Logical Relations as Types

Jonathan Sterling  jww. Robert Harper
Carnegie Mellon University

CCS Colloquium, April 2021
Thanks to Harley Eades III for the invitation!
Software engineering is about division of labor between users and machines, between clients and servers, and between different programmers and modules. Tension lies between abstraction (division of labor) and composition (harmony of labor). PL theory (equal.osf) is advancing linguistic solutions to the contradiction between abstraction and composition (Reynolds, equal.osf/nine.osf/eight.osf/three.osf).
Software engineering is about division of labor
Software engineering is about division of labor between users and machines
Software engineering is about division of labor
between users and machines
between clients and servers
Software engineering is about division of labor
between users and machines
between clients and servers
between different programmers
Software engineering is about division of labor between users and machines between clients and servers between different programmers between different modules.
Software engineering is about division of labor between users and machines between clients and servers between different programmers between different modules.

Tension lies between
Software engineering is about division of labor between users and machines between clients and servers between different programmers between different modules.

Tension lies between abstraction (division of labor)
Software engineering is about division of labor between users and machines between clients and servers between different programmers between different modules.

Tension lies between abstraction (division of labor) and composition (harmony of labor).
Software engineering is about division of labor
between users and machines
between clients and servers
between different programmers
between different modules.

Tension lies between
  abstraction (division of labor)
  and composition (harmony of labor).

**PL theory** = advancing linguistic solutions to the contradiction between
abstraction and composition (Reynolds, 1983).
Software engineering is about division of labor between users and machines between clients and servers between different programmers between different modules.

Tension lies between abstraction (division of labor) and composition (harmony of labor).

**PL theory** = advancing linguistic solutions to the contradiction between abstraction and composition (Reynolds, 1983).
Consider a queue data structure.

```plaintext
def QUEUE =
sig
t : type
emp : t
enq : string × t → t
deq : t → option (string × t)
end
```
Consider a *queue* data structure.

```plaintext
def QUEUE =
sig
t : type
emp : t
enq : string × t → t
deq : t → option (string × t)
end
```

*queue representation type*
Consider a queue data structure.

```plaintext
def QUEUE =
sig
  t : type
  emp : t
  enq : string × t → t
  deq : t → option (string × t)
end
```
Consider a *queue* data structure.

```plaintext
def QUEUE =
sig
  t : type
  emp : t
  enq : string × t → t
  deq : t → option (string × t)
end
```
Consider a *queue* data structure.

```ocaml
def QUEUE =
sig
  t : type
  emp : t
  enq : string × t → t
  deq : t → option (string × t)
end
```
Queue implementation (ListQueue)

def ListQueue : QUEUE =
struct
  def t = list string
  def emp = []
  def enq (x, q) = x :: q
  def deq q =
    case rev q of
      | [] ⇒ None
      | x :: xs ⇒
        Some (x, rev xs)
end
def BatchedQueue : QUEUE =
struct
  def t = list string × list string
  def emp = ([], [])
  def enq (x, (fs, rs)) = (fs, x :: rs)
  def deq (fs, rs) =
      case fs of
          | [] ⇒
              (case rev rs of
                | [] ⇒ None
                | x :: rs' ⇒ Some (x, rs', [])
              | x :: fs' ⇒ Some (x, fs', rs)
      end
Two unequal queue implementations

def ListQueue : QUEUE =
struct
def t = list string
def emp = []
def enq (x, q) = x :: q
def deq q =
  case rev q of
  | [] ⇒ None
  | x :: xs ⇒
    Some (x, rev xs)
end

def BatchedQueue : QUEUE =
struct
def t = list string × list string
def emp = ([], [])
def enq (x, (fs, rs)) = (fs, x :: rs)
def deq (fs, rs) =
  case fs of
  | [] ⇒
    (case rev rs of
      | [] ⇒ None
      | x :: rs' ⇒ Some (x, rs', [])
    | x :: fs' ⇒ Some (x, fs', rs)
  end

We have ListQueue.t ≠ BatchedQueue.t, hence ListQueue ≠ BatchedQueue. But it is not possible to observe the difference between the two!
What does it mean to be different?

Two implementations $M/zero.osf$, $M/one.osf$: $S$ are observably different if there exists a program $C:S \rightarrow \text{bool}$ with $C(M/zero.osf) = \text{true}$ and $C(M/one.osf) = \text{false}$.

We call two implementations observationally equivalent when there is no such $C$. 

/eight.osf / /three.osf/seven.osf
What does it mean to be different?

Two implementations $M_0, M_1 : S$ are **observably different** if there exists a program $C : S \rightarrow \text{bool}$ with $C(M_0) = \text{true}$ and $C(M_1) = \text{false}$. 
What does it mean to be different?

Two implementations $M_0, M_1 : S$ are **observably different** if there exists a program $C : S \rightarrow \text{bool}$ with $C(M_0) = \text{true}$ and $C(M_1) = \text{false}$.

We call two implementations **observationally equivalent** when there is no such $C$. 
def ListQueue : QUEUE =
struct
def t = list string
def emp = []
def enq (x, q) = x :: q
def deq q =
  case rev q of
  | [] ⇒ None
  | x :: xs ⇒
    Some (x, rev xs)
end

def BatchedQueue : QUEUE =
struct
def t = list string × list string
def emp = ([], [])
def enq (x, (fs, rs)) = (fs, x :: rs)
def deq (fs, rs) =
  case fs of
  | [] ⇒
    (case rev rs of
      | [] ⇒ None
      | x :: rs' ⇒ Some (x, rs', [])
    | x :: fs' ⇒ Some (x, fs', rs)
end

Parametricity theorem
For any program $C : \text{QUEUE} \rightarrow \text{bool}$, we have $C(\text{ListQueue}) = C(\text{BatchedQueue})$.

