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

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).

MODULES

Consider a *queue* data structure.

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

Consider a *queue* data structure.

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

queue representation type (annotation pointing to `t : type`)

Consider a *queue* data structure.

```
def QUEUE =
sig
  t : type          ← queue representation type
  emp : t           ← empty queue
  enq : string × t → t
  deq : t → option (string × t)
end
```

Consider a *queue* data structure.

```
def QUEUE =
sig
  t : type                                  queue representation type
  emp : t                                   empty queue
  enq : string × t → t                      enqueuing map
  deq : t → option (string × t)
end
```

Consider a *queue* data structure.

```
def QUEUE =
sig
  t : type                              ← queue representation type
  emp : t                               ← empty queue
  enq : string × t → t                  ← enqueuing map
  deq : t → option (string × t)         ← dequeuing map
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
```

Queue implementation (BatchedQueue)

```
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` $\neq$ `BatchedQueue.t`, hence `ListQueue` $\neq$ `BatchedQueue`. But it is not possible to *observe* the difference between the two!

What does it mean to be different?

What does it mean to be different?

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

What does it mean to be different?

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

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 $C : \texttt{QUEUE} \to \texttt{bool}$, we have $C(\texttt{ListQueue}) = C(\texttt{BatchedQueue})$.

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.

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.

In 1983, John Reynolds finally introduced the modern concept of relational parametricity as an explanation of this phenomenon.

“Types as logical relations”

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”.

“Types as logical relations”

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 $R_f : R_\sigma \to R_\tau$ is given by the following data:

“Types as logical relations”

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 $R_f : R_\sigma \to R_\tau$ is given by the following data:

- a closed function $f_L : \sigma_L \to \tau_L$,

“Types as logical relations”

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 $f_R : \sigma_R \rightarrow \tau_R$,

“Types as logical relations”

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 $R_f : R_\sigma \to R_\tau$ is given by the following data:

- a closed function $f_L : \sigma_L \to \tau_L$,
- a closed function $f_R : \sigma_R \to \tau_R$,
- 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_σ and R_τ, 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_σ and R_τ, the function type $R_{\sigma\to\tau}$ is interpreted like so:

$$R_{\sigma\to\tau} \subseteq (\cdot \vdash \sigma_L \to \tau_L) \times (\cdot \vdash \sigma_R \to \tau_R)$$
$$(f_L, f_R) \in R_{\sigma\to\tau} :\equiv \forall (x_L, x_R) \in R_\sigma . (f_L(x_L), f_R(x_R)) \in R_\tau$$

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

$$\frac{R_{\rho\times\sigma} \longrightarrow R_\tau}{R_\rho \longrightarrow R_{\sigma\to\tau}}$$

Type structure of relations: booleans

We may interpret the booleans along the *diagonal*:

$$R_{\texttt{bool}} \subseteq (\cdot \vdash \texttt{bool}) \times (\cdot \vdash \texttt{bool})$$
$$(b_L, b_R) \in R_{\texttt{bool}} :\equiv (b_L = b_R = \texttt{true}) \vee (b_L = b_R = \texttt{false})$$

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_α, 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 $f : \forall\alpha.(\alpha \to \texttt{bool})$, *we have* $f(\texttt{unit}, \star) = f(\texttt{bool}, \texttt{true}) : \texttt{bool}$.

Theorem

For $f : \forall\alpha.(\alpha \to \texttt{bool})$*, we have* $f(\texttt{unit}, \star) = f(\texttt{bool}, \texttt{true}) : \texttt{bool}$.

Proof.

By soundness we have $(f, f) \in R_{\forall\alpha.(\alpha\to\texttt{bool})}$ and hence:

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

Theorem

For $f : \forall\alpha.(\alpha \to \texttt{bool})$*, we have* $f(\texttt{unit}, \star) = f(\texttt{bool}, \texttt{true}) : \texttt{bool}$.

Proof.

By soundness we have $(f, f) \in R_{\forall\alpha.(\alpha\to\texttt{bool})}$ and hence:

