Gradual Guarantee via Step-Indexed Logical Relations in Agda

Jeremy G. Siek
School of Informatics, Computing, and Engineering
Indiana University
Bloomington, IN, USA
jsiek@iu.edu

The gradual guarantee is an important litmus test for gradually typed languages, that is, languages that enable a mixture of static and dynamic typing. The gradual guarantee states that changing the precision of a type annotation does not change the behavior of the program, except perhaps to trigger an error if the type annotation is incorrect. Siek et al. (2015) proved that the Gradually Typed Lambda Calculus (GTLC) satisfies the gradual guarantee using a simulation-based proof and mechanized their proof in Isabelle. In the following decade, researchers have proved the gradual guarantee for more sophisticated calculi, using step-indexed logical relations. However, given the complexity of that style of proof, there has not yet been a mechanized proof of the gradual guarantee using step-indexed logical relations. This paper reports on a mechanized proof of the gradual guarantee for the GTLC carried out in the Agda proof assistant.

1 Introduction

Gradually typed languages introduce the unknown type $\star$ as a way for programmers to control the amount of type precision, and compile-time type checking, in their program [15, 16]. If all type annotations are just $\star$, then the program behaves like a dynamically typed language. At the other end of the spectrum, if no type annotation contains $\star$, then the program behaves like a statically typed language.

Siek et al. [18] introduce the *gradual guarantee* as a litmus test for gradually typed languages. This property says that the behavior of a program should not change (except for errors) when the programmer changes type annotations to be more or less precise. Siek et al. [18] prove that the Gradually Typed Lambda Calculus (GTLC) satisfies the gradual guarantee using a simulation-based proof and mechanize the result in the Isabelle proof assistant [12]. Using logical relations, New and Ahmed [9] prove the gradual guarantee for the GTLC and New et al. [10] prove the gradual guarantee for a polymorphic calculus. Several researchers apply logical relations to prove other properties of gradually typed languages, such as noninterference[20], parametricity [1, 10, 7], and fully abstract embedding [5]. Of this later work, only the abstract embedding was mechanized (in Coq [19] using the Iris framework [6].)

There are several technical challenges to overcome in developing a mechanized proof of the gradual guarantee using step-indexed logical relations. As in any programming language mechanization, one must choose how to represent variables and perform substitution. Moreover, proofs based on logical relations rely on the fact that substitutions commute, and the proof of this standard result is quite technical and lengthy. Repeating these proofs for each new programming language is tedious, but this metatheory can be developed in a language-independent way using the notion of *abstract binding trees* [4]. To this end we developed the Abstract Binding Tree library in Agda [13], representing variables as de Bruijn indices and implementing substitution via parallel renaming and substitution [8, 21].

Bieniusa, Degen, Wehr (Eds.). A Second Soul:
Celebrating the Many Languages of Programming
EPTCS 413, 2024, pp. 27–42, doi:10.4204/EPTCS.413.3

The second technical challenge is that step-indexed logical relations "involve tedious, error-prone, and proof-obscuring step-index arithmetic" [3]. Dreyer, Ahmed, and Birkedal [3] propose to abstract over the step-indexing using a modal logic named LSLR. Dreyer and Birkedal, with many colleagues, implemented this logic as part of the Iris framework [6] in the Coq proof assistant. To make a similar modal logic available in Agda, we developed the Step-Indexed Logic (SIL) [14]. The proof of the gradual guarantee in this paper is the first application of SIL and we report on the experience. This paper is a literate Agda script, though we omit some of the proofs. The full Agda development is available from the following repository, in the `LogRel` directory.

`https://github.com/jsiek/gradual-typing-in-agda`

The semantics of the GTLC is defined by translation to a Cast Calculus, so we present the Cast Calculus in Section 2. We define the precision relation on types and terms in Section 3 and we review Step-Indexed Logic in Section 4. We define a logical relation for precision in Section 5. We prove the Fundamental Theorem of the logical relation in Section 6. To finish the proof of the gradual guarantee, in Section 7 we prove that the logical relation implies the gradual guarantee. Section 8 concludes this paper with a comparison of using logical-relations versus simulation to prove the gradual guarantee and it acknowledges Peter Thiemann and Philip Wadler for their contributions to this work.

2 Cast Calculus

The type structure of the Cast Calculus includes base types (integer and Boolean), function types, and the unknown type $\star$. The *ground types* include just the base types and function types from $\star$ to $\star$.

```
data Type : Set where
  ⋆ : Type
  $_t_ : (ι : Base) → Type
  _⇒_ : (A : Type) → (B : Type) → Type
```

```
data Ground : Set where
  $^g_ : (ι : Base) → Ground
  ⋆⇒⋆ : Ground

⌈_⌉ : Ground → Type
⌈ $^g ι ⌉ = $_t ι
⌈ ⋆⇒⋆ ⌉ = ⋆ ⇒ ⋆
```

There are three special features in the Cast Calculus:

1. injection $M\langle G! \rangle$, for casting from a ground type G to the unknown type $\star$,
2. projection $M\langle H? \rangle$, for casting from the unknown type $\star$ to a ground type H, and
3. blame which represents a runtime exception if a projection fails.

This Cast Calclulus differs from many of those in the literature [15, 17, 18] in that it does not include casts from one function type to another, a choice that reduces the number of reduction rules and simplifies the technical development. However, casts from one function type to another can be compiled into a combination of lambda abstractions, injections, and projections.

We define the terms of the Cast Calculus in Agda using the Abstract Binding Tree (ABT) library by instantiating it with an appropriate set of operators together with a description of their arity and variable-binding structure. To that end, the following `Op` data type includes one constructor for each term constructor that we have in mind for the Cast Calculus, except for variables which are always present. The `sig` function, shown below, specifies the variable binding structure for each operator. It returns a list of natural numbers where ■ represents zero and ν represents successor. The list includes one number for

each subterm; the number specifices how many variables come into scope for that subterm. Lambda abstraction (`op-lam`) has a single subterm and brings one variable into scope, whereas application (`op-app`) has two subterms but does not bind any variables.

```
data Lit : Set where
  Num : ℕ → Lit
  Bool : 𝔹 → Lit
```

```
data Op : Set where
  op-lam : Op
  op-app : Op
  op-lit : Lit → Op
  op-inject : Ground → Op
  op-project : Ground → Op
  op-blame : Op
```

```
sig : Op → List Sig
sig op-lam = (ν ■) :: []
sig op-app = ■ :: ■ :: []
sig (op-lit c) = []
sig (op-inject G) = ■ :: []
sig (op-project H) = ■ :: []
sig (op-blame) = []
```

We instantiate the ABT library as follows, by applying it to `Op` and `sig`. We rename the resulting `ABT` type to `Term`.

```
open import rewriting.AbstractBindingTree Op sig renaming (ABT to Term) public
```

We define Agda patterns to give succinct syntax to the construction of abstract binding trees.

```
pattern ƛ N = op-lam ⟪ cons (bind (ast N)) nil ⟫
infixl 7 _·_
pattern _·_ L M = op-app ⟪ cons (ast L) (cons (ast M) nil) ⟫
pattern $ c = (op-lit c) ⟪ nil ⟫
pattern _⟨_!⟩ M G = (op-inject G) ⟪ cons (ast M) nil ⟫
pattern _⟨_?⟩ M H = (op-project H) ⟪ cons (ast M) nil ⟫
pattern blame = op-blame ⟪ nil ⟫
```

The ABT library represents variables with de Bruijn indices and provides a definition of parallel substitution and many theorems about substitution. The de Bruijn indices are represented directly as natural numbers in Agda, with the constructors `0` and `suc` for zero and successor. It is helpful to think of a parallel substitution as an infinite stream, or equivalently, as a function from natural numbers (de Bruijn indices) to terms. The • operator is stream cons, that is, it adds a term to the front of the stream.

```
sub-zero : ∀ {V σ} → (V • σ) 0 ≡ V
sub-zero = refl
sub-suc : ∀ {V x σ} → (V • σ) (suc x) ≡ σ x
sub-suc {V}{x}{σ} = refl
```

