\gamma)$, This allows us to show that a contradiction in our type system corresponds to a contradiction in set theory.

Theorem 3.2 (Relative consistency).
We represent falsity as the type $\{x\colon Nat \mid False\}$*, then: there is no* e *such that* $\emptyset \vdash e\colon\ \{x\colon Nat \mid False\}$ *is derivable. If set theory is consistent.*

Proof. Suppose by contradiction that there exists an expression e such that it admits a type derivation of $\emptyset \vdash e\colon\ \{x\colon Nat \mid False\}$, then:

$$\begin{array}{rcr} [\![\emptyset \vdash e\colon\ \{x\colon Nat \mid False\}]\!] & : & \gamma \in [\![\vdash \emptyset]\!] \longrightarrow [\![\emptyset \vdash \{x\colon Nat \mid False\}]\!](\gamma) \\ & = & \mathbb{1} \longrightarrow \emptyset \end{array}$$

But if set theory is consistent, then there is no function from the non empty set $\mathbb{1}$ to $\emptyset$, hence we have a contradiction. □

4 Recursive bindings

Now we can extend our language with recursive bindings. A recursive let binding allows us to define a function f that can call itself recursively on arguments that are smaller according to a measure e_m.

$$\begin{array}{llll} \textbf{Expressions} & e ::= & \dots & \\ & \mid & \textbf{let } [e_1]\ f\ x\colon \tau_x = e_m \textbf{ in } e_r & \textit{Recursive let binding} \\ & \mid & \dots & \end{array}$$

The metric is expressed as a function from the type of the argument τ_x to the natural numbers Nat and is used to strengthen the type of the function f in the body of the definition to ensure that recursive calls are made on smaller arguments only. This guarantees termination of the recursion.

$$\frac{\begin{array}{c}\vdash \Gamma \qquad \Gamma \vdash \tau_x \qquad \Gamma,\ x\colon \tau_x \vdash \tau_r \\ \Gamma,\ x\colon \tau_x,\ f\colon (x\colon \{y\colon \tau_x \mid e_m\ x > e_m\ y\}) \to \tau_r \vdash e_1\colon \tau_r \\ \Gamma \vdash e_m\colon \tau_x \to Nat \qquad \Gamma,\ f\colon (x\colon \tau_x) \to \tau_r \vdash e_2\colon \tau\end{array}}{\Gamma \vdash \textbf{let } [e_m]\ f\ x\colon \tau_x \to \tau_r = e_1 \textbf{ in } e_2\colon \tau}\ \textsc{T-Let}$$

Our metric depends on a single variable x, but this does not reduce expressiveness: any lexicographic order on several variables can be encoded as a single natural number using, for example, the Cantor pairing function

$$\pi(x, y) \triangleq \frac{1}{2}(x + y)(x + y + 1) + y.$$

Let $\gamma \in [\![\vdash \Gamma]\!]$. The measure e_m allows us to separate the values of $[\![\Gamma \vdash \tau_x]\!]\ (\gamma)$ into $\mathbb{N}$ families.

$$Layer(n) \triangleq \left\{ w \;\middle|\; \begin{array}{c} w \in [\![\Gamma \vdash \tau_x]\!]\ (\gamma) \\ [\![\Gamma \vdash e_m\colon \tau_x \to Nat]\!]\ (\gamma)\ (w) = n \end{array} \right\} \tag{11}$$

Which is well defined from Theorem A.11.

Now we can give a recursive characterization of the semantics of the recursive binding, as we can define the functional F that destcibes the semantics of f at layer n in terms of the semantics at the previous layers. We can do this only because the strenghtened type of f in the body of the definition ensures that the semantics of each $Layer(n)$ depends only on the semantics of the previous layers.

$$\begin{aligned} F : n \in Nat \to & \left(w \in \bigcup_{i=0}^{n-1} Layer(i) \to [\![\Gamma,\ x\colon \tau_x \vdash \tau_r]\!]\ (\gamma,\ w) \right) \\ & \qquad \to w \in \bigcup_{i=0}^{n} Layer(i) \to [\![\Gamma,\ x\colon \tau_x \vdash \tau_r]\!]\ (\gamma,\ w) \end{aligned}$$

Where

$$F(n, X, w) \triangleq \begin{cases} X(w) & \text{if } w \notin Layer(n) \\ [\![\Gamma,\ x\colon \tau_x,\ f\colon \dots \vdash e_1\colon \tau_r]\!]\ (\gamma,\ w,\ X) & \text{if } w \in Layer(n) \end{cases} \tag{12}$$

Which is well defined from Theorem A.12. Now since F is defined in terms of functions over increasing layers, we can define the semantics layer by layer inductivly on the naturals starting from layer 0.

$$Rec : n \in Nat \to w \in \bigcup_{i=0}^{n} Layer(i) \to [\![\Gamma,\ x\colon \tau_x \vdash \tau_r]\!]\,(\gamma,\ w)$$

$$Rec(n) \triangleq \begin{cases} F(0, \bot) & \text{if } n = 0 \\ F(n, Rec(n-1)) & \text{if } n > 0 \end{cases} \tag{13}$$

Where $\bot$ is the empty function from the empty set, and it is well defined from Theorem A.13.

Now, finally, we can define the semantics of the recursive binding as a lookup in the recursive function defined above:

$$\mathscr{F} : w \in [\![\Gamma \vdash \tau_x]\!]\,(\gamma) \to [\![\Gamma,\ x\colon \tau_x \vdash \tau_r]\!]\,(\gamma,\ w)$$
$$\mathscr{F}(w) \triangleq Rec([\![\Gamma \vdash e_m\colon \tau_x \to Nat]\!]\,(\gamma)\,(w))(w)$$

$$\left[\!\!\left[\frac{\begin{array}{c} \vdash \Gamma \qquad \Gamma \vdash \tau_x \qquad \Gamma,\ x\colon \tau_x \vdash \tau_r \\ \Gamma,\ x\colon \tau_x,\ f\colon (x\colon \{y\colon \tau_x \mid e_m\ x > e_m\ y\}) \to \tau_r \vdash e_1\colon \tau_r \\ \Gamma \vdash e_m\colon \tau_x \to Nat \qquad \Gamma,\ f\colon (x\colon \tau_x) \to \tau_r \vdash e_2\colon \tau \end{array}}{\Gamma \vdash \textbf{let}\ [e_m]\ f\ x\colon \tau_x \to \tau_r = e_1\ \textbf{in}\ e_2\colon \tau} \right]\!\!\right]$$
$$\triangleq [\![\Gamma,\ f\colon (x\colon \tau_x) \to \tau_r \vdash e_2\colon \tau]\!]\,(\gamma,\ \mathscr{F}) \in [\![\Gamma,\ f\colon (x\colon \tau_x) \to \tau_r \vdash \tau]\!]\,(\gamma,\ \mathscr{F}) \tag{14}$$

Which is well defined from Theorem A.14.

It is important to note that we disallow syntactically the possibility of refining the type of f, as it would be unsound. For example, assume that the type of f was refined to $\{f\colon (x\colon \tau_x) \to \tau_r \mid False\}$, then in the body inside e_1 as f is in it's context we can prove the refinement type of f itself. Now this can solved syntactically by disallowing refinements on the type of f like we did, but it could also be solved by stripping the refinement from f inside e_1 only. Since we have the T-Refl rule and subtyping we can recover any refinement on f at the use sites, making both approaches equivalent.

5 Positive families

Inductively defined types are pervasive in functional programming, type families or indexed types generalize inductive types by allowing types to be indexed by values. For example, the type of vectors can be defined as an indexed family over the natural numbers, where the index indicates the length of the vector.

We can extend our language type variables, type application, case expressions and type declarations for indexed families as shown in Figure 9. Type variables are used to define the recursive occurrences of a type in its type constructor declaration, type applications are used to instantiate a type with an index, and case expressions are used to define function by pattern matching on the constructors of the indexed family. Type declarations are used to define the indexed family itself.

$$
\begin{array}{llcll}
\textbf{Types} & \tau & ::= & \ldots & \\
& & | & T\ e & \textit{Type application} \\
& & | & \ldots & \\
\textbf{Expressions} & e & ::= & \ldots & \\
& & | & \textbf{case}\ x\ @\ e_1\ \textbf{of}\ \{\ \overline{C\ \overline{y}^{\,j} \ \rightarrow\ e_2}^{\,i}\ \} & \textit{Case expression} \\
& & | & \textbf{def}\ D\ \textbf{in}\ e & \textit{Type declaration} \\
& & | & \ldots & \\
\textbf{Type declarations} & D & ::= & \textbf{data}\ T\langle\tau_\iota\rangle\{\overline{C \colon \tau_C}^{\,i}\} & \textit{Indexed family} \\
\textbf{Contexts} & \Gamma & ::= & \ldots & \\
& & | & \Gamma,\ D & \textit{Type declaration} \\
& & | & \Gamma,\ T\langle\tau\rangle & \textit{Type variable} \\
& & | & \ldots &
\end{array}
$$

Fig. 9. Extensions for positive type families.

5.1 Semantics of type variables and type applications

Now given a well formed context Γ we can add a type variable variables in the context as long as the index type is well formed in the context and obtain a well-formed extended context.

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \tau}{\vdash \Gamma,\ T\langle\tau\rangle}\ \textsc{Ctx-TyVar}$$

And the semantics of a type variable is a function from the semantics of the index type to sets.

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma \vdash \tau}{\vdash \Gamma,\ T\langle\tau\rangle} \right]\!\!\right] \triangleq \gamma \in [\![\vdash \Gamma]\!] \times ([\![\Gamma \vdash \tau]\!]\,(\gamma) \rightarrow \mathit{Set}) \tag{15}$$

Which is well defined by Theorem A.15.

Now a type application is well formed as long as the type variable is in the context and the index expression has the correct type:

$$\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ T\langle\tau\rangle,\ \ldots \qquad \Gamma \vdash e \colon \tau}{\Gamma \vdash T\ e}\ \textsc{Wf-TyVar}$$

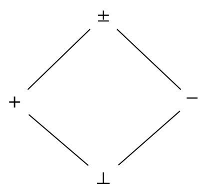
And the semantics is given by the application of the function in the context to the semantics of the index expression.

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ T\langle\tau\rangle,\ \ldots \qquad \Gamma \vdash e \colon \tau}{\Gamma \vdash T\ e} \right]\!\!\right] (\delta,\ t,\ \ldots) \triangleq t([\![\Gamma \vdash e \colon \tau]\!]\,(\delta,\ t,\ \ldots)) \tag{16}$$

And it is well defined by Theorem A.16.

5.2 Semantics of indexed family

To ensure logical consistency of the system we can't allow arbitrary inductive type declarations. We restrict to positive type families, where the recursive occurrences of the type being defined appear only in positive positions. To give a formal definition of positivity, we first define the polarity lattice and the operations on polarities and a procedure to compute the polarity of type occurrences in



(a) Polarity lattice.

$$\begin{aligned} {\sim}\pm &= \pm \\ {\sim}+ &= - \\ {\sim}- &= + \\ {\sim}\bot &= \bot \end{aligned}$$

(b) Polarity involution.

$$\begin{aligned} \delta(T, B) &= \bot \\ \delta(T, T\ e) &= + \\ \delta(T, U\ e) &= \bot \\ \delta(T, \{y{:}\ \tau \mid e\}) &= \delta(T, \tau) \\ \delta(T, (y{:}\ \tau_1) \to \tau_2) &= {\sim}\delta(T, \tau_1) \vee \delta(T, \tau_2) \end{aligned}$$

Where $B \in \{Bool, Unit, Nat\}$

(c) Polarity of type occurrences.

Fig. 10. Polarity operations.

Figure 10. Now the declaration of a type T is positive if the type of each constructor argument τ has positive polarity with respect to T *i.e.*, $\delta(T, \tau) < +$. Now we can define the well-formedness rule for positive type families:

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \tau_\iota \qquad \forall i, j.\ \delta(T, \tau_{i,j}) \leq + \qquad \forall i.\ \Gamma,\ T\langle\tau_\iota\rangle \vdash \overline{y{:}\ \tau}^{\,i,j} \to T\ e_\iota}{\Gamma \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{\overline{C{:}\ \overline{y{:}\ \tau}^{\,j} \to T\ e_\iota}^{\,i}\}}\ \textsc{Wf-Data}$$

Fixed $\gamma \in [\![\vdash \Gamma]\!]$ we pick as the semantics of inductive families the one of $[\![\Gamma \vdash \tau_\iota]\!]\,(\gamma) \to Set$. We can construct the functional F that applies every possible constructor to values in the semantics of the type being defined.

$$F : ([\![\Gamma \vdash \tau_\iota]\!]\,(\gamma) \to Set) \to ([\![\Gamma \vdash \tau_\iota]\!]\,(\gamma) \to Set)$$

$$F(X, w_\iota) = \left\{ (C_i,\ \overline{w}^{\,i,j}) \;\middle|\; \begin{aligned} w_{i,j} &\in [\![\Gamma,\ T\langle\tau_\iota\rangle, \overline{y{:}\ \tau}^{\,i,j-1} \vdash \tau]\!]\left(\gamma,\ X,\ \overline{w}^{\,i,j-1}\right) \\ w_\iota &= [\![\Gamma,\ T\langle\tau_\iota\rangle, \overline{y{:}\ \tau}^{\,i,j} \vdash e_\iota{:}\ \tau_\iota]\!]\left(\gamma,\ X,\ \overline{w}^{\,i,j}\right) \end{aligned} \right\} \tag{17}$$

And it is well defined from Theorem A.17.

