OPLSS 2023: Logical Relations *

Amal Ahmed

June 27th - 29th, 2023

Abstract

This document contains the lecture notes of the Logical Relations course at OPLSS 2023. During the lecture, we had an introduction to logical relations. We went through multiple examples of use of logical relation: normalization and type safety for Simple Typed Lambda Calculus (STLC), step-indexed logical relation for recursive types, and a glance of binary logical relation for parametric polymorphism.

Contents

1 Introduction

Logical relations are a prolific and highly useful proof method that can be used to prove a wide range of properties about programs and languages. For instance, logical relations can be used to prove:

- termination of well-typed programs in a language such as the simply-typed lambda calculus;
- type soundness/safety;
- program equivalence, which can take many forms:
  - Verifying that an optimized algorithm or implementation is equivalent to a simpler naive one;
  - Demonstrating the correctness of compiler optimizations or transformations;
  - Showing *representation independence* [Mit86], that is, that the implementation underlying a given interface does not influence the behavior of the client;

*transcribed by Gabriela Araujo Britto, Bruno da Rocha Paiva, Bastien Rousseau, and Priya Srikumar

    - For instance, a stack interface may be implemented using an array or a linked list, but this should be indistinguishable to the users of the interface. To show representation independence for the stack interface, we would show that a program that uses the array implementation is equivalent to a program that uses the linked list implementation;
  - Conceptualizing parametric polymorphism and corresponding free theorems [Wad89] as a relation loosely described as "related inputs go to related outputs;"
  - Proving noninterference for security-typed languages: showing that two runs of a given program are equivalent on low-security outputs for any variation on high-security (e.g. confidentiality) program data;

In recent work, logical relations have already been used for compilation (using cross-languages logical relations), foreign function interfaces or proving soundness of logic.

The terms *logical predicate* and *unary logical relation* are synonymous, as are the terms *logical relation* and *binary logical relation*. Both terms are used in the literature and common parlance. Logical predicates $P_\tau(e)$, are sets, and reason about a single program. Program termination and type safety and soundness are logical predicates. On the other hand, logical relations $R_\tau(e_1, e_2)$ are binary relations, and can be used to capture program equivalences more broadly.

2 Normalization of STLC

In this section, we show an example of a logical relation for proving normalization of STLC.

2.1 Formalization of STLC

We recall the syntax and operational semantic of STLC in Figure 1. We consider a slightly variant of STLC, with the base type of booleans, and the if-then-else expression.

$$
\begin{aligned}
\tau &::= \text{bool} \mid \tau \to \tau \\
e &::= x \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \mid \lambda x : \tau.\ e \mid e\ e \\
v &::= \text{true} \mid \text{false} \mid \lambda x : \tau.\ e \\
E &::= [] \mid \text{if E then } e \text{ else } e \mid \text{E } e \mid v \text{ E}
\end{aligned}
$$

$$
\textsc{E-IfThenElseTrue}\ \frac{}{\text{if true then } e_1 \text{ else } e_2 \mapsto e_1} \qquad \textsc{E-IfThenElseFalse}\ \frac{}{\text{if false then } e_1 \text{ else } e_2 \mapsto e_2} \qquad \textsc{E-App}\ \frac{}{(\lambda x : \tau.\ e)\ v \mapsto e[v/x]}
$$

$$
\textsc{E-Step}\ \frac{e \mapsto e'}{E[e] \mapsto E[e']}
$$

Figure 1: Call-by-value small step semantics of STLC with evaluation context. $\mapsto$ is the head reduction relation (also called *primitive reduction*).

The operational semantic implements a *call-by-value* evaluation order, meaning that the argument of a function needs to be evaluated into a value before applying the function. We write $e \mapsto e'$ to say that e reduces to e' in one computation step. We write $e \mapsto^* e'$ for the transitive and reflexive closure of the reduction relation.

Figure 2 shows the typing judgment rules. The relation $\Gamma \vdash e : \tau$ means that x is typed by τ under the typing context Γ. We do not go through the details of the typing rules, as they are fairly classic.

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

$$\text{T-True}\ \frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \text{T-False}\ \frac{}{\Gamma \vdash \text{false} : \text{bool}} \qquad \text{T-Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\text{T-IfThenElse}\ \frac{\Gamma \vdash e : \text{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$$

$$\text{T-Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2} \qquad \text{T-App}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

Figure 2: Typing judgments of STLC.

Then, we formally define normalization (N) of STLC, meaning that any well-typed closed terms actually terminates. We introduce the notation $e \Downarrow v$ to mean that the term e evaluates in to value v, that is $e \mapsto^* v$, and the notation $e \Downarrow$ to mean that e evaluates to some value , that is $\exists v. e \mapsto^* v$.

Theorem (Normalization)**.** *For all terms e, if $\cdot \vdash e : \tau$, then $e \Downarrow$.*

2.2 First attempt: Proving STLC is normalizing by induction on typing derivations

To see why we might want to use logical relations, let's first try to prove this theorem by induction on the typing derivation:

- Case T-True $\frac{}{\cdot \vdash \text{true} : \text{bool}}$: true is already a value, so it is terminated.
- Case T-False $\frac{}{\cdot \vdash \text{false} : \text{bool}}$: false is already a value, so it is terminated.
- Case T-Var $\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$: does not apply because the context is not empty.
- Case T-Abs $\frac{x : \tau_1 \vdash e : \tau_2}{\cdot \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$: $\lambda x : \tau_1.\ e$ is already a value, so it is terminated.
- Case T-IfThenElse $\frac{\cdot \vdash e : \text{bool} \qquad \cdot \vdash e_1 : \tau \qquad \cdot \vdash e_2 : \tau}{\cdot \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$: From the inductive hypothesis, we know that $e \Downarrow$, $e_1 \Downarrow$ and $e_2 \Downarrow$. Since $e \Downarrow v$ for some v, by inspecting our reduction semantics, we know that if e then e_1 else $e_2 \mapsto^*$ if v then e_1 else e_2. Thanks to the canonical forms lemma, we know that the only values of type bool are true and false. By case destruction over v, either if e then e_1 else $e_2 \mapsto^*$ if true then e_1 else $e_2 \mapsto^* e_1 \Downarrow$, or if e then e_1 else $e_2 \mapsto^*$ if false then e_1 else $e_2 \mapsto^* e_2 \Downarrow$.
- Case T-App $\frac{\cdot \vdash e_1 : \tau_2 \to \tau \qquad \cdot \vdash e_2 : \tau_2}{\cdot \vdash e_1\ e_2 : \tau}$: From the inductive hypothesis, we know that $e_1 \Downarrow$ and $e_2 \Downarrow v_2$. The values of type $\tau_2 \to \tau$ are lambda expressions, so since $e_1 \Downarrow$, we know $e_1 \Downarrow \lambda x : \tau_2.\ e'$, so $e_1\ e_2 \mapsto^* (\lambda x : \tau_2.\ e')\ e_2 \mapsto^* e'[v_2/x]$. But we don't know anything about whether e' terminates, so we are stuck and cannot prove this case.