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

Choose $R_\alpha \subseteq (\cdot \vdash \texttt{unit}) \times (\cdot \vdash \texttt{bool})$ to be the singleton $\{(\star, \texttt{true})\}$. $\square$

Back to the queues...

Theorem

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

Back to the queues...

Theorem

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

But how to prove? Reynolds says:

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

Back to the queues...

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

But how to prove? Reynolds says:

1. First restate C as a polymorphic function
$C' : \forall\alpha.(\alpha \to (\texttt{string} \times \alpha \to \alpha) \to (\alpha \to \texttt{option}(\texttt{string} \times \alpha)) \to \texttt{bool}))$
2. Instantiate C' in the relational model with the representation invariant
$R \subseteq (\cdot \vdash \texttt{ListQueue.t}) \times (\cdot \vdash \texttt{BatchedQueue.t})$, defining
$(\texttt{xs}, (\texttt{fs}, \texttt{rs})) \in R :\equiv (\texttt{xs} = (\texttt{fs} + \texttt{rev rs}))$

Back to the queues...

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

But how to prove? Reynolds says:

1. First restate C as a polymorphic function $C' : \forall\alpha.(\alpha \to (\texttt{string} \times \alpha \to \alpha) \to (\alpha \to \texttt{option}(\texttt{string} \times \alpha)) \to \texttt{bool}))$
2. Instantiate C' in the relational model with the representation invariant $R \subseteq (\cdot \vdash \texttt{ListQueue.t}) \times (\cdot \vdash \texttt{BatchedQueue.t})$, defining $(\texttt{xs}, (\texttt{fs}, \texttt{rs})) \in R :\equiv (\texttt{xs} = (\texttt{fs} + \texttt{rev rs}))$
3. The further arguments must be instantiated with proofs that, *e.g.* $(\texttt{ListQueue.emp}, \texttt{BatchedQueue.emp}) \in R$. Operations respect the queue invariant. □

Back to the queues...

Theorem

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

But how to prove? Reynolds says:

1. First restate C as a polymorphic function
   $C' : \forall\alpha.(\alpha \to (\mathtt{string} \times \alpha \to \alpha) \to (\alpha \to \mathtt{option}(\mathtt{string} \times \alpha)) \to \mathtt{bool}))$
2. Instantiate C' in the relational model with the representation invariant
   $R \subseteq (\cdot \vdash \mathtt{ListQueue.t}) \times (\cdot \vdash \mathtt{BatchedQueue.t})$, defining
   $(\mathtt{xs}, (\mathtt{fs}, \mathtt{rs})) \in R :\equiv (\mathtt{xs} = (\mathtt{fs} + \mathtt{rev}\ \ \mathtt{rs}))$
3. The further arguments must be instantiated with proofs that, *e.g.* $(\mathtt{ListQueue.emp}, \mathtt{BatchedQueue.emp}) \in R$. Operations respect the queue invariant. □

Works because $R_{\mathtt{bool}}$ 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 $\exists\alpha.\tau(\alpha) := \forall\rho.(\forall\alpha.\tau(\alpha) \to \rho) \to \rho$ 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 $\exists\alpha.\tau(\alpha) := \forall\rho.(\forall\alpha.\tau(\alpha) \to \rho) \to \rho$ 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 "$R_{\text{Type}} \subseteq (\cdot \vdash \text{Type}) \times (\cdot \vdash \text{Type})$".

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 $\exists\alpha.\tau(\alpha) := \forall\rho.(\forall\alpha.\tau(\alpha) \to \rho) \to \rho$ 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 "$R_{\mathsf{Type}} \subseteq (\cdot \vdash \mathsf{Type}) \times (\cdot \vdash \mathsf{Type})$".
Obstacle: there is no "relation of relations".

Abstract types (do not) have existential type

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

Encoding via existentials/weak sums $\exists\alpha.\tau(\alpha) := \forall\rho.(\forall\alpha.\tau(\alpha) \to \rho) \to \rho$ 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 "$R_{\mathsf{Type}} \subseteq (\cdot \vdash \mathsf{Type}) \times (\cdot \vdash \mathsf{Type})$".
Obstacle: there is no "relation of relations".
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 $C_\tau \longrightarrow (\cdot \vdash \tau_L) \times (\cdot \vdash \tau_R)$, writing $C_\tau[x_L, x_R]$ for the fiber of C_τ at a pair of closed terms (x_L, x_R).

$$C_{\sigma \to \tau}[f_L, f_R] := \prod_{x_L, x_R} C_\sigma[x_L, x_R] \to C_\tau[f_L(x_L), f_R(x_R)]$$
$$C_{\texttt{bool}}[b_L, b_R] := (b_L = b_R = \texttt{true}) + (b_L = b_R = \texttt{false})$$

We call such a family a *parametricity structure*.

The parametricity structure of types

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

