Formalizing logical relation for System F type safety in Coq

Bastien Rousseau

January 18, 2023

Abstract

Type safety is a language property that ensures that any well-typed, closed program is safe to execute. While syntactic approaches are widely used to prove type safety, another proof method based on logical relations has been shown to be efficient to prove such language properties. In the lectures, we have defined a logical relation and used it to prove type safety of System F, on paper. Because there are many details, paper proofs are prone to errors. Moreover, the encoding of some data structures, as well as their properties, are often implicit. Proof assistant such as Coq require everything explicit in the implementation, prove every single property, and help to keep track of every minute detail. To fill the gap between paper proof and a proof-assistant implementation, we describe an implementation in Coq of type safety of systemF, using a logical relation.

Contents

1 Introduction

Milner described type safety by its famous quote *"Well-typed programs cannot go wrong"* [4]. Type safety — also known as *type soundness* — is a language property ensuring that well-typed program are safe. In other words, well-typed closed term[1] of the language will never reach a state about which the semantics says nothing: it is either a value, or it can perform a step. It expresses that the semantics is complete, in the sense that the machine always knows what to do at any step of the execution.

A naive approach to prove type safety of a language would be to proceed by induction on the structure of types. However, this approach fails, because the induction hypothesis is too weak. To avoid naive induction, type safety is commonly proved using two auxiliary lemmas known as *progress* and *preservation* [13, 5]. Informally, the former states that a well-typed expression is not stuck, *i.e.,* it is either a value, or can perform a step. The latter says that if an expression is well-typed, the next step will also be well-typed.

Another way to prove type safety is based on logical relations. Logical relations are a proof method that scales better on more expressive language than syntactic approaches, and that can be used to prove others language properties, such as normalization, contextual equivalence, or non-interference. Moreover, the fundamental theorem of the logical relations has consequences besides type safety, as deriving free theorems.

When defining a logical relation, there is plenty of detail to keep track of, which makes a paper proof prone to miss an important detail. Moreover, the concrete data structures are sometimes quite implicit. Proof assistants, such as Coq, make sure that every single properties has been proved. The interactive theorem prover helps one to keep track of every minute details. Therefore, we get more confidence by implementing logical relation to prove

[1]In the document, we use "term" and "expression" interchangeably.

type safety in Coq. However, the implementation itself brings new challenges, that we will discuss in this document.

This report is organized as follows. In Section 2, we formalize on paper the language System F, the type safety property, the logical relation, and we establish the main theorems and lemmas. In Section 3, we present the different implementation options, and we highlight the challenges of the implementation in Coq, contrasting with the paper version of the proof. Finally, in Section 4, we propose ideas of improvement and alternatives of the implementation, as well as possible extensions of the project.

The Coq implementation is available in a Github repository. In the following, we use numbered circles to link the formal statement to the corresponding Coq code. For the purpose of the presentation, the representation in the paper and in the code might differ.

2 Type safety of System F on paper

In this section, we formalize on paper the language System F using a named representation of the binders. We present the call-by-value (CBV) small-step semantic and the typing judgment. Then, we formalize type safety, the logical relation and we formulate the main lemmas and theorems. Most of the formalization is standard [10].

2.1 Language

We consider a variant of System F that includes unit and products. We define the syntax and the operational semantics of the language in Figure 1. The named binders are represented as strings. The variable y in the expression $(tt,\ y)$ is a free variable (not bound to any lambda abstraction), while the variable x in the expression $\lambda x.\ x$ is bound to the lambda abstraction. In Section 3, we show a different representation of binders, based on the De Bruijn technique, and we explain why the named representation is not well-suited for implementation.

The dynamic semantics we consider is a small-step operational semantics with a call-by-value strategy. The notation $e.[v/x]$ denotes the capture-avoiding substitution of the expression variable x by the value v. We denote $\rightarrow^*$ the transitive closure of the non-head reduction relation $\rightarrow$. The syntax and the semantics of the language are standard. Figure 2 shows the types and the typing rules of System F. We denote $\Delta;\ \Gamma \vdash e : \tau$ the typing judgment stating that *"e is of type τ under the typing context $\Delta;\ \Gamma$"*. Γ is the context of expression variables, that associates a type to an expression variable. Δ is