The goal of this talk is to understand how to prove this.
A concept begging for a definition...

Strachey (1967) coined the term “parametricity” to informally describe the uniformity of polymorphic programs in their type arguments.
A concept begging for a definition...

Strachey (1967) coined the term “parametricity” to informally describe the uniformity of polymorphic programs in their type arguments.

Apparently independently, Lambek (1972) referred to this (as yet ill-defined) concept as “generality” in the context of formal deduction.
Strachey (1967) coined the term “parametricity” to informally describe the uniformity of polymorphic programs in their type arguments.

Apparently independently, Lambek (1972) referred to this (as yet ill-defined) concept as “generality” in the context of formal deduction.

In 1983, John Reynolds finally introduced the modern concept of relational parametricity as an explanation of this phenomenon.
Reynolds interprets types as binary relations $R_\tau \subseteq (\cdot \vdash \tau_L) \times (\cdot \vdash \tau_R)$ on the closed terms of a “left type” and a “right type”.
Reynolds interprets types as binary relations $R_\tau \subseteq (\vdash \tau_L) \times (\vdash \tau_R)$ on the closed terms of a “left type” and a “right type”.

A function from $R_f : R_\sigma \rightarrow R_\tau$ is given by the following data:
Reynolds interprets types as binary relations \( R_\tau \subseteq (\cdot \vdash \tau_L) \times (\cdot \vdash \tau_R) \) on the closed terms of a “left type” and a “right type”.

A function from \( R_f : R_\sigma \rightarrow R_\tau \) is given by the following data:

- a closed function \( f_L : \sigma_L \rightarrow \tau_L \),
Reynolds interprets types as binary relations $R_\tau \subseteq (\cdot \vdash \tau_L) \times (\cdot \vdash \tau_R)$ on the closed terms of a “left type” and a “right type”.

A function $R_f : R_\sigma \rightarrow R_\tau$ is given by the following data:

- a closed function $f_L : \sigma_L \rightarrow \tau_L$,
- a closed function $f_R : \sigma_R \rightarrow \tau_R$,
Reynolds interprets types as binary relations $R_\tau \subseteq (\cdot \vdash \tau_L) \times (\cdot \vdash \tau_R)$ on the closed terms of a “left type” and a “right type”.

A function from $R_f : R_\sigma \rightarrow R_\tau$ is given by the following data:

- a closed function $f_L : \sigma_L \rightarrow \tau_L$,
- a closed function $f_R : \sigma_R \rightarrow \tau_R$,
- such that $(x_L, x_R) \in R_\sigma \implies (f_L(x_L), f_R(x_R)) \in R_\tau$, i.e. the relations are preserved.
Type structure of relations: functions

Given relations $R_\sigma$ and $R_\tau$, the function type $R_{\sigma \to \tau}$ is interpreted like so:

$$R_{\sigma \to \tau} \subseteq (\cdot \vdash \sigma_L \to \tau_L) \times (\cdot \vdash \sigma_R \to \tau_R)$$

$$(f_L, f_R) \in R_{\sigma \to \tau} \equiv \forall (x_L, x_R) \in R_\sigma. (f_L(x_L), f_R(x_R)) \in R_\tau$$
**Type structure of relations: functions**

Given relations $R_\sigma$ and $R_\tau$, the function type $R_{\sigma \rightarrow \tau}$ is interpreted like so:

$$R_{\sigma \rightarrow \tau} \subseteq (\cdot \vdash \sigma_L \rightarrow \tau_L) \times (\cdot \vdash \sigma_R \rightarrow \tau_R)$$

$$(f_L, f_R) \in R_{\sigma \rightarrow \tau} \equiv \forall (x_L, x_R) \in R_\sigma. (f_L(x_L), f_R(x_R)) \in R_\tau$$

The above satisfies the universal property of the function type by definition:

$$\begin{align*}
R_\rho \times_\sigma & \longrightarrow R_\tau \\
R_\rho & \longrightarrow R_{\sigma \rightarrow \tau}
\end{align*}$$
We may interpret the booleans along the diagonal:

\[ R_{\text{bool}} \subseteq (\cdot \vdash \text{bool}) \times (\cdot \vdash \text{bool}) \]

\((b_L, b_R) \in R_{\text{bool}} \equiv (b_L = b_R = \text{true}) \lor (b_L = b_R = \text{false})\)
Type structure of relations: polymorphism

Given a family of relations $R_{\tau(\alpha)} \subseteq (\cdot \vdash \tau_L(\alpha_L)) \times (\cdot \vdash \tau_R(\alpha_R))$ varying in arbitrary relations $R_{\alpha}$, we define the polymorphic type $R_{\forall \alpha.\tau(\alpha)}$ like so:

\[
R_{\forall \alpha.\tau(\alpha)} \subseteq (\cdot \vdash \forall \alpha.\tau_L(\alpha)) \times (\cdot \vdash \forall \alpha.\tau_R(\alpha))
\]

\[
(f_L, f_R) \in R_{\forall \alpha.\tau(\alpha)} \equiv \forall R_{\alpha}.(f_L(\alpha_R), f_R(\alpha_R)) \in R_{\tau(\alpha)}
\]
Theorem

For $f : \forall \alpha. (\alpha \rightarrow \text{bool})$, we have $f(\text{unit}, \ast) = f(\text{bool}, \text{true}) : \text{bool}$. 
Theorem
For \( f : \forall \alpha. (\alpha \rightarrow \text{bool}) \), we have \( f(\text{unit}, \star) = f(\text{bool}, \text{true}) : \text{bool} \).