The main insight is, the induction hypothesis is too weak: the typing judgment does not tell us anything about the value produced by the normalization. To solve this issue, we will define a logical relation that precisely strengthens the induction hypothesis, allowing us to know more about the value produced by the evaluation.

2.3 Principles for defining a logical relation

The logical relation is a predicate $P_\tau(e)$ over an expression, indexed by a type. When defining a logical relation $P_\tau(e)$, there are three principles we typically follows:

- The logical relation should contain well-typed terms, ie. $\cdot \vdash e : \tau$
- The property of our interest P should be baked in the logical relation (i.e., normalization)
- The property of our interest P should be preserved by the elimination forms[1] for the corresponding type τ. Intuitively, it means that we have a way to continue after consuming a value of type τ.

We will now see how to construct a logical relation more concretely to prove that STLC is normalizing.

[1] The introduction forms creates a value of type τ, while elimination form consumes a value of type τ

2.4 Logical relation for normalization of STLC

We define the normalization logical predicate by induction on STLC types:

$$\begin{array}{rcl} N_{\texttt{bool}}(e) & \triangleq & \cdot \vdash e : \text{bool} \quad \wedge \quad e \Downarrow \\ N_{\tau_1 \to \tau_2}(e) & \triangleq & \cdot \vdash e : \tau_1 \to \tau_2 \quad \wedge \quad e \Downarrow \quad \wedge \quad \forall e'.\ N_{\tau_1}(e') \implies N_{\tau_2}(e\ e') \end{array}$$

One can easily check that the three principles are respected in the definition of N_τ.
From here, we can split the proof of normalization into two steps:

(A) For all terms e if $\cdot \vdash e : \tau$ then $N_\tau(e)$

(B) For all terms e if $N_\tau(e)$ then $e \Downarrow$

Step (B) follows quickly from the definition of the logical predicate N_τ. By induction on τ, we have two cases to consider, and in both cases $e \Downarrow$ follows directly from the definition of $N_\tau(e)$. One might remark that, it actually follows from the second principle: the property of our interest (normalization) is in the definition of the logical relation.

Step (A) is not so straightforward however due to the case of lambda abstraction. Suppose that $\cdot \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2$ so in particular we know $x : \tau_1 \vdash e : \tau_2$. To prove (A), we now need to show that $\cdot \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2$ which holds by assumption. We must also show that $\lambda x : \tau_1.\ e \Downarrow$, which holds as a lambda term is already a value. The final step is to prove that for any term e', if $N_{\tau_1}(e')$ then $N_{\tau_2}(\lambda x : \tau_1.\ e\ e')$. At this point we find ourselves stuck, since e is not a closed term and thus we cannot proceed by induction on its typing derivation. This leads us to generalising step (A) to allow for open terms:

(A') For all contexts Γ, substitutions γ and terms e if $\Gamma \vdash e : \tau$ and $\gamma \vDash \Gamma$ then $N_\tau(\gamma(e)))$

(B) For all terms e if $N_\tau(e)$ then $e \Downarrow$

The substitution $\gamma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ cannot be any arbitrary substitution. Intuitively, we say that $\gamma \vDash \Gamma$ (read γ satisfies the typing environment Γ) if they have the same domain and if all the inputs respects the property of our interest (in our case, normalization). This is required because a value in the context can be a lambda abstraction, which ends up with the same issue that we had in the first place.

$$\gamma \vDash \Gamma \triangleq \text{dom}(\gamma) = \text{dom}(\Gamma) \wedge (\forall x \in \text{dom}(\Gamma).\ N_{\Gamma(x)}(\gamma(x)))$$

Then, we define some auxiliary lemmas. In proving step (A') we will make use of two lemmas. The first is the substitution lemma, which will let us go from well typed-open terms to well-typed closed terms. The interested reader can check [Pie02, p. 9.3.8]

Lemma (Substitution Lemma)**.** *Substitution preserves types, that is if* $\Gamma \vdash e : \tau$ *and* $\gamma \vDash \Gamma$ *then* $\cdot \vdash \gamma(e) : \tau$.

The second lemma is the reduction lemma, which will be important for the lambda case when we will need to reduce terms.

Lemma (Reduction Lemma)**.** *The normalization logical predicate respects reduction, that is if* $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $N_\tau(e) \iff N_\tau(e')$.

Proof of generalized normalization.

$$\forall\ \Gamma, \gamma, e,\ \tau.\ \Gamma \vdash e : \tau \wedge \gamma \vDash \Gamma \Rightarrow N_\tau(\gamma(e))$$

We proceed by induction on the typing derivation. One might notice that, while the naive proof is easy for the abstraction case and gets stuck for the application case, this is the other way for this proof: the application case is easy, while the abstraction case is hard.

- Case T-True $\dfrac{}{\Gamma \vdash \text{true} : \text{bool}}$: We need to show $N_{\text{bool}}(\gamma(\text{true}))$. By definition of the substitution, it is equivalent to show $N_{\text{bool}}(\text{true})$. By unfolding the definition of N_{bool}, we need to show $\cdot \vdash \text{true} : \text{bool}$ and $\text{true} \Downarrow$. Both goals are trivial.
- Case T-False $\dfrac{}{\Gamma \vdash \text{false} : \text{bool}}$: similarly to the T-True case.
- Case T-Var $\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$: We need to show $N_\tau(x)$. Because $\gamma \in \Gamma$, we know that $x \in \Gamma$ and $N_{\Gamma(x)}(x)$, which is

equivalent to $N_\tau(x)$.