The ABT library provides the operator $\langle\!\langle \sigma \rangle\!\rangle M$ for applying a substitution to a term. Here are the equations for substitution applied to variables, application, and lambda abstraction. The ext operator transports a substitution over one variable binder.

```
_ : ∀ (σ : Subst) (x : ℕ) → ⟪ σ ⟫ (` x) ≡ σ x
_ = λ σ x → refl
_ : ∀ (σ : Subst) (L M : Term) → ⟪ σ ⟫ (L · M) ≡ ⟪ σ ⟫ L · ⟪ σ ⟫ M
_ = λ σ L M → refl
_ : ∀ (σ : Subst) (N : Term) → ⟪ σ ⟫ (ƛ N) ≡ ƛ (⟪ ext σ ⟫ N)
_ = λ σ N → refl
```

The bracket notation $M[N]$ is defined to replace the occurrences of variable 0 in M with N and decrement the other free variables. For example,

```
_ : ∀ (N : Term) → (` 1 · ` 0) [ N ] ≡ (` 0 · N)
_ = λ N → refl
```

Most importantly, the ABT library provides the following theorem which is both difficult to prove and needed later for the Compatibility Lemma for lambda abstraction.

```
ext-sub-cons : ∀ {σ N V} → (⟪ ext σ ⟫ N) [ V ] ≡ ⟪ V • σ ⟫ N
ext-sub-cons = refl
```

Figure 1 defines the type system and reduction for the Cast Calculus. The two rules specific to gradual typing are collapse and collide. The collapse rule states that when an injected value encounters a matching projection, the result is the underlying value. The collide says that if the injection and projection do not match, the result is blame. The reason we introduce the M variable and the equation $M \equiv V\langle G! \rangle$ in those rules is that we ran into difficulties with Agda when doing case analysis on reductions. The same is true for the $\xi\xi$ and $\xi\xi$-blame rules. The $\xi\xi$ and $\xi\xi$-blame rules handle the usual congruence rules that are needed for reduction. The Frame type is a shallow non-recursive evaluation context, representing just a single term constructor with a hole. The notation $F[M]$ plugs term M into the hole inside F.

Figure 1 defines $M \Downarrow$ to mean that M reduces to a value, $M \Uparrow$ to mean M diverges, and $M \Uparrow \uplus blame$ to mean that M either diverges or reduces to blame. (We ran into difficulties with the alternate formulation of $(M \Uparrow) \uplus (M \longrightarrow^* \mathsf{blame})$ and could not prove them equivalent.)

3 Precision on Types and Terms

To talk about the gradual guarantee, we first define when one type is less precise than another one.[1]

```
infixr 6 _⊑_
data _⊑_ : Type → Type → Set where
  unk⊑unk : ⋆ ⊑ ⋆
  unk⊑ : ∀{G}{B} → ⌈ G ⌉ ⊑ B → ⋆ ⊑ B
  base⊑ : ∀{ι} → $_t ι ⊑ $_t ι
  fun⊑ : ∀{A B C D} → A ⊑ C → B ⊑ D → A ⇒ B ⊑ C ⇒ D
```

The first two rules for precision are usually presented as a single rule that says $\star$ is less precise than any type. Instead we have separated out the case for when both types are $\star$ from the case when only the less-precise type is $\star$. Also, for the rule unk⊑, instead of writing $B \neq \star$ we have written $\lceil G \rceil \sqsubseteq B$, which turns out to be important later when we define the logical relation and use recursion on the precision relation. The Prec type bundles two types in the precision relation. Precision is reflexive and transitive, but we will not need transitivity.

```
Prec : Set
Prec = (∃[ A ] ∃[ B ] A ⊑ B)

Refl⊑ : ∀{A} → A ⊑ A
```

[1] Some of the literature instead defines a less imprecise relation, but here we stick to the original direction of the relation [15].

```
infix 3 _⊢_:_
data _⊢_:_ : List Type → Term → Type → Set where
  ⊢‘ : ∀ {Γ x A} → Γ ∋ x : A → Γ ⊢ ‘ x : A
  ⊢$ : ∀ {Γ} (l : Lit) → Γ ⊢ $ l : $_t (typeof l)
  ⊢· : ∀ {Γ L M A B} → Γ ⊢ L : (A ⇒ B) → Γ ⊢ M : A → Γ ⊢ L · M : B
  ⊢ƛ : ∀ {Γ N A B} → (A :: Γ) ⊢ N : B → Γ ⊢ ƛ N : (A ⇒ B)
  ⊢⟨!⟩ : ∀{Γ M G} → Γ ⊢ M : ⌈ G ⌉ → Γ ⊢ M ⟨ G !⟩ : ⋆
  ⊢⟨?⟩ : ∀{Γ M} → Γ ⊢ M : ⋆ → (H : Ground) → Γ ⊢ M ⟨ H ?⟩ : ⌈ H ⌉
  ⊢blame : ∀{Γ A} → Γ ⊢ blame : A

infix 2 _⟶_
data _⟶_ : Term → Term → Set where
  β : ∀ {N W} → Value W → (ƛ N) · W ⟶ N [ W ]
  collapse : ∀{G M V} → Value V → M ≡ V ⟨ G !⟩ → M ⟨ G ?⟩ ⟶ V
  collide : ∀{G H M V} → Value V → G ≢ H → M ≡ V ⟨ G !⟩ → M ⟨ H ?⟩ ⟶ blame
  ξξ : ∀ {M N M′ N′} → (F : Frame)
    → M′ ≡ F [ M ] → N′ ≡ F [ N ] → M ⟶ N → M′ ⟶ N′
  ξξ-blame : ∀ {M′} → (F : Frame) → M′ ≡ F [ blame ] → M′ ⟶ blame

data _⟶*_ : Term → Term → Set where
  _END : (M : Term) → M ⟶* M
  _⟶⟨_⟩_ : (L : Term) {M N : Term} → L ⟶ M → M ⟶* N → L ⟶* N

len : ∀{M N : Term} → (M⟶N : M ⟶* N) → ℕ
len (_ END) = 0
len (_ ⟶⟨ step ⟩ mstep) = suc (len mstep)

_⇓ : Term → Set
M ⇓ = ∃[ V ] (M ⟶* V) × Value V
_⇑ : Term → Set
M ⇑ = ∀ k → ∃[ N ] Σ[ r ∈ M ⟶* N ] k ≡ len r
_⇑⊎blame : Term → Set
M ⇑⊎blame = ∀ k → ∃[ N ] Σ[ r ∈ M ⟶* N ] ((k ≡ len r) ⊎ (N ≡ blame))
```