Proof.
By soundness we have \((f, f) \in R_{\forall \alpha. (\alpha \rightarrow \text{bool})}\) and hence:

\[
\forall R_\alpha. \forall (x_L, x_R) \in R_\alpha. f(x_L) = f(x_R)
\]

Choose \( R_\alpha \subseteq (\cdot \vdash \text{unit}) \times (\cdot \vdash \text{bool}) \) to be the singleton \( \{ (\star, \text{true}) \} \).
Theorem
For \( f : \forall \alpha. (\alpha \rightarrow \text{bool}) \), we have \( f(\text{unit}, \star) = f(\text{bool}, \text{true}) : \text{bool} \).

Proof.
By soundness we have \((f, f) \in R_{\forall \alpha. (\alpha \rightarrow \text{bool})}\) and hence:

\[
\forall R_\alpha. \forall (x_L, x_R) \in R_\alpha. f(x_L) = f(x_R)
\]

Choose \( R_\alpha \subseteq (\vdash \text{unit}) \times (\vdash \text{bool}) \) to be the singleton \{\((\star, \text{true})\)\}. \qed
Back to the queues...

Theorem
For any program \( C : \text{QUEUE} \to \text{bool} \), we have \( C(\text{ListQueue}) = C(\text{BatchedQueue}) \).
Back to the queues...

Theorem
For any program $C : \text{QUEUE} \rightarrow \text{bool}$, we have $C(\text{ListQueue}) = C(\text{BatchedQueue})$.

But how to prove? Reynolds says:

1. First restate $C$ as a polymorphic function
   $$C' : \forall \alpha. (\alpha \rightarrow (\text{string} \times \alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \text{option} (\text{string} \times \alpha)) \rightarrow \text{bool})$$
Back to the queues...

**Theorem**
For any program $C : \text{QUEUE} \to \text{bool}$, we have $C(\text{ListQueue}) = C(\text{BatchedQueue})$.

*But how to prove?* Reynolds says:

1. First restate $C$ as a polymorphic function
   
   $$C' : \forall \alpha. (\alpha \to (\text{string} \times \alpha \to \alpha) \to (\alpha \to \text{option}(\text{string} \times \alpha)) \to \text{bool})$$

2. Instantiate $C'$ in the relational model with the representation invariant
   
   $$R \subseteq (\cdot \vdash \text{ListQueue}.t) \times (\cdot \vdash \text{BatchedQueue}.t)$$
   
   defining
   
   $$(xs, (fs, rs)) \in R \equiv (xs = (fs + \text{rev } rs))$$
Back to the queues...

Theorem
For any program $C : \text{QUEUE} \to \text{bool}$, we have $C(\text{ListQueue}) = C(\text{BatchedQueue})$.

But how to prove? Reynolds says:

1. First restate $C$ as a polymorphic function
   $$C' : \forall \alpha. (\alpha \to (\text{string} \times \alpha \to \alpha) \to (\alpha \to \text{option}(\text{string} \times \alpha)) \to \text{bool})$$

2. Instantiate $C'$ in the relational model with the representation invariant
   $$R \subseteq (\cdot \vdash \text{ListQueue}.t) \times (\cdot \vdash \text{BatchedQueue}.t),$$
   defining
   $$(xs, (fs, rs)) \in R \equiv (xs = (fs + \text{rev } rs))$$

3. The further arguments must be instantiated with proofs that, e.g.
   $$(\text{ListQueue.emp}, \text{BatchedQueue.emp}) \in R.$$
   Operations respect the queue invariant.
Back to the queues...

**Theorem**
For any program $C : \text{QUEUE} \to \text{bool}$, we have $C(\text{ListQueue}) = C(\text{BatchedQueue})$.

*But how to prove?* Reynolds says:

1. First restate $C$ as a polymorphic function

   $$C' : \forall \alpha. (\alpha \to (\text{string} \times \alpha \to \alpha) \to (\alpha \to \text{option}(\text{string} \times \alpha)) \to \text{bool})$$

2. Instantiate $C'$ in the relational model with the representation invariant

   $$R \subseteq (\vdash \text{ListQueue}.t) \times (\vdash \text{BatchedQueue}.t),$$
   defining

   $$(xs, (fs, rs)) \in R \equiv (xs = (fs + \text{rev} \ rs))$$

3. The further arguments must be instantiated with proofs that, *e.g.*
   $$(\text{ListQueue}.\text{emp}, \text{BatchedQueue}.\text{emp}) \in R.$$  *Operations respect the queue invariant.*

Works because $R_{\text{bool}}$ is “discrete”, i.e. two booleans are related only when they are equal.
Abstract types (do not) have existential type

Reynolds’ trick is to treat type components of structures like QUEUE via polymorphism, but this only works if type components appear in negative positions.
Abstract types (do not) have existential type

Reynolds’ trick is to treat type components of structures like QUEUE via polymorphism, but this only works if type components appear in negative positions.

Encoding via existentials/weak sums \( \exists \alpha. \tau(\alpha) := \forall \rho. (\forall \alpha. \tau(\alpha) \rightarrow \rho) \rightarrow \rho \) is possible, but this does not directly model the “dot notation” Queue.t.
Abstract types (do not) have existential type

Reynolds’ trick is to treat type components of structures like QUEUE via polymorphism, but this only works if type components appear in negative positions.