- Case T-IfThenElse $\dfrac{\Gamma \vdash e : \text{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$: Proof is left as an exercise for the reader. Another exercise for the reader is to type up the proof and send it to us.

- Case T-App $\dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$: We need to show $N_\tau(\gamma(e_1 e_2)$, which is equivalent to showing $N_\tau(\gamma(e_1)\gamma(e_2)$. Because τ is arbitrary here, we should read the induction hypothesis. By induction hypothesis, we know that $N_{\tau_2 \to \tau}$. By definition, it means that $\forall e', N_{\tau_1}(e') \Rightarrow N_{\tau_2}(\gamma(e_1 e')$. We can instantiate e' and τ_1 with $\gamma(e_2)$ and τ, thanks to the other induction hypothesis. Then, we get $N_\tau(\gamma(e_1)\gamma(e_2)$, which is what we are showing.

- Case T-Abs $\dfrac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$:

We suppose that $\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2$ and $\gamma \vDash \Gamma$. We want to show that $N_{\tau_1 \to \tau_2}(\gamma(\lambda x : \tau_1.\ e))$. By unfolding the definition of N, there are three properties to prove:
(1) $\cdot \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2$
(2) $\lambda x : \tau_1.\ e \Downarrow$
(3) $\forall e'.\ N_\tau(e') \Rightarrow N_{\tau_2}((\lambda x : \tau_1.\ \gamma(e)\ e')$

(1) follows directly from the substitution lemma 2.4. (2) is trivial, because $\lambda x : \tau_1.\ e$ is already a value. Let us prove (3).

Suppose e' such that $N_{\tau_1}(e')$. We need to show that $N_{\tau 2}((\gamma(\lambda x : \tau_1.\ e)e')$. However, because $x \notin \Gamma$ (by induction hypothesis) and $\text{dom}(\gamma) = \text{dom}(\Gamma)$ (by $\gamma \vDash \Gamma$), we can push the substitution γ inside the lambda, and is equivalent to show that $N_{\tau 2}((\lambda x : \tau_1.\ \gamma(e))e')$.

From $N_{\tau_1}(e')$, we also know that it exists a value v' such that $e' \mapsto^* v'$. Using the backward reduction lemma 2.4, it then actually suffices to prove $N_{\tau 2}((\lambda x : \tau_1.\ \gamma(e))v')$. Taking another step, we eventually need to prove $N_{\tau 2}(\lambda x : \tau_1.\ \gamma(e)[v'/x])$.

We did not use the induction hypothesis yet, let's take a look at it. In this case, the induction hypothesis says: $\forall \gamma',\ \Gamma, x : \tau \vdash e : \tau_2 \wedge \gamma' \vDash \Gamma, x : \tau \Rightarrow N_{\tau_2}(\gamma'(e))$. It is important to understand what is happening when applying the induction principle here, because this is a common case of error. In particular, we get to pick a certain γ' that satisfies the new environment $\Gamma, x : \tau$. More specifically, we need to pick a value v such that $\gamma' = \gamma[x \mapsto v]$ and $N_\tau(v)$. We do have a candidate: v'. By hypothesis, we know that $N_{\tau_2}(e')$ and $e' \mapsto^* v'$, then using the forward propagation lemma we also know that $N_{\tau_2}(v')$. As such, we pick $\gamma' \triangleq \gamma[x \mapsto v']$.

Using γ', we can refine the induction hypothesis, giving us $N_{\tau_2}(\gamma'(e'))$, which is equivalent to $N_{\tau_2}(\gamma[x \mapsto v'](e))$. By definition of the substitution, it finally give us $N_{\tau_2}(\gamma(e[v'/x]))$, which is precisely what we wanted to prove.

3 Type safety of STLC

Type safety is informally defined by the slogan "well-typed programs do not go wrong", often attributed to Robin Milner. Alternatively, one can informally describe type safety as "well-typed programs do not get *stuck*." What this means is that, when a well-typed program runs, there won't be a nonsensical operation, such as trying to apply a string to a number (e.g. "foo" 5).

We define type safety as follows:

Theorem (Type safety)**.** *If* $\cdot \vdash e : \tau$ *and* $e \mapsto^* e'$*, then either* $\text{val}(e')$ *or* $\exists e''$ *such that* $e' \mapsto e''$*.*

Typically, type safety is proven using the progress and preservation lemmas:

Lemma (Progress)**.** *If* $\cdot \vdash e : \tau$ *then either* $\text{val}(e)$ *or* $\exists e'$ *such that* $e \mapsto e'$*.*

Lemma (Preservation)**.** *If* $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \tau$*.*

The progress and preservation lemmas refer to a single reduction step. To prove type safety, we use induction on the number of reduction steps and then apply the two lemmas.

3.1 Logical relation for type safety of STLC

To prove type safety of STLC using logical relations, we will define a logical relation on values ($\mathcal{V}[\![\tau]\!]$) and expressions ($\mathcal{E}[\![\tau]\!]$) separately by induction on type:

$$\begin{aligned}\mathcal{V}[\![\text{bool}]\!] &\triangleq \{\text{true}, \text{false}\}\\ \mathcal{V}[\![\tau_1 \to \tau_2]\!] &\triangleq \{\lambda x : \tau_1.\ e \mid \forall v \in \mathcal{V}[\![\tau_1]\!].e[v/x] \in \mathcal{E}[\![\tau_2]\!]\}\\ \mathcal{E}[\![\tau]\!] &\triangleq \{e \mid \forall e'.\ e \mapsto^* e' \wedge \text{irred}(e') \Rightarrow e' \in \mathcal{V}[\![\tau]\!]\}\end{aligned}$$

where

$$\text{irred}(e) \triangleq \nexists e'.\ e \mapsto^* e'$$

We can check that $\mathcal{V}[\![\tau]\!]$ ("values that behave like the type τ") and $\mathcal{E}[\![\tau]\!]$ ("expressions that behave like the type τ") are well-founded relations.

