# Logical Relations in Coq

Paul Wild

Friedrich-Alexander-Universität Erlangen-Nürnberg

2016-05-03

# Table of Contents

# Language and type system

$$
\begin{aligned}
\tau &::= \alpha \mid \texttt{unit} \mid \texttt{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \mu\alpha.\tau \\
e &::= x \mid \varepsilon \mid n \mid \langle e_1, e_2 \rangle \mid \pi_1\, e \mid \pi_2\, e \mid \\
&\quad \texttt{inl}\; e \mid \texttt{inr}\; e \mid \texttt{case}\; e\; \texttt{of}\; [\texttt{inl}\; x \Rightarrow e_1, \texttt{inr}\; x \Rightarrow e_2] \mid \\
&\quad \lambda x.e \mid e_1\, e_2 \mid \Lambda\alpha.e \mid e[\tau] \mid \\
&\quad \texttt{pack}\,(e, \tau_1)\,\texttt{as}\,\exists\alpha.\,\tau \mid \texttt{unpack}\,(x, \alpha) = e\;\texttt{in}\;e_1 \mid \\
&\quad \texttt{fold}\; e \mid \texttt{unfold}\; e
\end{aligned}
$$

# Language and type system

$$
\begin{aligned}
\tau &::= \alpha \mid \texttt{unit} \mid \texttt{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \mu\alpha.\tau \\
e &::= x \mid \varepsilon \mid n \mid \langle e_1, e_2\rangle \mid \pi_1 e \mid \pi_2 e \mid \\
&\quad\ \texttt{inl}\, e \mid \texttt{inr}\, e \mid \texttt{case}\, e\, \texttt{of}\, [\texttt{inl}\, x \Rightarrow e_1, \texttt{inr}\, x \Rightarrow e_2] \mid \\
&\quad\ \lambda x.e \mid e_1\, e_2 \mid \Lambda\alpha.e \mid e[\tau] \mid \\
&\quad\ \texttt{pack}\,(e, \tau_1)\,\texttt{as}\,\exists\alpha.\,\tau \mid \texttt{unpack}\,(x, \alpha) = e\,\texttt{in}\,e_1 \mid \\
&\quad\ \texttt{fold}\, e \mid \texttt{unfold}\, e
\end{aligned}
$$

This language is equipped with standard typing rules and a call-by-value (cbv) small-step operational semantics.

# Language and type system

$$
\begin{aligned}
\tau &::= \alpha \mid \texttt{unit} \mid \texttt{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \mu\alpha.\tau \\
e &::= x \mid \varepsilon \mid n \mid \langle e_1, e_2\rangle \mid \pi_1 e \mid \pi_2 e \mid \\
&\quad\ \texttt{inl}\, e \mid \texttt{inr}\, e \mid \texttt{case}\, e\, \texttt{of}\, [\texttt{inl}\, x \Rightarrow e_1, \texttt{inr}\, x \Rightarrow e_2] \mid \\
&\quad\ \lambda x.e \mid e_1\, e_2 \mid \Lambda\alpha.e \mid e[\tau] \mid \\
&\quad\ \texttt{pack}\,(e, \tau_1)\,\texttt{as}\,\exists\alpha.\,\tau \mid \texttt{unpack}\,(x, \alpha) = e\,\texttt{in}\,e_1 \mid \\
&\quad\ \texttt{fold}\, e \mid \texttt{unfold}\, e
\end{aligned}
$$

This language is equipped with standard typing rules and a call-by-value (cbv) small-step operational semantics.

In the Coq development, De Bruijn indices are used to represent both term and type level variables. Also, there are no types in the term language.

# Table of Contents

# Type safety

A property of many typed $\lambda$-calculi is *type safety*:

$$\forall e.\ \bullet \vdash e : \tau \Rightarrow \forall e'.\ e \mapsto^* e' \Rightarrow \mathrm{value}(e') \vee \exists e''.\ e' \mapsto e''$$

Informally it means that well-typed terms don't get stuck: computation only stops once a value is reached (or runs forever).

# Type safety

A property of many typed $\lambda$-calculi is *type safety*:

$$\forall e.\ \bullet \vdash e : \tau \Rightarrow \forall e'.\ e \mapsto^* e' \Rightarrow \mathrm{value}(e') \vee \exists e''.\ e' \mapsto e''$$

Informally it means that well-typed terms don't get stuck: computation only stops once a value is reached (or runs forever).

One can try to prove this by induction on the typing derivation, but this quickly fails. A stronger proof method is needed.

# Type safety

A property of many typed $\lambda$-calculi is *type safety*:

$$\forall e.\ \bullet \vdash e : \tau \Rightarrow \forall e'.\ e \mapsto^* e' \Rightarrow \mathrm{value}(e') \vee \exists e''.\ e' \mapsto e''$$

Informally it means that well-typed terms don't get stuck: computation only stops once a value is reached (or runs forever).

One can try to prove this by induction on the typing derivation, but this quickly fails. A stronger proof method is needed.

We define a new relation $\Gamma \vDash e : \tau$ (the logical relation) and show the following two properties, called the *Fundamental Property* and *Soundness*:

$$\Gamma \vdash e : \tau \Rightarrow \Gamma \vDash e : \tau$$

$$\bullet \vDash e : \tau \Rightarrow \forall e'.\ e \mapsto^* e' \Rightarrow \mathrm{value}(e') \vee \exists e''.\ e' \mapsto e''$$

# Natural numbers and products

Before we arrive at the definition of $\Gamma \vDash e : \tau$, we need to make our way through several layers.
First we need to define the notion of a semantic type, by induction on the syntax of types.