Now we can show how the positivity condition is used to ensure that the semantics that we give to inductive types is well defined. We use this fact to show that F is a monotone operator on the complete lattice of functions from $[\![\Gamma \vdash \tau_\iota]\!]\,(\gamma)$ to Set. First we state the general lemma that relates polarity to (anti)monotonicity.

Lemma 5.1 ((Mono/Anti)-tonicity of polarities). *Let $\gamma \in [\![\Gamma]\!]$, and define*

$$G(X) = [\![\Gamma,\ T\langle\tau_\iota\rangle \vdash \tau]\!]\,(\gamma,\ X)\,.$$

Let $P \leq Q$, where the order is the one obtained by the pointwise lift of the lattice generated by set inclusion. Then:

- *If $\delta(T, \tau) \leq +$, then $G(P) \leq G(Q)$.*
- *If $\delta(T, \tau) \leq -$, then $G(P) \geq G(Q)$.*

And then we use it to show that the functional F is monotonic, which allows us to use the Knaster-Tarski theorem to show that the least fixpoint of F is well defined.

We can give the semantics of the inductive family:

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma \vdash \tau_\iota \qquad \forall i, j.\, \delta(T, \tau_{i,j}) \leq + \quad \forall i.\, \Gamma,\ T\langle \tau_\iota \rangle \vdash \overline{y\colon \tau}^{\,i,j} \to T\ e_\iota}{\Gamma \vdash \mathbf{data}\ T\langle \tau_\iota \rangle \{\overline{C\colon \overline{y\colon \tau}^{\,j} \to T\ e_\iota}^{\,i}\}}\right]\!\!\right] (\gamma) = \mathbf{lfp}\ F \tag{18}$$

Witch is well defined from Theorem A.18.

Now we can give the typing rule for how type declaration interact with contexts: A context can be extended with a type declaration as long as the type declaration is well formed in the context; a data type declaration can be introduced in the context, and if a data type declaration is in the context then the type constructor can be used in the typing of types in that context.

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\}}{\vdash \Gamma,\ \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\}}\ \textsc{Ctx-Data} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\} \quad \Gamma,\ \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\} \vdash e\colon \tau}{\Gamma \vdash \mathbf{def\ data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\}\ \mathbf{in}\ e\colon \tau}\ \textsc{T-Data}$$

$$\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ \mathbf{data}\ T\langle \tau \rangle \{\mathcal{C}\},\ \ldots \qquad \Gamma \vdash e\colon \tau}{\Gamma \vdash T\ e}\ \textsc{Wf-Data-Var}$$

The semantics of extending the context with a data type declaration is the same as the semantics of the declaration, since the declaration has only one possible interpretation in a single context instantiation it is a singleton, all the other rules are straightforward.

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma \vdash \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\}}{\vdash \Gamma,\ \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\}}\right]\!\!\right] \triangleq \gamma \in [\![\vdash \Gamma]\!] \times \{[\![\Gamma \vdash \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\}]\!]\,(\gamma)\} \tag{19}$$

$$\begin{aligned} &\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma \vdash \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\} \quad \Gamma,\ \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\} \vdash e\colon \tau}{\Gamma \vdash \mathbf{def\ data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\}\ \mathbf{in}\ e\colon \tau}\right]\!\!\right] (\gamma) \\ &\quad \triangleq [\![\Gamma,\ \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\} \vdash e\colon \tau]\!]\,(\gamma,\ [\![\Gamma \vdash \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\}]\!]\,(\gamma)) \\ &\quad\quad \in [\![\Gamma,\ \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\} \vdash \tau]\!]\,(\gamma,\ [\![\Gamma \vdash \mathbf{data}\ T\langle \tau_\iota \rangle \{\mathcal{C}\}]\!]\,(\gamma)) \end{aligned} \tag{20}$$

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ \mathbf{data}\ T\langle \tau \rangle \{\mathcal{C}\},\ \ldots \qquad \Gamma \vdash e\colon \tau}{\Gamma \vdash T\ e}\right]\!\!\right] (\delta,\ t,\ \ldots) \triangleq t([\![\Delta \vdash e\colon \tau]\!]\,(\delta,\ t,\ \ldots)) \tag{21}$$

All these definitions are well defined by Theorems A.20 to A.22.

A reference to a constructor is well-typed as long as a data type with that constructor is in the context.

$$\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ \mathbf{data}\ T\langle \tau_\iota \rangle \{\ldots,\ C\colon \overline{x\colon \tau}^{\,i} \to T\ e,\ \ldots\},\ \ldots}{\Gamma \vdash C\colon \overline{x\colon \tau}^{\,i} \to T\ e}\ \textsc{Ty-Data-Ctor}$$

And semantically we have that since the semantics is defined as the least fixpoint of the functional F then the application of the constructor C_i to its arguments $\overline{w}^{\,i}$ must be in the semantics of the type.

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ \textbf{data}\ T\langle \tau_\iota \rangle \{\ldots,\ C\colon \overline{x\colon \tau}^{\,i} \to T\ e,\ \ldots\},\ \ldots}{\Gamma \vdash C\colon \overline{x\colon \tau}^{\,i} \to T\ e} \right]\!\!\right] (\delta,\ t,\ \ldots)$$

$$\triangleq \overline{w \in [\![\Gamma,\ \overline{x\colon \tau}^{\,i-1} \vdash \tau_i]\!] \left(\delta,\ t,\ \ldots,\ \overline{w}^{\,i-1}\right)}^{\,i} \mapsto (C_i,\ \overline{w}^{\,i})$$
$$\in [\![\Gamma,\ \overline{x\colon \tau}^{\,i} \vdash T\ e]\!] \left(\delta,\ t,\ \ldots,\ \overline{w}^{\,i}\right) \qquad (22)$$

And it is well defined by Theorem A.23.

Now we can give the typing rule for case expressions, we require that each branch handles one constructor of the data type being defined, and in addition we bind a whiteness of the fact that the scrutinee is equal to the constructor applied to its arguments, this is used as a proxy for dependent pattern matching.

$$\frac{\begin{array}{c}\vdash \Gamma \qquad \Gamma \vdash e_1\colon T\ e_\iota \\ \Delta = \Gamma,\ \textbf{data}\ T\langle \tau_\iota \rangle \{\overline{C_i\colon \overline{y\colon \tau_y}^{\,j} \to T\ e_\iota}^{\,i}\},\ \ldots \\ \Gamma,\ \overline{y\colon \tau_y}^{\,j,i},\ x\colon \{y\colon \mathit{Unit} \mid e_1 = C_i\ \overline{y}^{\,j,i}\} \vdash e_{2,i}\colon \tau\end{array}}{\Gamma \vdash \textbf{case}\ x\ @\ e_1\ \textbf{of}\ \{\ \overline{C\ \overline{y}^{\,j}\ \to\ e_2}^{\,i}\ \}\colon \tau}\ \textsc{T-Case-Data}$$

The semantics is given by cases on the value of the scrutinee, we first compute the semantics of the scrutinee, which must be a constructor application from the semantics of the data type declaration, this is due to the fact that it is a fixpoint of F then by cases we use the semantics of the corresponding branch, passing the arguments of the constructor as well as a proof about the equality of the scrutinee and the constructor application, which is just $\star$ as witnessed by the semantics of the scrutinee.

$$\left[\!\!\left[\frac{\begin{array}{c}\vdash \Gamma \qquad \Gamma \vdash e_1\colon T\ e_\iota \\ \Delta = \Gamma,\ \textbf{data}\ T\langle \tau_\iota \rangle \{\overline{C_i\colon \overline{y\colon \tau_y}^{\,j} \to T\ e_\iota}^{\,i}\},\ \ldots \\ \Gamma,\ \overline{y\colon \tau_y}^{\,j,i},\ x\colon \{y\colon \mathit{Unit} \mid e_1 = C_i\ \overline{y}^{\,j,i}\} \vdash e_{2,i}\colon \tau\end{array}}{\Gamma \vdash \textbf{case}\ x\ @\ e_1\ \textbf{of}\ \{\ \overline{C\ \overline{y}^{\,j}\ \to\ e_2}^{\,i}\ \}\colon \tau} \right]\!\!\right] (\delta,\ t,\ \ldots)$$

$$\triangleq [\![\Gamma,\ \overline{y\colon \tau_y}^{\,i,j},\ x\colon \{y\colon \mathit{Unit} \mid e_1 = C_i\ \overline{y}^{\,i,j}\} \vdash e_{2,i}\colon \tau]\!] \left(\delta,\ t,\ \ldots,\ \overline{w}^{\,j,i},\ \star\right)$$
$$\text{where } (C_i, \overline{w}^{\,j,i}) = [\![\Gamma \vdash e_1\colon T\ e_\iota]\!]\ (\delta,\ t,\ \ldots) \qquad (23)$$

And it is well defined by Theorem A.24.

Lastly we give the subtyping rule for data types, which is just a reflection of the equality of the indices.

$$\frac{\vdash \Gamma \qquad \Gamma \vdash T\ e_1 \qquad \Gamma \vdash T\ e_2 \qquad \Gamma \models e_1 = e_2}{\Gamma \vdash T\ e_1 \preceq T\ e_2}\ \textsc{Sub-TyApp}$$

And it satisfies the semantic subtyping of Theorem 3.1.

5.3 Stratified families

The positivity condition is a sufficient but not necessary requirement for the semantics to be well defined. In the context of representing graphs of functions defined by large elimination in dependent type theory, we can observe that the condition imposed on recursive definitions to ensure termination, hence well-definedness, is quite different from positivity. Nevertheless, it suggests

$$
\begin{aligned}
stratified(\Gamma, e_\iota, e_m, \overline{x\colon \tau}^{\,i}) &= \bigwedge_i strat(\Gamma, e_\iota, e_m, \tau_i)\\
strat(\Gamma, e_\iota, e_m, B) &= True\\
strat(\Gamma, e_\iota, e_m, U\ e_\iota') &= True\\
strat(\Gamma, e_\iota, e_m, T\ e_\iota') &= \Gamma \models e_m\ e_\iota' < e_m\ e_\iota\\
strat(\Gamma, e_\iota, e_m, \{x\colon \tau \mid e_r\}) &= strat(\Gamma, e_\iota, e_m, \tau)\\
strat(\Gamma, e_\iota, e_m, (x\colon \tau_x) \to \tau) &= strat(\Gamma, e_\iota, e_m, \tau_x) \wedge strat(\Gamma,\ x\colon \tau_x, e_\iota, e_m, \tau)
\end{aligned}
$$

$$\text{Where } B \in \{Bool, Unit, Nat\}.$$

Fig. 11. Stratified condition

an alternative, principled way to define well-formedness for inductive types: using the index to ensure “termination” in the same sense as for functions. Semantically, this corresponds to defining the interpretation of datatypes layer by layer.

However, this approach introduces a complication: the direction of dependency is reversed compared to functions. In recursive function definitions, the parameter of recursive calls depends on the function argument; in types, instead, the index of recursive occurrences influences the index of the current constructor. To handle this, we introduce the notion of *stratified occurrences*, which checks, for each parameter of a constructor, that the index of recursive occurrences is smaller than the return type's index for all possible assignments, as shown in Figure 11.

The stratified condition is parameterized by e_m, which assigns a size to each index, analogous to the termination metrics used for recursive definitions.

$$
\frac{\begin{array}{c}\vdash \Gamma \qquad \Gamma \vdash \tau_\iota \qquad \Gamma \vdash e_m\colon \tau_\iota \to Nat\\ \forall i.\ stratified(\Gamma,\ \overline{y\colon \tau}^{\,j_i}, e_\iota, e_m, \overline{y\colon \tau}^{\,j_i})\\ \forall i.\ \Gamma,\ T\langle \tau_\iota \rangle \vdash \overline{y\colon \tau}^{\,j} \to T\ e_\iota\end{array}}{\Gamma \vdash \mathbf{strat}\ T\langle \tau_\iota\rangle[e_m]\{\overline{C\colon \overline{y\colon \tau}^{\,j} \to T\ e_\iota}^{\,i}\}}\ \textsc{Wf-Stratified}
$$

In the same fashion of recursive bindings e_m allows us to separate the index in $\mathbb{N}$ distinct families:

$$
Layer(n) \triangleq \left\{ w \;\middle|\; \begin{array}{c} w \in [\![\Gamma \vdash \tau_\iota]\!]\,(\gamma)\\ [\![\Gamma \vdash e_m\colon \tau_\iota \to Nat]\!]\,(\gamma)\,(w) = n \end{array} \right\} \tag{24}
$$

Which is well defined from Theorem A.25.