$\mathcal{V}[\![\tau_1 \to \tau_2]\!]$ refers to itself in its definition, which may seem problematic. However, it refers to $\mathcal{V}[\![\tau_1]\!]$, which is $\mathcal{V}[\![-]\!]$ for a smaller type (τ_1) than $\tau_1 \to \tau_2$. It also refers to $\mathcal{E}[\![\tau_2]\!]$, which again is $\mathcal{E}[\![-]\!]$ for a smaller type (τ_2) than $\tau_1 \to \tau_2$. Similarly, The definition of $\mathcal{E}[\![-]\!]$ refers to $\mathcal{V}[\![-]\!]$ for the same type. So those two relations are well-founded. However, if our language had recursive types, we would not be able to define the relations by induction on types.

Now that we have defined our logical relations, we can use them to prove type safety for STLC.

Similar to normalization, the proof of type safety for STLC can be split into two steps:

(A) For all terms e if $\cdot \vdash e : \tau$ then $\cdot \vDash e : \tau$

(B) For all terms e if $\cdot \vDash e : \tau$ then safe(e)

Note that $\cdot \vdash e : \tau$ and $e \in \mathcal{E}[\![\tau]\!]$ are equivalent.

To prove (B), we assume that $\cdot \vDash e : \tau$, so for all terms e' if $e \mapsto^* e'$ and e' is irreducible, then $e' \in \mathcal{V}[\![\tau]\!]$. To prove safe($e$) we fix some e' and suppose $e \mapsto e'$, so we must show that either e' is a value or $\exists e''.e' \mapsto e''$. Now, either there exists some e'' such that $e' \mapsto e''$ in which case safe(e) or there exists no such e''. In the latter case, e' is irreducible and as $e \in \mathcal{E}[\![\tau]\!]$ we know that $e' \in \mathcal{V}[\![\tau]\!]$ and hence must be a value, so safe(e) holds in this case too.

Similar to before, we prove (A) as a corollary to the fundamental property of the type safety logical relation, although before we can state this theorem, we must introduce the notion of semantic well-typedness. First, we give a definition of substitutions into a context that respect type safety, which is done by induction on the structure of contexts

$$\begin{aligned}\mathcal{G}[\![\cdot]\!] &\triangleq \{\emptyset\}\\ \mathcal{G}[\![\Gamma, x : \tau]\!] &\triangleq \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \wedge v \in \mathcal{V}[\![\tau]\!]\}\end{aligned}$$

Then, the definition of semantic well-typedness is

$$\Gamma \vDash e : \tau \triangleq \forall \gamma \in \mathcal{G}[\![\Gamma]\!], \gamma(e) \in \mathcal{E}[\![\tau]\!]$$

Theorem (Fundamental Property/Basic Lemma of the Logical Relation $\mathcal{E}[\![-]\!]$)**.** *If a term is syntactically well-typed then it is semantically well-typed, that is, if* $\Gamma \vdash e : \tau$ *then* $\Gamma \vDash e : \tau$*.*

To prove the fundamental property we proceed by induction on the typing judgment.

- Case T-True $\dfrac{}{\Gamma \vdash \text{true} : \text{bool}}$: given a substitution $\gamma \in \mathcal{G}[\![\Gamma]\!]$, we must show that $\gamma(\text{true}) = \text{true} \in \mathcal{E}[\![\text{bool}]\!]$. Since true is already irreducible, it is enough to show that $\text{true} \in \mathcal{V}[\![\text{bool}]\!]$, which is true by definition.
- Case T-False $\dfrac{}{\Gamma \vdash \text{false} : \text{bool}}$: similarly to the above case, we have to show that $\text{false} \in \mathcal{E}[\![\text{bool}]\!]$. To show this it suffices to prove $\text{false} \in \mathcal{V}[\![\text{bool}]\!]$ and this holds by definition.
- Case T-Var $\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$: as the set $\mathcal{G}[\![\Gamma]\!]$ was defined inductively, we now proceed by induction on the structure of Γ. The case of the empty context cannot come up as the context contains at least the variable x, so we are left with two cases. If $\Gamma \triangleq \Gamma', x : \tau$, then since $\gamma \in \mathcal{G}[\![\Gamma', x : \tau]\!]$ we also know that $\gamma(x) \in \mathcal{V}[\![\tau]\!]$. As values are irreducible this means that $\gamma(x) \in \mathcal{E}[\![\tau]\!]$. The second case has $\Gamma \triangleq \Gamma', y : \tau'$ with y a distinct variable from x and $\gamma \triangleq \gamma'[y \mapsto v]$. In this case we know that we have $\Gamma' \vdash x : \tau$ and $\gamma' \in \mathcal{G}[\![\Gamma']\!]$ since $\gamma \in \mathcal{G}[\![\Gamma]\!]$, but by the inductive hypothesis we get that $\Gamma' \vDash x : \tau$ so $\gamma(x) = \gamma'(x) \in \mathcal{V}[\![\tau]\!]$ and hence in $\mathcal{E}[\![\tau]\!]$.
- Case T-IfThenElse $\dfrac{\Gamma \vdash e : \text{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$: to prove this case, we must inspect the reduction trace to decide which inductive hypothesis to use. This is a very important skill to have and as such we leave this case as a *useful* exercise to the reader.

- Case T-App $\dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$: the case for application follows similarly to the proof of normalisation and is left as a *fun* exercise for the reader.

- Case T-Abs $\dfrac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$:

We need to show $\Gamma \vDash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2$. By unfolding the definition of semantic typing, we suppose that $\gamma \in \mathcal{G}[\![\Gamma]\!]$, and we need to show that $\gamma(\lambda x : \tau_1.\ e) \in \mathcal{E}[\![\tau_1 \to \tau_2]\!]$. Because $x \notin \gamma$ (same reason as T-Abs case proof of normalization of STLC), we can push the substitution γ under the lambda. It thus remains to show that $\lambda x : \tau_1.\ \gamma(e) \in \mathcal{E}[\![\tau_1 \to \tau_2]\!]$.