# Natural numbers and products

Before we arrive at the definition of $\Gamma \vDash e : \tau$, we need to make our way through several layers.
First we need to define the notion of a semantic type, by induction on the syntax of types.

A first idea is to define a semantic type to be the set of (closed) values of that type. For $\mathtt{nat}, \mathtt{unit}, \tau_L \times \tau_R, \tau_L + \tau_R$, this works fine:

$$\begin{aligned}
\mathcal{V}[\![\mathtt{nat}]\!] &= \{n \mid n \in \mathbb{N}\} \\
\mathcal{V}[\![\mathtt{unit}]\!] &= \{\epsilon\} \\
\mathcal{V}[\![\tau_L \times \tau_R]\!] &= \{\langle e^L, e^R \rangle \mid e^L \in \mathcal{V}[\![\tau_L]\!], e^R \in \mathcal{V}[\![\tau_R]\!]\} \\
\mathcal{V}[\![\tau_L + \tau_R]\!] &= \{\mathtt{inl}\, e \mid e \in \mathcal{V}[\![\tau_L]\!]\} \cup \{\mathtt{inr}\, e \mid e \in \mathcal{V}[\![\tau_R]\!]\}
\end{aligned}$$

# Functions

As we are working with a cbv operational semantics, computation under $\lambda$s is suspended. We need a way to say that a closed term $e$ might itself not be in the value interpretation, but reduces down to a value $v$ that is:

# Functions

As we are working with a cbv operational semantics, computation under $\lambda$s is suspended. We need a way to say that a closed term $e$ might itself not be in the value interpretation, but reduces down to a value $v$ that is:

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] = \{\lambda x.\, e \mid \forall v \in \mathcal{V}[\![\tau_1]\!].\, e[v/x] \in \mathcal{E}[\![\tau_2]\!]\}$$
$$\mathcal{E}[\![\tau]\!] = \{e \mid \exists v.\, e \mapsto^* v, v \in \mathcal{V}[\![\tau]\!]\}$$

$\mathcal{E}[\![\tau]\!]$ is called the *evaluation closure* of $\mathcal{V}[\![\tau]\!]$.

# Functions

As we are working with a cbv operational semantics, computation under $\lambda$s is suspended. We need a way to say that a closed term $e$ might itself not be in the value interpretation, but reduces down to a value $v$ that is:

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] = \{\lambda x.\, e \mid \forall v \in \mathcal{V}[\![\tau_1]\!].\, e[v/x] \in \mathcal{E}[\![\tau_2]\!]\}$$
$$\mathcal{E}[\![\tau]\!] = \{e \mid \exists v.\, e \mapsto^* v, v \in \mathcal{V}[\![\tau]\!]\}$$

$\mathcal{E}[\![\tau]\!]$ is called the *evaluation closure* of $\mathcal{V}[\![\tau]\!]$.

For now, we ignore the issue of non-terminating programs (which we also want to have in our evaluation closure). We will have to tweak this definition once we have enough tools at our disposal.

# Universal and existential types

Let $T$ be the set of semantic types:

$$T = \{R \mid \forall v \in R.\, \mathrm{value}(v)\}$$

An environment of semantic types is a finite map $\eta$ from type variables to semantic types.

# Universal and existential types

Let $T$ be the set of semantic types:

$$T = \{R \mid \forall v \in R.\, \mathrm{value}(v)\}$$

An environment of semantic types is a finite map $\eta$ from type variables to semantic types.

Now we can give an interpretation for type variables:

$$\mathcal{V}[\![\alpha]\!]\eta = \eta(\alpha)$$

as well as interpretations for $\forall\alpha.\tau$ and $\exists\alpha.\tau$:

$$\mathcal{V}[\![\forall\alpha.\tau]\!]\eta = \{\Lambda e \mid \forall R.\, e \in \mathcal{E}[\![\tau]\!]\eta[\alpha \mapsto R]\}$$
$$\mathcal{V}[\![\exists\alpha.\tau]\!]\eta = \{\mathtt{pack}\, e \mid \exists R.\, e \in \mathcal{V}[\![\tau]\!]\eta[\alpha \mapsto R]\}$$

# Universal and existential types

Let $T$ be the set of semantic types:

$$T = \{R \mid \forall v \in R.\, \mathrm{value}(v)\}$$

An environment of semantic types is a finite map $\eta$ from type variables to semantic types.

Now we can give an interpretation for type variables:

$$\mathcal{V}[\![\alpha]\!]\eta = \eta(\alpha)$$

as well as interpretations for $\forall\alpha.\tau$ and $\exists\alpha.\tau$:

$$\mathcal{V}[\![\forall\alpha.\tau]\!]\eta = \{\Lambda e \mid \forall R.\, e \in \mathcal{E}[\![\tau]\!]\eta[\alpha \mapsto R]\}$$
$$\mathcal{V}[\![\exists\alpha.\tau]\!]\eta = \{\mathtt{pack}\, e \mid \exists R.\, e \in \mathcal{V}[\![\tau]\!]\eta[\alpha \mapsto R]\}$$

The previous definitions are changed to include type environments $\eta$ without ever modifying them.