$x, y \in$ string
$e ::= x \mid tt \mid (e, e) \mid (e,\ e) \mid \mathit{fst}\ e \mid \mathit{snd}\ e \mid \lambda x.\ e \mid e\ e \mid \Lambda e \mid e\ \_$
$v ::= x \mid tt \mid \lambda x.\ e \mid \Lambda e$
$\mathrm{K} ::= \bullet \mid \mathit{fst}\ \mathrm{K} \mid \mathit{snd}\ \mathrm{K} \mid (\mathrm{K},\ e) \mid (v,\ \mathrm{K}) \mid \mathrm{K}\ e \mid v\ \mathrm{K} \mid \mathrm{K}\ \_$

$$\begin{array}{c}\textsc{E-Fst}\\ \dfrac{}{\mathit{fst}\ (v_1,\ v_2) \leadsto v_1}\end{array} \qquad \begin{array}{c}\textsc{E-Snd}\\ \dfrac{}{\mathit{snd}\ (v_1,\ v_2) \leadsto v_2}\end{array} \qquad \begin{array}{c}\textsc{E-App}\\ \dfrac{}{(\lambda x.\ e)\ v \leadsto e.[v/x]}\end{array}$$

$$\begin{array}{c}\textsc{E-TApp}\\ \dfrac{}{\Lambda e\ \_ \leadsto e}\end{array} \qquad \begin{array}{c}\textsc{E-Step}\\ \dfrac{e \leadsto e'}{\mathrm{K}[e] \to \mathrm{K}\big[e'\big]}\end{array}$$

Figure 1: Call-by-value small-step semantic of systemF with evaluation context. $\leadsto$ is the head reduction relation. $\to$ is the reduction relation.

the context of type variables, that gathers the free type variables. We denote $\bullet$ an empty context. The typing rules are also standard.

With the language defined, we can now focus on a property of this language: type safety.

2.2 Type safety

A term is *safe* if it never gets stuck after any number of steps. In other words, any expression, or reduction of the expression, is either a value, or can perform a step:

Definition. *Safety*

$$\mathrm{Safe}(e) \triangleq \forall e'.\ e \to^* e' \Rightarrow (e' \in \mathrm{Val}) \vee \exists e''.\ e' \leadsto e''$$

A weaker version of safety is *parameterized safety.* Let P be a predicate of values. A term is safe according to P if any expression can either perform a step, or is a value that respects the predicate P:

Definition. *Parameterized safety* ①

$$\mathrm{Safe}_P(e) \triangleq \forall e'.\ e \to^* e' \Rightarrow (e' \in \mathrm{Val} \wedge P(v)) \vee \exists e''.\ e' \to e''$$

We notice that, for any P, $\mathrm{Safe}_P(e) \Rightarrow \mathrm{Safe}(e)$.

A language is *type safe* when any closed well-typed term is safe:

Theorem. *Type safety* ② $\forall e,\ \tau.\ \bullet \vdash e : \tau \Rightarrow \mathrm{Safe}(e)$

$\alpha, \beta \in \text{string}$
$\tau ::= \text{unit} \mid \alpha \mid \tau \times \tau \mid \tau \to \tau \mid \forall \alpha.\ \tau$
$\Gamma ::= \bullet \mid x : \tau, \Gamma$
$\Delta ::= \bullet \mid \alpha, \Delta$

$$\boxed{\Delta; \Gamma \vdash \mathrm{e} : \tau}$$

$$\textsc{T-Unit}\ \frac{}{\Delta; \Gamma \vdash tt : \text{unit}} \qquad \textsc{T-Var}\ \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau} \qquad \textsc{T-Prod}\ \frac{\Delta; \Gamma \vdash e_1 : \tau_1 \qquad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash (e_1,\, e_2) : \tau_1 \times \tau_2}$$

$$\textsc{T-Fst}\ \frac{\Delta; \Gamma \vdash e : \tau_1 \times \tau_2}{\Delta; \Gamma \vdash \mathit{fst}\ e : \tau_1} \qquad \textsc{T-Snd}\ \frac{\Delta; \Gamma \vdash e : \tau_1 \times \tau_2}{\Delta; \Gamma \vdash \mathit{snd}\ e : \tau_2} \qquad \textsc{T-Abs}\ \frac{\Delta; x : \tau_1, \Gamma \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

$$\textsc{T-App}\ \frac{\Delta; \Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_1}$$

$$\textsc{T-TAbs}\ \frac{\alpha, \Delta; \Gamma \vdash e : \tau \qquad \alpha \text{ is not free in } \Gamma}{\Delta; \Gamma \vdash \Lambda e : \forall \alpha.\ \tau} \qquad \textsc{T-TApp}\ \frac{\Delta; \Gamma \vdash e : \forall \alpha.\ \tau}{\Delta; \Gamma \vdash e\ \_ : \tau.[\tau'/\alpha]}$$

Figure 2: Type syntax and typing judgment of SystemF

2.3 Logical relation

As pointed out in Section 1, type safety of System F can be proved using the syntactic approach of progress and preservation [13], but can also be proved using a logical relation. In order to define the logical relation, we need to introduce the interpretation context ξ, a mapping from type variable to an expression property $P \in (\text{Expr} \to \mathbb{P})$.

$$\xi ::= \bullet \mid (\alpha \mapsto P) :: \xi$$

We denote the logical relation $[\![\tau]\!]_\xi(v)$, also read *"v is in the logical relation for the type τ, under the interpretation context ξ"*. We define our logical relation by induction on the structure of types in Figure 3.