By unfolding the definition of $\mathcal{E}[\![\tau_1 \to \tau_2]\!]$, suppose that we have an e' such that $\mapsto^* \lambda x : \tau_1.\ \gamma(e)e'$ and irred(e'), and we then need to show $e' \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$. One might notice that the operational semantic actually takes no step, which means that $e' = \lambda x : \tau_1.\ \gamma(e)$. As such, we need to show that $\lambda x : \tau_1.\ \gamma(e) \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$.

Let us take a look at the definition of $\mathcal{V}[\![\tau_1 \to \tau_2]\!]$. Suppose $v \in \mathcal{V}[\![\tau_1]\!]$, and show that $\gamma(e)[v/x] \in \mathcal{E}[\![\tau_2]\!]$.

By induction hypothesis, we have $\Gamma, x : \tau_1 \vDash e : \tau_2$. By unfolding the definition of semantic typing, we can provide a certain γ' such that $\gamma' \vDash \Gamma, x : \tau_1$. We can instantiate γ' with $\gamma[x \mapsto v]$, because (by definition of satisfaction of the environment) $\gamma \vDash \Gamma$ and $v \in \mathcal{V}[\![ty_1]\!]$. Finally, the induction hypothesis gives us $\gamma'(e) \in \mathcal{E}[\![\tau_2]\!]$, which is equivalent to $\gamma[x \mapsto v](e) \in \mathcal{E}[\![\tau_2]\!]$. By definition of substitution, it is then equivalent to $\gamma(e[v/x]) \in \mathcal{E}[\![\tau_2]\!]$, which is exactly what we wanted to prove.

4 Adding Recursive Types to the STLC

Recursive types can be used to capture potentially infinite data structures such as lists, streams, and trees, among others. The addition of recursive types to a language permit non-terminating programs to be type checked.

To demonstrate the utility of recursive types, let us examine the Ω combinator from the untyped lambda calculus, which loops infinitely:

$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

If we try to apply the first term to the second, we step to the the Ω combinator again. Now, let's suppose we tried to type this term in the simply-typed lambda calculus:

$$(\lambda x :?.xx)(\lambda x :?.xx)$$

In one appearance of x in the term $(\lambda x.xx)$, it is being applied to itself, so we would expect its type to be something like $\tau_1 \to \tau_2$. However, the x supplied as argument to the function x would not be of the correct input type. We will soon see how recursive types can be helpful in typing the Ω combinator.

Consider the recursive type describing a tree (here written using OCaml syntax):

```
type tree = Leaf | Node of int * tree * tree
```

We can rewrite this definition using the unit type 1 to represent tree leaves:

$$\text{type } tree = 1 + (\text{int} \times tree \times tree)$$

tree appears in its own definition, which prevents us from inspecting it further. We will use the type variable α in order to allow the type of tree to appear in its definition.

Note that we can *unfold* the definition of a tree to describe a tree with a larger depth by substituting the definition of a tree within the definition.

$$1 + (\text{int} \times \alpha \times \alpha) = 1 + (\text{int} \times (1 + (\text{int} \times \alpha \times \alpha)) \times (1 + (\text{int} \times \alpha \times \alpha)))$$

A fixpoint is a function f such that $x = f(x)$ for all $x \in dom(f)$. Given the following recursive function, we will define a fixpoint for it, namely some F such that $tree = F(tree)$:

$$F = \lambda\alpha :: \text{type}.1 + (\text{int} \times \alpha \times \alpha)$$

We will write the fixpoint of F as follows, and refer to it as the *fixed point type constructor* for α:

$$\mu\alpha.F(\alpha) = F(\mu\alpha.F(\alpha))$$

We will substitute τ for $F(\alpha)$; note that α is free to appear in τ:

$$\mu\alpha.\tau = F(\mu\alpha.\tau)$$

Note that we can rewrite the right-hand side straightforwardly, since $F(\alpha) = \tau$:

$$\mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha]$$

Intuitively, the left-hand side of this equation represents the fixpoint for τ, which in the case of trees, represents our initial definition of the tree type. The right-hand side represents the *unfolded* definition of trees that we saw afterwards. Indeed, going from the left-hand side of the above equation to the right-hand side represents *unfolding* a recursive type, while going the opposite way represents *folding* a recursive type.

We will extend the STLC with recursive types.

$$\begin{gathered}\tau ::= 1 \mid \text{bool} \mid \tau \to \tau \mid \alpha \\ e ::= x \mid \text{true} \mid \text{false} \mid \lambda x.\ e \mid e\ e \mid \text{fold}\ e \mid \text{unfold}\ e \\ v ::= \text{true} \mid \text{false} \mid \lambda x : \tau.\ e \mid \text{fold}\ v \\ E ::= [] \mid \text{if E then } e_1 \text{ else } e_2 \mid \text{E}\ e \mid v\ \text{E} \mid \text{fold}\ E \mid \text{unfold}\ E\end{gathered}$$

$$\textsc{E-IfThenElseTrue}\ \frac{}{\text{if true then } e_1 \text{ else } e_2 \mapsto e_1} \qquad \textsc{E-IfThenElseFalse}\ \frac{}{\text{if false then } e_1 \text{ else } e_2 \mapsto e_2} \qquad \textsc{E-App}\ \frac{}{(\lambda x : \tau.\ e)\ v \mapsto e[v/x]}$$

$$\textsc{E-Step}\ \frac{e \mapsto e'}{E[e] \mapsto E[e']} \qquad \textsc{E-Fold}\ \frac{}{\text{unfold}(\text{fold}\ v) \mapsto v}$$

Figure 3: Call-by-value small step semantics of STLC + recursive types with evaluation context.

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

$$\textsc{T-True}\ \frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \textsc{T-False}\ \frac{}{\Gamma \vdash \text{false} : \text{bool}} \qquad \textsc{T-Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{T-IfThenElse}\ \frac{\Gamma \vdash e : \text{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$$

$$\textsc{T-Abs}\ \frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2} \qquad \textsc{T-App}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1} \qquad \textsc{T-Fold}\ \frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \text{fold}\ e : \mu\alpha.\tau}$$

$$\textsc{T-Unfold}\ \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \text{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]}$$

Figure 4: Typing judgments of STLC + recursive types.