Encoding via existentials/weak sums $\exists \alpha.\tau(\alpha) := \forall \rho. (\forall \alpha.\tau(\alpha) \rightarrow \rho) \rightarrow \rho$ is possible, but this does not directly model the “dot notation” Queue.t.

**Goal:** a version of the relational interpretation where Queue.t makes sense. Therefore we need something like “$R_{Type} \subseteq (\cdot \vdash Type) \times (\cdot \vdash Type)$.”
Abstract types (do not) have existential type

Reynolds’ trick is to treat type components of structures like QUEUE via polymorphism, but this only works if type components appear in negative positions.

Encoding via existentials/weak sums $\exists \alpha.\tau(\alpha) := \forall \rho. (\forall \alpha.\tau(\alpha) \rightarrow \rho) \rightarrow \rho$ is possible, but this does not directly model the “dot notation” $\text{Queue.t}$.

**Goal:** a version of the relational interpretation where $\text{Queue.t}$ makes sense. Therefore we need something like “$R_{\text{Type}} \subseteq (\cdot \vdash \text{Type}) \times (\cdot \vdash \text{Type})$”.

**Obstacle:** there is no “relation of relations”.
Abstract types (do not) have existential type

Reynolds’ trick is to treat type components of structures like \texttt{QUEUE} via polymorphism, but \textbf{this only works if type components appear in negative positions}.

Encoding via existentials/weak sums \( \exists \alpha.\tau(\alpha) := \forall \rho.(\forall \alpha.\tau(\alpha) \rightarrow \rho) \rightarrow \rho \) is possible, but this \textit{does not} directly model the “dot notation” \texttt{Queue.t}.

\textbf{Goal:} a version of the relational interpretation where \texttt{Queue.t} makes sense. Therefore we need something like “\( R_{\text{Type}} \subseteq (\cdot \vdash \text{Type}) \times (\cdot \vdash \text{Type}) \)”. \textbf{Obstacle:} there is no “relation of relations”. \textbf{Solution:} proof-relevant parametricity.
Proof-relevant parametricity

Instead of interpreting a type as a relation $R_\tau \subseteq (\cdot \vdash \tau_L) \times (\cdot \vdash \tau_R)$, interpret it as a family of sets $C_\tau \rightarrow (\cdot \vdash \tau_L) \times (\cdot \vdash \tau_R)$, writing $C_\tau[x_L, x_R]$ for the fiber of $C_\tau$ at a pair of closed terms $(x_L, x_R)$.

$$C_{\sigma \rightarrow \tau}[f_L, f_R] := \prod_{x_L, x_R} C_{\sigma[x_L, x_R]} \to C_{\tau[f_L(x_L), f_R(x_R)]}$$

$$C_{\text{bool}}[b_L, b_R] := (b_L = b_R = \text{true}) + (b_L = b_R = \text{false})$$

We call such a family a parametricity structure.
The parametricity structure of types

Given a universe $\mathcal{U}$ of small sets, we are now able to define:

$$C_{\text{Type}} \to (\cdot \vdash \text{Type}) \times (\cdot \vdash \text{Type})$$

$$C_{\text{Type}}[\sigma_L, \sigma_R] = \{ A \to (\cdot \vdash \sigma_L) \times (\cdot \vdash \sigma_R) \mid \forall x_L, x_R. A[x_L, x_R] \in \mathcal{U} \}$$

We can close parametricity structures under strong sums ($\Sigma$) and dependent products ($\Pi$). Hence we have a compositional interpretation of QUEUE:

$$\text{QUEUE} \cong \Sigma \alpha : \text{Type}. \alpha \times (\text{bool} \times \alpha \to \alpha) \times (\alpha \to 1 + \text{bool} \times \alpha)$$
Proving parametricity results is painful and non-modular.
Proving parametricity results is painful and non-modular.

By studying the structure of the category of parametricity structures, we can abstract a new language for synthetic parametricity arguments.
Reynolds slogan:

TYPES AS LOGICAL RELATIONS
Reynolds slogan:

TYPES AS LOGICAL RELATIONS
S.–Harper slogan:

LOGICAL RELATIONS AS TYPES
S.–Harper slogan:

LOGICAL RELATIONS AS TYPES
The syntax-semantics prism

A purely syntactic parametricity structure $C_\tau$ is one where each fiber is the terminal set, i.e., $C_\tau \{x_L, x_R\} \sim = \text{one.osf}$. A purely semantic parametricity structure $C_\tau$ is one where the base is the terminal type, i.e., $\tau_L \sim = \tau_R \sim = \text{unit}$. Artin, Grothendieck, and Verdier teach us: every $C_\tau$ refracts into purely syntactic and purely semantic parts $\text{Syn}(C_\tau), \text{Sem}(C_\tau)$ respectively. $C_\tau \text{Syn}(C_\tau), \text{Sem}(C_\tau)$ are (open, closed) modalities in the language of parametricity structures!
The syntax-semantics prism

A purely syntactic parametricity structure $C_{\tau}$ is one where each fiber is the terminal set, i.e. $C_{\tau}[x_L, x_R] \cong 1$. 

Artin, Grothendieck, and Verdier teach us: every $C_{\tau}$ refracts into purely syntactic and purely semantic parts $\text{Syn}(C_{\tau}), \text{Sem}(C_{\tau})$ respectively.