Figure 1: Type System and Reduction for the Cast Calculus

```
infix 3 _⊩_⊑_:_
data _⊩_⊑_:_ : List Prec → Term → Term → ∀{A B : Type} → A ⊑ B → Set where
  ⊑-var : ∀ {Γ x A⊑B} → Γ ∋ x : A⊑B → Γ ⊩ (` x) ⊑ (` x) : proj₂ (proj₂ A⊑B)
  ⊑-lit : ∀ {Γ c} → Γ ⊩ ($ c) ⊑ ($ c) : base⊑{typeof c}
  ⊑-app : ∀{Γ L M L′ M′ A B C D}{c : A ⊑ C}{d : B ⊑ D}
    → Γ ⊩ L ⊑ L′ : fun⊑ c d → Γ ⊩ M ⊑ M′ : c
    → Γ ⊩ L · M ⊑ L′ · M′ : d
  ⊑-lam : ∀{Γ N N′ A B C D}{c : A ⊑ C}{d : B ⊑ D}
    → (A , C , c) :: Γ ⊩ N ⊑ N′ : d → Γ ⊩ ƛ N ⊑ ƛ N′ : fun⊑ c d
  ⊑-inj-L : ∀{Γ M M′}{G B}{c : ⌈ G ⌉ ⊑ B}
    → Γ ⊩ M ⊑ M′ : c  → Γ ⊩ M ⟨ G !⟩ ⊑ M′ : unk⊑{G}{B} c
  ⊑-inj-R : ∀{Γ M M′}{G}{c : ⋆ ⊑ ⌈ G ⌉}
    → Γ ⊩ M ⊑ M′ : c  → Γ ⊩ M ⊑ M′ ⟨ G !⟩ : unk⊑unk
  ⊑-proj-L : ∀{Γ M M′ H B}{c : ⌈ H ⌉ ⊑ B}
    → Γ ⊩ M ⊑ M′ : unk⊑ c → Γ ⊩ M ⟨ H ?⟩ ⊑ M′ : c
  ⊑-proj-R : ∀{Γ M M′ H}{c : ⋆ ⊑ ⌈ H ⌉}
    → Γ ⊩ M ⊑ M′ : unk⊑unk → Γ ⊩ M ⊑ M′ ⟨ H ?⟩ : c
  ⊑-blame : ∀{Γ M A} → map proj₁ Γ ⊢ M : A → Γ ⊩ M ⊑ blame : Refl⊑{A}
```

Figure 2: Precision Relation on Terms

```
Refl⊑ {⋆} = unk⊑unk
Refl⊑ {$ᵗ ι} = base⊑
Refl⊑ {A ⇒ B} = fun⊑ Refl⊑ Refl⊑
```

Figure 2 defines the precision relation on terms. The judgment has the form $\Gamma \Vdash M \sqsubseteq M' : c$ where Γ is a list of precision-related types and $c : A \sqsubseteq A'$ is a precision derivation for the types of M and M'. There are two rules for injection and also for projection, allowing either to appear on the left or right across from an arbitrary term. However, when injection is on the right, the term on the left must have type $\star$ (rule ⊑-inj-R). Similarly, when projection is on the right, the term on the left must have type $\star$ (rule ⊑-proj-R). The term blame is at least as precise as any term.

With precision defined, we are ready to discuss the gradual guarantee. It states that if M is less precise than M', then M and M' behave in a similar way, as defined below by the predicate gradual $M\,M'$. In particular, it says that if the more-precise term terminates or diverges, then the less-precise term does likewise. On the other hand the more-precise term may reduce to blame even though the less-precise term does not.

```
gradual : (M M′ : Term) → Set
gradual M M′ = (M′ ⇓ → M ⇓) × (M′ ⇑ → M ⇑) × (M ⇓ → M′ ⇓ ⊎ M′ ⟶* blame)
  × (M ⇑ → M′ ⇑⊎blame) × (M ⟶* blame → M′ ⟶* blame)
```

4 Step-Indexed Logic

The Step-Indexed Logic (SIL) library [14] is a shallow embedding of a modal logic into Agda. The formulas of this logic have type Set°, which is a record with three fields, the most important of which is named $\#$ and is a function from $\mathbb{N}$ to Set which expresses the meaning of the formula in Agda. Think of the $\mathbb{N}$ as a count-down clock, with smaller numbers representing later points in time. The other two fields of the record contain proofs of the LSLR invariants: (1) that the formula is true at 0, and (2) if the formula is true at some number, then it is true at all smaller numbers (monotonicity). Each of the constructors for SIL formulas proves these two properties, thereby saving the client of SIL from these tedious proofs.

SIL includes the connectives of first-order logic (conjunction, disjunction, existential and universal quantification, etc.). Each connective comes in two versions, one with a superscript "o" and another with superscript "s". The "o" version has type Set° whereas the "s" version has type $\mathsf{Set}^s\,\Gamma\,\Delta$, which we explain next. What makes SIL special is that it includes an operator μ° for defining recursive predicates. In the body of the μ°, de Bruijn index 0 refers to itself, that is, the entire μ°. However, variable 0 may only be used "later", that is, underneath at least one use of the modal operator $\triangleright^s$. The formula in the body of a μ° has type $\mathsf{Set}^s\,\Gamma\,\Delta$, where Γ is a list of types, one for each recursive predicate in scope (one can nest μ^s an arbitrary number of times).

The Δ records when each recursive predicate is used (now or later). It is represented by a list-like data structured indexed by Γ to ensure they have the same length, with $\emptyset$ as the empty list and cons to add an element to the front of a list. Set^s is a record whose field $\#$ is a function from a tuple of step-indexed predicates to Set°. (These tuples are structured like cons-lists with the always-true predicate tt^p playing the role of nil.) From the client's perspective, use the "s" connectives when writing formulas under a μ° and use the "o" connectives otherwise. During this work we found that the "s" versus "o" distinction created unnecessary complexity for the client and have developed a new version of the SIL (file `StepIndexedLogic2.lagda`) that has one version of each logical connective.

The majority of the lines of code in the SIL library are dedicated to proving the $\mathsf{fixpoint}^\circ$ theorem, which states that a recursive predicate is equivalent to one unrolling of itself. The proof of $\mathsf{fixpoint}^\circ$ is an adaptation of the fixpoint theorem of Appel and McAllester [2].

```
_ : ∀(A : Set) (P : A → Setˢ (A :: []) (cons Later ∅)) (a : A)
    → μ° P a ≡° # (P a) (μ° P , ttᵖ)
_ = λ A P a → fixpoint° P a
```

5 A Logical Relation for Precision

To define a logical relation for precision, we adapt the logical relation of New [11], which used explicit step indexing, into the Step-Indexed Logic. The logical relation has two directions (of type Dir): the $\preceq$ direction requires the more-precise term to simulate the less-precise term whereas the $\succeq$ direction requires the less-precise term to simulate the more-precise. logical relation consists of mutually-recursive relations on both terms and values. SIL does not directly support mutual recursion, but it can be expressed by combining the two relations into a single relation whose input is a disjoint sum. The formula for expressing membership in these recursive relations is verbose, so we define the below shorthands. Note that these shorthands are only intended for use inside the definition of the logical relation.

```
LR-type : Set
LR-type = (Prec × Dir × Term × Term) ⊎ (Prec × Dir × Term × Term)

LR-ctx : List Set
LR-ctx = LR-type :: []

_|_ ˢ⊑^LR_t_:_ : Dir → Term → Term → ∀{A}{A′} (c : A ⊑ A′) → Setˢ LR-ctx (cons Now ∅)
dir | M ˢ⊑^LR_t M′ : c = (inj₂ ((_ , _ , c) , dir , M , M′)) ∈ zeroˢ

_|_ ˢ⊑^LR_v_:_ : Dir → Term → Term → ∀{A}{A′} (c : A ⊑ A′) → Setˢ LR-ctx (cons Now ∅)
dir | V ˢ⊑^LR_v V′ : c = (inj₁ ((_ , _ , c) , dir , V , V′)) ∈ zeroˢ
```

The logical relation is defined in Figure 3 and explained in the following paragraphs. The definition of the logical relation for terms is based on the requirements of the gradual guarantee, but it only talks about one step at a time of the term begin simulated. In the $\preceq$ direction, the first clause says that the less-precise M takes a step to N and that N is related to M' at one tick later in time. The second clause allows the more-precise M' to reduce to an error. The third clause says that the less-precise M is already a value, and requires M' to reduce to a value that is related at the current time to M. The other direction $\succeq$ is defined in a similar way, but with the more precise term M' taking one step at a time.

