Logical Relations as Types
Proof-Relevant Parametricity for Program Modules

Robert Harper
Carnegie Mellon University

(Joint work with Jon Sterling)

April 2021

Acknowledgments

Please see `http://arxiv.org/abs/2010.08599` for full development, including citations and comparisons to related work. This work is to accepted to appear in *J. ACM*.

This work was supported in part by AFOSR under grants MURI FA9550-15-1-0053 and FA9550-19-1-0216, Tristan Nguyen, program manager. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

Types for Modularity

The ML module system structures programs into separable components:

- Signatures, or interfaces declaring types and values.

Types for Modularity

The ML module system structures programs into separable components:

- Signatures, or interfaces declaring types and values.
- Structures that define them.

Types for Modularity

The ML module system structures programs into separable components:

- Signatures, or interfaces declaring types and values.
- Structures that define them.
- Hierarchical dependencies, sub-structures.

Types for Modularity

The ML module system structures programs into separable components:

- Signatures, or interfaces declaring types and values.
- Structures that define them.
- Hierarchical dependencies, sub-structures.
- Functional dependencies, functors.

Queue Signature

```
signature QUEUE = sig
  type t
  val emp : t
  val ins : bool * t -> t
  val rem : t -> bool * t
end
```

Queue Implementation I

```
structure Q₀ : QUEUE = struct
  type t = bool list
  val emp = nil
  fun ins (x, q) = ret (x :: q)
  fun rem q =
    bind val rev_q ← rev q in
    case rev_q of
    | nil ⇒ throw
    | x :: xs ⇒
      bind val rev_xs ← rev xs in
      ret (f, rev_xs)
end
```

Queue Implementation II

```
structure Q₁ : QUEUE = struct
  type t = bool list * bool list
  val emp = (nil, nil)
  fun ins (x, (fs, rs)) = ret (fs, x :: rs)
  fun rem (fs, rs) =
    case fs of
    | nil ⇒
      bind val rev_rs ← rev rs in
      (case rev_rs of
       | nil ⇒ throw
       | x::rs' ⇒ ret (x, rs', nil))
    | x::fs' ⇒ ret (x, fs', rs)
end
```

Coherence Specifications

Coherence is specified by equational sharing specifications.

```
functor Layer
    (structure Lower : LAYER and Packet : PACKET
     sharing Lower.Packet.T = Packet.T)
```

Supports composition from pre-existing components!

- Avoids anticipation of all possible combination patterns.
- Encourages off-the-shelf re-use.

Coherence Specifications

Coherence is specified by equational sharing specifications.

```
functor Layer
    (structure Lower : LAYER and Packet : PACKET
     sharing Lower.Packet.T = Packet.T)
```

Supports composition from pre-existing components!

- Avoids anticipation of all possible combination patterns.
- Encourages off-the-shelf re-use.

But what do sharing specifications mean?

The Phase Distinction

Moggi introduced the phase distinction:

- Static, or compile-time.
- Dynamic, or run-time.

Sharing specifications are static constraints!

- Enforced during type checking (compile time).
- Governs static components, not dynamic (no code comparison).

Types for Program Modules

MacQueen proposed using dependent types for modularity:

- Basic type and value declarations.

Types for Program Modules

MacQueen proposed using dependent types for modularity:

- Basic type and value declarations.
- A universe of “small” types.

Types for Program Modules

MacQueen proposed using dependent types for modularity:

- Basic type and value declarations.
- A universe of "small" types.
- Dependent sums: $x{:}\sigma_1 \times \sigma_2$.

  `Lower : PROTOCOL` $\times$ `type T`

Types for Program Modules

MacQueen proposed using dependent types for modularity:

- Basic type and value declarations.
- A universe of "small" types.
- Dependent sums: $x{:}\sigma_1 \times \sigma_2$.

  ```
  Lower : PROTOCOL × type T
  ```

- Dependent functions: $x{:}\sigma_1 \rightarrow \sigma_2$.

  ```
  Lower : PROTOCOL → Upper : PROTOCOL sharing Lower.T=Upper.T
  ```

Types for Program Modules

MacQueen proposed using dependent types for modularity:

- Basic type and value declarations.
- A universe of "small" types.
- Dependent sums: $x{:}\sigma_1 \times \sigma_2$.