$C_{\tau}$, $\text{Sem}(\text{Syn}(C_{\tau}))$ are (open, closed) modalities in the language of parametricity structures!
The syntax-semantics prism

- A **purely syntactic** parametricity structure $C_\tau$ is one where each fiber is the terminal set, i.e. $C_\tau[x_L, x_R] \cong 1$.

- A **purely semantic** parametricity structure $C_\tau$ is one where the base is the terminal type, i.e. $\tau_L \cong \tau_R \cong \text{unit}$. 

Artin, Grothendieck, and Verdier teach us: every $C_\tau$ refracts into purely syntactic and purely semantic parts $\Syn(C_\tau), \Sem(C_\tau)$ respectively.

$C_\tau^{\Syn}(C_\tau) = \Sem(C_\tau)$ are (open, closed) modalities in the language of parametricity structures!
A purely syntactic parametricity structure $C_\tau$ is one where each fiber is the terminal set, i.e. $C_\tau[x_L, x_R] \cong 1$.

A purely semantic parametricity structure $C_\tau$ is one where the base is the terminal type, i.e. $\tau_L \cong \tau_R \cong \text{unit}$.

Artin, Grothendieck, and Verdier (1972) teach us: every $C_\tau$ refracts into purely syntactic and purely semantic parts $\text{Syn}(C_\tau)$, $\text{Sem}(C_\tau)$ respectively.

\[
\begin{array}{ccc}
C_\tau & \longrightarrow & \text{Sem}(C_\tau) \\
\downarrow & & \downarrow \\
\text{Syn}(C_\tau) & \longrightarrow & \text{Sem}(\text{Syn}(C_\tau))
\end{array}
\]
The syntax-semantics prism

- A purely syntactic parametricity structure $C_\tau$ is one where each fiber is the terminal set, i.e. $C_\tau[x_L, x_R] \cong 1$.
- A purely semantic parametricity structure $C_\tau$ is one where the base is the terminal type, i.e. $\tau_L \cong \tau_R \cong \text{unit}$.

Artin, Grothendieck, and Verdier (1972) teach us: every $C_\tau$ refracts into purely syntactic and purely semantic parts $\text{Syn}(C_\tau)$, $\text{Sem}(C_\tau)$ respectively.

\[
\begin{array}{ccc}
C_\tau & \longrightarrow & \text{Sem}(C_\tau) \\
\downarrow & & \downarrow \\
\text{Syn}(C_\tau) & \longrightarrow & \text{Sem}(\text{Syn}(C_\tau))
\end{array}
\]

$\text{Syn}$, $\text{Sem}$ are (open, closed) modalities in the language of parametricity structures!
There is a proof-irrelevant parametricity structure $\mathbb{S}_{\text{syn}}$ over the unit type such that for any other parametricity structure $C_{\tau}$, we have $\text{Syn}(C_{\tau}) \cong (\mathbb{S}_{\text{syn}} \to C_{\tau})$. 

Big idea: the semantic part $\mathbb{S}_{\text{syn}}$ is the empty set, zeroing out the semantic part of $C_{\tau}$. We can also redefine $\text{Sem}(C_{\tau})$ as the join $C_{\tau} \vee \mathbb{S}_{\text{syn}}$.

Bigger idea: all we need to talk about parametricity is a proof-irrelevant proposition $\mathbb{S}_{\text{syn}}$; all the remaining structure is unfurled from this.
The “syntactic lock”

There is a proof-irrelevant parametricity structure $\mathcal{S}_{\text{syn}}$ over the unit type such that for any other parametricity structure $C_\tau$, we have $\text{Syn}(C_\tau) \cong (\mathcal{S}_{\text{syn}} \to C_\tau)$.

**Big idea:** the semantic part $\mathcal{S}_{\text{syn}}$ is the empty set, zeroing out the semantic part of $C_\tau$. We can also redefine $\text{Sem}(C_\tau)$ as the join $C_\tau \lor \mathcal{S}_{\text{syn}}$. 
The “syntactic lock”

There is a proof-irrelevant parametricity structure $\text{syn}$ over the unit type such that for any other parametricity structure $C_\tau$, we have $\text{Syn}(C_\tau) \cong (\text{syn} \rightarrow C_\tau)$.

**Big idea:** the semantic part $\text{syn}$ is the empty set, zeroing out the semantic part of $C_\tau$. We can also redefine $\text{Sem}(C_\tau)$ as the join $C_\tau \lor \text{syn}$.

**Bigger idea:** all we need to talk about parametricity is a proof-irrelevant proposition $\text{syn}$; all the remaining structure is unfurled from this.
We define a type theory \textbf{ParamTT} of parametricity structures.
We define a type theory \textit{ParamTT} of parametricity structures.

1. Start with plain extensional type theory.
Logical Relations As Types

We define a type theory \textsc{ParamTT} of parametricity structures.

1. Start with plain extensional type theory.
2. Add some abstract propositions \( \mathcal{P}_{\text{syn}/l}, \mathcal{P}_{\text{syn}/r}, \mathcal{P}_{\text{syn}} : \text{Prop} \) satisfying the following laws:

\[
\mathcal{P}_{\text{syn}/l} \land \mathcal{P}_{\text{syn}/r} = \bot \\
\mathcal{P}_{\text{syn}/l} \lor \mathcal{P}_{\text{syn}/r} = \mathcal{P}_{\text{syn}}
\]
We define a type theory \textit{ParamTT} of parametricity structures.