4.1 Step-indexed logical relations for type safety of STLC with recursive types

To prove type safety for the STLC extended with recursive types, we might be tempted to try to extend the $\mathcal{V}[\![-]\!]$ logical relation for the recursive type case:

$$\mathcal{V}[\![\mu\alpha.\tau]\!] \triangleq \{\text{fold } v \mid \text{unfold (fold } v) \in \mathcal{E}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\}$$

but notice that unfold (fold v) $\mapsto v$, so simplifying this definition, and noticing that v is a value, we get:

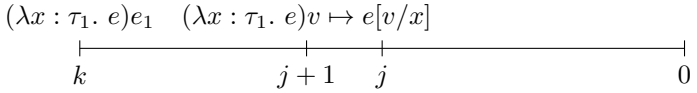
$$\mathcal{V}[\![\mu\alpha.\tau]\!] \triangleq \{\text{fold } v \mid v \in \mathcal{V}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\}$$

But now $\mathcal{V}[\![\tau]\!]$ is no longer well-founded, because the definition of $\mathcal{V}[\![\mu\alpha.\tau]\!]$ refers to $\mathcal{V}[\![\tau[\mu\alpha.\tau/\alpha]]\!]$, which is indexed by a larger type. To address this problem, we will index our logical relations with a step index k, in addition to a type. The idea is that we will define new relations, $\mathcal{V}_k[\![\tau]\!]$ and $\mathcal{E}_k[\![\tau]\!]$, that contain values and expressions that behave like type τ *for up to k reduction steps.*

$$\begin{aligned}
\mathcal{V}_k[\![\text{bool}]\!] &\triangleq \{\text{true}, \text{false}\} \\
\mathcal{V}_k[\![\tau_1 \to \tau_2]\!] &\triangleq \{\lambda x : \tau_1.\ e \mid \forall j < k. \forall v \in \mathcal{V}_j[\![\tau_1]\!]. e[v/x] \in \mathcal{E}_j[\![\tau_2]\!]\} \\
\mathcal{V}_k[\![\mu\alpha.\tau]\!] &\triangleq \{\text{fold } v \mid \forall j < k. v \in \mathcal{V}_j[\![\tau[\mu\alpha.\tau/\alpha]]\!]\} \\
\mathcal{E}_k[\![\tau]\!] &\triangleq \{e \mid \forall j < k. \forall e'.\ e \mapsto_j e' \wedge \text{irred}(e') \Rightarrow e' \in \mathcal{V}_{k-j}[\![\tau]\!]\} \\
\mathcal{G}_k[\![\cdot]\!] &\triangleq \{\emptyset\} \\
\mathcal{G}_k[\![\Gamma, x : \tau]\!] &\triangleq \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}_k[\![\Gamma]\!] \wedge v \in \mathcal{V}_k[\![\tau]\!]\}
\end{aligned}$$

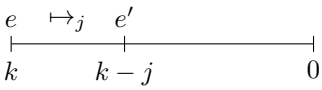
We remark that we do not define the logical relation for the type α. The reason is quite subtle: the type α exists only under the $\mu\alpha.\tau$, which is immediately replaced. More precisely, it is because the α only appears under the μ.

To understand the definition of $\mathcal{V}_k[\![\tau_1 \to \tau_2]\!]$, consider the following timeline:



At the beginning, we have k reduction steps left to take. At some point in the future, after the program runs for some number of steps, say when we have $j+1$ steps left (where $j < k$ by definition), the lambda expression $\lambda x : \tau_1.\ e$ receives a value as an argument, and we take one reduction step to reduce the application $(\lambda x : \tau_1.\ e)v$ to $e[v/x]$, so we are left with j reduction steps to take.

The definition of $\mathcal{E}_k[\![\tau]\!]$ uses a different timeline:



At the beginning, we have k reduction steps left to reduce e. We take j (for some $j < k$) reduction steps and end up reducing e to e', and at that point, we have $k - j$ reduction steps left.

The fundamental property of $\mathcal{E}[\![-]\!]$ has the same form, namely $\Gamma \vdash e : \tau \Rightarrow \Gamma \vDash e : \tau$. However, the definition of semantic type safety has changed:

$$\Gamma \vDash e : \tau \triangleq \forall k \geq 0. \forall \gamma \in \mathcal{G}_k[\![\Gamma]\!]. \gamma(e) \in \mathcal{E}_k[\![\tau]\!]$$

The proof of the fundamental property for this step-indexed logical relation will again proceed by induction on the typing judgement. Let us walk through the case T-Fold (repeated below for reference):

$$\text{T-Fold}\ \frac{\Delta; \Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Delta; \Gamma \vdash \text{fold } e : \mu\alpha.\tau}$$

We want to show that $\Gamma \vDash \text{fold } e : \tau$.

It will be helpful for us to have a small lemma:

Lemma (Monotonicity)**.** *If $v \in \mathcal{V}_k[\![\tau]\!]$ and $j \leq k$ then $v \in \mathcal{V}_j[\![\tau]\!]$.*

The proof for the lemma proceeds by induction on τ. The reader is encouraged to try proving the lemma for the $\tau_1 \to \tau_2$ case. It may also be illustrative to try proving the lemma with $j = k-1$ and observe where the proof fails.

We will now continue with proving the fundamental property.

Suppose $k \geq 0$ and $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$. We want to show that

$$\gamma(\text{fold } e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$$

Note that we can apply the substitution to e alone, so this is equivalent to $\text{fold}(\gamma(e)) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$.

Suppose $j < k$ and $\text{fold}(\gamma(e)) \mapsto^j e' \wedge \text{irred}(e')$. We want to show that

$$e' \in \mathcal{V}_{k-j}[\![\mu\alpha.\tau]\!]$$

We know that $\text{fold}(\gamma(e)) \mapsto^{j_1} \text{fold}(e_1)$, where $j_1 \leq j$ and $\text{irred}(e_1)$. By the inductive hypothesis, $\Gamma \vDash e : \tau[\mu\alpha.\tau/\alpha]$. We will instantiate the definition of type safety with γ and k, and note that $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$. By the definition of $\vDash$, we have that

$$\gamma(e) \in \mathcal{E}_k[\![\tau[\mu\alpha.\tau/\alpha]]\!]$$

We instantiate $\mathcal{E}_k[\![\tau[\mu\alpha.\tau/\alpha]]\!]$ with j_1 and noting that since $j_1 \leq j < k$, $j_1 < k$.