# Recursive types

For recursive types, we would want to define

$$\mathcal{V}[\![\mu\alpha.\tau]\!]\eta = \{\texttt{fold}\, e \mid e \in \mathcal{V}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\eta\}.$$

However, the value interpretation $\mathcal{V}[\![\cdot]\!]$ is defined by induction on types, and the above would make this definition not well-founded.

# Recursive types

For recursive types, we would want to define

$$\mathcal{V}[\![\mu\alpha.\tau]\!]\eta = \{\texttt{fold}\, e \mid e \in \mathcal{V}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\eta\}.$$

However, the value interpretation $\mathcal{V}[\![\cdot]\!]$ is defined by induction on types, and the above would make this definition not well-founded.

Instead we are going to use the technique of *step-indexing*: we index the value interpretation by a natural number $k$ and take $v \in \mathcal{V}_k[\![\tau]\!]$ to mean that $v$ is "good" for $k$ steps: if it is used in any program context and that program is run for up to $k$ steps, the computation does not get stuck.

# Recursive types

For recursive types, we would want to define

$$\mathcal{V}[\![\mu\alpha.\tau]\!]\eta = \{\texttt{fold}\, e \mid e \in \mathcal{V}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\eta\}.$$

However, the value interpretation $\mathcal{V}[\![\cdot]\!]$ is defined by induction on types, and the above would make this definition not well-founded.

Instead we are going to use the technique of *step-indexing*: we index the value interpretation by a natural number $k$ and take $v \in \mathcal{V}_k[\![\tau]\!]$ to mean that $v$ is "good" for $k$ steps: if it is used in any program context and that program is run for up to $k$ steps, the computation does not get stuck.

Now, we can give the correct definition of $\mathcal{V}[\![\mu\alpha.\tau]\!]$:

$$\mathcal{V}_k[\![\mu\alpha.\tau]\!]\eta = \{\texttt{fold}\, e \mid e \in \mathcal{V}_{k-1}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\eta\}.$$

# Functions, revisited

With step-indexing added to our toolset, the definition of $\mathcal{V}[\![\tau_1 \to \tau_2]\!]$ now looks like this:

$$\mathcal{V}_k[\![\tau_1 \to \tau_2]\!]\eta = \{\lambda x.\, e \mid \forall j \leq k.\, \forall v \in \mathcal{V}_j[\![\tau_1]\!].\, e[v/x] \in \mathcal{E}_j[\![\tau_2]\!]\eta\}$$

# Functions, revisited

With step-indexing added to our toolset, the definition of $\mathcal{V}[\![\tau_1 \to \tau_2]\!]$ now looks like this:

$$\mathcal{V}_k[\![\tau_1 \to \tau_2]\!]\eta = \{\lambda x.\, e \mid \forall j \leq k.\, \forall v \in \mathcal{V}_j[\![\tau_1]\!].\, e[v/x] \in \mathcal{E}_j[\![\tau_2]\!]\eta\}$$

The evaluation closure also changes considerably:

$$\mathcal{E}_k[\![\tau]\!]\eta = \{e \mid \forall j < k.\, \forall e'.\, e \mapsto^j e' \wedge \mathrm{irred}(e') \Rightarrow e' \in \mathcal{V}_{k-j}[\![\tau]\!]\eta\}$$

# The logical relation

There is one final ingredient before we can define the logical relation itself; a semantic interpretation of typing contexts:

$$\mathcal{G}_k[\![\bullet]\!]\eta = \{\emptyset\}$$
$$\mathcal{G}_k[\![\Gamma, x : \tau]\!]\eta = \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}_k[\![\Gamma]\!]\eta \wedge v \in \mathcal{V}_k[\![\tau]\!]\eta\}$$

# The logical relation

There is one final ingredient before we can define the logical relation itself; a semantic interpretation of typing contexts:

$$\mathcal{G}_k[\![\bullet]\!]\eta = \{\emptyset\}$$
$$\mathcal{G}_k[\![\Gamma, x : \tau]\!]\eta = \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}_k[\![\Gamma]\!]\eta \wedge v \in \mathcal{V}_k[\![\tau]\!]\eta\}$$

Finally, the definition of the logical relation: the open term $e$ is *semantically* of type $\tau$, if at any step level $k$ and for any way to substitute values for the variables in $e$, the resulting term is in $\mathcal{E}_k[\![\tau]\!]$:

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

# The Fundamental Property

It remains to establish the Fundamental Property and Soundness of the logical relation:

$$\Gamma \vdash e : \tau \Rightarrow \Gamma \vDash e : \tau$$

- $\vDash e : \tau \Rightarrow \forall e'.\, e \mapsto^* e' \Rightarrow \mathrm{value}(e') \vee \exists e''.\, e' \mapsto e''$

# The Fundamental Property

It remains to establish the Fundamental Property and Soundness of the logical relation:

$$\Gamma \vdash e : \tau \Rightarrow \Gamma \vDash e : \tau$$

- $\bullet \vDash e : \tau \Rightarrow \forall e'.\, e \mapsto^* e' \Rightarrow \mathrm{value}(e') \vee \exists e''.\, e' \mapsto e''$

The latter follows in a simple manner from the definition of the logical relation.