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

We can close parametricity structures under strong sums (Σ) and dependent products (Π). Hence we have a compositional interpretation of `QUEUE`:

$$\mathtt{QUEUE} \cong \Sigma\alpha : \mathtt{Type}.\alpha \times (\mathtt{bool} \times \alpha \to \alpha) \times (\alpha \to \mathbf{1} + \mathtt{bool} \times \alpha)$$

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

The syntax-semantics prism

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

The syntax-semantics prism

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

The syntax-semantics prism

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

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

$$\begin{array}{ccc} C_\tau & \longrightarrow & \mathbf{Sem}(C_\tau) \\ \downarrow & \lrcorner & \downarrow \\ \mathbf{Syn}(C_\tau) & \longrightarrow & \mathbf{Sem}(\mathbf{Syn}(C_\tau)) \end{array}$$

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

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

$$\begin{array}{ccc} C_\tau & \longrightarrow & \mathbf{Sem}(C_\tau) \\ \big\downarrow & \lrcorner & \big\downarrow \\ \mathbf{Syn}(C_\tau) & \longrightarrow & \mathbf{Sem}(\mathbf{Syn}(C_\tau)) \end{array}$$

Syn, **Sem** are (open, closed) modalities in the language of parametricity structures!

The “syntactic lock”

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

The "syntactic lock"

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

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

The "syntactic lock"

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

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

Bigger idea: all we need to talk about parametricity is a proof-irrelevant proposition $🔓_{\mathrm{syn}}$; all the remaining structure is unfurled from this.

Logical Relations As Types

We define a type theory ParamTT of parametricity structures.

Logical Relations As Types

We define a type theory ParamTT of parametricity structures.

1. Start with plain extensional type theory.

Logical Relations As Types

We define a type theory ParamTT of parametricity structures.

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

$$🔓_{\mathsf{syn}/l} \wedge 🔓_{\mathsf{syn}/r} = \bot \qquad\qquad 🔓_{\mathsf{syn}/l} \vee 🔓_{\mathsf{syn}/r} = 🔓_{\mathsf{syn}}$$

Logical Relations As Types

We define a type theory ParamTT of parametricity structures.

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

$$\text{🔓}_{\mathsf{syn}/\mathsf{l}} \wedge \text{🔓}_{\mathsf{syn}/\mathsf{r}} = \bot \qquad\qquad \text{🔓}_{\mathsf{syn}/\mathsf{l}} \vee \text{🔓}_{\mathsf{syn}/\mathsf{r}} = \text{🔓}_{\mathsf{syn}}$$

3. Define $\mathbf{Syn}(A) := \{\_ : \text{🔓}_{\mathsf{syn}}\} \to A$ and $\mathbf{Sem}(A) := A \vee \text{🔓}_{\mathsf{syn}}$, satisfies $\mathbf{Syn}(\mathbf{Sem}(A)) \cong \mathbf{1}$.

Logical Relations As Types

We define a type theory ParamTT of parametricity structures.

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

$$¶_{\mathsf{syn}/\mathsf{l}} \wedge ¶_{\mathsf{syn}/\mathsf{r}} = \bot \qquad\qquad ¶_{\mathsf{syn}/\mathsf{l}} \vee ¶_{\mathsf{syn}/\mathsf{r}} = ¶_{\mathsf{syn}}$$

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

Logical Relations As Types

We define a type theory ParamTT of parametricity structures.

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

$$\text{🔓}_{\mathsf{syn/l}} \wedge \text{🔓}_{\mathsf{syn/r}} = \bot \qquad\qquad \text{🔓}_{\mathsf{syn/l}} \vee \text{🔓}_{\mathsf{syn/r}} = \text{🔓}_{\mathsf{syn}}$$

3. Define $\mathbf{Syn}(A) := \{\_ : \text{🔓}_{\mathsf{syn}}\} \to A$ and $\mathbf{Sem}(A) := A \vee \text{🔓}_{\mathsf{syn}}$, satisfies $\mathbf{Syn}(\mathbf{Sem}(A)) \cong \mathbf{1}$.
4. Can define elements of $\mathbf{Syn}(A)$ by case analysis $[\text{🔓}_{\mathsf{syn/l}} \hookrightarrow a, \text{🔓}_{\mathsf{syn/r}} \hookrightarrow b]$.

We can use this language to abstractly prove parametricity theorems.

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

$$(\mathtt{A}\ \mathtt{where}\ \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathtt{a}) :\equiv \{\mathtt{x} : \mathtt{A} \mid \mathbf{Syn}(\mathtt{a} =_{\mathtt{A}} \mathtt{x})\}$$

Getting started with LRAT

To study a language $\mathcal{L}$, first define $\mathcal{L}$ as a signature (dependent record) in the language of ParamTT.

Getting started with LRAT

To study a language $\mathcal{L}$, first define $\mathcal{L}$ as a signature (dependent record) in the language of ParamTT.