We can now define the semantics of an inductive type in a layer-by-layer fashion, where the operator G applies all possible data constructors of the type at each stage, and F simply takes the definition of the datatypes at the previous layer and adds the next layer.

$$
F : (n \in \mathbb{N}) \to (\bigcup_{i=0}^{n-1} Layer(i) \to Set) \to (\bigcup_{i=0}^{n} Layer(i) \to Set)
$$

$$
F(n, X, w_\iota) = \begin{cases} G(w_\iota) & [\![\Gamma \vdash e_m\colon \tau_\iota \to Nat]\!]\,(\gamma)\,(w_\iota) = n\\ X(w_\iota) & [\![\Gamma \vdash e_m\colon \tau_\iota \to Nat]\!]\,(\gamma)\,(w_\iota) < n \end{cases} \tag{25}
$$

$$
G : Layer(n) \to Set
$$

$$G(w_\iota) = \left\{ (C_i, \overline{w}^{\,i,j}) \;\middle|\; \begin{array}{l} w_{i,j} \in \llbracket \Gamma, T\langle \{x\colon \tau_\iota \mid e_m x < n\}\rangle, \overline{y\colon \tau}^{\,i,j-1} \vdash \tau \rrbracket \left(\gamma, X, \overline{w}^{\,i,j-1}\right) \\ w_\iota = \llbracket \Gamma, T\langle \{x\colon \tau_\iota \mid e_m x < n\}\rangle, \overline{y\colon \tau}^{\,i,j} \vdash e_\iota\colon \tau_\iota \rrbracket \left(\gamma, X, \overline{w}^{\,i,j}\right) \end{array} \right\}$$

And all of this is well defined from Theorem A.26.

And now we can define the semantics of the stratified declaration as the iteration of the functional F.

$$\begin{aligned} &Rec : n \in Nat \to w \in \bigcup_{i=0}^{n} Layer(i) \to Set \\ &Rec(n) \triangleq \begin{cases} F(0, \bot) & \text{if } n = 0 \\ F(n, Rec(n-1)) & \text{if } n > 0 \end{cases} \end{aligned} \tag{26}$$

Where $\bot$ is the empty function from the empty set, and we can show that is well defined from Theorem A.27.

And now we can give the semantics as:

$$\left\llbracket \frac{\begin{array}{c} \vdash \Gamma \qquad \Gamma \vdash \tau_\iota \qquad \Gamma \vdash e_m\colon \tau_\iota \to Nat \\ \forall i.\ stratified(\Gamma,\ \overline{y\colon \tau}^{\,j_i}, e_\iota, e_m, \overline{y\colon \tau}^{\,j_i}) \\ \forall i.\ \Gamma,\ T\langle \tau_\iota \rangle \vdash \overline{y\colon \tau}^{\,j} \to T\ e_\iota \end{array}}{\Gamma \vdash \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\overline{C\colon \overline{y\colon \tau}^{\,j} \to T\ e_\iota}^{\,i}\}} \right\rrbracket (\gamma)$$

$$= w_\iota \mapsto Rec(\llbracket \Gamma \vdash e_m\colon \tau_\iota \to Nat \rrbracket (\gamma)\ (w_\iota))(w_\iota)$$

From this the semantics of contexts and lookups, and type declarations are identical as positive families:

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\}}{\vdash \Gamma,\ \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\}} \textsc{Ctx-Strat} \qquad \frac{\begin{array}{c} \vdash \Gamma \qquad \Gamma \vdash \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\} \\ \Gamma,\ \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\} \vdash e\colon \tau \end{array}}{\Gamma \vdash \mathbf{def\ strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\}\ \mathbf{in}\ e\colon \tau} \textsc{T-Strat}$$

$$\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ \mathbf{strat}\ T\langle \tau \rangle [e_m] \{\mathcal{C}\},\ \ldots \qquad \Gamma \vdash e\colon \tau}{\Gamma \vdash T\ e} \textsc{Wf-Strat-Var}$$

$$\left\llbracket \frac{\vdash \Gamma \qquad \Gamma \vdash \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\}}{\vdash \Gamma,\ \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\}} \right\rrbracket$$

$$\triangleq \gamma \in \llbracket \vdash \Gamma \rrbracket \times \{\llbracket \Gamma \vdash \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\} \rrbracket (\gamma)\}$$

$$\left\llbracket \frac{\begin{array}{c} \vdash \Gamma \qquad \Gamma \vdash \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\} \\ \Gamma,\ \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\} \vdash e\colon \tau \end{array}}{\Gamma \vdash \mathbf{def\ strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\}\ \mathbf{in}\ e\colon \tau} \right\rrbracket (\gamma)$$

$$\begin{aligned} &\triangleq \llbracket \Gamma,\ \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\} \vdash e\colon \tau \rrbracket (\gamma,\ \llbracket \Gamma \vdash \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\} \rrbracket (\gamma)) \\ &\in \llbracket \Gamma,\ \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\} \vdash \tau \rrbracket (\gamma,\ \llbracket \Gamma \vdash \mathbf{strat}\ T\langle \tau_\iota \rangle [e_m] \{\mathcal{C}\} \rrbracket (\gamma)) \end{aligned}$$

AF: Here technically the well definedness of G follows from info that are not onyl stored in the types but also in the stratification condition so maybe is not super clear

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ \textbf{strat}\ T\langle \tau \rangle[e_m]\{\mathscr{c}\},\ \ldots \qquad \Gamma \vdash e\colon \tau}{\Gamma \vdash T\ e}\right]\!\!\right](\delta,\ t,\ \ldots)$$
$$\triangleq t(\llbracket \Delta \vdash e\colon \tau \rrbracket\, (\delta,\ t,\ \ldots))$$

And the well definedness argument is identical to the one for positive families.

Now, we can give the semantics of constructors:

$$\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ \textbf{strat}\ T\langle \tau_\iota \rangle[e_m]\{\ldots,\ C\colon \overline{x\colon \tau}^{\,i} \to T\ e,\ \ldots\},\ \ldots}{\Gamma \vdash C\colon \overline{x\colon \tau}^{\,i} \to T\ e}\ \textsc{Ty-Strat-Ctor}$$

And the semantics is given by:

$$\left[\!\!\left[\frac{\vdash \Gamma \qquad \Gamma = \Delta,\ \textbf{strat}\ T\langle \tau_\iota \rangle[e_m]\{\ldots,\ C\colon \overline{x\colon \tau}^{\,i} \to T\ e,\ \ldots\},\ \ldots}{\Gamma \vdash C\colon \overline{x\colon \tau}^{\,i} \to T\ e}\right]\!\!\right](\delta,\ t,\ \ldots)$$
$$\triangleq \overline{w \in \llbracket \Gamma,\ \overline{x\colon \tau}^{\,i-1} \vdash \tau_i \rrbracket \left(\delta,\ t,\ \ldots,\ \overline{w}^{\,i-1}\right)}^{\,i} \mapsto (C_i,\ \overline{w}^{\,i})$$
$$\in \llbracket \Gamma,\ \overline{x\colon \tau}^{\,i} \vdash T\ e \rrbracket \left(\delta,\ t,\ \ldots,\ \overline{w}^{\,i}\right) \tag{27}$$

And it is well defined by Theorem A.28.

Now we can give the typing rules and semantics of case expressions:

$$\frac{\begin{array}{c}\vdash \Gamma \qquad \Gamma \vdash e_1\colon T\ e_\iota \\ \Delta = \Gamma,\ \textbf{strat}\ T\langle \tau_\iota \rangle[e_m]\{\overline{C_i\colon \overline{y\colon \tau_y}^{\,j} \to T\ e_\iota}^{\,i}\},\ \ldots \\ \Gamma,\ \overline{y\colon \tau_y}^{\,j,i},\ x\colon \left\{y\colon Unit \,\middle|\, e_1 = C_i\ \overline{y}^{\,j,i}\right\} \vdash e_{2,i}\colon \tau\end{array}}{\Gamma \vdash \textbf{case}\ x\ @\ e_1\ \textbf{of}\ \{\ \overline{C\ \overline{y}^{\,j}\ \to\ e_2}^{\,i}\ \}\colon \tau}\ \textsc{T-Case-Strat}$$

$$\left[\!\!\left[\frac{\begin{array}{c}\vdash \Gamma \qquad \Gamma \vdash e_1\colon T\ e_\iota \\ \Delta = \Gamma,\ \textbf{strat}\ T\langle \tau_\iota \rangle[e_m]\{\overline{C_i\colon \overline{y\colon \tau_y}^{\,j} \to T\ e_\iota}^{\,i}\},\ \ldots \\ \Gamma,\ \overline{y\colon \tau_y}^{\,j,i},\ x\colon \left\{y\colon Unit \,\middle|\, e_1 = C_i\ \overline{y}^{\,j,i}\right\} \vdash e_{2,i}\colon \tau\end{array}}{\Gamma \vdash \textbf{case}\ x\ @\ e_1\ \textbf{of}\ \{\ \overline{C\ \overline{y}^{\,j}\ \to\ e_2}^{\,i}\ \}\colon \tau}\right]\!\!\right](\delta,\ t,\ \ldots)$$
$$\triangleq \left\llbracket \Gamma,\ \overline{y\colon \tau_y}^{\,i,j},\ x\colon \left\{y\colon Unit \,\middle|\, e_1 = C_i\ \overline{y}^{\,i,j}\right\} \vdash e_{2,i}\colon \tau \right\rrbracket \left(\delta,\ t,\ \ldots,\ \overline{w}^{\,j,i},\ \star\right)$$
$$\text{where } (C_i, \overline{w}^{\,j,i}) = \llbracket \Gamma \vdash e_1\colon T\ e_\iota \rrbracket\, (\delta,\ t,\ \ldots) \tag{28}$$

And it is well defined by Theorem A.29.

6 Elaboration

In Section 3 we defined the defined indexed inductive and stratified types and gave them different syntaxes and typing rules, this has been done to make ease the construction of the semantic model and presentation easier. In practice however, it is more convenient from the implementation point of view to encode both as simple inductive types. In this section we present an elaboration from the surface language with indexed inductive and stratified types to a core language with only simple inductive types.

6.1 Elaboration of Indexed types

The elaboration of indexed inductive types follows the same approach teased in Section 1 and used by Borkowski [2] and [3].

The idea boils down to introducing an uninterpreted symbol, say prop, to represent a function that maps values of the type to their index, and then refine the constructors of such types to axiomaitze the behavior of this function on the constructors. As an example, consider the indexed inductive type of sized integer vectors:

```
data Vec <Nat> where
  VNil  :: Vec 0
  VCons :: Int -> n:Nat -> Vec n -> Vec (n + 1)
```

Will be elaborated to the following simple inductive type:

```
data Vec where
  VNil  :: {v:Vec | prop v = 0}
  VCons :: Int -> n:Nat -> {v:Vec | prop v = n}
        -> {v:Vec | prop v = n + 1}
```

It is important to note that the uninterpreted symbol prop is not given any definition, but rather axiomatized through the refinement types in the constructors. This way, we can still reason about the indexes of the values of the type, without having to define a function that computes them, in particular defining prop explicitly as a function from values to maybe indexes it is not possible, as we can express properties that are not computable, take for example the type of infinitely branching trees indexed by their depth.

```
data Tree <Nat> where
  Node :: n:Nat -> (Nat -> Tree n) -> Tree (n + 1)
  Leaf :: Tree 0
```

Clearly we check that the trees have the right indexes, because it would require us to check that the function passed to the constructor Node returns a valid tree for every possible input.

This definition of prop as an uninterpreted symbol also validates the subtyping rule Sub-TyApp as $\{x\colon T \mid e_1 = prop\ x\}$ is a subtype of $\{x\colon T \mid e_2 = prop\ x\}$ only if we can prove that e_1 and e_2 are equal as *prop* is a function. The positivity condition required for wellformedness instead is already implied by the one for standard datatypes.

6.2 Elaboration of Stratified types

Stratified types can be elaborated in a similar way, as they are indexed in the same way as indexed inductive types. Hence the type of well typed STLC values:

```
strat Val <Ty> where
  VInt :: Int -> Val TyInt
  VAbs :: ty1:Ty -> ty2:Ty -> (Val ty1 -> Val ty2)
       -> Val (TyArr ty1 ty2)
```

Will be elaborated to:

```
data Val where
  VInt :: Int -> {v:Val | prop v = TyInt}
  VAbs :: ty1:Ty -> ty2:Ty
       -> ({v:Val | prop v = ty1} -> {v:Val | prop v = ty2})
       -> {v:Val | prop v = TyArr ty1 ty2}
```

The only difference from the previous elaboration is that the welldefinedness condition for stratified types requires is not the same as the one for standard data types, hence for stratified types we need to disable the positivity check and implement an ad-hoc stratification check, but the rest remains the same.

7 Discussion

7.1 Mixing positivity and stratification

Since the positivity condition disallows negative occurrences of inductive types in their own definitions, it is natural to wonder whether the stratification condition can be restricted to only negative occurrences as well, making stratified types a generalization of indexed inductive types, sadly, this is not the case as the two conditions used to give the semantics to these types are incompatible with each other. This is exemplified by the following counterexample:

```
data Curry <Nat> where
  Lower :: n:Nat -> Curry (n + 1) -> Curry n
  Curry :: n:Nat -> (Curry n -> Void) -> Curry (n + 1)

bad :: n:Nat -> Curry n -> Void
bad n (Lower _ m) = bad (n + 1) m
bad n (Curry _ f) = f (Lower (n - 1) (Curry (n - 1) f))

verybad :: Curry 0
verybad = Lower 0 (Curry 0 (bad 0))

terrible :: Void
terrible = bad 0 verybad
```

We can rephrase the well-known Curry paradox in our setting using the constructor `Lower` to conceal a non-decreasing self-reference in negative position of the constructor `Curry`.

This also highlight that it is not trivial to define the conditions in which we can define mutually recursive standard indexed inductive types and stratified types, as it can be captured in the same manner as mutual indexed inductive types *i.e.* by elaborating them to a single inductive type with an extra index distinguishing the different types.