The definition of the logical relation for values is by recursion on the precision relation and by cases on the values and their types. When both values are of the same base type (base $\sqsubseteq$), they are related if they are identical constants. When the values are of function type (fun $\sqsubseteq$), then they are related if they are both lambda abstractions that, when later applied to related arguments, behave in a related way. When the values are both of unkown type (unk $\sqsubseteq$ unk), then they are related if they are both injections from the same ground type and the underlying values are related one step later. If the less-precise value is of unknown type but the more-precise value is not (unk $\sqsubseteq$), then they are related if (1) the less-precise value is an injection and (2) the ground type of the injection is less-precise than the type of the more-precise value. Furthermore, for direction $\preceq$, (3a) the underlying value of the injection is related one step later to the more-precise value. For direction $\succeq$, (3b) the underlying value of the injection is related now to the more-precise value. Note that the recursive call to $\mathsf{LR_v}$ is fine from a termination perspective because argument d is a subterm of unk $\sqsubseteq d$. This is why the unk $\sqsubseteq$ rule needs to be recursive, with the premise $\lceil G \rceil \sqsubseteq B$.

The following definitions combine the LR_v and LR_t functions into a single function, pre-$\mathrm{LR}_t \uplus \mathrm{LR}_v$, and than applies the μ° operator to produce the recursive relation $\mathrm{LR}_t \uplus \mathrm{LR}_v$. Finally, we define some shorthand for the logical relation on values, written $\sqsubseteq^{LR}{}_v$, and the logical relation on terms, $\sqsubseteq^{LR}{}_t$.

```
pre-LR_t⊎LR_v : LR-type → Setˢ LR-ctx (cons Later ∅)
pre-LR_t⊎LR_v (inj₁ (c , dir , V , V′)) = LR_v c dir V V′
pre-LR_t⊎LR_v (inj₂ (c , dir , M , M′)) = LR_t c dir M M′

LR_t⊎LR_v : LR-type → Set°
LR_t⊎LR_v X = μ° pre-LR_t⊎LR_v X

_|_⊑^LR_v_:_ : Dir → Term → Term → ∀{A A′} → A ⊑ A′ → Set°
dir | V ⊑^LR_v V′ : A⊑A′ = LR_t⊎LR_v (inj₁ ((_ , _ , A⊑A′) , dir , V , V′))

_|_⊑^LR_t_:_ : Dir → Term → Term → ∀{A A′} → A ⊑ A′ → Set°
dir | M ⊑^LR_t M′ : A⊑A′ = LR_t⊎LR_v (inj₂ ((_ , _ , A⊑A′) , dir , M , M′))
```

$$
\begin{aligned}
&\mathsf{LR}_t : \mathsf{Prec} \to \mathsf{Dir} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Set}^s\ \mathsf{LR\text{-}ctx}\ (\mathsf{cons}\ \mathsf{Later}\ \emptyset)\\
&\mathsf{LR}_v : \mathsf{Prec} \to \mathsf{Dir} \to \mathsf{Term} \to \mathsf{Term} \to \mathsf{Set}^s\ \mathsf{LR\text{-}ctx}\ (\mathsf{cons}\ \mathsf{Later}\ \emptyset)\\
\\
&\mathsf{LR}_t\ (A\ ,\ A'\ ,\ c) \preceq M\ M' =\\
&\quad (\exists^s[\ N\ ]\ (M \longrightarrow N)^s \times^s \triangleright^s\ (\preceq \mid N\ {}^s\sqsubseteq^{LR}_t\ M' : c))\\
&\quad \uplus^s\ (M' \longrightarrow^* \mathsf{blame})^s\\
&\quad \uplus^s\ ((\mathsf{Value}\ M)^s \times^s (\exists^s[\ V'\ ]\ (M' \longrightarrow^* V')^s \times^s (\mathsf{Value}\ V')^s \times^s (\mathsf{LR}_v\ (\_\ ,\ \_\ ,\ c) \preceq M\ V')))\\
&\mathsf{LR}_t\ (A\ ,\ A'\ ,\ c) \succeq M\ M' =\\
&\quad (\exists^s[\ N'\ ]\ (M' \longrightarrow N')^s \times^s \triangleright^s\ (\succeq \mid M\ {}^s\sqsubseteq^{LR}_t\ N' : c))\\
&\quad \uplus^s\ (\mathsf{Blame}\ M')^s\\
&\quad \uplus^s\ ((\mathsf{Value}\ M')^s \times^s (\exists^s[\ V\ ]\ (M \longrightarrow^* V)^s \times^s (\mathsf{Value}\ V)^s \times^s (\mathsf{LR}_v\ (\_\ ,\ \_\ ,\ c) \succeq V\ M')))\\
\\
&\mathsf{LR}_v\ (.(\$_t\ \iota)\ ,\ .(\$_t\ \iota)\ ,\ \mathsf{base}\sqsubseteq\{\iota\})\ dir\ (\$\ c)\ (\$\ c') = (c \equiv c')^s\\
&\mathsf{LR}_v\ (.(\$_t\ \iota)\ ,\ .(\$_t\ \iota)\ ,\ \mathsf{base}\sqsubseteq\{\iota\})\ dir\ V\ V' = \bot^s\\
&\mathsf{LR}_v\ (.(A \Rightarrow B)\ ,\ .(A' \Rightarrow B')\ ,\ \mathsf{fun}\sqsubseteq\{A\}\{B\}\{A'\}\{B'\}\ A\sqsubseteq A'\ B\sqsubseteq B')\ dir\ (\lambda\ N)(\lambda\ N') =\\
&\qquad \forall^s[\ W\ ]\ \forall^s[\ W'\ ]\ \triangleright^s\ (dir \mid W\ {}^s\sqsubseteq^{LR}_v\ W' : A\sqsubseteq A')\\
&\qquad\qquad \to^s \triangleright^s\ (dir \mid (N\ [\ W\ ])\ {}^s\sqsubseteq^{LR}_t\ (N'\ [\ W'\ ]) : B\sqsubseteq B')\\
&\mathsf{LR}_v\ (.(A \Rightarrow B)\ ,\ .(A' \Rightarrow B')\ ,\ \mathsf{fun}\sqsubseteq\{A\}\{B\}\{A'\}\{B'\}\ A\sqsubseteq A'\ B\sqsubseteq B')\ dir\ V\ V' = \bot^s\\
&\mathsf{LR}_v\ (.\star\ ,\ .\star\ ,\ \mathsf{unk}\sqsubseteq\mathsf{unk})\ dir\ (V\ \langle\ G\ !\rangle)\ (V'\ \langle\ H\ !\rangle)\\
&\quad \mathsf{with}\ G \equiv^g H\\
&\ldots \mid \mathsf{yes}\ \mathsf{refl} = (\mathsf{Value}\ V)^s \times^s (\mathsf{Value}\ V')^s \times^s (\triangleright^s\ (dir \mid V\ {}^s\sqsubseteq^{LR}_v\ V' : \mathsf{Refl}\sqsubseteq\{\lceil\ G\ \rceil\}))\\
&\ldots \mid \mathsf{no}\ neq = \bot^s\\
&\mathsf{LR}_v\ (.\star\ ,\ .\star\ ,\ \mathsf{unk}\sqsubseteq\mathsf{unk})\ dir\ V\ V' = \bot^s\\
&\mathsf{LR}_v\ (.\star\ ,\ .A'\ ,\ \mathsf{unk}\sqsubseteq\{H\}\{A'\}\ d) \preceq (V\ \langle\ G\ !\rangle)\ V'\\
&\quad \mathsf{with}\ G \equiv^g H\\
&\ldots \mid \mathsf{yes}\ \mathsf{refl} = (\mathsf{Value}\ V)^s \times^s (\mathsf{Value}\ V')^s \times^s \triangleright^s\ (\preceq \mid V\ {}^s\sqsubseteq^{LR}_v\ V' : d)\\
&\ldots \mid \mathsf{no}\ neq = \bot^s\\
&\mathsf{LR}_v\ (.\star\ ,\ .A'\ ,\ \mathsf{unk}\sqsubseteq\{H\}\{A'\}\ d) \succeq (V\ \langle\ G\ !\rangle)\ V'\\
&\quad \mathsf{with}\ G \equiv^g H\\
&\ldots \mid \mathsf{yes}\ \mathsf{refl} = (\mathsf{Value}\ V)^s \times^s (\mathsf{Value}\ V')^s \times^s (\mathsf{LR}_v\ (\lceil\ G\ \rceil\ ,\ A'\ ,\ d) \succeq V\ V')\\
&\ldots \mid \mathsf{no}\ neq = \bot^s\\
&\mathsf{LR}_v\ (\star\ ,\ .A'\ ,\ \mathsf{unk}\sqsubseteq\{H\}\{A'\}\ d)\ dir\ V\ V' = \bot^s
\end{aligned}
$$