We highlight the important part of the definition. A value is in the logical relation of the type variable α, if it respects the property defined in the interpretation context ξ. A value is in the logical relation of the arrow type $\tau_1 \to \tau_2$ if applying the term to any value that is in the logical relation

$$
\begin{aligned}
[\![\alpha]\!]_\xi(v) &\triangleq \xi(\alpha)(v)\\
[\![\text{unit}]\!]_\xi(v) &\triangleq v = tt\\
[\![\tau_1 \times \tau_2]\!]_\xi(v) &\triangleq \exists v_1,\ v_2.\ v = (v_1,\ v_2) \wedge [\![\tau_1]\!]_\xi(v_1) \wedge [\![\tau_2]\!]_\xi(v_2)\\
[\![\tau_1 \to \tau_2]\!]_\xi(v) &\triangleq \exists e.\ v = \lambda x.\ e \wedge (\forall v'.\ [\![\tau_1]\!]_\xi(v') \Rightarrow \text{Safe}_{[\![\tau_2]\!]_\xi}(e.[v'/x]))\\
[\![\forall\alpha.\ \tau]\!]_\xi(v) &\triangleq \exists e.\ v = \Lambda e \wedge (\forall P.\ \text{Safe}_{[\![\tau]\!]_{((\alpha \mapsto P)::\xi)}}(e))
\end{aligned}
$$

Figure 3: Logical relation

of τ_1 leads to a safe value in the logical relation of τ_2. A value is in the logical relation of the polymorphic type $\forall\alpha.\ \tau$ if it is safe to interpret the type variable with any expression property P.

The type safety theorem follows from the composition of two lemmas about the logical relation:

1. any well-typed closed term is in the logical relation

$$\forall e,\ \tau.\ \bullet \vdash e : \tau \Rightarrow [\![e]\!]_\bullet(\tau)$$

2. any term in the logical relation is safe

$$\forall e,\ \tau.\ [\![e]\!]_\bullet(\tau) \Rightarrow \text{Safe}(e)$$

Following the *rules of thumb* in [7], we baked the safety in the definition of the logical relation. This makes the second lemma straightforward to prove. Although, the first lemma is the more interesting to prove. We need to generalize it. The generalized version is called the **Fundamental Theorem of the Logical Relation** (FTLR).

To define the FTLR, we first need to introduce the semantic substitution. We denote γ a substitution, which maps expression variables to expressions.

$$\gamma ::= \bullet \mid (x \mapsto e) :: \gamma$$

Assuming that the domain of γ and Γ are equals, we say that *the substitution γ satisfies the typing context Γ for the predicate $P \in$ Type $\to$ Expr $\to \mathbb{P}$*, written $\gamma \Mapsto_P \Gamma$, when for all expression variables in γ, the predicate P holds for the mapped expression and their corresponding type in Γ:

Definition. *Typing context satisfaction* ③

$$\gamma \Mapsto_P \Gamma \triangleq \forall x \in \text{Dom}(\gamma).\ P(\gamma(x))(\Gamma(x))$$

The FTLR states that if a closed term is well-typed, the term substituted with γ is in the logical relation, for any γ that satisfies the typing context Γ for the logical relation:

Theorem. *Fundamental Theorem of the Logical Relation* $\textcircled{4}$

$$\forall e,\ \tau,\ \Delta,\ \Gamma.\ \Delta;\Gamma \vdash e : \tau \Rightarrow (\forall \xi,\ \gamma.\ (\gamma \mapsto_P \Gamma) \Rightarrow [\![\tau]\!]_\xi(\gamma(e)))$$

with $P = \lambda\tau,\ e.\ [\![\tau]\!]_\xi(e)$.

In the end of the section, we highlight some important intermediate lemmas. We refer the reader to the Coq proof, or the lecture notes [10] for the details.