Therefore, we see that

$$e_1 \in \mathcal{V}_{k-j}[\![\tau[\mu\alpha.\tau/\alpha]]\!]$$

Therefore, $\text{fold}(e_1) = \text{fold}(v_1) = e'$ and $j_1 = j$. It then suffices to show that

$$\text{fold}(v_1) \in \mathcal{V}_{k-j}[\![\mu\alpha.\tau]\!]$$

Suppose $j' < k - j$. We must show that

$$v_1 \in \mathcal{V}_{j'}[\![\tau[\mu\alpha.\tau/\alpha]]\!]$$

This follows from the monotonicity lemma.

4.2 Step-indexed logical relation with mutable state

A little digression, to go further in the step-indexed logical relation. If the language also contains a heap with mutable references, then not only must the operational semantics describes the heap state, but the logical relation must also be parameterized by a *world* describing the state of the heap.

This kind of world comes from the so-called *Kripke Logical Relation*, describing the set of values behaving like a type, in a particular world. However, the logical relation also have to describe the future worlds $W \sqsubseteq W'$, meaning that W' is reachable (in the future) by the world W.

For instance, the future reachable worlds are only those that have less steps to do. In the case of a typed heap without deallocation (only allocation), the future reachable worlds are only those for which the domain of references grows. For more information, we refer the reader to Amal Ahmed's dissertation [Ahm04, Chapters 2-3]

5 Polymorphism

In this last section, we give a flavor about defining a logical relation for parametric polymorphism.

5.1 Formalization of SystemF

$$\tau ::= \text{bool} \mid \tau \to \tau \mid \alpha \mid \forall\alpha.\tau$$
$$e ::= x \mid \text{true} \mid \text{false} \mid \lambda x.\ e \mid \Lambda\alpha.\ e \mid e[\tau] \mid e\ e \mid \text{fold}\ e \mid \text{unfold}\ e$$
$$v ::= \text{true} \mid \text{false} \mid \lambda x : \tau.\ e \mid \text{fold}\ v \mid \Lambda\alpha.v$$
$$E ::= [] \mid \text{if E then } e_1 \text{ else } e_2 \mid \text{E}\ e \mid v\ \text{E} \mid \text{E}[\tau] \mid \text{fold}\ E \mid \text{unfold}\ E$$

$$\text{E-IfThenElseTrue}\ \frac{}{\text{if true then } e_1 \text{ else } e_2 \mapsto e_1} \qquad \text{E-IfThenElseFalse}\ \frac{}{\text{if false then } e_1 \text{ else } e_2 \mapsto e_2} \qquad \text{E-App}\ \frac{}{(\lambda x : \tau.\ e)\ v \mapsto e[v/x]}$$

$$\text{E-Step}\ \frac{e \mapsto e'}{E[e] \mapsto E[e']} \qquad \text{E-Fold}\ \frac{}{\text{unfold}(\text{fold}\ v) \mapsto v}$$

Figure 5: Call-by-value small step semantics of System F with evaluation context. $(\Lambda\alpha.e)[\tau] \mapsto e[\tau/\alpha]$.

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

$$\text{T-True}\ \frac{}{\Delta; \Gamma \vdash \text{true} : \text{bool}} \qquad \text{T-False}\ \frac{}{\Delta; \Gamma \vdash \text{false} : \text{bool}} \qquad \text{T-Var}\ \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}$$

$$\text{T-IfThenElse}\ \frac{\Delta; \Gamma \vdash e : \text{bool} \qquad \Delta; \Gamma \vdash e_1 : \tau \qquad \Delta; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau} \qquad \text{T-Abs}\ \frac{x : \tau_1, \Delta; \Gamma \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$$

$$\text{T-App}\ \frac{\Delta; \Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_1} \qquad \text{T-Fold}\ \frac{\Delta; \Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Delta; \Gamma \vdash \text{fold}\ e : \mu\alpha.\tau} \qquad \text{T-Unfold}\ \frac{\Delta; \Gamma \vdash e : \mu\alpha.\tau}{\Delta; \Gamma \vdash \text{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]}$$

$$\text{T-TAbs}\ \frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \qquad \text{T-Tapp}\ \frac{\Delta; \Gamma \vdash e : \forall\alpha.\tau \qquad \Delta \vdash \tau'}{\Delta; \Gamma \vdash e[\tau'] : \tau[\tau'/\alpha]}$$

Figure 6: Typing judgments of System F. Δ is a *type environment* consisting of type variables α.

5.2 Motivation and free theorems

The most well-known example of a free theorem concerns terms $\cdot \vdash e : \Lambda\alpha.\alpha \to \alpha$. In a language supporting parametric polymorphism such as System F, we know that e must behave like the polymorphic identity function. By this, we mean that for any closed type τ and and term $\cdot \vdash e' : \tau$, if $e\ [\tau]\ e'$ terminates then we know that it will reduce to e'. Intuitively, in a language with parametric polymorphism we are unable to inspect the type of terms, so we must produce something of type α in a uniform way. In a similar vein, parametric polymorphism also tells us that any term $\cdot \vdash e : \Lambda\alpha.\alpha \to \tau$ will behave like a constant function or that any term $\cdot \vdash e : \Lambda\alpha.\alpha$ must not terminate.

5.3 Logical relation for parametricity of SystemF

In this section, we define step-by-step the logical relation for polymorphism and explain what misses in a naive approach, and how to solve the problem.

We define a binary logical relation, $(v_1, v_2) \in \mathcal{V}[\![\tau]\!]$, saying that the values v_1 and v_2 are related under the type τ.