We suspect by the connection between stratified types and large elimination in dependent type theory that the conditions for mutual definitions should be similar to the one for inductive-recursive definitions.

8 Related Work

8.1 Stratified types in Beluga

The idea of stratified types was first explored in [5], although their system differs substantially from ours. In their approach, non-stratified inductive types are allowed without any (strict-)positivity restriction, but as a consequence, they can only be manipulated through Mendler-style recursion schemes. This separation between stratified and ordinary inductive types makes the approach difficult to reconcile with Haskell-style programming, where uniform pattern matching and recursion are expected. In our system, by contrast, stratified and ordinary datatypes are treated uniformly: they elaborate to the same core representation and differ only in their declaration form.

Their system also starts from an already indexed language, whereas in our setting indices are used only in the metatheoretical development: all definitions are expressed directly in terms of refinement types. This design opens the door to extending existing programming languages with theorem-proving capabilities, since our system requires no changes to the core implementation or proof automation of refinement type systems. In Beluga, the index language is the logical framework itself, while in our case it coincides with the host language itself, Haskell in our implementation.

8.2 Dependent type theory

The standard set-theoretic model can be extended to show the relative consistency of dependent type theories (see, for example, [13]). Even when restricted to the predicative fragment, these systems differ from ours in two key respects. First, dependently typed theories require a stratification of universes, which in the set-theoretic model corresponds to a hierarchy of inaccessible cardinals. This is reflected in the model construction: if we had $Type : Type$, it would be interpreted as $[\![Type]\!] \in [\![Type]\!]$. Second, inductive families in dependent type theory require *strict* positivity to ensure consistency, whereas our system only requires positivity or stratification.

The exact strength of our system is still under investigation. It remains unclear which classes of properties may be inexpressible, or whether all constructions from predicative dependent type theories can be reformulated, maybe less cleanly, in our framework. Intuitively, our system appears more predicative than standard predicative dependent type theories, though a full comparison is still ongoing. The situation becomes more subtle once polymorphism is reintroduced: it could potentially recover some of the expressive strength that may be lost in the predicative fragment, but it is not yet clear what restrictions, if any, would be required to ensure the consistency of the resulting system.

References

[1] Agda Developers. 2024. Agda. https://agda.readthedocs.io/. Version 2.9.0.
[2] Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. 2024. Mechanizing Refinement Types. 8 (2024), 70:2099–70:2128. Issue POPL. doi:10.1145/3632912
[3] A. Ferrarini, N. Vazou, and Swierstra W. 2025. PLEX: Normalization for refinemenet types. (2025).
[4] Catarina Gamboa, Paulo Canelas, Christopher Timperley, and Alcides Fonseca. 2023. Usability-Oriented Design of Liquid Types for Java. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia, 2023-07-26) *(ICSE '23)*. IEEE Press, 1520–1532. doi:10.1109/ICSE48619.2023.00132
[5] Rohan Jacob-Rao, Brigitte Pientka, and David Thibodeau. 2018. Index-Stratified Types. 108 (2018), 19:1–19:17. doi:10.4230/LIPICS.FSCD.2018.19
[6] Ranjit Jhala and Niki Vazou. 2020. *Refinement Types: A Tutorial.* arXiv:2010.07763 [cs] doi:10.48550/arXiv.2010.07763
[7] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. 7 (2023), 169:1533–169:1557. Issue PLDI. doi:10.1145/3591283
[8] J. Parker, N. Vazou, and M. Hicks. 2019. LWeb: Information Flow Security for Multi-tier Web Applications. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* (2019). doi:10.1145/3290388
[9] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. 43, 6 (2008), 159–169. doi:10.1145/1379022.1375602
[10] Wouter Swierstra. 2023. A Correct-by-Construction Conversion from Lambda Calculus to Combinatory Logic. 33 (2023), e11. doi:10.1017/S0956796823000084
[11] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. 49, 9 (2014), 269–282. doi:10.1145/2692915.2628161
[12] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. 2 (2017), 53:1–53:31. Issue POPL. doi:10.1145/3158141
[13] Benjamin Werner. 1997. Sets in Types, Types in Sets. In *Theoretical Aspects of Computer Software* (Berlin, Heidelberg, 1997), Martín Abadi and Takayasu Ito (Eds.). Springer, 530–546. doi:10.1007/BFb0014566
[14] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1998-05-01) *(PLDI '98)*. Association for Computing Machinery, 249–257. doi:10.1145/277650.277732

A Well definedness of the semantics

Lemma A.1 (Equation (1) is well-defined). *We know that*

$$\gamma \in [\![\vdash \Gamma]\!] \,.$$

By the definition of $[\![\Gamma \vdash \tau_x]\!]$,

$$[\![\Gamma \vdash \tau_x]\!] : [\![\vdash \Gamma]\!] \to \mathit{Set},$$

so taking $w \in [\![\Gamma \vdash \tau_x]\!]\,(\gamma)$ *we obtain*

$$(\gamma, w) \in [\![\vdash \Gamma]\!] \times [\![\Gamma \vdash \tau_x]\!]\,(\gamma) = [\![\vdash \Gamma,\ x\colon \tau_x]\!] \,.$$

Therefore the codomain of our function is

$$[\![\Gamma,\ x\colon \tau_x \vdash \tau]\!] : [\![\vdash \Gamma,\ x\colon \tau_x]\!] \to \mathit{Set},$$

and in particular

$$[\![\Gamma,\ x\colon \tau_x \vdash \tau]\!]\,(\gamma,\ w) \in \mathit{Set}.$$

Hence the dependent function type semantics above is well defined.

Lemma A.2 (Equation (2) is well-defined). *First,*

$$\gamma \in [\![\vdash \Gamma]\!] \,.$$

By the definition of $[\![\Gamma \vdash \tau]\!]$ *we have*

$$[\![\Gamma \vdash \tau]\!] : [\![\vdash \Gamma]\!] \to \mathit{Set},$$

so taking $w \in [\![\Gamma \vdash \tau]\!]\,(\gamma)$ *gives*

$$(\gamma, w) \in [\![\vdash \Gamma]\!] \times [\![\Gamma \vdash \tau]\!]\,(\gamma) = [\![\vdash \Gamma,\ x\colon \tau]\!] \,.$$

Next, by the definition of the semantic typing judgment for expressions and types,

$$[\![\Gamma,\ x\colon \tau \vdash \mathit{Bool}]\!] : [\![\vdash \Gamma,\ x\colon \tau]\!] \to \mathit{Set}$$
$$[\![\Gamma,\ x\colon \tau \vdash r\colon \mathit{Bool}]\!] : \gamma \in [\![\vdash \Gamma,\ x\colon \tau]\!] \to [\![\Gamma,\ x\colon \tau \vdash \mathit{Bool}]\!]\,(\gamma)\,,$$

so in particular

$$[\![\Gamma,\ x\colon \tau \vdash r\colon \mathit{Bool}]\!]\,(\gamma,\ w) \in [\![\Gamma,\ x\colon \tau \vdash \mathit{Bool}]\!]\,(\gamma,\ w) = \mathbb{2} \ni \mathit{tt}.$$

Lemma A.3 (Equation (3) is well-defined). *First,*

$$\gamma \in [\![\vdash \Gamma]\!]\,,$$
$$\left[\!\left[\Gamma \vdash e_g\colon \mathit{Bool}\right]\!\right] : \gamma \in [\![\vdash \Gamma]\!] \to [\![\Gamma \vdash \mathit{Bool}]\!]\,(\gamma)$$

so in particular

$$\left[\!\left[\Gamma \vdash e_g\colon \mathit{Bool}\right]\!\right]\,(\gamma) \in [\![\Gamma \vdash \mathit{Bool}]\!]\,(\gamma) = \mathbb{2} = \{\mathit{tt},\ \mathit{ff}\}$$

Thus the two cases above are exhaustive. Next, each branch has the right signature:

$$[\![\Gamma \vdash e_t\colon \tau]\!] : \gamma \in [\![\vdash \Gamma]\!] \to [\![\Gamma \vdash \tau]\!]\,(\gamma)\,,$$
$$[\![\Gamma \vdash e_e\colon \tau]\!] : \gamma \in [\![\vdash \Gamma]\!] \to [\![\Gamma \vdash \tau]\!]\,(\gamma)\,,$$

and therefore

$$[\![\Gamma \vdash e_t\colon \tau]\!]\,(\gamma) \in [\![\Gamma \vdash \tau]\!]\,(\gamma)\,,$$
$$[\![\Gamma \vdash e_e\colon \tau]\!]\,(\gamma) \in [\![\Gamma \vdash \tau]\!]\,(\gamma)\,.$$

Lemma A.4 (Equation (4) is well-defined). *First,*

$$\begin{aligned}&\gamma \in [\![\vdash \Gamma]\!]\,,\\ &[\![\Gamma \vdash e_l\colon Nat]\!] : \gamma \in [\![\vdash \Gamma]\!] \longrightarrow [\![\Gamma \vdash Nat]\!]\,(\gamma)\,,\\ &[\![\Gamma \vdash e_r\colon Nat]\!] : \gamma \in [\![\vdash \Gamma]\!] \longrightarrow [\![\Gamma \vdash Nat]\!]\,(\gamma)\,.\end{aligned}$$

So in particular

$$\begin{aligned}&[\![\Gamma \vdash e_l\colon Nat]\!]\,(\gamma) \in [\![\Gamma \vdash Nat]\!]\,(\gamma) = \mathbb{N},\\ &[\![\Gamma \vdash e_r\colon Nat]\!]\,(\gamma) \in [\![\Gamma \vdash Nat]\!]\,(\gamma) = \mathbb{N}.\end{aligned}$$

Thus the two cases are exhaustive (either the left value is strictly less than the right, or it is greater or equal). Finally, each branch is in the correct codomain:

$$[\![\Gamma \vdash Bool]\!]\,(\gamma) = \mathbb{2} = \{tt,\ ff\}$$

Lemma A.5 (Equation (5) is well-defined). *First,*

$$\begin{aligned}&\gamma \in [\![\vdash \Gamma]\!]\,,\\ &[\![\Gamma \vdash e_l\colon \tau]\!] : \gamma \in [\![\vdash \Gamma]\!] \longrightarrow [\![\Gamma \vdash \tau]\!]\,(\gamma)\,,\\ &[\![\Gamma \vdash e_r\colon \tau]\!] : \gamma \in [\![\vdash \Gamma]\!] \longrightarrow [\![\Gamma \vdash \tau]\!]\,(\gamma)\,.\end{aligned}$$

So in particular

$$\begin{aligned}&[\![\Gamma \vdash e_l\colon \tau]\!]\,(\gamma) \in [\![\Gamma \vdash \tau]\!]\,(\gamma)\,,\\ &[\![\Gamma \vdash e_r\colon \tau]\!]\,(\gamma) \in [\![\Gamma \vdash \tau]\!]\,(\gamma)\,.\end{aligned}$$

Thus the two cases above are exhaustive (either the two values are equal or they are not). Finally, each branch is in the correct codomain:

$$[\![\Gamma \vdash Bool]\!]\,(\gamma) = \mathbb{2} = \{tt,\ ff\}$$

Lemma A.6 (Equation (6) is well-defined). *Indeed, by assumption*

$$\begin{aligned}(\delta, w, \dots) \in [\![\Gamma]\!] &= [\![\Delta,\ x\colon \tau,\ \dots]\!]\\ &= \delta \in [\![\Delta]\!] \times [\![\Delta \vdash \tau]\!]\,(\delta) \times \dots,\end{aligned}$$

so in particular

$$\delta \in [\![\Delta]\!]\,, \qquad w \in [\![\Delta \vdash \tau]\!]\,(\delta)\,.$$

By Theorems D.1 and D.3 we have

$$[\![\Delta \vdash \tau]\!]\,(\delta) = [\![\Gamma \vdash \tau]\!]\,(\delta,\ x\colon \tau,\ \dots)\,,$$

hence

$$w \in [\![\Gamma \vdash \tau]\!]\,(\delta,\ x\colon \tau,\ \dots)$$

as required.