# The Fundamental Property

It remains to establish the Fundamental Property and Soundness of the logical relation:

$$\Gamma \vdash e : \tau \Rightarrow \Gamma \vDash e : \tau$$

- $\bullet \vDash e : \tau \Rightarrow \forall e'.\, e \mapsto^* e' \Rightarrow \mathrm{value}(e') \vee \exists e''.\, e' \mapsto e''$

The latter follows in a simple manner from the definition of the logical relation.

The proof of the former works by induction on the typing derivation. The cases for each of the typing rules are traditionally called *Compatibility lemmas*, e.g.:

$$\Gamma, x : \tau_1 \vDash e : \tau_2 \Rightarrow \Gamma \vDash \lambda x.\, e : \tau_1 \to \tau_2$$

$$\Gamma \vDash e_1 : \tau_1 \to \tau_2, \Gamma \vDash e_2 : \tau_1 \Rightarrow \Gamma \vDash e_1\, e_2 : \tau_2$$

# Table of Contents

# Contextual equivalence

A context $C[\cdot]$ is a term with a hole. It is straightforward to define substitution of terms into and typing $C[\cdot] : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')$ of contexts by induction.

# Contextual equivalence

A context $C[\cdot]$ is a term with a hole. It is straightforward to define substitution of terms into and typing $C[\cdot] : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')$ of contexts by induction.

Let $\Gamma \vdash e_1 : \tau, \Gamma \vdash e_2 : \tau$. Then we can define the notion of *contextual equivalence*:

$$\Gamma \vdash e_1 \approx_{\mathrm{ctx}} e_2 : \tau \Leftrightarrow$$
$$(\forall \Gamma', C[\cdot] : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \mathtt{unit}).\ C[e_1] \Downarrow \Leftrightarrow C[e_2] \Downarrow)$$

# Contextual equivalence

A context $C[\cdot]$ is a term with a hole. It is straightforward to define substitution of terms into and typing $C[\cdot] : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')$ of contexts by induction.

Let $\Gamma \vdash e_1 : \tau, \Gamma \vdash e_2 : \tau$. Then we can define the notion of *contextual equivalence*:

$$\Gamma \vdash e_1 \approx_{\mathrm{ctx}} e_2 : \tau \Leftrightarrow$$
$$(\forall \Gamma', C[\cdot] : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \mathtt{unit}).\ C[e_1] \Downarrow \Leftrightarrow C[e_2] \Downarrow)$$

As with type safety before, proving contextual equivalence of two terms directly can be quite hard, so we are looking for a logical relation that is *sound* with respect to contextual equivalence:

$$\Gamma \vdash e_1 \approx_{\log} e_2 : \tau \Rightarrow \Gamma \vdash e_1 \approx_{\mathrm{ctx}} e_2 : \tau$$

# Contextual approximation

As it turns out, it is often easier to instead build a logical relation that is sound w.r.t. contextual approximation:

$$\Gamma \vdash e_1 \preceq_{\mathrm{ctx}} e_2 : \tau \Leftrightarrow$$
$$(\forall \Gamma', C[\cdot] : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \texttt{unit}).\, C[e_1] \Downarrow \Rightarrow C[e_2] \Downarrow)$$

Contextual equivalence is then shown by showing approximation in both directions.

# Contextual approximation

As it turns out, it is often easier to instead build a logical relation that is sound w.r.t. contextual approximation:

$$\Gamma \vdash e_1 \preceq_{\mathrm{ctx}} e_2 : \tau \Leftrightarrow$$
$$(\forall \Gamma', C[\cdot] : (\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \mathtt{unit}).\, C[e_1] \Downarrow \Rightarrow C[e_2] \Downarrow)$$

Contextual equivalence is then shown by showing approximation in both directions.

As in the unary case, the logical relation will consist of several layers:

- relations $\mathcal{V}_k[\![\tau]\!]\eta$ of pairs of closed values
- relations $\mathcal{E}_k[\![\tau]\!]\eta$ of pairs of closed terms
- relations $\Gamma \vdash \cdot \preceq_{\log} \cdot : \tau$ of pairs of open terms

# Value interpretations

Defining the $\mathcal{V}_k[\![\tau]\!]\eta$ is straightforward:

$$
\begin{aligned}
\mathcal{V}_k[\![\mathtt{nat}]\!]\eta &= \{(n, n) \mid n \in \mathbb{N}\} \\
\mathcal{V}_k[\![\mathtt{unit}]\!]\eta &= \{(\epsilon, \epsilon)\} \\
\mathcal{V}_k[\![\tau_L \times \tau_R]\!]\eta &= \{(\langle e_1^L, e_1^R\rangle, \langle e_2^L, e_2^R\rangle) \mid \\
&\qquad (e_1^L, e_2^L) \in \mathcal{V}_k[\![\tau_L]\!]\eta, (e_1^R, e_2^R) \in \mathcal{V}_k[\![\tau_R]\!]\eta\} \\
\mathcal{V}_k[\![\tau_L + \tau_R]\!]\eta &= \{(\mathtt{inl}\, e_1, \mathtt{inl}\, e_2) \mid (e_1, e_2) \in \mathcal{V}_k[\![\tau_L]\!]\eta\} \\
&\quad \cup \{(\mathtt{inr}\, e_1, \mathtt{inr}\, e_2) \mid (e_1, e_2) \in \mathcal{V}_k[\![\tau_R]\!]\eta\} \\
\mathcal{V}_k[\![\tau_1 \to \tau_2]\!]\eta &= \{(\lambda x.\, e_1, \lambda x.\, e_2) \mid \forall j \leq k.\, \forall (v_1, v_2) \in \mathcal{V}_j[\![\tau_1]\!]\eta. \\
&\qquad (e_1[v_1/x], e_2[v_2/x]) \in \mathcal{E}_j[\![\tau_2]\!]\} \\
\mathcal{V}_k[\![\forall\alpha.\tau]\!]\eta &= \{(\Lambda e_1, \Lambda e_2) \mid \forall R.\, (e_1, e_2) \in \mathcal{E}_k[\![\tau]\!]\eta[\alpha \mapsto R]\} \\
\mathcal{V}_k[\![\exists\alpha.\tau]\!]\eta &= \{(\mathtt{pack}\, e_1, \mathtt{pack}\, e_2) \mid \exists R.\, (e_1, e_2) \in \mathcal{V}_k[\![\tau]\!]\eta[\alpha \mapsto R]\} \\
\mathcal{V}_k[\![\mu\alpha.\tau]\!]\eta &= \{(\mathtt{fold}\, e_1, \mathtt{fold}\, e_2) \mid (e_1, e_2) \in \mathcal{V}_{k-1}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\eta\}.
\end{aligned}
$$