Figure 3: Logical Relation for Precision on Terms LR_t and Values LR_v

```
_⊑^LR_t_:_ : Term → Term → ∀{A A′} → A ⊑ A′ → Set°
M ⊑^LR_t M′ : A⊑A′ = (⊑ | M ⊑^LR_t M′ : A⊑A′) ×° (⊒ | M ⊑^LR_t M′ : A⊑A′)
```

The relations that we have defined so far, $\sqsubseteq^{LR}_v$ and $\sqsubseteq^{LR}_t$, only apply to closed terms, that is, terms with no free variables. We also need to relate open terms. The standard way to do that is to apply two substitutions to the two terms, replacing each free variable with related values. We relate a pair of substitutions γ and γ' with the following definition, which says that the substitutions must be point-wise related using the logical relation for values.

```
_|_⊨_⊑^LR_ : (Γ : List Prec) → Dir → Subst → Subst → List Set°
[] | dir ⊨ γ ⊑^LR γ′ = []
((_ , _ , A⊑A′) :: Γ) | dir ⊨ γ ⊑^LR γ′ =
    (dir | (γ 0) ⊑^LR_v (γ′ 0) : A⊑A′) :: (Γ | dir ⊨ (λ x → γ (suc x)) ⊑^LR (λ x → γ′ (suc x)))
```

We then define two open terms M and M' to be logically related if there are a pair of related substitutions γ and γ' such that applying them to M and M' produces related terms.

```
_|_⊨_⊑^LR_:_ : List Prec → Dir → Term → Term → Prec → Set
Γ | dir ⊨ M ⊑^LR M′ : (_ , _ , A⊑A′) = ∀ (γ γ′ : Subst)
    → (Γ | dir ⊨ γ ⊑^LR γ′) ⊢° dir | (⟪ γ ⟫ M) ⊑^LR_t (⟪ γ′ ⟫ M′) : A⊑A′
```

We use the following notation for the conjunction of the two directions and define the proj function for accessing each direction.

```
_⊨_⊑^LR_:_ : List Prec → Term → Term → Prec → Set
Γ ⊨ M ⊑^LR M′ : c = (Γ | ⊑ ⊨ M ⊑^LR M′ : c) × (Γ | ⊒ ⊨ M ⊑^LR M′ : c)

proj : ∀ {Γ}{c} → (dir : Dir) → (M M′ : Term) → Γ ⊨ M ⊑^LR M′ : c → Γ | dir ⊨ M ⊑^LR M′ : c
proj ⊑ M M′ M⊑M′ = proj₁ M⊑M′
proj ⊒ M M′ M⊑M′ = proj₂ M⊑M′
```

The definition of $\sqsubseteq^{LR}_v$ includes several clauses that ensured that the related values are indeed syntactic values. Here we make use of that to prove that indeed, logically related values are syntactic values.

```
LR_v⇒Value° : ∀ {dir}{A}{A′}{𝒫} (A⊑A′ : A ⊑ A′) M M′
  → 𝒫 ⊢° (dir | M ⊑^LR_v M′ : A⊑A′) → 𝒫 ⊢° (Value M)° ×° (Value M′)°
```

If two values are related via $\sqsubseteq^{LR}_v$, then they are also related via $\sqsubseteq^{LR}_t$.

```
LR_v⇒LR_t : ∀{A}{A′}{A⊑A′ : A ⊑ A′}{𝒫}{V V′}{dir}
  → 𝒫 ⊢° dir | V ⊑^LR_v V′ : A⊑A′ → 𝒫 ⊢° dir | V ⊑^LR_t V′ : A⊑A′
```

6 Fundamental Theorem of the Logical Relation

The fundamental theorem of the logical relation states that if two terms are related by precision, then they are in the logical relation. The fundamental theorem is proved by induction on the term precision relation. Each case of that proof is split out into a separate lemma, which by tradition are called Compatibility Lemmas.

Compatibility for Literals, Blame, and Variables The proof of compatibility for literals uses the $\mathrm{LR}_v{\Rightarrow}\mathrm{LR}_t$ lemma.

```
compatible-literal : ∀{Γ}{c}{ι} → Γ ⊨ $ c ⊑LR $ c : ($t ι , $t ι , base⊑)
```

The proof of compatibility for blame is direct from the definitions.

```
compatible-blame : ∀{Γ}{A}{M} → map proj₁ Γ ⊢ M : A → Γ ⊨ M ⊑LR blame : (A , A , Refl⊑)
```

The proof of compatibility for variables relies on the following lemma regarding related substitutions.

```
lookup-⊑LR : ∀{dir} (Γ : List Prec) → (γ γ′ : Subst) → ∀ {A}{A′}{A⊑A′}{x}
  → Γ ∋ x : (A , A′ , A⊑A′) → (Γ | dir ⊨ γ ⊑LR γ′) ⊢° dir | γ x ⊑LRv γ′ x : A⊑A′
```

```
compatibility-var : ∀ {Γ A A′ A⊑A′ x} → Γ ∋ x : (A , A′ , A⊑A′)
  → Γ ⊨ ‘ x ⊑LR ‘ x : (A , A′ , A⊑A′)
```

Compatibility for Lambda The proof of compatibility for lambda abstraction has a premise that says the bodies of the two lambdas are in the logical relation, which is the induction hypothesis in this case of the fundamental theorem. The logical relation for lambda requires us to prove

$$\mathscr{P} \vdash^\circ (\mathsf{dir} \mid (\langle\!\langle\ \mathsf{ext}\ \gamma\ \rangle\!\rangle\ \mathsf{N})\ [\ \mathsf{W}\ ] \sqsubseteq^{LR}_t (\langle\!\langle\ \mathsf{ext}\ \gamma'\ \rangle\!\rangle\ \mathsf{N}')\ [\ \mathsf{W}'\ ] : \mathsf{d})$$

Using the premise we obtain

$$\mathscr{P} \vdash^\circ (\mathsf{dir} \mid \langle\!\langle\ \mathsf{W} \bullet \gamma\ \rangle\!\rangle\ \mathsf{N} \sqsubseteq^{LR}_t \langle\!\langle\ \mathsf{W}' \bullet \gamma'\ \rangle\!\rangle\ \mathsf{N}' : \mathsf{d})$$

which is equivalent to what is required thanks to the ext-sub-cons theorem from the ABT library. As an example of a proof using SIL, here is the proof in full of compatibility for lambda.

```
compatible-lambda : ∀{Γ : List Prec}{A}{B}{C}{D}{N N′ : Term}{c : A ⊑ C}{d : B ⊑ D}
  → ((A , C , c) :: Γ) ⊨ N ⊑LR N′ : (B , D , d)
  → Γ ⊨ (ƛ N) ⊑LR (ƛ N′) : (A ⇒ B , C ⇒ D , fun⊑ c d)
compatible-lambda{Γ}{A}{B}{C}{D}{N}{N′}{c}{d} ⊨N⊑N′ =
  (λ γ γ′ → ⊢ℰƛNƛN′) , (λ γ γ′ → ⊢ℰƛNƛN′)
  where ⊢ℰƛNƛN′ : ∀{dir}{γ}{γ′} → (Γ | dir ⊨ γ ⊑LR γ′)
                    ⊢° (dir | ⟪ γ ⟫ (ƛ N) ⊑LRt ⟪ γ′ ⟫ (ƛ N′) : fun⊑ c d)
        ⊢ℰƛNƛN′ {dir}{γ}{γ′} = LRv⇒LRt (subst° (≡°-sym LRv-fun)
          (Λ°[ W ] Λ°[ W′ ] →°I {P = ▷° (dir | W ⊑LRv W′ : c)}
            let IH = (proj dir N N′ ⊨N⊑N′) (W • γ) (W′ • γ′) in
            (app° (S° (▷→ (mono° (→°I IH)))) Z°)))
```

We note that the use of SIL in the above proof comes with some tradeoffs. On the one hand, there is no explicit reasoning about step indices. On the other hand, there is some added verbosity compared to a proof in raw Agda such as the use of app° for modus-ponens, the use of de Bruijn indices Z° to refer to premises, and extra annotations such as {P = ▷° (dir | W ⊑LRv W′ : c)} that are needed when Agda's type inference fails.