  `Lower : PROTOCOL` $\times$ `type T`

- Dependent functions: $x{:}\sigma_1 \rightarrow \sigma_2$.

  `Lower : PROTOCOL` $\rightarrow$ `Upper : PROTOCOL sharing Lower.T=Upper.T`

But what are sharing specifications?

Types for Program Modules

A skeletal module system ("modularity framework"):

- Dependent sums and functions (above).

Types for Program Modules

A skeletal module system ("modularity framework"):

- Dependent sums and functions (above).
- Lax modality for effects (monads).

Types for Program Modules

A skeletal module system ("modularity framework"):

- Dependent sums and functions (above).
- Lax modality for effects (monads).
- Modal account of the phase distinction.

Types for Program Modules

A skeletal module system ("modularity framework"):

- Dependent sums and functions (above).
- Lax modality for effects (monads).
- Modal account of the phase distinction.

A programming language is an instance of this framework!

Types for Program Modules

A skeletal module system ("modularity framework"):

- Dependent sums and functions (above).
- Lax modality for effects (monads).
- Modal account of the phase distinction.

A programming language is an instance of this framework!

- Choice of core type structure.

Types for Program Modules

A skeletal module system ("modularity framework"):

- Dependent sums and functions (above).
- Lax modality for effects (monads).
- Modal account of the phase distinction.

A programming language is an instance of this framework!

- Choice of core type structure.
- Choice of monadic effects.

Modal Formulation of Phases

Modules are, intrinsically, mixed phase entities.

- Static part, the types (but see later).
- Dynamic part, the types and the code.

Isolate the static part using an open lock, $🔓_{st}$.

- A proof-irrelevant proposition: "at most true".
- *Static equivalence*, $\Gamma, 🔓_{st} \vdash M \equiv N : \sigma$, disregards dynamic components.

The lock induces open and closed modalities, $\bigcirc_{st}(\sigma)$ and $\bullet_{st}(\sigma)$.

- Static part: $\bigcirc_{st}(\sigma) \cong 🔓_{st} \to \sigma$.
- Dynamic part: $\bullet_{st}(\bigcirc_{st}(\sigma)) \cong \mathbf{1}$.

Static Extent

The modal formulation accounts for static sharing:

FORMATION

$$\frac{\Gamma \vdash \sigma \ \mathit{sig} \qquad \Gamma, \unlock_{\mathsf{st}} \vdash V : \sigma}{\Gamma \vdash \{\sigma \mid \unlock_{\mathsf{st}} \hookrightarrow V\} \ \mathit{sig}}$$

INTRODUCTION

$$\frac{\Gamma \vdash U : \sigma \qquad \Gamma, \unlock_{\mathsf{st}} \vdash U \equiv V : \sigma}{\Gamma \vdash U : \{\sigma \mid \unlock_{\mathsf{st}} \hookrightarrow V\}}$$

ELIMINATION

$$\frac{\Gamma \vdash U : \{\sigma \mid \unlock_{\mathsf{st}} \hookrightarrow V\}}{\Gamma \vdash U : \sigma \qquad \Gamma, \unlock_{\mathsf{st}} \vdash U \equiv V : \sigma}$$

Interlude

Other forms of phase distinction are also possible, and useful:

- Compilation: abstraction vs visibility.

Interlude

Other forms of phase distinction are also possible, and useful:

- Compilation: abstraction vs visibility.
- Verification: specification vs structure.

Interlude

Other forms of phase distinction are also possible, and useful:

- Compilation: abstraction vs visibility.
- Verification: specification vs structure.
- Resource usage: cost vs behavior.

Interlude

Other forms of phase distinction are also possible, and useful:

- Compilation: abstraction vs visibility.
- Verification: specification vs structure.
- Resource usage: cost vs behavior.

Questions?

Relational Parametricity

Reynolds introduced parametricity to explain data abstraction.

- Implementors provide the type and its implementation.

Relational Parametricity

Reynolds introduced parametricity to explain data abstraction.

- Implementors provide the type and its implementation.
- Clients are polymorphic in the abstract type.

Relational Parametricity

Reynolds introduced parametricity to explain data abstraction.

- Implementors provide the type and its implementation.
- Clients are polymorphic in the abstract type.

Parametricity theorem: well-typed programs respect relational intepretations of abstract types.

Two implementations are co-correct when they correspond. By parametricity no client can distinguish them.