# Binary semantic types

The set $T$ of semantic types is now:

$$T = \{(R_n)_{n \in \mathbb{N}} \mid \forall n \in \mathbb{N}, (v_1, v_2) \in R_n . \, \mathrm{value}(v_1) \wedge \mathrm{value}(v_2)\}$$

# Binary semantic types

The set $T$ of semantic types is now:

$$T = \{(R_n)_{n\in\mathbb{N}} \mid \forall n \in \mathbb{N}, (v_1, v_2) \in R_n.\, \mathrm{value}(v_1) \wedge \mathrm{value}(v_2)\}$$

Note that it is not required that the terms on the left side have the same types as the terms on the right side. This allows us to use the logical relation to prove, for example, that

$$(\mathtt{pack}(\langle 5, \mathtt{even}\rangle, \mathtt{nat}), \mathtt{pack}(\langle \mathtt{true}, \mathtt{not}\rangle, \mathtt{bool})) \in \mathcal{E}_k[\![\exists\alpha.\alpha \times (\alpha \to \mathtt{bool})]\!]\emptyset$$

We will look at a more complex example later.

# Evaluation closure

The evaluation closure changes again quite dramatically:

$$\mathcal{E}_k[\![\tau]\!]\eta = \{(e_1, e_2) \mid (\mathrm{value}(e_1) \Rightarrow \exists e_2'.\, e_2 \mapsto^* e_2' \wedge (e_1, e_2') \in \mathcal{V}_k[\![\tau]\!]\eta) \wedge (\forall e_1.\, e_1 \mapsto e_1' \Rightarrow (e_1', e_2) \in \mathcal{E}_{k-1}[\![\tau]\!]\eta)\}$$

# Logical approximation

As before, we need a semantic interpretation of typing contexts:

$$\mathcal{G}_k[\![\bullet]\!]\eta = \{\emptyset\}$$

$$\mathcal{G}_k[\![\Gamma, x : \tau]\!]\eta = \{(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \mid (\gamma_1, \gamma_2) \in \mathcal{G}_k[\![\Gamma]\!]\eta \wedge (v_1, v_2) \in \mathcal{V}_k[\![\tau]\!]\eta\}$$

# Logical approximation

As before, we need a semantic interpretation of typing contexts:

$$\mathcal{G}_k[\![\bullet]\!]\eta = \{\emptyset\}$$
$$\mathcal{G}_k[\![\Gamma, x:\tau]\!]\eta = \{(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \mid (\gamma_1, \gamma_2) \in \mathcal{G}_k[\![\Gamma]\!]\eta \wedge (v_1, v_2) \in \mathcal{V}_k[\![\tau]\!]\eta\}$$

Finally, here is the definition of logical approximation:

$$\Gamma \vdash e_1 \preceq_{\log} e_2 : \tau \Leftrightarrow \forall k \geq 0.\, \forall \eta.\, \forall (\gamma_1, \gamma_2) \in \mathcal{G}_k[\![\Gamma]\!]\eta.\, (\gamma_1(e_1), \gamma_2(e_2)) \in \mathcal{E}_k[\![\tau]\!]\eta$$

# Fundamental Property & Soundness

The Fundamental Property and Soundness of the logical relation can now be stated as follows:

$$\Gamma \vdash e : \tau \Rightarrow \Gamma \vdash e \preceq_{\log} e : \tau$$

$$\Gamma \vdash e_1 \preceq_{\log} e_2 : \tau \Rightarrow \Gamma \vdash e_1 \preceq_{\mathrm{ctx}} e_2 : \tau$$

# Fundamental Property & Soundness

The Fundamental Property and Soundness of the logical relation can now be stated as follows:

$$\Gamma \vdash e : \tau \Rightarrow \Gamma \vdash e \preceq_{\log} e : \tau$$

$$\Gamma \vdash e_1 \preceq_{\log} e_2 : \tau \Rightarrow \Gamma \vdash e_1 \preceq_{\mathrm{ctx}} e_2 : \tau$$

The Fundamental property is again shown by induction on the typing derivation, with the compatibility lemmas as cases.