```
def ℒ = sig
  type : 𝒰
  tm : type → 𝒰
  arr : type → type → type
  lam : {σ,τ : type} → (tm σ → tm τ) ≅ tm (arr σ τ)
  bool : type
  true : tm bool
  false : tm bool
end
```

Where's the FTLR??

The fundamental theorem of logical relations for $\mathcal{L}$ is to define a suitable section to the projection $\mathcal{L} \to \mathbf{Syn}(\mathcal{L})$, *i.e.* a dependent function:

$$\mathtt{M*} : (\mathtt{M} : \mathbf{Syn}(\mathcal{L})) \to (\mathcal{L}\ \mathtt{where}\ \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathtt{M})$$

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.

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.

```
def M*.type : 𝒰 where 🔓syn ↪ M.type = ?
```

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.

```
def M*.type : 𝒰 where 🔓syn ↪ M.type =
sig
  syn : Syn M.type
  sem : 𝒰 where 🔓syn ↪ 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.

```
def M*.type : 𝒰 where 🔓syn ↪ M.type =
sig
  syn : Syn M.type
  sem : 𝒰 where 🔓syn ↪ M.el syn
end

def tm A = A.sem
```

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.

```
def M*.type : 𝒰 where 🔓syn ↪ M.type =
sig
  syn : Syn M.type
  sem : 𝒰 where 🔓syn ↪ M.el syn
end

def tm A = A.sem
```

(Automatic coercion from `M*.type` to `M.type` under $🔓_{\text{syn}}$/**Syn**.)