For any value predicates P and Q, and any expression e, the Safe predicate is monotone over the parameterized predicate:

Lemma. *Safe monotonicity* $\textcircled{5}$ $(\forall v.\ P(v) \Rightarrow Q(v)) \Rightarrow \mathrm{Safe}_P(e) \Rightarrow \mathrm{Safe}_Q(e)$

For any value predicates P that holds for a value v, the Safe predicate also holds:

Lemma. *Safe value* $\textcircled{6}$ $P(v) \Rightarrow \mathrm{Safe}_P(v)$

For any expression e that steps to an expression e', the Safe predicate is (backward)-preserved for any value predicate P:

Lemma. *Safe step backward* $\textcircled{7}$ $e \to e' \Rightarrow \mathrm{Safe}_P(e') \Rightarrow \mathrm{Safe}_P(e)$

To prove that an expression $\mathrm{K}[e]$ is Safe, it actually suffices to show that $\mathrm{K}[v]$ is safe for any value v:

Lemma. *Safe bind* $\textcircled{8}$

$$\forall P\ Q\ e,\ \mathrm{Safe}_Q(e) \Rightarrow (\forall v,\ Q(v) \Rightarrow \mathrm{Safe}_P(\mathrm{K}[v])) \Rightarrow \mathrm{Safe}_P(\mathrm{K}[e]))$$

A value v is in the logical relation of $\tau.[\tau'/\alpha]$ for a certain interpretation context ξ if and only if the value is in the logical relation of τ in which the interpretation of τ' in ξ is the logical relation itself:

Lemma. *Logrel subst* $\textcircled{9}$ $[\![\tau.[\tau'/\alpha]]\!]_\xi(v) \Leftrightarrow [\![\tau]\!]_{(\alpha \mapsto [\![\tau']\!]_\xi)::\xi}(v)$

If α is not free in τ, we can associate any predicate P to α in ξ:

Lemma. *Logrel weaken* $\textcircled{10}$ $[\![\tau]\!]_\xi(v) \Leftrightarrow [\![\tau]\!]_{(\alpha \mapsto P)::\xi}(v)$

2.4 Free theorems

The FTLR has other consequences, besides type safety. For instance, it can also be used to derive free theorems, *à la Wadler* [12]. We prove the two following theorems.

Any polymorphic expression that is typed with the identity type $\forall\alpha.\ \alpha \to \alpha$ is the identity function, *i.e.,* if we apply the expression to a value v, it will reduce the value v itself, or run forever.

Theorem. *Polymorphic identity* ⑪

$$\forall e\ , v.\ \bullet ; \bullet \vdash e : \forall\alpha.\ \alpha \to \alpha \Rightarrow \mathrm{Safe}_{(\lambda e.\ e=v)}((e\ \_)\ v)$$

Any polymorphic expression that is typed with the type $\forall\alpha.\ \alpha$ is actually the empty type, which is as expected uninhabited:

Theorem. *Empty type* ⑫

$$\forall e\ , v.\ \bullet ; \bullet \vdash e : \forall\alpha.\ \alpha \Rightarrow \mathrm{Safe}_{(\lambda e.\ \bot)}(e\ \_)$$

3 Implementation in Coq

In this section, we describe the different implementation options that we had, and we explain the choices that we made.

3.1 Semantic with evaluation context

Our operational semantics of System F follows a call-by-value evaluation strategy, small-step semantic. We had in mind two possible solutions to implement the reduction relation of System F CBV.

1. A semantic with an explicit reduction rule for every inductive case ⑬, with a single reduction relation which contains all the rules.

2. An evaluation context based semantics in two layers ⑭: a head reduction relation, which expresses how to reduce the expression when the redex is in head position; and a non-head reduction relation, when the redex is not is the head position. The evaluation context determines where is the redex in the term.

The two semantics are equivalent ⑮, and both implementations have their own pros and cons. On the one hand, the structural induction is easier with the one-step semantic (1), but the lack of evaluation context prevents defining the safe-bind lemma. As a consequence, the binding property has to be defined *on-the-fly* for each induction case of the FTLR. On the other hand, the evaluation context semantic (2) is convenient to define the safe-bind lemma, but the proof by induction tends to be more tedious. This is caused by the fact that the semantics has two reduction relations: in particular, the non-head reduction relation requires us to destruct the evaluation context.

In the end, both approaches are equivalent in term of proof effort, and the choice is mainly a matter of taste. We wanted to stick as much as possible to the lecture notes [10], so we decided to use the evaluation context semantic to define the logical relation. Moreover, we show in Section 4 another argument that scales in favour of the evaluation context semantic.

3.2 Nameless binders

In the expression $\lambda x.\ e$, we call λx a *binder*. When an expression variable is bound, we say that it points to a (specific) binder. Named binders are a way to represent binders, such that all the occurrences of x in $\lambda x.\ e$ point to the binder λx, until another nested binder with the same name appears. It is a convenient way to represent the binders, because the proofs, especially on paper, are more readable. However, this representation has also some downsides. First, the same term might have different representation, because λ-terms are equal *up-to renaming of the bound variables*. For instance, $\lambda x.\ x$ represents the same term as $\lambda y.\ y$. Second, we have to make sure that the substitution is capture-avoiding. Since the proofs are more readable with named binders, we first tried to use them in the implementation. At some point, we had to define parallel (or simultaneous) substitution, and using this definition in the Coq proof was tedious.