However, there is a bigger problem regarding incremental proof development in SIL. It is common practice to create a partial proof with a hole, written ?, and one can ask Agda to print what need to be

proved in the hole. For example, instead of ($\rightarrow^\circ$I IH) in the above proof, one might have started with ($\rightarrow^\circ$I ?). Unfortunately, Agda's message describing what needs to be proved fills an entire computer screen because Agda normalizes the SIL formulas into their Agda encodings. We are working on new version of SIL that uses the `abstract` feature of Agda to hide the internals of SIL from its clients, but that also has its challenges. It seems that the `prop` extension is necessary so that the fields of Set$^\circ$ that contain proofs are ignored when proving equations such as fixpoint$^\circ$.

Anti-reduction and Bind Lemmas The remaining compatibility lemmas, for function application and for injections and projections, require several anti-reduction lemmas which state that if two terms are in the logical relation, then walking backwards with one or both of them yields terms that are still in the logical relation. We formulated these lemmas with explicit step indices and the meaning function # because working with the raw Agda encoding enables the use of Agda's Auto command for automatic proof search.

$$\begin{aligned}&\text{anti-reduction-}\preceq\text{-R-one} : \forall\{A\}\{A'\}\{c : A \sqsubseteq A'\}\{M\}\{M'\}\{N'\}\{i\}\\ &\quad \rightarrow \#(\preceq \mid M \sqsubseteq^{LR}_t N' : c)\ i \rightarrow M' \longrightarrow N' \rightarrow \#(\preceq \mid M \sqsubseteq^{LR}_t M' : c)\ i\end{aligned}$$

$$\begin{aligned}&\text{anti-reduction-}\preceq\text{-R} : \forall\{A\}\{A'\}\{c : A \sqsubseteq A'\}\{M\}\{M'\}\{N'\}\{i\}\\ &\quad \rightarrow \#(\preceq \mid M \sqsubseteq^{LR}_t N' : c)\ i \rightarrow M' \longrightarrow^* N' \rightarrow \#(\preceq \mid M \sqsubseteq^{LR}_t M' : c)\ i\end{aligned}$$

$$\begin{aligned}&\text{anti-reduction-}\succeq\text{-R-one} : \forall\{A\}\{A'\}\{c : A \sqsubseteq A'\}\{M\}\{M'\}\{N'\}\{i\}\\ &\quad \rightarrow \#(\succeq \mid M \sqsubseteq^{LR}_t N' : c)\ i \rightarrow M' \longrightarrow N' \rightarrow \#(\succeq \mid M \sqsubseteq^{LR}_t M' : c)\ (\mathsf{suc}\ i)\end{aligned}$$

$$\begin{aligned}&\text{anti-reduction-}\succeq\text{-L-one} : \forall\{A\}\{A'\}\{c : A \sqsubseteq A'\}\{M\}\{N\}\{M'\}\{i\}\\ &\quad \rightarrow \#(\succeq \mid N \sqsubseteq^{LR}_t M' : c)\ i \rightarrow M \longrightarrow N \rightarrow \#(\succeq \mid M \sqsubseteq^{LR}_t M' : c)\ i\end{aligned}$$

$$\begin{aligned}&\text{anti-reduction-}\succeq\text{-L} : \forall\{A\}\{A'\}\{c : A \sqsubseteq A'\}\{M\}\{N\}\{M'\}\{i\}\\ &\quad \rightarrow \#(\succeq \mid N \sqsubseteq^{LR}_t M' : c)\ i \rightarrow M \longrightarrow^* N \rightarrow \#(\succeq \mid M \sqsubseteq^{LR}_t M' : c)\ i\end{aligned}$$

$$\begin{aligned}&\text{anti-reduction-}\preceq\text{-L-one} : \forall\{A\}\{A'\}\{c : A \sqsubseteq A'\}\{M\}\{N\}\{M'\}\{i\}\\ &\quad \rightarrow \#(\preceq \mid N \sqsubseteq^{LR}_t M' : c)\ i \rightarrow M \longrightarrow N \rightarrow \#(\preceq \mid M \sqsubseteq^{LR}_t M' : c)\ (\mathsf{suc}\ i)\end{aligned}$$

$$\begin{aligned}&\text{anti-reduction} : \forall\{A\}\{A'\}\{c : A \sqsubseteq A'\}\{M\}\{N\}\{M'\}\{N'\}\{i\}\{dir\}\\ &\quad \rightarrow \#(dir \mid N \sqsubseteq^{LR}_t N' : c)\ i \rightarrow M \longrightarrow N \rightarrow M' \longrightarrow N'\\ &\quad \rightarrow \#(dir \mid M \sqsubseteq^{LR}_t M' : c)\ (\mathsf{suc}\ i)\end{aligned}$$

The remaining compatibility lemmas all involve language features with subexpressions, and one needs to reason about the reduction of those subexpressions to values. The following LR_t-bind lemma performs that reasoning once and for all. It says that if you are trying to prove that $N \sqsubseteq^{LR}_t N'$, if M is a direct subexpression of N and M' is a direct subexpression of N', so $N = F[\![M]\!]$ and $N' = F'[\![M']\!]$ (F and F' are non-empty frames), then one can replace M and M' with any related values $V \sqsubseteq^{LR}_v V'$ and it suffices prove $F[\![V]\!] \sqsubseteq^{LR}_t F'[\![V']\!]$. The proof of the LR_t-bind lemma relies on two of the anti-reduction lemmas.

$$\begin{aligned}&\mathsf{LR}_t\text{-bind} : \forall\{B\}\{B'\}\{c : B \sqsubseteq B'\}\{A\}\{A'\}\{d : A \sqsubseteq A'\}\{F\}\{F'\}\{M\}\{M'\}\{i\}\{dir\}\\ &\quad\to \#(dir \mid M \sqsubseteq^{LR}_t M' : d)\ i\\ &\quad\to (\forall\, j\ V\ V' \to j \le i \to M \longrightarrow^* V \to \mathsf{Value}\ V \to M' \longrightarrow^* V' \to \mathsf{Value}\ V'\\ &\qquad\to \#(dir \mid V \sqsubseteq^{LR}_v V' : d)\ j \to \#(dir \mid (F \llbracket V \rrbracket) \sqsubseteq^{LR}_t (F' \llbracket V' \rrbracket) : c)\ j)\\ &\quad\to \#(dir \mid (F \llbracket M \rrbracket) \sqsubseteq^{LR}_t (F' \llbracket M' \rrbracket) : c)\ i\end{aligned}$$