Lemma A.7 (Equation (7) is well-defined). *First, by assumption*

$$\begin{aligned}&\gamma \in [\![\vdash \Gamma]\!]\,,\\ &[\![\Gamma \vdash e_1\colon (x\colon \tau_x) \longrightarrow \tau]\!] : \gamma \in [\![\vdash \Gamma]\!] \longrightarrow [\![\Gamma \vdash (x\colon \tau_x) \longrightarrow \tau]\!]\,(\gamma)\,,\\ &[\![\Gamma \vdash e_2\colon \tau_x]\!] : \gamma \in [\![\vdash \Gamma]\!] \longrightarrow [\![\Gamma \vdash \tau_x]\!]\,(\gamma)\,.\end{aligned}$$

So in particular,

$$\llbracket \Gamma \vdash e_1 \colon (x\colon \tau_x) \to \tau \rrbracket\,(\gamma) \in \llbracket \Gamma \vdash \tau_x \rrbracket\,(\gamma) \to \llbracket \Gamma,\ x\colon \tau_x \vdash \tau \rrbracket\,(\gamma,\ w)\,,$$
$$\llbracket \Gamma \vdash e_2 \colon \tau_x \rrbracket\,(\gamma) \in \llbracket \Gamma \vdash \tau_x \rrbracket\,(\gamma)\,,$$

so the application

$$\big(\llbracket \Gamma \vdash e_1 \colon (x\colon \tau_x) \to \tau \rrbracket\,(\gamma)\ \ \llbracket \Gamma \vdash e_2 \colon \tau_x \rrbracket\,(\gamma)\big) \in \llbracket \Gamma,\ x\colon \tau_x \vdash \tau \rrbracket\,(\gamma,\ \llbracket \Gamma \vdash e_2 \colon \tau_x \rrbracket\,(\gamma))\,.$$

Finally, by Theorems D.5 and D.7,

$$\llbracket \Gamma,\ x\colon \tau_x \vdash \tau \rrbracket\,(\gamma,\ \llbracket \Gamma \vdash e_2 \colon \tau_x \rrbracket\,(\gamma)) = \llbracket \Gamma \vdash \tau[x/e_2] \rrbracket\,(\gamma)\,.$$

Lemma A.8 (Equation (8) is well-defined). *First, by assumption*

$$\gamma \in \llbracket \vdash \Gamma \rrbracket\,,$$
$$w \in \llbracket \Gamma \vdash \tau_x \rrbracket\,(\gamma)\,,$$
$$\llbracket \Gamma,\ x\colon \tau_x \vdash e\colon \tau \rrbracket : \gamma \in \llbracket \vdash \Gamma,\ x\colon \tau_x \rrbracket \to \llbracket \Gamma,\ x\colon \tau_x \vdash \tau \rrbracket\,(\gamma)\,.$$

So in particular,

$$(\gamma,\ w) \in (\gamma \in \llbracket \vdash \Gamma \rrbracket \times \llbracket \Gamma \vdash \tau_x \rrbracket\,(\gamma)) = \llbracket \vdash \Gamma,\ x\colon \tau_x \rrbracket\,,$$
$$\llbracket \Gamma,\ x\colon \tau_x \vdash e\colon \tau \rrbracket\,(\gamma,\ w) \in \llbracket \Gamma,\ x\colon \tau_x \vdash \tau \rrbracket\,(\gamma,\ w)\,.$$

Hence we conclude

$$\big(w \in \llbracket \Gamma \vdash \tau_x \rrbracket\,(\gamma) \mapsto \llbracket \Gamma,\ x\colon \tau_x \vdash e\colon \tau \rrbracket\,(\gamma,\ w)\big) \in \big(w \in \llbracket \Gamma \vdash \tau_x \rrbracket\,(\gamma) \to \llbracket \Gamma,\ x\colon \tau_x \vdash \tau \rrbracket\,(\gamma,\ w)\big) = \llbracket \Gamma \vdash (x\colon \tau_x) \to \tau \rrbracket\,(\gamma)\,.$$

Lemma A.9 (Equation (9) is well-defined). *First, by assumption*

$$\gamma \in \llbracket \vdash \Gamma \rrbracket\,,$$
$$\llbracket \Gamma \vdash e\colon \tau \rrbracket : \gamma \in \llbracket \vdash \Gamma \rrbracket \to \llbracket \Gamma \vdash \tau \rrbracket\,(\gamma)\,,$$
$$\llbracket \Gamma \vdash \{x\colon \tau \mid x = e\} \rrbracket : \llbracket \vdash \Gamma \rrbracket \to \mathit{Set}.$$

So in particular,

$$\llbracket \Gamma \vdash e\colon \tau \rrbracket\,(\gamma) \in \llbracket \Gamma \vdash \tau \rrbracket\,(\gamma)\,,$$
$$\llbracket \Gamma \vdash \{x\colon \tau \mid x = e\} \rrbracket\,(\gamma) = \left\{ w \,\middle|\, \begin{array}{c} w \in \llbracket \Gamma \vdash e\colon \tau \rrbracket\,(\gamma) \\ \llbracket \Gamma,\ x\colon \tau \vdash x = e\colon \mathit{Bool} \rrbracket\,(\gamma,\ w) = \mathit{tt} \end{array} \right\}.$$

Finally, we have

- $\llbracket \Gamma \vdash e\colon \tau \rrbracket\,(\gamma) \in \llbracket \Gamma \vdash \tau \rrbracket\,(\gamma)$ *by assumption.*
- $\begin{array}{l} \llbracket \Gamma,\ x\colon \tau \vdash x = e\colon \mathit{Bool} \rrbracket\,(\gamma,\ \llbracket \Gamma \vdash e\colon \tau \rrbracket\,(\gamma)) \\ \quad = \big(\llbracket \Gamma \vdash e\colon \tau \rrbracket\,(\gamma) = \llbracket \Gamma \vdash e\colon \tau \rrbracket\,(\gamma)\big) = \mathit{tt} \end{array}$ *by definition.*

Lemma A.10 (Equation (10) is well-defined). *First, by assumption:*

$$\gamma \in \llbracket \vdash \Gamma \rrbracket\,,$$
$$\llbracket \Gamma \vdash e\colon \tau_2 \rrbracket : \gamma \in \llbracket \vdash \Gamma \rrbracket \to \llbracket \Gamma \vdash \tau_2 \rrbracket\,(\gamma)\,,$$
$$\llbracket \Gamma \vdash \tau_1 \rrbracket : \llbracket \vdash \Gamma \rrbracket \to \mathit{Set}.$$

So in particular:

$$\llbracket \Gamma \vdash e\colon \tau_2 \rrbracket\,(\gamma) \in \llbracket \Gamma \vdash \tau_2 \rrbracket\,(\gamma)\,.$$

Finally, from the subtyping assumption $\Gamma \vdash \tau_2 \preceq \tau_1$ *and Theorem 3.1, we have:*

$$[\![\Gamma \vdash \tau_2]\!]\,(\gamma) \subseteq [\![\Gamma \vdash \tau_1]\!]\,(\gamma)\,.$$

Hence:

$$[\![\Gamma \vdash e\colon \tau_2]\!]\,(\gamma) \in [\![\Gamma \vdash \tau_1]\!]\,(\gamma)\,.$$

Lemma A.11 (Equation (11) is well-defined). *First,*

$$[\![\Gamma \vdash \tau_x]\!] : [\![\vdash \Gamma]\!] \to Set,$$
$$[\![\Gamma \vdash e_m\colon \tau_x \to Nat]\!] : \gamma \in [\![\vdash \Gamma]\!] \to [\![\Gamma \vdash \tau_x \to Nat]\!]\,(\gamma)\,,$$

In particular,

$$[\![\Gamma \vdash e_m\colon \tau_x \to Nat]\!]\,(\gamma) \in [\![\Gamma \vdash \tau_x]\!]\,(\gamma) \to [\![\Gamma \vdash Nat]\!]\,(\gamma) = \mathbb{N}.$$

Lemma A.12. *[Equation (12) is well-defined] First,*

$$[\![\Gamma,\ x\colon \tau_x,\ f\colon (x\colon \{y\colon \tau_x \mid e_m\ x > e_m\ y\}) \to \tau_r \vdash e_r\colon \tau_r]\!]$$
$$: \gamma \in [\![\vdash \Gamma,\ x\colon \tau_x,\ f\colon \ldots]\!] \to [\![\Gamma,\ x\colon \tau_x,\ f\colon \ldots \vdash \tau_r]\!]\,(\gamma)$$

In particular,

$$[\![\vdash \Gamma,\ x\colon \tau_x,\ f\colon (x\colon \{y\colon \tau_x \mid e_m\ x > e_m\ y\}) \to \tau_r]\!] = \gamma^* \in [\![\vdash \Gamma]\!]$$
$$\times\ w^* \in [\![\Gamma \vdash \tau_x]\!]\,(\gamma^*) \times [\![\Gamma,\ x\colon \tau_x \vdash (x\colon \{y\colon \tau_x \mid e_m\ x > e_m\ y\}) \to \tau_r]\!]\,(\gamma^*,\ w^*)$$

And since $\gamma \in [\![\vdash \Gamma]\!]$ *by assumption and* $w \in Layer(n)$ *implies that* $w \in [\![\Gamma \vdash \tau_x]\!]\,(\gamma)$*, hence:*

$$\begin{aligned}
&[\![\Gamma,\ x\colon \tau_x \vdash (x\colon \{y\colon \tau_x \mid e_m\ x > e_m\ y\}) \to \tau_r]\!]\,(\gamma,\ w)\\
&= v \in [\![\Gamma,\ x\colon \tau_x \vdash \{y\colon \tau_x \mid e_m\ x > e_m\ y\}]\!]\,(\gamma,\ w)\\
&\quad \to [\![\Gamma,\ x\colon \tau_x,\ y\colon \{y\colon \tau_x \mid e_m\ x > e_m\ y\} \vdash \tau_r]\!]\,(\gamma,\ w,\ v)\\
&\quad \textit{Since } [\![\Gamma,\ x\colon \tau_x \vdash e_m x\colon Nat]\!]\,(\gamma, w) = n \textit{ and by Theorems D.6 and D.8}\\
&= v \in [\![\Gamma,\ x\colon \tau_x \vdash \{y\colon \tau_x \mid n > e_m\ y\}]\!]\,(\gamma,\ w)\\
&\quad \to [\![\Gamma,\ x\colon \tau_x,\ y\colon \{y\colon \tau_x \mid e_m\ x > e_m\ y\} \vdash \tau_r]\!]\,(\gamma,\ w,\ v)\\
&= v \in \bigcup_{i=0}^{n} Layer(i) \to [\![\Gamma,\ x\colon \tau_x,\ y\colon \{y\colon \tau_x \mid e_m\ x > e_m\ y\} \vdash \tau_r]\!]\,(\gamma,\ w,\ v)\\
&\quad \textit{By Theorem D.3}\\
&= v \in \bigcup_{i=0}^{n} Layer(i) \to [\![\Gamma,\ x\colon \tau_x \vdash \tau_r]\!]\,(\gamma,\ w)\\
&\ni X
\end{aligned}$$

Thus

$$\begin{aligned}
[\![\Gamma,\ x\colon \tau_x,\ f\colon \ldots \vdash e_1\colon \tau_r]\!]\,(\gamma,\ w,\ X) &\in [\![\Gamma,\ x\colon \tau_x,\ f\colon \ldots \vdash \tau_r]\!]\,(\gamma,\ w,\ X)\\
&\quad \textit{By Theorem D.3}\\
&= [\![\Gamma,\ x\colon \tau_x \vdash \tau_r]\!]\,(\gamma,\ w)
\end{aligned}$$

Lemma A.13 (Equation (13) is well-defined). *We can show that is well defined by induction on* n*:*

- $n = 0$*: then* $\bigcup_{i=0}^{0-1} Layer(i) = \emptyset$*, hence* $\bot \in (w \in \emptyset \to [\![\Gamma,\ x\colon \tau_x \vdash \tau_r]\!]\,(\gamma,\ w)$ *vacuously and thus* $F(0, \bot) : w \in \bigcup_{i=0}^{0} Layer(i) \to [\![\Gamma,\ x\colon \tau_x \vdash \tau_r]\!]\,(\gamma,\ w)$

- $n > 0$: then by inductive hypothesis $Rec(n-1) : w \in \bigcup_{i=0}^{n-1} Layer(i) \to [\![\Gamma,\ x\colon \tau_x \vdash \tau_r]\!]\,(\gamma,\ w)$, hence $F(n, Rec(n-1)) : w \in \bigcup_{i=0}^{n} Layer(i) \to$ $[\![\Gamma,\ x\colon \tau_x \vdash \tau_r]\!]\,(\gamma,\ w)$.

Lemma A.14 (Equation (14) is well-defined). *First, $\mathscr{F}$ is well defined because: By definition of Layer,*

$$w \in Layer\big([\![\Gamma \vdash e_m\colon \tau_x \to Nat]\!]\,(\gamma)\,(w)\big),$$

hence

$$Rec\big([\![\Gamma \vdash e_m\colon \tau_x \to Nat]\!]\,(\gamma)\,(w)\big)(w) \in [\![\Gamma,\ x\colon \tau_x \vdash \tau_r]\!]\,(\gamma,\ w)\,.$$

And since

$$[\![\Gamma,\ f\colon (x\colon \tau_x) \to \tau_r \vdash e_2\colon \tau]\!] : (\gamma \in [\![\vdash \Gamma,\ f\colon \ldots]\!]) \to [\![\Gamma,\ f\colon \ldots \vdash \tau]\!]\,(\gamma)\,,$$

we obtain

$$[\![\Gamma,\ f\colon \ldots \vdash e_2\colon \tau]\!]\,(\gamma,\ \mathscr{F}) \in [\![\Gamma,\ f\colon \ldots \vdash \tau]\!]\,(\gamma,\ \mathscr{F})\,.$$

Lemma A.15 (Equation (15) is well-defined). *By assumption $\gamma \in [\![\vdash \Gamma]\!]$ and*

$$[\![\Gamma \vdash \tau]\!] : [\![\vdash \Gamma]\!] \to Set,$$

hence $[\![\Gamma \vdash \tau]\!]\,(\gamma)$ is a set.