The named representation of the binders is not satisfactory when implementing the language in Coq. The question of the implementation of binders is a well-known issue when implementing a language [5]. An alternative solution is to use a representation using De Bruijn indices. It is a canonical, unique and nameless representation of the binder. Informally, variables point directly to their binder: the named variables are replaced by a natural number that expresses the distance to its binder. More precisely, the DeBruijn index k points to the k-th enclosing λ.

In a more formal way, the expression variables are $k \in \mathbb{N}$. A variable k is free when it ranges outside of the enclosing λ. The notation $e.[v/]$ is the substitution of the *first* free variable: it replaces the free variable 0 — or under n lambda abstraction, it is represented by n — by v, and renames (here, renumbers) all the other variable accordingly, by subtracting. For instance, in the expression $(0, 1).[tt/] = (tt, 0)$, the first free variable is 0, so it replaces 0 by the expression tt. Moreover, the next free variable 1 is renamed to 0. In the expression $1.[tt/] = 0$, the first free variable should be 0 (even if it does not appear in the expression), so the substitution only performs the renaming. Finally, in the expression, $(\lambda.\ (0, (1, 2))).[tt/] = (\lambda.\ (0, (tt, 1)))$, the substitution enters the lambda-abstraction, where the first free variable is now represented as 1. In a similar way, we also use the DeBruijn representation for type variables $\alpha \in \mathbb{N}$. The formal definition of the substitution can be found in the Appendix A.

$k \in \mathbb{N}$
$e ::= \boldsymbol{k} \mid tt \mid (e, e) \mid fst\ e \mid snd\ e \mid \boldsymbol{\lambda.\ e} \mid e\ e \mid \Lambda e \mid e\ \_$
$v ::= \boldsymbol{k} \mid tt \mid \boldsymbol{\lambda.\ e} \mid \Lambda e$
$\alpha \in \mathbb{N}$
$\tau ::= \text{unit} \mid \alpha \mid \tau \times \tau \mid \tau \to \tau \mid \boldsymbol{\forall.\ \tau}$
$\Gamma ::= \bullet \mid \tau, \Gamma$

$$\boxed{\Gamma \vdash e :\ \tau}$$

$$\textsc{E-App-DeBruijn}\ \frac{}{(\lambda.\ e)\ v \rightsquigarrow e.[v/]} \qquad \textsc{T-TAbs-DeBruijn}\ \frac{(\text{map } (+1)\ \Gamma) \vdash e : \tau}{\Gamma \vdash \Lambda e : \forall.\ \tau} \qquad \textsc{T-TApp-DeBruijn}\ \frac{\Gamma \vdash e : \forall.\ \tau}{\Gamma \vdash e\ \_ : \tau.[\tau'/]}$$

Figure 4: Modifications of the syntax (**bold** symbol), the operational semantic and the typing rules for the De Bruijn representation of the binders.

Figure 4 shows the modifications on the syntax due to the new representation. As type variable are also represented using the De Bruijn indices, the free variable are the α that range outside the number of enclosing $\forall$. Thus, there is no need to maintain the type variable context Δ. Moreover, the expression variable context becomes an ordered sequence of types, such that the k-th element of the sequence Γ is the type of the free expression variable represented by k.

The main modification is in the rule T-TAbs-DeBruijn. Indeed, in the rule T-Abs with named binders, the binder of the type variable α is added in the context, and makes sure that α does not appear freely in the context Γ. If necessary, α can be renamed to a fresh type variable. Using De Bruijn representation, the new binder is represented by the type variable 0. All the type variables in the context Γ have to be renamed: it both ensures that the type points to the right binder, and the freshness of the new binder. The renaming consists on incrementing the free type variables by 1, because they are now under one more $\forall$.

The De Bruijn technique has been widely used to represent binders. *autosubst* [6] is a Coq library that helps implement and automate the DeBruijn representation. It automatically derives and proves some basic lemmas about (parallel) substitution. Moreover, it provides useful tactics to reason about substitution. Our implementation uses *autosubst* to represent binders and leverages the automation to simplify the proofs, in particular of the substitution lemma and the weakening lemma.

$\xi ::= \bullet \mid P :: \xi$

$$\llbracket \tau_1 \to \tau_2 \rrbracket_\xi(v) \triangleq \exists e.\ v = \lambda.\ e \wedge (\forall v'.\ \llbracket \tau_1 \rrbracket_\xi(v') \Rightarrow \mathrm{Safe}_{\llbracket \tau_2 \rrbracket_\xi}(e.[v'/]))$$
$$\llbracket \forall.\ \tau \rrbracket_\xi(v) \triangleq \exists e.\ v = \Lambda e \wedge (\forall P.\ \mathrm{Safe}_{\llbracket \tau \rrbracket_{(P::\xi)}}(e))$$

Figure 5: Modification of the logical relation due to the De Bruijn representation.