Synthetic parametricity structure of functions

```
def M*.arr A B : M*.type where 🔓syn ↪ M.arr A B =
struct
  def syn = ?
  def sem = ?
end
```

Synthetic parametricity structure of functions

```
def M*.arr A B : M*.type where 🔓syn ↪ M.arr A B =
struct
  def syn = M.arr A B
  def sem = ?
end
```

Synthetic parametricity structure of functions

```
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
```

Synthetic parametricity structure of booleans

```
def M*.bool : M*.type where 🔓syn ↪ M.bool =
struct
  def syn = M.bool
  def sem = ?
end

def M*.true : M*.tm M*.bool where 🔓syn ↪ M.true = ?
```

Synthetic parametricity structure of booleans

```
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 = ?
```

Synthetic parametricity structure of booleans

```
def M*.bool : M*.type where 🔓syn ↪ 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 ↪ M.true = ?
```

Synthetic parametricity structure of booleans

```
def M*.bool : M*.type where 🔓syn ↪ 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 ↪ M.true =
struct
  def b = M.true
  def p = returnSem inl(⋆)
end
```

Back to the queues again...

We started with two implementations of the `QUEUE` structure.

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:

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:

$$\texttt{def } Q_{LR} : \textbf{Syn } \texttt{QUEUE} =$$
$$[\blacksquare_{\mathsf{syn}/l} \hookrightarrow \texttt{ListQueue},$$
$$\blacksquare_{\mathsf{syn}/r} \hookrightarrow \texttt{BatchedQueue}]$$

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:

```
def Q : QUEUE where 🔓_syn ↪ Q_LR =
struct
  def t = sig
    q : Syn Q_LR.t,
    p : Sem {x,y,z | x = (y + rev z) ∧ q = [🔓_syn/l ↪ x, 🔓_syn/r ↪ (y,z)]}
  end

  (* ... *)
end
```

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:

```
def Q : QUEUE where 🔓syn ↪ Q_LR =
struct
  def t = sig
    q : Syn Q_LR.t,
    p : Sem {x,y,z | x = (y + rev z) ∧ q = [🔓syn/l ↪ x, 🔓syn/r ↪ (y,z)]}
  end

  def emp = struct
    def q = Q_LR.emp
    def p = return_Sem ([],[],[])
  end

  (* ... *)
end
```

The Bigger Picture

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

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.

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

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.
  - [JACM] *Logical Relations As Types: Proof-Relevant Parametricity for Program Modules* (S., Harper).
  - [LICS '21] *Normalization for cubical type theory* (S., Angiuli).

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.
  - [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).

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.
  - [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:

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.
  - [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)

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.
    - [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)

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.
  - [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)

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.
  - [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)

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.
  - [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)

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.
  - [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)

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.
  - [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)

Thanks!

References I

Abadi, Martín et al. (1999). “A Core Calculus of Dependency”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’99. San Antonio, Texas, USA: Association for Computing Machinery, pp. 147–160. ISBN: 1-58113-095-3. DOI: 10.1145/292540.292555.

Artin, Michael, Alexander Grothendieck, and Jean-Louis Verdier (1972). *Théorie des topos et cohomologie étale des schémas.* Séminaire de Géométrie Algébrique du Bois-Marie 1963–1964 (SGA 4), Dirigé par M. Artin, A. Grothendieck, et J.-L. Verdier. Avec la collaboration de N. Bourbaki, P. Deligne et B. Saint-Donat, Lecture Notes in Mathematics, Vol. 269, 270, 305. Berlin: Springer-Verlag.

Birkedal, Lars and Rasmus E. Møgelberg (2005). “Categorical models for Abadi and Plotkin’s logic for parametricity”. In: *Mathematical Structures in Computer Science* 15.4, pp. 709–772. DOI: 10.1017/S0960129505004834.

Cardelli, Luca and Xavier Leroy (1990). “Abstract types and the dot notation”. In: *Proceedings IFIP TC2 working conference on programming concepts and methods*. North-Holland, pp. 479–504.

Coquand, Thierry (2019). “Canonicity and normalization for dependent type theory”. In: *Theoretical Computer Science* 777. In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I, pp. 184–191. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2019.01.015. arXiv: 1810.09367 [cs.PL].