# Fundamental Property & Soundness

The Fundamental Property and Soundness of the logical relation can now be stated as follows:

$$\Gamma \vdash e : \tau \Rightarrow \Gamma \vdash e \preceq_{\log} e : \tau$$

$$\Gamma \vdash e_1 \preceq_{\log} e_2 : \tau \Rightarrow \Gamma \vdash e_1 \preceq_{\mathrm{ctx}} e_2 : \tau$$

The Fundamental property is again shown by induction on the typing derivation, with the compatibility lemmas as cases.

The proof of Soundness goes by induction on the typing derivation of contexts $C[\cdot]$ and uses the Fundamental Property as well as the compatibility lemmas throughout.

# Table of Contents

# Queue example

Consider the types

$$\begin{aligned}\tau_{\texttt{list}} &= \mu\beta.\,\texttt{unit} + \texttt{nat} \times \beta \\ \tau_{\texttt{queue}} &= \exists\alpha.\,(\texttt{unit} \to \alpha) \times (\alpha \times \texttt{nat} \to \alpha) \\ &\qquad \times (\alpha \to \alpha) \times (\alpha \to \texttt{unit} + \texttt{nat})\end{aligned}$$

# Queue example

Consider the types

$$\begin{aligned}\tau_{\mathtt{list}} &= \mu\beta.\,\mathtt{unit} + \mathtt{nat} \times \beta \\ \tau_{\mathtt{queue}} &= \exists\alpha.\,(\mathtt{unit} \to \alpha) \times (\alpha \times \mathtt{nat} \to \alpha) \\ &\qquad \times (\alpha \to \alpha) \times (\alpha \to \mathtt{unit} + \mathtt{nat})\end{aligned}$$

First, there is a simple queue implementation using lists:

$$\begin{aligned}\mathtt{new1} &= \lambda x.\,\mathtt{nil} \\ \mathtt{push1} &= \lambda q.\lambda n.\,\mathtt{append}\,q\,[n] \\ \mathtt{pop1} &= \lambda q.\,\mathtt{case}\,\mathtt{unfold}\,q\,\mathtt{of}\,[\mathtt{inl}\,x \Rightarrow \mathtt{nil}, \mathtt{inr}\,x \Rightarrow \pi_2\,x] \\ \mathtt{top1} &= \lambda q.\,\mathtt{case}\,\mathtt{unfold}\,q\,\mathtt{of}\,[\mathtt{inl}\,x \Rightarrow \mathtt{inl}\varepsilon, \mathtt{inr}\,x \Rightarrow \mathtt{inr}\,(\pi_1\,x)] \\ \mathtt{queue1} &= \mathtt{pack}\,(\langle \mathtt{new1}, \mathtt{push1}, \mathtt{pop1}, \mathtt{top1}\rangle, \tau_{\mathtt{list}})\,\mathtt{as}\,\tau_{\mathtt{queue}}\end{aligned}$$

# Queue example

Consider the types

$$\begin{aligned}\tau_{\mathtt{list}} &= \mu\beta.\,\mathtt{unit} + \mathtt{nat} \times \beta \\ \tau_{\mathtt{queue}} &= \exists\alpha.\,(\mathtt{unit} \to \alpha) \times (\alpha \times \mathtt{nat} \to \alpha) \\ &\quad \times (\alpha \to \alpha) \times (\alpha \to \mathtt{unit} + \mathtt{nat})\end{aligned}$$

Then, there is this more efficient one using pairs of lists:

$$\begin{aligned}\mathtt{norm} &= \lambda q.\,\mathtt{case}\,\mathtt{unfold}\,(\pi_1\,q)\,\mathtt{of} \\ &\qquad [\mathtt{inl}\,x \Rightarrow \mathtt{reverse}\,(\pi_2\,q), \mathtt{nil}), \mathtt{inr}\,x \Rightarrow q] \\ \mathtt{new2} &= \lambda x.\,(\mathtt{nil}, \mathtt{nil}) \\ \mathtt{push2} &= \lambda q.\lambda n.\,\mathtt{norm}\,(\pi_1\,q, n :: \pi_2\,q) \\ \mathtt{pop2} &= \lambda q.\,\mathtt{case}\,\mathtt{unfold}\,(\pi_1\,q)\,\mathtt{of} \\ &\qquad [\mathtt{inl}\,x \Rightarrow (\mathtt{nil}, \mathtt{nil}), \mathtt{inr}\,x \Rightarrow \mathtt{norm}\,(\pi_2\,x, \pi_2\,q)] \\ \mathtt{top2} &= \lambda q.\,\mathtt{case}\,\mathtt{unfold}\,(\pi_1\,q)\,\mathtt{of}\,[\mathtt{inl}\,x \Rightarrow \mathtt{inl}\varepsilon, \mathtt{inr}\,x \Rightarrow \mathtt{inr}\,(\pi_1\,x)] \\ \mathtt{queue2} &= \mathtt{pack}\,(\langle \mathtt{new2}, \mathtt{push2}, \mathtt{pop2}, \mathtt{top2}\rangle, \tau_{\mathtt{list}} \times \tau_{\mathtt{list}})\,\mathtt{as}\,\tau_{\mathtt{queue}}\end{aligned}$$

# Queue example

`push1` has time complexity $\mathcal{O}(n)$, while `push2` has amortized time complexity $\mathcal{O}(1)$.

# Queue example

`push1` has time complexity $\mathcal{O}(n)$, while `push2` has amortized time complexity $\mathcal{O}(1)$.