Figure 5 highlights the modifications to the logical relation according to the De Bruijn representation of the binders. In particular, the mapping ξ is a sequence of expression properties instead of a mapping of type variables. Indeed, as we have already done with the context Γ, the α-th element of ξ is the property mapped to the type variable α.

3.3 Substitution lemmas

One of the most useful properties of the logical relation is the *substitution lemma*. We recall the substitution lemma below, but with the De Bruijn representation of the binders.

Lemma. *Logrel subst - De Bruijn*

$$\forall \xi,\ \tau,\ \tau',\ v.\ \llbracket \tau.[\tau'/] \rrbracket_\xi(v) \Leftrightarrow \llbracket \tau \rrbracket_{(\llbracket \tau' \rrbracket_\xi :: \xi)}(v)$$

It states that a value v is in the logical relation for the type $\tau.[\tau'/]$ if and only if we can associate its own logical relation to the corresponding free type variable in the interpretation mapping. However, while the string representation of the binder allows to prove this by straightforward induction on τ, we cannot proceed directly by induction with our representation based on DeBruijn indices. The induction hypothesis is actually not strong enough, and the inductive case for the polymorphic type does not work. Indeed, the induction hypothesis is then

$$\forall \xi,\ \tau,\ \tau',\ v.\ [\![\tau.[\tau'/]]\!]_{\xi}(v) \Leftrightarrow [\![\tau]\!]_{([\![\tau']\!]_{\xi} :: \xi)}(v)$$

and therefore the proof obligation for the polymorphic case is

$$[\![(\forall.\ \tau).[\tau'/]]\!]_{\xi}(v) \Leftrightarrow [\![\forall.\ \tau]\!]_{([\![\tau']\!]_{\xi} :: \xi)}(v)$$

If we unfold the definition of the logical relation and simplify the goal, the proof obligation ends up being

$$[\![\tau.[\tau'/]]\!]_{P :: \xi}(v) \Leftrightarrow [\![\tau]\!]_{P :: ([\![\tau']\!]_{\xi} :: \xi)}(v)$$

where an additional predicate P is the head of the mapping ξ. While we would like to use the induction hypothesis, it is not possible because the head of the mapping has to be the property of the substituted type variable τ'.

The solution is then to generalize the substitution lemma, such that the predicate that maps the substituted type variable τ' to the logical relation may be anywhere in the new mapping. At a high level, this means that the induction has already gone through a certain number of type abstractions $\forall \tau_1.\ \forall \tau_2.\ \ldots \forall \tau_n.$.

Lemma. *Generalized logrel subst - De Bruijn* (16)

$$\forall \xi_1,\ \xi_2,\ \tau',\ v.\ [\![\tau.[\text{upn } (\text{len } \xi_1)\ \tau'/]]\!]_{\xi_1 ++ \xi_2}(v) \Leftrightarrow [\![\tau]\!]_{\xi_1 ++ ([\![\tau']\!]_{\xi_2} :: \xi_2)}(v)$$

where $\tau.[\text{upn } \kappa\ \tau'/]$ *substitutes* τ' *in the type* τ *by renaming the variables after* κ*.*

It suffices to instantiate the generalized theorem with $\xi_1 = \bullet$ to get the original substitution lemma.

3.4 Proving type safety in Coq

With this setup, the Coq implementation mainly follows the paper proof. The main difference between resides in the binder representation: the paper proof uses named representation whereas the Coq implementation uses De Bruijn indices. However, the main hurdles have been tackled in the intermediates lemmas such as the substitution lemma in Section 3.3. In this way, the FTLR is proved by induction on the type judgment, and is completely independent of the binder representation.

4 Possible improvement / Future work

In this section, we discuss different ways in which the project could be improved. First, we propose an improvement to make the implementation more generic and modular. Then, we propose some directions in which we could extend the project. Finally, we propose an alternative way to implement a logical relation for type safety in Coq, which could lead to a convenient way to extend the language with non-trivial features.

4.1 Language typeclass

The main interest of the project was to implement type safety of System F in Coq, using logical relations. In Section 2.2, we defined the parameterized Safe predicate and a few intermediate lemmas about this predicate. The Safe predicate does not really depend on the language, unlike the logical relation, which is defined over the type structure.

To follow steps of previous work, we propose to make the implementation more generic and modular, such that we can define the Safe predicate independently to the language. In a Coq implementation, it results in the definition of a class that expresses what is a valid language. This is the way that Iris [2] deals with such abstract language.