Crary, Karl (2017). “Modules, Abstraction, and Parametric Polymorphism”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: Association for Computing Machinery, pp. 100–113. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009892.

Gratzer, Daniel (2021). *Normalization for Multimodal Type Theory.* Unpublished manuscript. URL: https://jozefg.github.io/papers/normalization-for-multimodal-type-theory.pdf.

Harper, Robert (2020). “**PFPL** Supplement: Types for Program Modules”. URL: http://www.cs.cmu.edu/~rwh/pfpl/supplements/modules.pdf.

References II

Harper, Robert, John C. Mitchell, and Eugenio Moggi (1990). "Higher-Order Modules and the Phase Distinction". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, USA: Association for Computing Machinery, pp. 341–354. ISBN: 0-89791-343-4. DOI: 10.1145/96709.96744.

Lambek, Joachim (1972). "Deductive systems and categories III. Cartesian closed categories, intuitionist propositional calculus, and combinatory logic". In: *Toposes, Algebraic Geometry and Logic*. Ed. by F. W. Lawvere. Lecture Notes in Mathematics. From a conference on "Connections between Category Theory and Algebraic Geometry & Intuitionistic Logic" held at Halifax, Nova Scotia, Canada, January 16–19, 1971 at Dalhousie University. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 57–82. ISBN: 978-3-540-37609-5.

MacQueen, David B. (1986). "Using Dependent Types to Express Modular Structure". In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. St. Petersburg Beach, Florida: Association for Computing Machinery, pp. 277–286. ISBN: 978-1-4503-7347-0. DOI: 10.1145/512644.512670.

Melliès, Paul-André and Noam Zeilberger (2015). "Functors are Type Refinement Systems". In: *POPL '15: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Mumbai, India: ACM. ISBN: 978-1-4503-3300-9. URL: https://hal.inria.fr/hal-01096910.

Mitchell, John C. and Gordon D. Plotkin (1985). "Abstract Types Have Existential Types". In: *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New Orleans, Louisiana, USA: Association for Computing Machinery, pp. 37–51. ISBN: 0-89791-147-4. DOI: 10.1145/318593.318606.

Plotkin, Gordon and Martín Abadi (1993). "A logic for parametric polymorphism". In: *Typed Lambda Calculi and Applications*. Ed. by Marc Bezem and Jan Friso Groote. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 361–375. ISBN: 978-3-540-47586-6.

Reynolds, John C. (1983). "Types, Abstraction, and Parametric Polymorphism". In: *Information Processing*.

References III

Rijke, Egbert, Michael Shulman, and Bas Spitters (Jan. 2020). "Modalities in homotopy type theory". In: *Logical Methods in Computer Science* Volume 16, Issue 1. DOI: 10.23638/LMCS-16(1:2)2020. arXiv: 1706.07526 [math.CT]. URL: https://lmcs.episciences.org/6015.

Shulman, Michael (2011). *Internalizing the External, or The Joys of Codiscreteness.* blog. URL: https://golem.ph.utexas.edu/category/2011/11/internalizing_the_external_or.html.

— (2015). "Univalence for inverse diagrams and homotopy canonicity". In: *Mathematical Structures in Computer Science* 25.5, pp. 1203–1277. DOI: 10.1017/S0960129514000565.

Sterling, Jonathan and Carlo Angiuli (2021). "Normalization for Cubical Type Theory". In: *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science.* To appear. New York, NY, USA: ACM. arXiv: 2101.11479 [cs.LO].

Sterling, Jonathan and Robert Harper (2020). "Logical Relations As Types: Proof-Relevant Parametricity for Program Modules". In: *Journal of the ACM.* To appear. arXiv: 2010.08599 [cs.PL].

Strachey, Christopher (Aug. 1967). *Fundamental Concepts in Programming Languages.* Lecture Notes, International Summer School in Computer Programming, Copenhagen. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000.