Lemma A.16 (Equation (16) is well-defined). *Indeed, by assumption*

$$\begin{aligned}(\delta, t, \ldots) \in [\![\Gamma]\!] &= [\![\Delta,\ T\langle\tau\rangle,\ \ldots]\!]\\ &= \delta \in [\![\Delta]\!] \times ([\![\Delta \vdash \tau]\!]\,(\delta) \to Set) \times \ldots,\end{aligned}$$

and

$$[\![\Gamma \vdash e\colon \tau]\!] : \gamma \in [\![\vdash \Gamma]\!] \to [\![\Gamma \vdash \tau]\!]\,(\gamma)\,,$$

in particular

$$\begin{aligned}[\![\Gamma \vdash e\colon \tau]\!]\,(\delta,\ t,\ \ldots) &\in [\![\Gamma \vdash \tau]\!]\,(\delta,\ t,\ \ldots)\\ &\quad \text{By Theorem D.3}\\ &= [\![\Delta \vdash \tau]\!]\,(\delta)\end{aligned}$$

thus $t([\![\Gamma \vdash e\colon \tau]\!]\,(\delta,\ t,\ \ldots)) : Set$.

Lemma A.17 (Equation (17) is well-defined). *First, by assumption*

$$\begin{aligned}\left[\!\!\left[\Gamma,\ T\langle\tau_\iota\rangle,\ \overline{y\colon \tau}^{\,i,j-1} \vdash \tau\right]\!\!\right] &: \left[\!\!\left[\vdash \Gamma,\ T\langle\tau_\iota\rangle,\ \overline{y\colon \tau}^{\,i,j-1}\right]\!\!\right] \to Set,\\ \left[\!\!\left[\Gamma,\ T\langle\tau_\iota\rangle,\ \overline{y\colon \tau}^{\,i,j} \vdash e_\iota\colon \tau_\iota\right]\!\!\right] &: (\gamma \in \left[\!\!\left[\vdash \Gamma,\ T\langle\tau_\iota\rangle,\ \overline{y\colon \tau}^{\,i,j}\right]\!\!\right])\\ &\qquad \to \left[\!\!\left[\Gamma,\ T\langle\tau_\iota\rangle,\ \overline{y\colon \tau}^{\,i,j} \vdash \tau_\iota\right]\!\!\right](\gamma)\,.\end{aligned}$$

Moreover, the context unfolds as

$$\begin{aligned}\left[\!\!\left[\vdash \Gamma,\ T\langle\tau_\iota\rangle,\ \overline{y\colon \tau}^{\,i,j-1}\right]\!\!\right] &= \gamma^* \in [\![\vdash \Gamma]\!] \times X^* \in ([\![\Gamma \vdash \tau_\iota]\!]\,(\gamma^*) \to Set)\\ &\qquad \bigtimes_{k=0}^{j-1} w_k^* \in \left[\!\!\left[\Gamma,\ T\langle\tau_\iota\rangle, \overline{y\colon \tau}^{\,k-1} \vdash\right]\!\!\right] \left(\gamma^*,\ X^*,\ \overline{w^*}^{\,k-1}\right).\end{aligned}$$

Finally, since $\gamma \in [\![\vdash \Gamma]\!]$ and $X \in [\![\Gamma \vdash \tau_\iota]\!]\,(\gamma) \to Set$, we have

$$w_{i,j} \in \left[\!\!\left[\Gamma,\ T\langle\tau_\iota\rangle,\ \overline{y\colon \tau}^{\,i,j-1} \vdash \tau\right]\!\!\right] \left(\gamma,\ X,\ \overline{w}^{\,i,j-1}\right)$$

and

$$\begin{aligned}
&\left[\!\left[\Gamma,\ T\langle\tau_\iota\rangle,\ \overline{y\colon\tau}^{\,i,j} \vdash \tau\right]\!\right] \left(\gamma,\ X,\ \overline{w}^{\,i,j}\right) \\
&\in \left[\!\left[\Gamma,\ T\langle\tau_\iota\rangle,\ \overline{y\colon\tau}^{\,i,j-1} \vdash \tau\right]\!\right] \left(\gamma,\ X,\ \overline{w}^{\,i,j-1}\right) \\
&\quad \textit{by Theorem D.3} \\
&= \left[\!\left[\Gamma \vdash \tau_\iota\right]\!\right](\gamma)\,.
\end{aligned}$$

Lemma A.18 (Equation (18) is well-defined). *It is well defined since from Theorem 5.1 we have that F is monotone on a complete lattice, hence it has a least fixpoint by Knaster-Tarski theorem. Now we can state the substitution of type variable theorem:*

Theorem A.19 (Substitution of type variable for data). *If* $\Gamma,\ T\langle\tau_\iota\rangle,\ \ldots \vdash \tau$ *and* $\Gamma \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}$ *then* $\Gamma,\ \mathbf{data}\ T\langle\tau_\iota\rangle\{c\},\ \ldots \vdash \tau$.

Lemma A.20 (Equation (19) is well-defined). *It is well defined because* $\gamma \in [\![\vdash \Gamma]\!]$ *by assumption and* $[\![\Gamma \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!] : [\![\vdash \Gamma]\!] \longrightarrow Set$, *hence* $[\![\Gamma \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!](\gamma) : Set$.

Lemma A.21 (Equation (20) is well-defined). *It is well defined because* $\gamma \in [\![\vdash \Gamma]\!]$ *by assumption and:*

$$\begin{aligned}
[\![\Gamma \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!] &: [\![\vdash \Gamma]\!] \longrightarrow Set, \\
[\![\Gamma,\ \mathbf{data}\ T\langle\tau_\iota\rangle\{c\} \vdash e\colon\tau]\!] &: (\gamma \in [\![\vdash \Gamma,\ \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!]) \\
&\qquad \longrightarrow [\![\Gamma,\ \mathbf{data}\ T\langle\tau_\iota\rangle\{c\} \vdash \tau]\!](\gamma)\,, \\
[\![\vdash \Gamma,\ \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!] &= \gamma^* \in [\![\vdash \Gamma]\!] \times \{[\![\Gamma \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!](\gamma^*)\},
\end{aligned}$$

hence

$$(\gamma, [\![\Gamma \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!](\gamma)) \in [\![\vdash \Gamma,\ \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!]\,,$$

and thus

$$\begin{aligned}
&[\![\Gamma,\ \mathbf{data}\ T\langle\tau_\iota\rangle\{c\} \vdash e\colon\tau]\!](\gamma,\ [\![\Gamma \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!](\gamma)) \\
&\qquad\qquad \in [\![\Gamma,\ \mathbf{data}\ T\langle\tau_\iota\rangle\{c\} \vdash \tau]\!](\gamma,\ [\![\Gamma \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!](\gamma))\,.
\end{aligned}$$

Lemma A.22 (Equation (21) is well-defined). *Indeed, by assumption*

$$\begin{aligned}
(\delta, t, \ldots) \in [\![\Gamma]\!] &= [\![\Delta,\ \Delta \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{c\},\ \ldots]\!] \\
&= \delta \in [\![\Delta]\!] \times \{[\![\Delta \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!](\delta)\} \times \ldots,
\end{aligned}$$

and

$$\begin{aligned}
[\![\Delta \vdash \mathbf{data}\ T\langle\tau_\iota\rangle\{c\}]\!](\delta) &: [\![\Delta \vdash \tau]\!](\delta) \longrightarrow Set, \\
[\![\Gamma \vdash e\colon\tau]\!] &: \gamma \in [\![\vdash \Gamma]\!] \longrightarrow [\![\Gamma \vdash \tau]\!](\gamma)\,,
\end{aligned}$$

in particular

$$\begin{aligned}
[\![\Gamma \vdash e\colon\tau]\!](\delta,\ t,\ \ldots) &\in [\![\Gamma \vdash \tau]\!](\delta,\ t,\ \ldots) \\
&\quad \textit{By Theorem D.3} \\
&= [\![\Delta \vdash \tau]\!](\delta)
\end{aligned}$$

thus $t([\![\Gamma \vdash e\colon\tau]\!](\delta,\ t,\ \ldots)) : Set$.

Lemma A.23 (Equation (22) is well-defined). *Since*

$$\begin{aligned}
&\vdash \Gamma \\
\implies\ & \Delta \vdash \textbf{data}\ T\langle \tau_\iota\rangle\{\dots,\ C\colon \overline{x\colon\tau}^{\,i} \to T\ e,\ \dots\} \\
\implies\ & \Delta,\ T\langle\tau_\iota\rangle \vdash \overline{x\colon\tau}^{\,i} \to T\ e \\
& \textit{By Theorem A.19} \\
\implies\ & \Delta,\ \textbf{data}\ T\langle\tau_\iota\rangle\{\dots,\ C\colon\overline{x\colon\tau}^{\,i} \to T\ e,\ \dots\} \vdash \overline{x\colon\tau}^{\,i} \to T\ e
\end{aligned}$$

And

$$\begin{aligned}
(\delta, t, \dots) \in [\![\Gamma]\!] &= [\![\Delta,\ \Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\overline{c}\},\ \dots]\!] \\
&= \delta \in [\![\Delta]\!] \times \{[\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\overline{c}\}]\!]\,(\delta)\} \times \dots,
\end{aligned}$$

It follows from Theorem D.3 that:

$$\begin{aligned}
&[\![\Gamma,\ \overline{x\colon\tau}^{\,i-1} \vdash \tau_i]\!]\left(\delta,\ t,\ \dots,\ \overline{w}^{\,i-1}\right) \\
&\qquad = [\![\Delta,\ \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\},\ \overline{x\colon\tau}^{\,i-1} \vdash \tau_i]\!]\left(\delta,\ t,\ \overline{w}^{\,i-1}\right)
\end{aligned}$$

and

$$\begin{aligned}
&[\![\Gamma,\ \overline{x\colon\tau}^{\,i} \vdash T\ e]\!]\left(\delta,\ t,\ \dots,\ \overline{w}^{\,i}\right) \\
&\qquad = [\![\Delta,\ \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\},\ \overline{x\colon\tau}^{\,i} \vdash T\ e]\!]\left(\delta,\ t,\ \overline{w}^{\,i}\right) \\
&\qquad = [\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\}]\!]\,(\delta)\,\left([\![\Gamma,\ \overline{x\colon\tau}^{\,i} \vdash e\colon\tau_\iota]\!]\left(\delta,\ t,\ \overline{w}^{\,i}\right)\right)
\end{aligned}$$

since $[\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\}]\!]\,(\delta)$ *is defined as the least fixpoint of the functional* F *we have that:*

$$[\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\}]\!]\,(\delta) = F([\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\}]\!]\,(\delta))$$

AF: Here it could be more precise because we pass t where actually the type is a var

And thus

$$\begin{aligned}
(C,\ \overline{w}^{\,i}) &\in F([\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\}]\!]\,(\delta))\left([\![\Gamma,\ \overline{x\colon\tau}^{\,i} \vdash e\colon\tau_\iota]\!]\left(\delta,\ \overline{w}^{\,i}\right)\right) \\
&= [\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\}]\!]\,(\delta)\,\left([\![\Gamma,\ \overline{x\colon\tau}^{\,i} \vdash e\colon\tau_\iota]\!]\left(\delta,\ t,\ \dots,\ \overline{w}^{\,i}\right)\right) \\
&= [\![\Gamma \vdash T\ e]\!]\left(\delta,\ t,\ \dots,\ \overline{w}^{\,i}\right)
\end{aligned}$$

Lemma A.24 (Equation (23) is well-defined). *By assumption*

$$\begin{aligned}
(\delta, t, \dots) \in [\![\Gamma]\!] &= [\![\Delta,\ \Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\overline{c}\},\ \dots]\!] \\
&= \delta \in [\![\Delta]\!] \times \{[\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\overline{c}\}]\!]\,(\delta)\} \times \dots,
\end{aligned}$$

and

$$[\![\Gamma \vdash e_1\colon T\ e_\iota]\!] : (\gamma \in [\![\vdash \Gamma]\!]) \to [\![\Gamma \vdash T\ e_\iota]\!]\,(\gamma)\,,$$

Hence

$$\begin{aligned}
[\![\Gamma \vdash e_1\colon T\ e_\iota]\!]\,(\delta,\ t,\ \dots) &\in [\![\Gamma \vdash T\ e_\iota]\!]\,(\delta,\ t,\ \dots) \\
&= [\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\}]\!]\,(\delta)\,([\![\Gamma \vdash e_\iota\colon\tau_\iota]\!]\,(\delta,\ t,\ \dots))
\end{aligned}$$

since $[\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\}]\!]\,(\delta)$ *is defined as the least fixpoint of the functional* F *we have that:*

$$[\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\}]\!]\,(\delta) = F([\![\Delta \vdash \textbf{data}\ T\langle\tau_\iota\rangle\{\dots\}]\!]\,(\delta))$$

Hence by the definition of F

$$(C_i, \overline{w}^{\,j,i}) = [\![\Gamma \vdash e_1 \colon T\ e_\iota]\!]\,(\delta,\ t,\ \ldots)$$
$$w^{j,i} \in \left[\!\!\left[\Delta,\ \overline{y\colon \tau_y}^{\,j-1,i} \vdash \tau_y^{j,i}\right]\!\!\right]\left(\delta,\ \overline{w}^{\,j-1,i}\right)$$

Since

$$\begin{aligned}&[\![\Gamma,\ \overline{y\colon \tau_y}^{\,i,j},\ x\colon \{y\colon Unit \mid e_1 = C_i\ \} \vdash e_{2,i}\colon \tau]\!]\\ &\qquad\qquad :\gamma \in [\![\vdash \Gamma,\ \overline{y\colon \tau_y}^{\,i,j},\ x\colon \{y\colon Unit \mid e_1 = C_i\ \}]\!]\\ &\qquad\qquad\qquad\qquad \to [\![\Gamma,\ \overline{y\colon \tau_y}^{\,i,j},\ x\colon \left\{y\colon Unit \,\middle|\, e_1 = C_i\ \overline{y}^{\,i,j}\right\} \vdash \tau]\!]\ (\gamma)\end{aligned}$$