We can show the two implementations to be logically (and therefore contextually) equivalent by providing a suitable interpretation $R$ for the type variable $\alpha$.

# Queue example

`push1` has time complexity $\mathcal{O}(n)$, while `push2` has amortized time complexity $\mathcal{O}(1)$.

We can show the two implementations to be logically (and therefore contextually) equivalent by providing a suitable interpretation $R$ for the type variable $\alpha$.

$$R_n = \{(\texttt{xs}, \langle \texttt{ys}, \texttt{zs} \rangle) \mid \texttt{append ys (reverse zs)} \mapsto^* \texttt{xs}\}$$

# Table of Contents

# Complete ordered families of equivalences

## Definition

An *ordered family of equivalences (ofe)* consists of a set $X$ and equivalence relations $\overset{n}{=}$, $n \in \mathbb{N}$, such that $\forall x, x'.\, x \overset{0}{=} x'$ and $\forall n, x, x'.\, x \overset{n+1}{=} x' \Rightarrow x \overset{n}{=} x'$.

# Complete ordered families of equivalences

## Definition

An *ordered family of equivalences (ofe)* consists of a set $X$ and equivalence relations $\overset{n}{=}$, $n \in \mathbb{N}$, such that $\forall x, x'.\, x \overset{0}{=} x'$ and $\forall n, x, x'.\, x \overset{n+1}{=} x' \Rightarrow x \overset{n}{=} x'$.

A sequence $(x_i)_{i \in \mathbb{N}}$, such that $\forall k.\, \exists n.\, \forall i, j \geq n.\, x_i \overset{k}{=} x_j$ is called *Cauchy chain*. An element $x \in X$ is called *limit* of $(x_i)_{i \in \mathbb{N}}$, if $\forall k.\, \exists n.\, \forall i \geq n.\, x_i \overset{k}{=} x$.

# Complete ordered families of equivalences

## Definition

An *ordered family of equivalences (ofe)* consists of a set $X$ and equivalence relations $\overset{n}{=}$, $n \in \mathbb{N}$, such that $\forall x, x'.\, x \overset{0}{=} x'$ and $\forall n, x, x'.\, x \overset{n+1}{=} x' \Rightarrow x \overset{n}{=} x'$.

A sequence $(x_i)_{i \in \mathbb{N}}$, such that $\forall k.\, \exists n.\, \forall i, j \geq n.\, x_i \overset{k}{=} x_j$ is called *Cauchy chain*. An element $x \in X$ is called *limit* of $(x_i)_{i \in \mathbb{N}}$, if $\forall k.\, \exists n.\, \forall i \geq n.\, x_i \overset{k}{=} x$.

An ofe $(X, (\overset{n}{=})_{n \in \mathbb{N}})$ is *complete* if all Cauchy chains have a limit.

# Complete ordered families of equivalences

## Definition

An *ordered family of equivalences (ofe)* consists of a set $X$ and equivalence relations $\overset{n}{=}$, $n \in \mathbb{N}$, such that $\forall x, x'.\, x \overset{0}{=} x'$ and $\forall n, x, x'.\, x \overset{n+1}{=} x' \Rightarrow x \overset{n}{=} x'$.

A sequence $(x_i)_{i\in\mathbb{N}}$, such that $\forall k.\, \exists n.\, \forall i, j \geq n.\, x_i \overset{k}{=} x_j$ is called *Cauchy chain*. An element $x \in X$ is called *limit* of $(x_i)_{i\in\mathbb{N}}$, if $\forall k.\, \exists n.\, \forall i \geq n.\, x_i \overset{k}{=} x$.

An ofe $(X, (\overset{n}{=})_{n\in\mathbb{N}})$ is *complete* if all Cauchy chains have a limit.

Remark: a cofe is the same as a complete bisected ultra-metric space where we set $d(x, x') = 0.5\hat{}(\sup\{n \mid x \overset{n}{=} x'\})$.

# Contractive maps

## Definition

A function $f : X \to Y$ between cofes is
*non-expansive*, if $\forall n, x, x'.\, x \overset{n}{=} x' \Rightarrow f(x) \overset{n}{=} f(x')$ and
*contractive*, if $\forall n, x, x'.\, x \overset{n}{=} x' \Rightarrow f(x) \overset{n+1}{=} f(x')$.

## Theorem (Banach)

*Let $f : X \to X$ be a contractive map on $X$. Then $f$ has a unique fixed point $\mu f$.*

# Uniform predicates as cofe

## Definition

A *uniform predicate* on a set $X$ is a set $P \subseteq \mathbb{N} \times X$, such that $\forall (k, x) \in P, j \leq k. (j, x) \in P$.

For $k \in \mathbb{N}$, define $\lfloor P \rfloor_k = \{(j, x) \in P \mid j < k\}$.

Also define $\triangleright P = \{(j, x) \mid j = 0 \vee (j - 1, x) \in P\}$, ("later" P).

# Uniform predicates as cofe

## Definition

A *uniform predicate* on a set $X$ is a set $P \subseteq \mathbb{N} \times X$, such that $\forall (k, x) \in P, j \leq k. (j, x) \in P$.
For $k \in \mathbb{N}$, define $\lfloor P \rfloor_k = \{(j, x) \in P \mid j < k\}$.
Also define $\triangleright P = \{(j, x) \mid j = 0 \vee (j - 1, x) \in P\}$, ("later" P).