A generic language is a tuple (Expr, is_value, head_step, is_ectx), where Expr is the type of expressions of the language and the name of the other functions are self explanatory. The future work is to determine which properties (expressed in terms of the generic language) are necessary to make the language a *valid* language. By valid language, we mean a language that allows one to derive the properties over the Safe predicate, *i.e.,* safe-mono, safe-val, safe-bind and safe-step.

This improvement brings more modularity in the implementation. One can indeed define its own language, prove that the language is a valid language and the typeclass derives automatically the lemmas about the safety. Our variant of System F would be an instance of such valid language, and we could easily extend the project with type safety of another language, such as Simply Typed Lambda Calculus (STLC), without proving the safety lemmas again.

4.2 Other language properties

As mentioned earlier, logical relations are a proof technique that can be used to prove language properties [7]. We propose two others properties to extend the project.

4.2.1 Normalization

A term normalizes if it reduces to a value. Formally,

$$\text{Norm}(e) \triangleq \exists v \in \text{Val}.\ e \to^* v$$

and the parameterized version

$$\text{Norm}_P(e) \triangleq \exists v \in \text{Val}.\ e \to^* v \land P(v)$$

We can derive the lemmas equivalent to safe-mono, safe-val, safe-bind and safe-step for the Norm predicate. This extension could leverage the modularity of the previous proposition.

4.2.2 Contextual equivalence

Contextual equivalence, or observational equivalence, is a language property saying that, if two program are contextually equivalent, it does not exist any context able to differentiate them. It is also a way to derive free theorems.

Formally, the contextual equivalence is defined as follows:

$$\Delta'; \Gamma' \vdash e_1 \approx^{\text{ctx}} e_2 : \tau' \triangleq \forall \text{K} \ :\ (\Delta; \Gamma \vdash \tau) \Rightarrow (\bullet; \bullet \vdash \text{unit}).\ (\text{K}[e_1] \Downarrow v \Leftrightarrow \text{K}[e_2] \Downarrow v)$$

where $e \Downarrow v \triangleq e \to^* v$ and

$$\textsc{Ctx-Typing}\quad \frac{\Delta; \Gamma \vdash e : \tau \qquad \Delta'; \Gamma' \vdash \text{K}[e] : \tau'}{\text{K} \ :\ (\Delta; \Gamma \vdash \tau) \Rightarrow (\Delta'; \Gamma' \vdash \tau')}$$

It states that two expressions e_1 and e_2 of type τ are contextually equivalent if and only if, for any context that has a hole of type τ, and produces a closed expression of type unit (see Ctx-Typing), filling the hole with e_1 or e_2 will both reduces to the same value v (which actually has to be tt).

4.3 Logical relation using Iris

Iris [2, 3] is a higher-order separation logic framework, implemented and verified in Coq. Iris as been shown to be an efficient framework to implement logical relation. In particular, an alternative way to implement the logical relation and prove type soundness of System F [11] could have been to use the Iris framework.

A possible extension of System F is to add recursive types. However, adding recursive types is a feature that leads to a non-trivial extension of the logical relation, as shown in [7]: indeed, we have defined the logical relation inductively on the structure of type, but unfolding a recursive type does not guarantee the resulting type to be smaller than the folded one. Thus, it is impossible to simply extend the logical relation defined in this project. Similarly, another extension of the language is to add mutable state (for instance, with pointers) [1, 8] which leads to a similar issue (we can encode recursion through the heap, thanks to the Landin's knot technique).

However, Iris is a step-indexed logic. It provides the logical tools to easily manage recursive types. Moreover, because Iris is a logic of resources, it allows to define resources describing the heap, and easily implement mutable state.

4.4 Logical relation as an interpretation of types

The logical relation as we defined it can be seen as an interpretation of types. The logical relation for the type τ can be actually understand as the set of expression that behaves as the type τ. In Figure 6, we define a logical relation in a way that it highlight this interpretation of types.

$$\begin{aligned}
[\![\alpha]\!]_\xi &\triangleq \{e \in \text{Expr} \mid \exists v.\ e \to^* v \land \xi(\alpha)(v)\} \\
[\![\text{unit}]\!]_\xi &\triangleq \{e \in \text{Expr} \mid e \to^* tt\} \\
[\![\tau_1 \times \tau_2]\!]_\xi &\triangleq \{e \in \text{Expr} \mid \exists v_1,\ v_2.\ v_1 \in [\![\tau_1]\!]_\xi \land v_2 \in [\![\tau_1]\!]_\xi \land e \to^* (v_1,\ v_2)\} \\
[\![\tau_1 \to \tau_2]\!]_\xi &\triangleq \{e \in \text{Expr} \mid \forall v \in [\![\tau_1]\!]_\xi.\ (e\ v) \in [\![\tau_2]\!]_\xi\} \\
[\![\forall \alpha.\ \tau]\!]_\xi &\triangleq \left\{e \in \text{Expr} \mid \forall P.\ e\ \_ \in [\![\tau]\!]_{((\alpha \mapsto P)::\xi)}\right\}
\end{aligned}$$

Figure 6: Logical relation as interpretation of type

In his notes about logical relations [9], Jon Sterling points out that some refactoring allows to get an instance of denotational semantics from a logical relation. In other words, it shows how to transform the logical relation into a compositionnal interpretation of terms, *i.e.,* define the meaning of a term with the meaning of its sub-terms. This kind of properties is desirable because it guides the approaches to prove semantics properties, and allows to abstract the reasoning.