And

$$\begin{aligned}&[\![\vdash \Gamma,\ \overline{y\colon \tau_y}^{\,i,j},\ x\colon \{y\colon Unit \mid e_1 = C_i\ \}]\!]\\ &= \gamma^* \in [\![\vdash \Gamma]\!] \times \left(\bigtimes_{k=0}^{j-1,i} w_k^* \in \left[\!\!\left[\Gamma,\ \overline{y\colon \tau_y}^{\,k-1,i} \vdash\right]\!\!\right]\left(\gamma^*,\ \overline{w^*}^{\,k-1,i}\right)\right)\\ &\times\ [\![\Gamma,\ \overline{y\colon \tau_y}^{\,j,i} \vdash x\colon \left\{y\colon Unit \,\middle|\, e_1 = C_i\ \overline{y}^{\,i,j}\right\}]\!]\ \left(\gamma^*,\ \overline{w^*}^{\,j,i}\right)\end{aligned}$$

And since $(\delta,\ t,\ \ldots) \in [\![\Gamma]\!]$ *and by Theorem D.3:*

$$w^{j,i} \in \left[\!\!\left[\Gamma,\ \overline{y\colon \tau_y}^{\,j-1,i} \vdash \tau_y^{j,i}\right]\!\!\right]\left(\delta,\ t,\ \ldots,\ \overline{w}^{\,j-1,i}\right)$$

And thus:

$$\begin{aligned}&[\![\Gamma,\ \overline{y\colon \tau_y}^{\,i,j} \vdash \left\{y\colon Unit \,\middle|\, e_1 = C_i\ \overline{y}^{\,i,j}\right\}]\!]\ \left(\delta,\ t,\ \ldots,\ \overline{w}^{\,j,i}\right)\\ &= \left\{\star \mid [\![\Gamma,\ \overline{y\colon \tau_y}^{\,i,j},\ x\colon unit \vdash e_1 = C_i\ \overline{y}^{\,i,j}\colon Bool]\!]\ \left(\gamma,\ t,\ \ldots,\ \overline{w}^{\,j,i},\ \star\right)\right\}\\ &= \left\{\star \mid [\![\Gamma,\ \overline{y\colon \tau_y}^{\,i,j},\ x\colon unit \vdash e_1\colon T\ e_\iota]\!]\ \left(\gamma,\ t,\ \ldots,\ \overline{w}^{\,j,i},\ \star\right) = (C_i,\ \overline{w}^{\,j,i})\right\}\\ &\quad \text{By Theorem D.4}\\ &= \{\star \mid tt\}\end{aligned}$$

Hence $(\gamma,\ t,\ \ldots,\ \overline{w}^{\,j,i},\ \star) \in \vdash \Gamma,\ \overline{y\colon \tau_y}^{\,i,j},\ x\colon \left\{y\colon Unit \,\middle|\, e_1 = C_i\ \overline{y}^{\,i,j}\right\}$ *and thus*

$$\begin{aligned}&[\![\Gamma,\ \overline{y\colon \tau_y}^{\,i,j},\ x\colon \left\{y\colon Unit \,\middle|\, e_1 = C_i\ \overline{y}^{\,i,j}\right\} \vdash e_{2,i}\colon \tau]\!]\ \left(\gamma,\ t,\ \ldots,\ \overline{w}^{\,j,i},\ \star\right)\\ &\in [\![\Gamma,\ \overline{y\colon \tau_y}^{\,i,j},\ x\colon \left\{y\colon Unit \,\middle|\, e_1 = C_i\ \overline{y}^{\,i,j}\right\} \vdash \tau]\!]\ \left(\gamma,\ t,\ \ldots,\ \overline{w}^{\,j,i},\ \star\right)\\ &\quad \text{By Theorem D.3}\\ &= [\![\Gamma \vdash \tau]\!]\,(\delta,\ t,\ \ldots)\end{aligned}$$

Lemma A.25 (Equation (24) is well-defined). *Is well defined because:*

$$[\![\Gamma \vdash \tau_\iota]\!] : [\![\vdash \Gamma]\!] \to Set,$$
$$[\![\Gamma \vdash e_m\colon \tau_\iota \to Nat]\!] : \gamma \in [\![\vdash \Gamma]\!] \to [\![\Gamma \vdash \tau_\iota \to Nat]\!]\,(\gamma)\,,$$

In particular,

$$[\![\Gamma \vdash e_m\colon \tau_\iota \to Nat]\!]\,(\gamma) \in [\![\Gamma \vdash \tau_\iota]\!]\,(\gamma) \to [\![\Gamma \vdash Nat]\!]\,(\gamma) = \mathbb{N}.$$

Lemma A.26 (Equation (25) is well-defined). *By assumption,*

$$\left[\!\left[\Gamma, T\langle\{x\colon \tau_\iota \mid e_m x < n\}\rangle, \overline{y\colon\tau}^{\,i,j-1} \vdash \tau\right]\!\right] : \left[\!\left[\vdash \Gamma, T\langle\{x\colon \tau_\iota \mid e_m x < n\}\rangle, \overline{y\colon\tau}^{\,i,j-1}\right]\!\right] \longrightarrow Set$$
$$\left[\!\left[\Gamma, T\langle\tau_\iota\rangle, \overline{y\colon\tau}^{\,i,j} \vdash e_\iota\colon \tau_\iota\right]\!\right] : (\gamma \in \left[\!\left[\vdash \Gamma, T\langle\{x\colon \tau_\iota \mid e_m x < n\}\rangle, \overline{y\colon\tau}^{\,i,j}\right]\!\right])$$
$$\longrightarrow \left[\!\left[\Gamma, T\langle\{x\colon \tau_\iota \mid e_m x < n\}\rangle, \overline{y\colon\tau}^{\,i,j} \vdash \tau_\iota\right]\!\right](\gamma)\,.$$

Moreover, the context unfolds as

$$\begin{aligned}\left[\!\left[\vdash \Gamma,\ T\langle\{x\colon \tau_\iota \mid e_m x < n\}\rangle,\ \overline{y\colon\tau}^{\,i,j-1}\right]\!\right] = \gamma^* &\in \left[\!\left[\vdash \Gamma\right]\!\right]\\ \times X^* &\in (\left[\!\left[\Gamma \vdash \{x\colon \tau_\iota \mid e_m x < n\}\right]\!\right](\gamma^*) \longrightarrow Set)\\ \bigtimes_{k=0}^{j-1} w_k^* &\in \left[\!\left[\Gamma,\ T\langle\{x\colon \tau_\iota \mid e_m x < n\}\rangle, \overline{y\colon\tau}^{\,k-1} \vdash\right]\!\right]\left(\gamma^*,\ X^*,\ \overline{w^*}^{\,k-1}\right).\end{aligned}$$

Finally, $\gamma \in \left[\!\left[\vdash \Gamma\right]\!\right]$ *and* $X \in \left[\!\left[\Gamma \vdash \{x\colon \tau_\iota \mid e_m x < n\}\right]\!\right](\gamma) \longrightarrow Set$ *because* $\left[\!\left[\Gamma \vdash \{x\colon \tau_\iota \mid e_m x < n\}\right]\!\right](\gamma) = \bigcup_{i=0}^{n-1} Layer(i)$. *Now given that we have* $\Gamma,\ T\langle\tau_\iota\rangle \vdash \overline{y\colon\tau}^{\,j} \longrightarrow T\,e_\iota$ *and* $stratified(\Gamma, \overline{y\colon\tau}^{\,j_i}, e_\iota, e_m, \overline{y\colon\tau}^{\,j_i})$, *hence*

$$w_{i,j} \in \left[\!\left[\Gamma,\ T\langle\{x\colon \tau_\iota \mid e_m x < n\}\rangle,\ \overline{y\colon\tau}^{\,i,j-1} \vdash \tau\right]\!\right]\left(\gamma,\ X,\ \overline{w}^{\,i,j-1}\right)$$

is well defined, and

$$\begin{aligned}&\left[\!\left[\Gamma,\ T\langle\tau_\iota\rangle,\ \overline{y\colon\tau}^{\,i,j} \vdash \tau\right]\!\right]\left(\gamma,\ X,\ \overline{w}^{\,i,j}\right)\\ &\in \left[\!\left[\Gamma,\ T\langle\tau_\iota\rangle,\ \overline{y\colon\tau}^{\,i,j-1} \vdash \tau\right]\!\right]\left(\gamma,\ X,\ \overline{w}^{\,i,j-1}\right)\\ &\quad \text{by Theorem D.3}\\ &= \left[\!\left[\Gamma \vdash \tau_\iota\right]\!\right](\gamma)\,.\end{aligned}$$

Lemma A.27 (Equation (26) is well-defined). *By induction on* n*:*

- $n = 0$: *then* $\bigcup_{i=0}^{0-1} Layer(i) = \emptyset$, *hence* $\bot \in (w \in \emptyset \longrightarrow Set)$ *vacuously and thus* $F(0, \bot) : w \in \bigcup_{i=0}^{0} Layer(i) \longrightarrow Set$
- $n > 0$: *then by inductive hypothesis* $Rec(n-1) : w \in \bigcup_{i=0}^{n-1} Layer(i) \longrightarrow Set$, *hence* $F(n, Rec(n-1)) : w \in \bigcup_{i=0}^{n} Layer(i) \longrightarrow Set$.

Lemma A.28 (Equation (27) is well-defined). *Well definedness follows the same argument as positive data constructor (Theorem A.23) but where the codomain* $\left[\!\left[\Delta \vdash \textbf{strat}\ T\langle\tau_\iota\rangle[e_m]\{\ldots\}\right]\!\right](\delta)$ *is not characterized as a lfp but as:*

$$w_\iota \longmapsto Rec(\left[\!\left[\Delta \vdash e_m\colon \tau_\iota \longrightarrow Nat\right]\!\right](\delta)\,(w_\iota))(w_\iota)$$

$$Rec(n) \triangleq \begin{cases} F(0, \bot) & \textit{if } n = 0\\ F(n, Rec(n-1)) & \textit{if } n > 0\end{cases}$$

Hence $(C_i,\ \overline{w}^{\,i}) \in Rec(\left[\!\left[\Gamma \vdash e_m\colon \tau_\iota \longrightarrow Nat\right]\!\right](\delta)\,(w_\iota))$

Lemma A.29 (Equation (28) is well-defined). *Well definedness follows the same argument as positive data constructor (Theorem A.24) but where the codomain* $\left[\!\left[\Delta \vdash \textbf{strat}\ T\langle\tau_\iota\rangle[e_m]\{\ldots\}\right]\!\right](\delta)$ *is not characterized as a lfp but as:*

$$w_\iota \longmapsto Rec(\left[\!\left[\Delta \vdash e_m\colon \tau_\iota \longrightarrow Nat\right]\!\right](\delta)\,(w_\iota))(w_\iota)$$

$$Rec(n) \triangleq \begin{cases} F(0, \bot) & \textit{if } n = 0\\ F(n, Rec(n-1)) & \textit{if } n > 0\end{cases}$$

B Polarity lemma

Proof of Theorem 5.1. By structural induction on τ:

- Basic types:
  For $\tau \in \{Bool, Nat, Unit\}$,
  $$[\![\Gamma,\ T\langle\tau_\iota\rangle \vdash \tau]\!]\,(\gamma,\ X)$$
  is independent of X: it equals $\mathbb{2}$, $\mathbb{N}$, or $\mathbb{1}$, respectively. Therefore $G(P) \lesseqgtr G(Q)$.
- Different type variable or reference to type:
  For $\tau = U$ with $U \neq T$,
  $$[\![\Gamma,\ T\langle\tau_\iota\rangle \vdash U]\!]\,(\gamma,\ X) = \mathbb{U}([\![\Gamma \vdash e\colon \tau_\iota]\!]\,(\gamma,\ X))$$
  is independent of X from Theorem D.9. Therefore $G(P) \lesseqgtr G(Q)$.
- Same type variable:
  For $\tau = T$,
  $$[\![\Gamma,\ T\langle\tau_\iota\rangle \vdash T\ e_\iota]\!]\,(\gamma,\ X) = X([\![\Gamma,\ T\langle\tau_\iota\rangle \vdash e_\iota\colon \tau_\iota]\!]\,(\gamma,\ X))$$
  By Theorem D.9, we have $X([\![\Gamma,\ T\langle\tau_\iota\rangle \vdash e_\iota\colon \tau_\iota]\!]\,(\gamma,\ X)) = X(w)$ for some w, since T is appearing positively we need to verify that $P(w) \leq Q(w)$, which holds by assumption.
- Refinement type:
  For $\tau = \{x\colon \tau \mid e\}$,
  $$\begin{aligned}&[\![\Gamma,\ T\langle\tau_\iota\rangle \vdash \{x\colon \tau \mid e\}]\!]\,(\gamma,\ X)\\ =&\left\{w\ \middle|\ \begin{array}{c} w \in [\![\Gamma,\ T\langle\tau_\iota\rangle \vdash \tau]\!]\,(\gamma,\ X)\\ [\![\Gamma,\ T\langle\tau_\iota\rangle,\ x\colon \tau \vdash r\colon Bool]\!]\,(\gamma,\ X,\ w) = tt\end{array}\right\}\end{aligned}$$
  By inductive hypothesis, we have
  $$[\![\Gamma,\ T\langle\tau_\iota\rangle \vdash \tau]\!]\,(\gamma,\ P) \lesseqgtr [\![\Gamma,\ T\langle\tau_\iota\rangle \vdash \tau]\!]\,(\gamma,\ Q)$$
  and Theorem D.9 it follows $G(P) \lesseqgtr G(Q)$.
- Function type:
  For $\tau = x\colon \tau_x \longrightarrow \tau'$,
  $$\begin{aligned}&[\![\Gamma,\ T\langle\tau_\iota\rangle \vdash x\colon \tau_x \longrightarrow \tau]\!]\,(\gamma,\ X)\\ =&\ w \in [\![\Gamma,\ T\langle\tau_\iota\rangle \vdash \tau_x]\!]\,(\gamma,\ X) \longrightarrow [\![\Gamma,\ T\langle\tau_\iota\rangle,\ x\colon \tau_x \vdash \tau']\!]\,(\gamma,\ X,\ w)\end{aligned}$$
  We show the case for positive occurrence (the negative one is similar). We have by definition that τ_x appears negatively in τ and τ' appears positively in τ. Thus, by the inductive hypothesis,
  $$[\![\Gamma,\ T\langle\tau_\iota\rangle \vdash \tau_x]\!]\,(\gamma,\ P) \geq [\![\Gamma,\ T\langle\tau_\iota\rangle \vdash \tau_x]\!]\,(\gamma,\ Q)$$
  and
  $$[\![\Gamma,\ T\langle\tau_\iota\rangle,\ x\colon \tau_x \vdash \tau']\!]\,(\gamma,\ P,\ w) \leq [\![\Gamma,\ T\langle\tau_\iota\rangle,\ x\colon \tau_x \vdash \tau']\!]\,(\gamma,\ Q,\ w)$$
  And thus by co(ntra)variance of functions we have $G(P) \leq G(Q)$.