1. Start with plain extensional type theory.
2. Add some abstract propositions $\text{syn}/l, \text{syn}/r, \text{syn} : \text{Prop}$ satisfying the following laws:

   $$\text{syn}/l \wedge \text{syn}/r = \bot \quad \text{syn}/l \lor \text{syn}/r = \text{syn}$$

3. Define $\text{Syn}(A) := \{ _- : \text{syn} \} \rightarrow A$ and $\text{Sem}(A) := A \lor \text{syn}$, satisfies $\text{Syn}(\text{Sem}(A)) \cong 1$. 

Logical Relations As Types

We define a type theory \texttt{ParamTT} of parametricity structures.

1. Start with plain extensional type theory.
2. Add some abstract propositions $\mathsf{syn}_l, \mathsf{syn}_r, \mathsf{syn} : \text{Prop}$ satisfying the following laws:

\[
\mathsf{syn}_l \land \mathsf{syn}_r = \bot \quad \text{and} \quad \mathsf{syn}_l \lor \mathsf{syn}_r = \mathsf{syn}
\]

3. Define $\text{Syn}(A) := \{ _, \mathsf{syn} \} \rightarrow A$ and $\text{Sem}(A) := A \lor \mathsf{syn}$, satisfies $\text{Syn}(\text{Sem}(A)) \cong 1$.

4. Can define elements of $\text{Syn}(A)$ by case analysis $[\mathsf{syn}_l \leftrightarrow a, \mathsf{syn}_r \leftrightarrow b]$. 

We can use this language to abstractly prove parametricity theorems.
Logical Relations As Types

We define a type theory \texttt{ParamTT} of parametricity structures.

1. Start with plain extensional type theory.
2. Add some abstract propositions \( \mathsf{syn}/l, \mathsf{syn}/r, \mathsf{Syn} : \text{Prop} \) satisfying the following laws:

\[
\mathsf{syn}/l \land \mathsf{syn}/r = \bot \quad \mathsf{syn}/l \lor \mathsf{syn}/r = \mathsf{Syn}
\]

3. Define \( \mathsf{Syn}(A) := \{ - : \mathsf{syn} \} \rightarrow A \) and \( \mathsf{Sem}(A) := A \lor \mathsf{syn} \), satisfies \( \mathsf{Syn}(\mathsf{Sem}(A)) \cong 1 \).
4. Can define elements of \( \mathsf{Syn}(A) \) by case analysis \([\mathsf{syn}/l \hookrightarrow a, \mathsf{syn}/r \hookrightarrow b]\).

We can use this language to abstractly prove parametricity theorems.
Syntactic extents

**Syntactic extent.** For a parametricity structure $A$ and an element of its syntactic part $a : \text{Syn}(A)$, define the *syntactic extent* $(A \text{ where } \text{Syn} \hookrightarrow a)$ to be the subset of $A$ that agrees syntactically with $a$:

$$(A \text{ where } \text{Syn} \hookrightarrow a) \equiv \{ x : A \mid \text{Syn}(a =_A x) \}$$
To study a language $\mathcal{L}$, first define $\mathcal{L}$ as a signature (dependent record) in the language of \texttt{ParamTT}.
To study a language $\mathcal{L}$, first define $\mathcal{L}$ as a signature (dependent record) in the language of ParamTT.

```plaintext
def $\mathcal{L}$ = sig
type : $\mathcal{U}$
tm : type $\rightarrow$ $\mathcal{U}$
arr : type $\rightarrow$ type $\rightarrow$ type
lam : $\{\sigma, \tau : \text{type}\} \rightarrow (\text{tm } \sigma \rightarrow \text{tm } \tau) \cong \text{tm } (\text{arr } \sigma \tau)$
bool : type
true : tm bool
false : tm bool
end
```
The fundamental theorem of logical relations for $\mathcal{L}$ is to define a suitable section to the projection $\mathcal{L} \rightarrow \text{Syn}(\mathcal{L})$, i.e. a dependent function:

$$M^*: (M : \text{Syn}(\mathcal{L})) \rightarrow (\mathcal{L} \text{ where } \square_{\text{syn}} \leftrightarrow M)$$
An $\mathcal{L}$-type is interpreted by a pair of a syntactic $\mathcal{L}$-type and a small parametricity structure that agrees syntactically with its collection of elements.
Synthetic parametricity structure of types

An $\mathcal{L}$-type is interpreted by a pair of a syntactic $\mathcal{L}$-type and a small parametricity structure that agrees syntactically with its collection of elements.

```python
def M*.type : U where □syn ↦ M.type = ?
```
An $\mathcal{L}$-type is interpreted by a pair of a syntactic $\mathcal{L}$-type and a small parametricity structure that agrees syntactically with its collection of elements.

```python
def M*.type : $\mathcal{U}$ where $\mathcal{U}$syn $\hookrightarrow$ M.type =
sig
    syn : Syn M.type
    sem : $\mathcal{U}$ where $\mathcal{U}$syn $\hookrightarrow$ M.el syn
end
```
Synthetic parametricity structure of types

An $\mathcal{L}$-type is interpreted by a pair of a syntactic $\mathcal{L}$-type and a small parametricity structure that agrees syntactically with its collection of elements.

```plaintext
def M*.type : $\mathcal{U}$ where $\emptyset_{\text{syn}} \hookrightarrow M\.type =
    \text{sig}
    \quad \text{syn} : \textbf{Syn} M\.type
    \quad \text{sem} : $\mathcal{U}$ where $\emptyset_{\text{syn}} \hookrightarrow M\.el$ \text{syn}
end

def tm A = A.sem
```