Compatibility for Application Here is where the logical relation demonstrates its worth. Using the LR_t-bind lemma twice, we go from needing to prove $L \cdot M \sqsubseteq^{LR}_t L' \cdot M'$ to $V \cdot W \sqsubseteq^{LR}_t V' \cdot W'$. Then because $V \sqsubseteq^{LR}_v V'$ at function type, the logical relation tells us that $V = \lambda N$, $V' = \lambda N'$, and $\mathsf{N}[\mathsf{W}] \sqsubseteq^{LR}_t \mathsf{N}'[\mathsf{W}']$ at one step later in time. So we back up one step of β-reduction using anti-reduction to show that $(\lambda\mathsf{N}) \cdot \mathsf{W} \sqsubseteq^{LR}_t (\lambda\mathsf{N}') \cdot \mathsf{W}'$.

$$\begin{aligned}&\mathsf{compatible\text{-}app} : \forall\{\Gamma\}\{A\ A'\ B\ B'\}\{c : A \sqsubseteq A'\}\{d : B \sqsubseteq B'\}\{L\ L'\ M\ M'\}\\ &\quad\to \Gamma \vDash L \sqsubseteq^{LR} L' : (A \Rightarrow B\ ,\ A' \Rightarrow B'\ ,\ \mathsf{fun}\sqsubseteq\ c\ d) \to \Gamma \vDash M \sqsubseteq^{LR} M' : (A\ ,\ A'\ ,\ c)\\ &\quad\to \Gamma \vDash L \cdot M \sqsubseteq^{LR} L' \cdot M' : (B\ ,\ B'\ ,\ d)\end{aligned}$$

Compatibility for Injections We have two cases to deal with, the injection can be on the left or the right. Starting with a projection on the left, LR_t-bind takes us from need to prove $M\langle G!\rangle \sqsubseteq^{LR} M'$ to needing $V\langle G!\rangle \sqsubseteq^{LR} V'$, assuming $V \sqsubseteq^{LR} V'$. We proceed by case analysis on the direction ($\preceq$ or $\succeq$). For $\preceq$, we need to prove $\triangleright^\circ(V \sqsubseteq^{LR}_v V')$, which follows from $V \sqsubseteq^{LR} V'$ by monotonicity. For $\succeq$, we just need to prove $V \sqsubseteq^{LR} V'$, which we have by the assumption.

$$\begin{aligned}&\mathsf{compatible\text{-}inj\text{-}L} : \forall\{\Gamma\}\{G\ A'\}\{c : \lceil\ G\ \rceil \sqsubseteq A'\}\{M\ M'\}\\ &\quad\to \Gamma \vDash M \sqsubseteq^{LR} M' : (\lceil\ G\ \rceil\ ,\ A'\ ,\ c)\\ &\quad\to \Gamma \vDash M\ \langle\ G\ !\rangle \sqsubseteq^{LR} M' : (\star\ ,\ A'\ ,\ \mathsf{unk}\sqsubseteq\{G\}\{A'\}\ c)\end{aligned}$$

Next consider when the injection is on the right. The LR_t-bind lemma takes us from needing to prove $M \sqsubseteq^{LR} M'\langle G!\rangle$ to needing $V \sqsubseteq^{LR} V'\langle G!\rangle$ where $V \sqsubseteq^{LR}_v V'$. We know that V is of type $\star$ (rule $\sqsubseteq$-inj-R) so $V = W\langle G!\rangle$ and $W \sqsubseteq^{LR}_v V'$ (the $\mathsf{unk}\sqsubseteq$ clause of LR_v). So we conclude that $W\langle G!\rangle \sqsubseteq^{LR} V'\langle G!\rangle$ by the $\mathsf{unk}\sqsubseteq\mathsf{unk}$ clause of LR_v.

$$\begin{aligned}&\mathsf{compatible\text{-}inj\text{-}R} : \forall\{\Gamma\}\{G\}\{c : \star \sqsubseteq \lceil\ G\ \rceil\ \}\{M\ M'\}\\ &\quad\to \Gamma \vDash M \sqsubseteq^{LR} M' : (\star\ ,\ \lceil\ G\ \rceil\ ,\ c)\\ &\quad\to \Gamma \vDash M \sqsubseteq^{LR} M'\ \langle\ G\ !\rangle : (\star\ ,\ \star\ ,\ \mathsf{unk}\sqsubseteq\mathsf{unk})\end{aligned}$$

Compatibility for Projections We can have a projection on the left (rule $\sqsubseteq$-proj-L) or the right (rule $\sqsubseteq$-proj-R). Starting on the left, LR_t-bind changes the goal to $V\langle H?\rangle \sqsubseteq^{LR}_t V'$ assuming that $V \sqsubseteq^{LR} V'$. We need to ensure that $V\langle H?\rangle$ reduces to a value without error. Thankfully, $\sqsubseteq$-proj-L says the types of V and V' are related by $\mathsf{unk}\sqsubseteq\ \mathsf{c}$ with $c : H \sqsubseteq A'$, and that clause of LR_v tells us that $V = W\langle H!\rangle$ and $W \sqsubseteq^{LR}_v V'$. So $W\langle H!\rangle\langle H?\rangle \longrightarrow W$ and by anti-reduction we conclude that $W\langle H!\rangle\langle H?\rangle \sqsubseteq^{LR}_t V'$.

$$\begin{aligned}&\mathsf{compatible\text{-}proj\text{-}L} : \forall\{\Gamma\}\{H\}\{A'\}\{c : \lceil\ H\ \rceil \sqsubseteq A'\}\{M\}\{M'\}\\ &\quad\to \Gamma \vDash M \sqsubseteq^{LR} M' : (\star\ ,\ A'\ ,\ \mathsf{unk}\sqsubseteq\ c)\\ &\quad\to \Gamma \vDash M\ \langle\ H\ ?\rangle \sqsubseteq^{LR} M' : (\lceil\ H\ \rceil\ ,\ A'\ ,\ c)\end{aligned}$$

When the projection is on the right, there is less to worry about. LR$_t$-bind changes the goal to $V \sqsubseteq^{LR}_t V'\langle H?\rangle$ assuming that $V \sqsubseteq^{LR}_v V'$. We have $V' = W'\langle G!\rangle$ and $V \sqsubseteq^{LR}_v W'$. If $G = H$ then $W'\langle G!\rangle\langle H?\rangle \longrightarrow W'$ and by anti-reduction, $V \sqsubseteq^{LR}_t W'\langle G!\rangle\langle H?\rangle$. If $G \neq H$, then $W'\langle G!\rangle\langle H?\rangle \longrightarrow$ blame. Since $V \sqsubseteq^{LR}_t$ blame, we use anti-reduction to conclude $V \sqsubseteq^{LR}_t W'\langle G!\rangle\langle H?\rangle$.

```
compatible-proj-R : ∀{Γ}{H}{c : ⋆ ⊑ ⌈ H ⌉}{M}{M′}
  → Γ ⊨ M ⊑^LR M′ : (⋆ , ⋆ , unk⊑unk)
  → Γ ⊨ M ⊑^LR M′ ⟨ H ?⟩ : (⋆ , ⌈ H ⌉ , c)
```

Fundamental Theorem The proof is by induction on $M \sqsubseteq M'$, using the appropriate Compatibility Lemma in each case.

```
fundamental : ∀ {Γ}{A}{A′}{A⊑A′ : A ⊑ A′} → (M M′ : Term)
  → Γ ⊩ M ⊑ M′ : A⊑A′ → Γ ⊨ M ⊑^LR M′ : (A , A′ , A⊑A′)
```