## Lemma

*The set* $\mathrm{UPred}(X)$ *of uniform predicates on* $X$ *becomes a cofe by defining* $P \stackrel{n}{=} P' \Leftrightarrow \lfloor P \rfloor_n = \lfloor P' \rfloor_n$.

# Uniform predicates as cofe

## Definition

A *uniform predicate* on a set $X$ is a set $P \subseteq \mathbb{N} \times X$, such that $\forall (k, x) \in P, j \leq k. (j, x) \in P$.
For $k \in \mathbb{N}$, define $\lfloor P \rfloor_k = \{(j, x) \in P \mid j < k\}$.
Also define $\triangleright P = \{(j, x) \mid j = 0 \vee (j - 1, x) \in P\}$, ("later" P).

## Lemma

*The set* $\mathrm{UPred}(X)$ *of uniform predicates on* $X$ *becomes a cofe by defining* $P \overset{n}{=} P' \Leftrightarrow \lfloor P \rfloor_n = \lfloor P' \rfloor_n$.

The sets of semantic types for our logical relations turn out to be $\mathrm{UPred}(\mathrm{Exp})$ and $\mathrm{UPred}(\mathrm{Exp} \times \mathrm{Exp})$, respectively.

# Uniform predicates as cofe

## Definition

A *uniform predicate* on a set $X$ is a set $P \subseteq \mathbb{N} \times X$, such that $\forall (k, x) \in P, j \leq k. (j, x) \in P$.
For $k \in \mathbb{N}$, define $\lfloor P \rfloor_k = \{(j, x) \in P \mid j < k\}$.
Also define $\triangleright P = \{(j, x) \mid j = 0 \vee (j - 1, x) \in P\}$, ("later" P).

## Lemma

*The set* $\mathrm{UPred}(X)$ *of uniform predicates on* $X$ *becomes a cofe by defining* $P \overset{n}{=} P' \Leftrightarrow \lfloor P \rfloor_n = \lfloor P' \rfloor_n$.

The sets of semantic types for our logical relations turn out to be $\mathrm{UPred}(\mathrm{Exp})$ and $\mathrm{UPred}(\mathrm{Exp} \times \mathrm{Exp})$, respectively.

The definition of $\mathcal{V}[\![\mu\alpha.\tau]\!]$ can now be restated as applying the Banach fixed point operator to the contractive map:

$$R \mapsto \{\mathtt{fold}\, e \mid e \in \triangleright(\mathcal{V}[\![\tau]\!]\eta[\alpha \mapsto R])\}.$$

# The category $\mathcal{U}$ and ModuRes

**Lemma**

*Cofes and non-expanding maps form a cartesian closed category $\mathcal{U}$.*

# The category $\mathcal{U}$ and ModuRes

## Lemma

*Cofes and non-expanding maps form a cartesian closed category $\mathcal{U}$.*

ModuRes is a project aimed at developing mathematical models for reasoning about programming languages with features such as higher-order functions, mutable references and concurrency.

# The category $\mathcal{U}$ and ModuRes

**Lemma**

*Cofes and non-expanding maps form a cartesian closed category $\mathcal{U}$.*

ModuRes is a project aimed at developing mathematical models for reasoning about programming languages with features such as higher-order functions, mutable references and concurrency.

Its Coq development formalizes the theory of the category $\mathcal{U}$ via Coq typeclasses and comes together with a tutorial that (among other things) presents the construction of a unary logical relation for a smaller language that does not contain unit, sum and existential types.

# My work in the project

I extended the unary model to include these type constructors and then built the binary model for contextual approximation, including proofs of the Fundamental Properties and Soundness.

# My work in the project

I extended the unary model to include these type constructors and then built the binary model for contextual approximation, including proofs of the Fundamental Properties and Soundness.

Then I used the binary logical relation to formalize some examples in Coq, but...

# My work in the project

I extended the unary model to include these type constructors and then built the binary model for contextual approximation, including proofs of the Fundamental Properties and Soundness.

Then I used the binary logical relation to formalize some examples in Coq, but...

The current logical relations do not mention any syntactic types. Therefore it is not possible to show that terms that are in the logical relation at a certain type are also syntactically of that type.

# My work in the project

I extended the unary model to include these type constructors and then built the binary model for contextual approximation, including proofs of the Fundamental Properties and Soundness.

Then I used the binary logical relation to formalize some examples in Coq, but...

The current logical relations do not mention any syntactic types. Therefore it is not possible to show that terms that are in the logical relation at a certain type are also syntactically of that type.

I intend to add syntactic types into the logical relation to broaden the range of equivalences that can be shown using the logical relation. This requires changing the set of semantic types to be

$$T = \{(\tau_1, \tau_2, R) \mid R \in \mathrm{UPred}(\mathrm{Exp} \times \mathrm{Exp}),$$
$$R \text{ contains only closed values of the correct closed type}\}$$

This can be shown to be a cofe by using the fact that $\mathcal{U}$ is a ccc.

# References


D. Dreyer, A. Ahmed, L. Birkedal.
*Logical step-indexed logical relations.*
Logical Methods in Computer Science

B.C. Pierce.
*Types and Programming Languages.*
The MIT Press

F. Sieczkowski, A. Bizjak, Y. Zakowski, L. Birkedal.
*Modular Reasoning about Concurrent Higher-Order Imperative Programs: a Coq Tutorial.*

A. Ahmed.
*Logical Relations.*
OPLSS'15 (video lectures)