An $\mathcal{L}$-type is interpreted by a pair of a syntactic $\mathcal{L}$-type and a small parametricity structure that agrees syntactically with its collection of elements.

$$
\text{def } M^{\ast}.\text{type} : \mathcal{U} \text{ where } \mathcal{U}_{\text{syn}} \hookrightarrow M.\text{type} = \\
\text{sig} \\
\qquad \text{syn} : \textbf{Syn} \ M.\text{type} \\
\qquad \text{sem} : \mathcal{U} \text{ where } \mathcal{U}_{\text{syn}} \hookrightarrow M.\text{el} \ \text{syn} \\
\text{end} \\

\text{def } \text{tm} \ A = A.\text{sem}
$$

(Automatic coercion from $M^{\ast}.\text{type}$ to $M.\text{type}$ under $\mathcal{U}_{\text{syn}}/\text{Syn}$.)
def M*.arr A B : M*.type where \( \text{syn} \maps M\text{.arr} A B = \)
struct
  def syn = ?
  def sem = ?
end
Synthetic parametricity structure of functions

```python
def M*.arr A B : M*.type where syn ↦ M.arr A B =
struct
  def syn = M.arr A B
  def sem = ?
end
```
def M*.arr A B : M*.type where ⬢syn ➔ M.arr A B =
struct
    def syn = M.arr A B
    def sem = A.sem ➔ B.sem
end
def M*.bool : M*.type where \( \Omega_{\text{syn}} \rightarrow M.\text{bool} \) =

struct
def syn = M.\text{bool}
def sem = ?
end

def M*.true : M*.tm M*.bool where \( \Omega_{\text{syn}} \rightarrow M.\text{true} \) = ?

def M*.bool : M*.type where  □_syn  ↦  M.bool =
struct
  def syn = M.bool
  def sem = sig
    b :  Syn  M.bool
    p :  b = M.true + b = M.false
  end
end

def M*.true : M*.tm M*.bool where  □_syn  ↦  M.true = ?
def M*.bool : M*.type where \( \mathbb{S}_{\text{syn}} \mapsto M\text{-bool} = \)
struct
  def syn = M.bool
  def sem = sig
    b : Syn M.bool
    p : Sem (b = M.true + b = M.false)
  end
end
end

def M*.true : M*.tm M*.bool where \( \mathbb{S}_{\text{syn}} \mapsto M\text{-true} = ? \)
def M*.bool : M*.type where ⨿_{syn} \leftrightarrow M.bool = 
struct 
  def syn = M.bool 
  def sem = sig 
    b : Syn M.bool 
    p : Sem (b = M.true + b = M.false) 
  end 
end 

def M*.true : M*.tm M*.bool where ⨿_{syn} \leftrightarrow M.true = 
struct 
  def b = M.true 
  def p = return Sem inl(\star) 
end
Back to the queues again...

We started with two implementations of the QUEUE structure.
We started with two implementations of the QUEE structure. We will define these in ParamTT as a single syntactic queue by case analysis:
Back to the queues again...

We started with two implementations of the QUEUE structure. We will define these in ParamTT as a single syntactic queue by case analysis:

```
def Q_LR : Syn QUEUE =
  [\text{unlock syn l} \mapsto \text{ListQueue},
   \text{unlock syn r} \mapsto \text{BatchedQueue}]
```
To prove the representation independence theorem, we need only program a third queue whose type component carries the representation invariant:

```plaintext
def Q : QUEUE where \( {\text{unlock}} \rightarrow Q_{LR} = \) 
struct 
  def t = sig 
    q : Syn \( Q_{LR}.t, \)  
    p : Sem \{x,y,z | x = (y + \text{rev } z) \land q = [\( {\text{unlock}} \rightarrow x, \space {\text{unlock}} \rightarrow (y,z) ] \} \) 
  end

(* ... *)
end
```

Back to the queues again...
Back to the queues again...

To prove the representation independence theorem, we need only **program a third queue** whose type component carries the representation invariant:

```plaintext
def Q : QUEUE where $\mathcal{S}_\text{syn} \leftrightarrow Q_{LR} =
struct
def t = sig
def q : Syn Q_{LR}.t,
def p : Sem \{x,y,z \mid x = (y + \text{rev} z) \land q = [\mathcal{S}_\text{syn}/l \mapsto x, \mathcal{S}_\text{syn}/r \mapsto (y,z)]\}
end
def emp = struct
def q = Q_{LR}.emp
def p = return Sem ([],[],[],[])
end

(* ... *)
end
```
Our modal account of parametricity is a special case of Synthetic Tait Computability, described in my forthcoming thesis.
Our modal account of parametricity is a special case of Synthetic Tait Computability, described in my forthcoming thesis.

- Logical relations is a phase distinction between syntactic and semantic.
Our modal account of parametricity is a special case of Synthetic Tait Computability, described in my forthcoming thesis.

- Logical relations is a phase distinction between syntactic and semantic.

Thanks!
Our modal account of parametricity is a special case of Synthetic Tait Computability, described in my forthcoming thesis.

▶ Logical relations is a phase distinction between syntactic and semantic.
The Bigger Picture

Our modal account of parametricity is a special case of Synthetic Tait Computability, described in my forthcoming thesis.

▶ Logical relations is a phase distinction between syntactic and semantic.
  ▶ Normalization for multimodal type theory (Gratzer).