We think that exploring this direction is another way to extend the project.

5 Conclusion

Logical relations are a proof technique that has been widely studied in the past decades, and that have proven very useful to prove programming language properties. While doing proof on paper is prone to mistakes, proof assistants such as Coq tend to give some stronger guarantees. Coq carefully manages every minute detail and ensures that every single lemma is proven. On the other hand, it requires carefully choosing the implementation representations in order to make the proofs more manageable.

[1] Amal Ahmed. “Semantics of Types for Mutable State”. PhD thesis. Princeton University, 2004.

[2] *Iris Project.* https://iris-project.org/.

[3] Ralf Jung et al. “Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic”. In: *J. Funct. Prog.* 28 (2018).

[4] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Sciences* 17.3 (1978).

[5] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[6] Steven Schäfer, Tobias Tebbi, and Gert Smolka. “Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions”. In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015.* 2015.

[7] Lau Skorstengaard. “An Introduction to Logical Relations”. In: ().

[8] Lau Skorstengaard. *Logical Relations And References.* 2016.

[9] Jon Sterling. *Practical Semantics.* https://www.jonmsterling.com/papers/sterling-2022-lr-tutorial.pdf.

[10] Amin Timany. “Logical Relations: Safety of System F”.

[11] Amin Timany et al. “A Logical Approach to Type Soundness”. In: (2022).

[12] Philip Wadler. “Theorems for Free!” In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture - FPCA ’89.* 1989.

[13] A.K. Wright and M. Felleisen. “A Syntactic Approach to Type Soundness”. In: *Inf. Comput.* 115.1 (1994).

A Substitution De Bruijn binders

$$
\begin{array}{l}
\uparrow_c^d (tt) \triangleq tt \\
\uparrow_c^d (k) \triangleq k, \text{ if } k < c \\
\uparrow_c^d (k) \triangleq k + d, \text{ if } k \geq c \\
\uparrow_c^d ((e_1, e_2)) \triangleq \left(\uparrow_c^d (e_1), \uparrow_c^d (e_2)\right) \\
\uparrow_c^d ((\mathit{fst}\ e)) \triangleq \mathit{fst}\ \ \uparrow_c^d (e) \\
\uparrow_c^d ((\mathit{snd}\ e)) \triangleq \mathit{snd}\ \ \uparrow_c^d (e) \\
\uparrow_c^d ((\lambda.\ e)) \triangleq \lambda.\ \ \uparrow_c^{d+1} (e) \\
\uparrow_c^d ((e_1\ e_2)) \triangleq \uparrow_c^d (e_1)\ \ \uparrow_c^d (e_2) \\
\uparrow_c^d ((\Lambda e)) \triangleq \Lambda(\uparrow_c^d (e)) \\
\uparrow_c^d ((e\ \_)) \triangleq (\uparrow_c^d (e))\ \_
\end{array}
$$

Figure 7: Definition of shifting of De Bruijn indices.

$$
\begin{array}{l}
tt.[s/n] \triangleq tt \\
n.[s/n] \triangleq s \\
k.[s/n] \triangleq k, \text{ if } k \neq n \\
(e_1, e_2).[s/n] \triangleq (e_1.[s/n], e_2.[s/n]) \\
(\mathit{fst}\ e).[s/n] \triangleq \mathit{fst}\ e.[s/n] \\
(\mathit{snd}\ e).[s/n] \triangleq \mathit{snd}\ e.[s/n] \\
(\lambda.\ e).[s/n] \triangleq \lambda.\ e.\left[\uparrow_0^1 (s)/n+1\right] \\
(e_1\ e_2).[s/n] \triangleq e_1.[s/n]\ e_2.[s/n] \\
(\Lambda e).[s/n] \triangleq \Lambda(e.[s/n]) \\
(e\ \_).[s/n] \triangleq (e.[s/n])\ \_
\end{array}
$$

Figure 8: Definition of substitution of De Bruijn indices.

$\uparrow_c^d (e)$ is the shifting is of the variables above c by d in the term e. $e.[s/n]$ is the substitution of expression variable n by s in the expression e. We note $e.[s/] \triangleq e.[s/0]$.