7 Proof of the Gradual Guarantee

The next step in the proof of the gradual guarantee is to connect the logical relation to the behavior that's required by the gradual guarantee. (Recall the gradual predicate defined in Section 3.) The proof goes through an intermediate step that uses the following notion of semantic approximation for a fixed number of reduction steps.

```
_⊨_⊑_for_ : Dir → Term → Term → ℕ → Set
⊒ ⊨ M ⊑ M′ for k = (M ⇓ × M′ ⇓) ⊎ (M′ —→* blame) ⊎ ∃[ N ] Σ[ r ∈ M —→* N ] len r ≡ k
⊑ ⊨ M ⊑ M′ for k = (M ⇓ × M′ ⇓) ⊎ (M′ —→* blame) ⊎ ∃[ N′ ] Σ[ r ∈ M′ —→* N′ ] len r ≡ k

⊨_⊑_for_ : Term → Term → ℕ → Set
⊨ M ⊑ M′ for k = (⊒ ⊨ M ⊑ M′ for k) × (⊑ ⊨ M ⊑ M′ for k)
```

The proof that the logical relation implies semantic approximation is a straightforward induction on the step index k.

```
LR⇒sem-approx : ∀{A}{A′}{A⊑A′ : A ⊑ A′}{M}{M′}{k}{dir}
  → #(dir | M ⊑^LR_t M′ : A⊑A′) (suc k) → dir ⊨ M ⊑ M′ for k
```

The proof that semantic approximation implies the gradual guarantee relies on a proof of determinism for the Cast Calculus.

```
sem-approx⇒GG : ∀{A}{A′}{A⊑A′ : A ⊑ A′}{M}{M′}
  → (∀ k → ⊨ M ⊑ M′ for k) → gradual M M′
```

We put these two proofs together to show that the logical relation implies the gradual guarantee.

```
LR⇒GG : ∀{A}{A′}{A⊑A′ : A ⊑ A′}{M}{M′} → [] ⊢° M ⊑^LR_t M′ : A⊑A′ → gradual M M′
```

The gradual guarantee then follows by use of the Fundamental Lemma to obtain $M \sqsubseteq^{LR}_t M'$ and then LR⇒GG to conclude that gradual M M′.

```
gradual-guarantee : ∀ {A}{A′}{A⊑A′ : A ⊑ A′} → (M M′ : Term)
  → [] ⊩ M ⊑ M′ : A⊑A′ → gradual M M′
gradual-guarantee {A}{A′}{A⊑A′} M M′ M⊑M′ =
  let (⊨≼M⊑LRM′ , ⊨≽M⊑LRM′) = fundamental M M′ M⊑M′ in
  LR⇒GG (⊨≼M⊑LRM′ id id , ⊨≽M⊑LRM′ id id)
```

8 Conclusion and Acknowledgments

This paper presented the first mechanized proof of the gradual guarantee using step-indexed logical relations. One naturally wonders how using step-indexed logical relations compares to a simulation-based proof. One rough comparison is the number of lines of code in Agda. Wadler, Thiemann, and I developed a simulation-based proof of the gradual guarantee for a similar cast calculus in Agda, which came in at 3,200 LOC of which 232 lines are proofs about substitution (equivalent to what is provided in the ABT library). The logical-relations proof presented here was somewhat shorter, at 2,300 LOC, though it makes use of the Abstract Binding Tree Library (900 LOC) and the Step-Indexed Logic Library (2100 LOC). These LOC numbers confirm my feeling that the total effort to create the SIL and ABT libraries and prove the gradual guarantee via logical relations was higher than to prove the gradual guarantee via simulation. However, if one discounts the SIL and ABT libraries because they are reusable and language independent, then the remaining effort to prove the gradual guarantee via logical relations was lower than via simulation.

As mentioned at various points in this paper, there are some rough edges to the SIL and ABT libraries, primarily due to challenges regarding leaky abstractions. For SIL, we mentioned how Agda's output shows normalized versions of the SIL formulas, which exposes the underlying encodings and are too large to be readable. We have created a new version of SIL that uses Agda's `abstract` feature and look forward to updating the proof of the gradual guarantee to use the new version of SIL. Regarding the ABT library, there are also challenges regarding (1) Agda output not always using the concise pattern syntax and (2) Agda's automated case splitting does not work for ABT-generated languages.

This work was conducted in the context of a collaboration with Peter Thiemann and Philip Wadler where we have been exploring how to mechanize blame calculi in Agda and study the polymorphic blame calculus. My understanding of step-indexed logical relations was improved by reading Peter Thiemann's proof in Agda of type safety for a typed λ-calculus with fix (for defining recursive function) using step-indexed logical relations. The initiative to build an Agda version of the LSLR logic came from Philip Wadler.

References

[1] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek & Philip Wadler (2011): *Blame for All*. In: *Symposium on Principles of Programming Languages*.

[2] Andrew W. Appel & David McAllester (2001): *An Indexed Model of Recursive Types for Foundational Proof-carrying Code*. *ACM Trans. Program. Lang. Syst.* 23(5), pp. 657–683.

[3] Derek Dreyer, Amal Ahmed & Lars Birkedal (2011): *Logical Step-Indexed Logical Relations*. *Logical Methods in Computer Science* 7.

[4] Professor Robert Harper (2012): *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA.

[5] Koen Jacobs, Amin Timany & Dominique Devriese (2021): *Fully Abstract from Static to Gradual*. *Proc. ACM Program. Lang.* 5(POPL), doi:10.1145/3434288.

[6] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, AleŠ Bizjak, Lars Birkedal & Derek Dreyer (2018): *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*. *Journal of Functional Programming* 28, p. 73, doi:10.1017/S0956796818000151.

[7] Elizabeth Labrada, Matías Toro & Éric Tanter (2020): *Gradual System F*. *CoRR* abs/1807.04596. arXiv:1807.04596.

[8] Conor McBride (2005): *Type-Preserving Renaming and Substitution*.

[9] Max S. New & Amal Ahmed (2018): *Graduality from embedding-projection pairs*. *Proc. ACM Program. Lang.* 2(ICFP), doi:10.1145/3236768.

[10] Max S. New, Dustin Jamner & Amal Ahmed (2019): *Graduality and Parametricity: Together Again for the First Time*. *Proc. ACM Program. Lang.* 4(POPL), doi:10.1145/3371114.

[11] Max Stewart New (2020): *A Semantic Foundation for Sound Gradual Typing*. Ph.D. thesis, Northeastern University.

[12] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2007): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer.

[13] Jeremy G. Siek (2021): *Abstract Binding Trees*. Available at `https://github.com/jsiek/abstract-binding-trees`.

[14] Jeremy G. Siek (2023): *Step-Indexed Logic*. Available at `https://github.com/jsiek/step-indexed-logic`.

[15] Jeremy G. Siek & Walid Taha (2006): *Gradual typing for functional languages*. In: *Scheme and Functional Programming Workshop*, pp. 81–92. Available at `http://scheme2006.cs.uchicago.edu/13-siek.pdf`.

[16] Jeremy G. Siek & Walid Taha (2007): *Gradual Typing for Objects*. In: *European Conference on Object-Oriented Programming*, *LCNS* 4609, pp. 2–27, doi:10.1007/978-3-540-73589-2_2.

[17] Jeremy G. Siek, Peter Thiemann & Philip Wadler (2021): *Blame and coercion: Together again for the first time*. *Journal of Functional Programming* 31, p. e20, doi:10.1017/S0956796821000101.

[18] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini & John Tang Boyland (2015): *Refined Criteria for Gradual Typing*. In: *SNAPL: Summit on Advances in Programming Languages*, LIPIcs: Leibniz International Proceedings in Informatics, doi:10.4230/LIPIcs.SNAPL.2015.274.

[19] The Coq Dev. Team (2004): *The Coq Proof Assistant Reference Manual – Version V8.0*. `http://coq.inria.fr`.

[20] Matías Toro, Ronald Garcia & Éric Tanter (2018): *Type-Driven Gradual Security with References*. *ACM Trans. Program. Lang. Syst.* 40(4), pp. 16:1–16:55, doi:10.1145/3229061.

[21] Philip Wadler, Wen Kokke & Jeremy G. Siek (2020): *Programming Language Foundations in Agda*. Available at `http://plfa.inf.ed.ac.uk/20.07/`.