We first start by lift the unary logical relation of type safety of STLC to a binary logical relation, which states how to values of the same type behaves similarly.

$$\begin{aligned}\mathcal{V}[\![\text{bool}]\!] &\triangleq \{(\text{true},\ \text{true}),\ (\text{false},\ \text{false})\}\\ \mathcal{V}[\![\tau_1 \to \tau_2]\!] &\triangleq \{(\lambda x:\tau_1.\ e_1,\ \lambda x:\tau_1.\ e_2) \mid \forall (v_1,\ v_2) \in \mathcal{V}[\![\tau_1]\!].\ (e_1[v_1/x],\ e_2[v_2/x]) \in \mathcal{E}[\![\tau_2]\!]\}\end{aligned}$$

Then, let us try to extend the definition with the polymorphic case. Intuitively, what is expected is that, two values of type $\forall\alpha.\ \tau$ needs to be of the form $\Lambda\alpha.e$, and they relate if giving them a type τ to instantiate α, they should be related in the expression relation.

$$\mathcal{V}[\![\forall\alpha.\ \tau]\!] \triangleq \{(\Lambda\alpha.e_1,\ \Lambda\alpha.e_2) \mid \forall(\tau_1,\ \tau_2).\ (e_1[\tau_1/\alpha],\ e_2[\tau_2/\alpha])) \in \mathcal{E}[\![\tau[?/\alpha]]\!]\}$$

However, both values might receive a different type, and then it is not clear with which type we are supposed α in the expression relation. Actually, there is no good answer: it cannot be either τ_1 or τ_2. The solution is to keep τ unchanged, and keep track of the type substitutions done all over the execution, recorded into a structure $\rho = \{\alpha \mapsto (\tau_1, \tau_2), \ldots\}$. We parameterize the logical relation by the type substitution ρ.

$$\begin{aligned}\mathcal{V}[\![\text{bool}]\!]_\rho &\triangleq \{(\text{true},\ \text{true}),\ (\text{false},\ \text{false})\}\\ \mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho &\triangleq \{(\lambda x:\tau_1.\ e_1,\ \lambda x:\tau_1.\ e_2) \mid \forall (v_1,\ v_2) \in \mathcal{V}[\![\tau_1]\!]_\rho.\ (e_1[v_1/x],\ e_2[v_2/x]) \in \mathcal{E}[\![\tau_2]\!]_\rho\}\\ \mathcal{V}[\![\forall\alpha.\ \tau]\!]_\rho &\triangleq \{(\Lambda\alpha.e_1,\ \Lambda\alpha.e_2) \mid \forall(\tau_1,\ \tau_2).\ (e_1[\tau_1/\alpha],\ e_2[\tau_2/\alpha])) \in \mathcal{E}[\![\tau[\tau/\alpha]]\!]_{\rho[\alpha\mapsto(\tau_1,\tau_2)]}\}\end{aligned}$$

It now remains to define the α case. Let us give it a shot.

$$\mathcal{V}[\![\alpha]\!]_\rho \triangleq \{(v_1,\ v_2) \mid \rho(\alpha) = (\tau_1, \tau_2) \wedge \ldots\}$$

And at this point, we do not know how to relate the values. This is actually where the polymorphism shines and reveal it's full power. The relation between the values of two type can actually be chosen: the true power of parametric polymorphism is the choice of this relation.

For each type variable, we get to choose a relation $R \in \mathcal{R}[\tau_1, \tau_2]$ that relates values v_1 of type τ_1 with values v_2 of type τ_2. This relation is also recorded in the substitution $\rho = \{\alpha \mapsto (\tau_1, \tau_2, R), \ldots\}$. It finally gives us a way to relates the values of type α.

The final definition of the binary logical relation is the following:

$$\begin{aligned}\mathcal{V}[\![\text{bool}]\!]_\rho &\triangleq \{(\text{true},\ \text{true}),\ (\text{false},\ \text{false})\}\\ \mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho &\triangleq \{(\lambda x:\tau_1.\ e_1,\ \lambda x:\tau_1.\ e_2) \mid \forall (v_1,\ v_2) \in \mathcal{V}[\![\tau_1]\!]_\rho.\ (e_1[v_1/x],\ e_2[v_2/x]) \in \mathcal{E}[\![\tau_2]\!]_\rho\}\\ \mathcal{V}[\![\forall\alpha.\ \tau]\!]_\rho &\triangleq \{(\Lambda\alpha.e_1,\ \Lambda\alpha.e_2) \mid \forall(\tau_1,\ \tau_2).\ R \in \mathcal{R}[\tau_1, \tau_2].\ (e_1[\tau_1/\alpha],\ e_2[\tau_2/\alpha])) \in \mathcal{E}[\![\tau[\tau/\alpha]]\!]_{\rho[\alpha\mapsto(\tau_1,\tau_2,R)]}\}\\ \mathcal{V}[\![\alpha]\!]_\rho &\triangleq \{(v_1,\ v_2) \mid \rho(\alpha) = (\tau_1, \tau_2, R) \wedge (v_1,\ v_2) \in R\}\end{aligned}$$

We didn't define the expression relation, nor the fundamental theorem during the lecture. We refer the interested reader to the Logical Relation course from OPLSS 2015, Lectures 3, 4, and 5. The videos of the lectures are available online [Ahm15].

References

[Mit86] John C. Mitchell. "Representation Independence and Data Abstraction". In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86. St. Petersburg Beach, Florida: Association for Computing Machinery, 1986, pp. 263–276. ISBN: 9781450373470. DOI: 10.1145/512644.512669. URL: https://doi.org/10.1145/512644.512669.

[Wad89] Philip Wadler. "Theorems for Free!" In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, pp. 347–359. ISBN: 0897913280. DOI: 10.1145/99370.99404. URL: https://doi.org/10.1145/99370.99404.

[Pie02] Benjamin C Pierce. *Types and programming languages*. eng. Cambridge, Massachusetts ; London, 2002.

[Ahm04] Amal Jamil Ahmed. "Semantics of Types for Mutable State". AAI3136691. PhD thesis. USA, 2004.

[Ahm15] Amal Ahmed. *Logical Relations*. Eugene, Oregon, July 2015. URL: https://www.youtube.com/watch?v=F8E2_8b1hS4&list=PLiHLLF-foEeykV1sxQFZdQZvchYVkW5Tw&index=29.