□

C Sub-typing theorem

Proof of Theorem 3.1. By structural induction on the derivation of $\Gamma \vdash \tau_1 \preceq \tau_2$.

- Case Sub-Base: Have $\gamma \in [\![\vdash \Gamma]\!]$ and $w \in [\![\{x\colon \tau_1 \mid e_1\}]\!]\,(\gamma)$, i.e., by the definition of the semantics of refinement types:
  - $w \in [\![\Gamma \vdash \tau_1]\!]\,(\gamma)$,
  - $[\![\Gamma,\ x\colon \tau_1 \vdash e_1\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{tt}$.

  By inductive hypothesis and the subtyping assumption $\Gamma \vdash \tau_1 \preceq \tau_2$, we have
  $$w \in [\![\Gamma \vdash \tau_2]\!]\,(\gamma)\,.$$
  Moreover, from the entailment
  $$\Gamma,\ x\colon \tau_1 \models e_1 \implies e_2[y/x],$$
  we get:
  $$\begin{aligned} & [\![\Gamma,\ x\colon \tau_1 \vdash e_1 \implies e_2[y/x]\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{tt} \\ & \quad \text{By definition} \\ = & [\![\Gamma,\ x\colon \tau_1 \vdash e_1\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{tt} \implies [\![\Gamma,\ x\colon \tau_1 \vdash e_2\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{tt} \\ & \quad \text{Since } [\![\Gamma,\ x\colon \tau_1 \vdash e_1\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{tt}, \text{ we conclude} \\ \implies & [\![\Gamma,\ x\colon \tau_1 \vdash e_2[y/x]\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{tt} \\ & \quad \text{By renaming of variables} \\ = & [\![\Gamma,\ y\colon \tau_1 \vdash e_2\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{tt}. \end{aligned}$$
  Hence,
  $$w \in [\![\{x\colon \tau_2 \mid e_2\}]\!]\,(\gamma)\,.$$
- Case Sub-Flat: Have $\gamma \in [\![\vdash \Gamma]\!]$ and $w \in [\![\{x\colon\ \{y\colon \tau \mid e_i\} \mid e_o\}]\!]\,(\gamma)$, i.e., by the definition of the semantics of refinement types:
  - $w \in [\![\Gamma \vdash \tau]\!]\,(\gamma)$,
  - $[\![\Gamma,\ y\colon \tau \vdash e_i\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{tt}$,
  - $[\![\Gamma,\ x\colon\ \{y\colon \tau \mid e_i\} \vdash e_o\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{tt}$.

  Now,
  $$\begin{aligned} & [\![\Gamma,\ x\colon \tau \vdash \textbf{if } e_i[y/x] \textbf{ then } e_o \textbf{ else } \mathit{False}\colon \mathit{Bool}]\!]\,(\gamma,\ w) \\ & \quad \text{By definition of the semantics of if-then-else} \\ = & \begin{cases} [\![\Gamma \vdash e_o\colon \mathit{Bool}]\!]\,(\gamma,\ w) & \text{if } [\![\Gamma \vdash e_i[y/x]\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{tt}, \\ [\![\Gamma \vdash \mathit{False}\colon \mathit{Bool}]\!]\,(\gamma,\ w) & \text{if } [\![\Gamma \vdash e_i[y/x]\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{ff} \end{cases} \\ & \quad \text{From the assumption } [\![\Gamma,\ y\colon \tau \vdash e_i\colon \mathit{Bool}]\!]\,(\gamma,\ w) = \mathit{tt} \text{ (Modulo renaming)} \\ = & [\![\Gamma \vdash e_o\colon \mathit{Bool}]\!]\,(\gamma,\ w) \\ & \quad \text{From the assumptions} \\ = & \mathit{tt}. \end{aligned}$$
  Hence $w \in [\![\Gamma \vdash \{x\colon \tau \mid \textbf{if } e_i[y/x] \textbf{ then } e_o \textbf{ else } \mathit{False}\}]\!]\,(\gamma)$.
- Sub-Arr: By inductive hypothesis, we have
  $$[\![\Gamma \vdash \tau_y]\!]\,(\gamma) \subseteq [\![\Gamma \vdash \tau_x]\!]\,(\gamma)\,,$$
  and for all $w_y \in [\![\Gamma \vdash \tau_y]\!]\,(\gamma)$,
  $$[\![\Gamma,\ y\colon \tau_y \vdash \tau_1[x/y]]\!]\,(\gamma,\ w_y) \subseteq [\![\Gamma,\ y\colon \tau_y \vdash \tau_2]\!]\,(\gamma,\ w_y)\,.$$

Hence,

$$w \in [\![\Gamma \vdash \tau_x]\!] (\gamma) \rightarrow [\![\Gamma,\ x{:}\ \tau_x \vdash \tau_1]\!] (\gamma,\ w)$$
$$\subseteq w \in [\![\Gamma \vdash \tau_y]\!] (\gamma) \rightarrow [\![\Gamma,\ y{:}\ \tau_y \vdash \tau_2]\!] (\gamma,\ w)\,.$$

Therefore,

$$[\![\Gamma \vdash (x{:}\ \tau_x) \rightarrow \tau_1]\!] (\gamma) \subseteq [\![\Gamma \vdash (y{:}\ \tau_y) \rightarrow \tau_2]\!] (\gamma)\,.$$

- Sub-TyApp: $\gamma \in [\![\vdash \Gamma]\!]$, $\Gamma \models e_1 = e_2$ implies $[\![\Gamma \vdash e_1{:}\ \tau]\!] (\gamma) = [\![\Gamma \vdash e_2{:}\ \tau]\!] (\gamma)$ and thus

$$\begin{aligned}[\![\Gamma \vdash T\ e_1]\!] (\gamma) &= [\![\Gamma \vdash T]\!] (\gamma) ([\![\Gamma \vdash e_1{:}\ \tau]\!] (\gamma)) \\ &= [\![\Gamma \vdash T]\!] (\gamma) ([\![\Gamma \vdash e_2{:}\ \tau]\!] (\gamma)) \\ &= [\![\Gamma \vdash T\ e_2{:}\ \tau]\!] (\gamma)\,.\end{aligned}$$

□

D Substitution lemmas

Lemma D.1 (Type weakening). *If $\Gamma \vdash \tau$ and $\Gamma \subseteq \Delta$ then $\Delta \vdash \tau$*

Proof. By induction on the derivation of $\Gamma \vdash \tau$. The only interesting case is for expressions resolved by Theorem D.2 and references to type in the context but all the typing rule admits weakening. □

Lemma D.2 (Expression weakening). *If $\Gamma \vdash e{:}\ \tau$ and $\Gamma \subseteq \Delta$ then $\Delta \vdash e{:}\ \tau$*

Proof. By induction on the derivation of $\Gamma \vdash e{:}\ \tau$. The only interesting case is for types resolved by Theorem D.1 and variables in the context but the typing rule admits weakening. □

Lemma D.3 (Semantic type weakening). *If $\Gamma \vdash \tau$, $\Delta \vdash \tau$ and $\gamma \in [\![\Gamma]\!] \subseteq \delta \in [\![\Delta]\!]$ then $[\![\Gamma \vdash \tau]\!] (\gamma) = [\![\Delta \vdash \tau]\!] (\delta)$*

Proof. By induction on the definition of $[\![\Gamma \vdash \tau]\!] (\gamma)$ since all the extra part of the context Δ are ignored in the definition of the semantics as they are not free in τ. □

Lemma D.4 (Semantic expression weakening). *If $\Gamma \vdash e{:}\ \tau$, $\Delta \vdash e{:}\ \tau$ and $\gamma \in [\![\Gamma]\!] \subseteq \delta \in [\![\Delta]\!]$ then $[\![\Gamma \vdash e{:}\ \tau]\!] (\gamma) = [\![\Delta \vdash e{:}\ \tau]\!] (\delta)$*

Proof. By induction on the definition of $[\![\Gamma \vdash e{:}\ \tau]\!] (\gamma)$ since all the extra part of the context Δ are ignored in the definition of the semantics as they are not free in e. □

Lemma D.5 (Type substitution). *If $\Gamma,\ x{:}\ \tau_x \vdash \tau$ and $\Gamma \vdash e{:}\ \tau_x$ then $\Gamma \vdash \tau[x/e]$.*

Proof. By induction on the derivation of $\Gamma,\ x{:}\ \tau_x \vdash \tau$. The only interesting case is for expressions resolved by Theorem D.6. □

Lemma D.6 (Expression substitution). *If $\Gamma,\ x{:}\ \tau_x \vdash e_1{:}\ \tau$ and $\Gamma \vdash e_2{:}\ \tau_x$ then $\Gamma \vdash e_1[x/e_2]{:}\ \tau[x/e_2]$.*

Proof. By induction on the derivation of $\Gamma,\ x{:}\ \tau_x \vdash e_1{:}\ \tau$. The only interesting case is for variables but we can replace them by the derivation of $\Gamma \vdash e_2{:}\ \tau_x$. □

Lemma D.7 (Semantic type substitution). *If $\Gamma,\ x{:}\ \tau_x \vdash \tau$ and $\Gamma \vdash e{:}\ \tau_x$ then $[\![\Gamma \vdash \tau[x/e]]\!] (\gamma) = [\![\Gamma,\ x{:}\ \tau_x \vdash \tau]\!] (\gamma,\ [\![\Gamma \vdash e{:}\ \tau_x]\!] (\gamma))$.*

Proof. By induction on the definition of $[\![\Gamma,\ x{:}\ \tau_x \vdash \tau]\!] (\gamma,\ [\![\Gamma \vdash e{:}\ \tau_x]\!] (\gamma))$ since only the semantics of expressions depends on x it is resolved by Theorem D.8. □

Lemma D.8 (Semantic expression substitution). *If* $\Gamma,\ x\colon \tau_x\ \vdash\ e_1\colon \tau$ *and* $\Gamma\ \vdash\ e_2\colon \tau_x$ *then* $[\![\Gamma \vdash e_1[x/e_2]\colon \tau[x/e_2]]\!]\,(\gamma) = [\![\Gamma,\ x\colon \tau_x \vdash e_1\colon \tau]\!]\,(\gamma,\ [\![\Gamma \vdash e_2\colon \tau_x]\!]\,(\gamma))$.

Proof. By induction on the definition of $[\![\Gamma,\ x\colon \tau_x \vdash e_1\colon \tau]\!]\,(\gamma,\ [\![\Gamma \vdash e_2\colon \tau_x]\!]\,(\gamma))$ since all the occurrences of x in e_1 and τ are replaced by e_2 in the definition of the semantics. □

Lemma D.9 (Type variables semantics irrelevance). *If* $\Gamma,\ T\langle\tau_\iota\rangle \vdash e\colon \tau_\iota$ *then for any* $\gamma \in [\![\vdash \Gamma]\!]$ *and* $X, Y \in [\![\Gamma \vdash \tau_\iota]\!]\,(\gamma)$, $[\![\Gamma,\ T\langle\tau_\iota\rangle \vdash e\colon \tau_\iota]\!]\,(\gamma,\ X) = [\![\Gamma,\ T\langle\tau_\iota\rangle \vdash e\colon \tau_\iota]\!]\,(\gamma,\ Y)$

Proof. By induction on the definition of $[\![\Gamma,\ T\langle\tau_\iota\rangle \vdash e\colon \tau_\iota]\!]\,(\gamma,\ X)$ The semantics is independent of X. □