Thanks!
Our modal account of parametricity is a special case of *Synthetic Tait Computability*, described in my forthcoming thesis.

▶ Logical relations is a **phase distinction** between syntactic and semantic.

  ▶ [*JACM*] *Logical Relations As Types: Proof-Relevant Parametricity for Program Modules* (S., Harper).

  ▶ [*LICS ’21*] *Normalization for cubical type theory* (S., Angiuli).

  ▶ *Normalization for multimodal type theory* (Gratzer).

▶ Other phase distinctions abound:
Our modal account of parametricity is a special case of Synthetic Tait Computability, described in my forthcoming thesis.

- Logical relations is a phase distinction between syntactic and semantic.
  - [JACM] Logical Relations As Types: Proof-Relevant Parametricity for Program Modules (S., Harper).
  - [LICS ’21] Normalization for cubical type theory (S., Angiuli).
  - Normalization for multimodal type theory (Gratzer).

- Other phase distinctions abound:
  - **ML languages**: static vs. dynamic (Harper, Mitchell, and Moggi, 1990)
Our modal account of parametricity is a special case of Synthetic Tait Computability, described in my forthcoming thesis.

- Logical relations is a phase distinction between syntactic and semantic.
  - [JACM] Logical Relations As Types: Proof-Relevant Parametricity for Program Modules (S., Harper).
  - [LICS '21] Normalization for cubical type theory (S., Angiuli).
  - Normalization for multimodal type theory (Gratzer).

- Other phase distinctions abound:
  - **ML languages**: static vs. dynamic (Harper, Mitchell, and Moggi, 1990)
  - **Compilation**: inlining vs. abstraction (jww. Harper)
Our modal account of parametricity is a special case of Synthetic Tait Computability, described in my forthcoming thesis.

- Logical relations is a phase distinction between syntactic and semantic.
  - [JACM] Logical Relations As Types: Proof-Relevant Parametricity for Program Modules (S., Harper).
  - [LICS ’21] Normalization for cubical type theory (S., Angiuli).
  - Normalization for multimodal type theory (Gratzer).

- Other phase distinctions abound:
  - **ML languages**: static vs. dynamic (Harper, Mitchell, and Moggi, 1990)
  - **Compilation**: inlining vs. abstraction (jww. Harper)
  - **Resource usage**: behavior vs. cost (jww. Niu and Harper)
Our modal account of parametricity is a special case of Synthetic Tait Computability, described in my forthcoming thesis.

▶ Logical relations is a phase distinction between syntactic and semantic.
  ▶ Normalization for multimodal type theory (Gratzer).

▶ Other phase distinctions abound:
  ▶ ML languages: static vs. dynamic (Harper, Mitchell, and Moggi, 1990)
  ▶ Compilation: inlining vs. abstraction (jww. Harper)
  ▶ Resource usage: behavior vs. cost (jww. Niu and Harper)
  ▶ Refinement: extraction vs. verification (Melliès and Zeilberger, 2015)
Our modal account of parametricity is a special case of Synthetic Tait Computability, described in my forthcoming thesis.

- **Logical relations** is a phase distinction between syntactic and semantic.
  - [JACM] Logical Relations As Types: Proof-Relevant Parametricity for Program Modules (S., Harper).
  - [LICS ’21] Normalization for cubical type theory (S., Angiuli).
  - Normalization for multimodal type theory (Gratzer).

- Other phase distinctions abound:
  - **ML languages**: static vs. dynamic (Harper, Mitchell, and Moggi, 1990)
  - **Compilation**: inlining vs. abstraction (jww. Harper)
  - **Resource usage**: behavior vs. cost (jww. Niu and Harper)
  - **Refinement**: extraction vs. verification (Melliès and Zeilberger, 2015)
  - **Information flow**: low vs. high security (Abadi et al., 1999)
Our modal account of parametricity is a special case of Synthetic Tait Computability, described in my forthcoming thesis.

- Logical relations is a phase distinction between syntactic and semantic.
  - [JACM] Logical Relations As Types: Proof-Relevant Parametricity for Program Modules (S., Harper).
  - [LICS ’21] Normalization for cubical type theory (S., Angiuli).
  - Normalization for multimodal type theory (Gratzer).

- Other phase distinctions abound:
  - **ML languages**: static vs. dynamic (Harper, Mitchell, and Moggi, 1990)
  - **Compilation**: inlining vs. abstraction (jww. Harper)
  - **Resource usage**: behavior vs. cost (jww. Niu and Harper)
  - **Refinement**: extraction vs. verification (Melliès and Zeilberger, 2015)
  - **Information flow**: low vs. high security (Abadi et al., 1999)
  - **Topology**: open vs. closed subspace (Artin, Grothendieck, and Verdier, 1972)
Our modal account of parametricity is a special case of *Synthetic Tait Computability*, described in my forthcoming thesis.

- Logical relations is a **phase distinction** between syntactic and semantic.
  - *Normalization for multimodal type theory* (Gratzer).

- Other phase distinctions abound:
  - **ML languages**: static vs. dynamic (Harper, Mitchell, and Moggi, 1990)
  - **Compilation**: inlining vs. abstraction (jww. Harper)
  - **Resource usage**: behavior vs. cost (jww. Niu and Harper)
  - **Refinement**: extraction vs. verification (Melliès and Zeilberger, 2015)
  - **Information flow**: low vs. high security (Abadi et al., 1999)
  - **Topology**: open vs. closed subspace (Artin, Grothendieck, and Verdier, 1972)

Thanks!
References