In the case of queues define

$$R(\vec{x}, \langle \vec{y}, \vec{z} \rangle) \quad \text{iff} \quad \vec{x} = (\vec{y} + \mathit{rev}(\vec{z}))$$

and check that the operations preserve the correspondence.

Queue Signature

```
signature QUEUE = sig
  type t
  val emp : t
  val ins : bool * t -> t
  val rem : t -> bool * t
end
```

Queue Implementation I

```
structure Q0 : QUEUE = struct
  type t = bool list
  val emp = nil
  fun ins (x, q) = ret (x :: q)
  fun rem q =
    bind val rev_q ← rev q in
    case rev_q of
    | nil ⇒ throw
    | x :: xs ⇒
      bind val rev_xs ← rev xs in
      ret (f, rev_xs)
end
```

Queue Implementation II

```
structure Q₁ : QUEUE = struct
  type t = bool list * bool list
  val emp = (nil, nil)
  fun ins (x, (fs, rs)) = ret (fs, x :: rs)
  fun rem (fs, rs) =
    case fs of
    | nil ⇒
      bind val rev_rs ← rev rs in
      (case rev_rs of
       | nil ⇒ throw
       | x::rs' ⇒ ret (x, rs', nil))
    | x::fs' ⇒ ret (x, fs', rs)
end
```

Relational Interpretation

The key to Reynolds' method is to interpret types as heterogenous binary relations.

- First, choose $R_t \subseteq \tau_{\mathsf{L}} \times \tau_{\mathsf{R}}$.

Relational Interpretation

The key to Reynolds' method is to interpret types as heterogenous binary relations.

- First, choose $R_t \subseteq \tau_{\mathsf{L}} \times \tau_{\mathsf{R}}$.
- Extend to $R_\tau \subseteq [\tau_{\mathsf{L}}/t]\tau \times [\tau_{\mathsf{R}}/t]\tau$ by the action of type constructors.

Relational Interpretation

The key to Reynolds' method is to interpret types as heterogenous binary relations.

- First, choose $R_t \subseteq \tau_{\mathsf{L}} \times \tau_{\mathsf{R}}$.
- Extend to $R_\tau \subseteq [\tau_{\mathsf{L}}/t]\tau \times [\tau_{\mathsf{R}}/t]\tau$ by the action of type constructors.

Importantly,

Relational Interpretation

The key to Reynolds' method is to interpret types as heterogenous binary relations.

- First, choose $R_t \subseteq \tau_\mathsf{L} \times \tau_\mathsf{R}$.
- Extend to $R_\tau \subseteq [\tau_\mathsf{L}/t]\tau \times [\tau_\mathsf{R}/t]\tau$ by the action of type constructors.

Importantly,

- $e \; R_{\mathsf{bool}} \; e'$ iff either $e \equiv \#\mathsf{t} \equiv e'$ or $e \equiv \#\mathsf{f} \equiv e'$.

Relational Interpretation

The key to Reynolds' method is to interpret types as heterogenous binary relations.

- First, choose $R_t \subseteq \tau_{\mathsf{L}} \times \tau_{\mathsf{R}}$.
- Extend to $R_\tau \subseteq [\tau_{\mathsf{L}}/t]\tau \times [\tau_{\mathsf{R}}/t]\tau$ by the action of type constructors.

Importantly,

- $e \, R_{\mathsf{bool}} \, e'$ iff either $e \equiv \#\mathsf{t} \equiv e'$ or $e \equiv \#\mathsf{f} \equiv e'$.
- $e \, R_{\tau_1 \to \tau_2} \, e'$ iff $e_1 \, R_{\tau_1} \, e_1'$ implies $e(e_1) \, R_{\tau_2} \, e'(e_1')$.

Relational Interpretation

The key to Reynolds' method is to interpret types as heterogenous binary relations.

- First, choose $R_t \subseteq \tau_\mathsf{L} \times \tau_\mathsf{R}$.
- Extend to $R_\tau \subseteq [\tau_\mathsf{L}/t]\tau \times [\tau_\mathsf{R}/t]\tau$ by the action of type constructors.

Importantly,

- $e \, R_{\mathsf{bool}} \, e'$ iff either $e \equiv \#\mathsf{t} \equiv e'$ or $e \equiv \#\mathsf{f} \equiv e'$.
- $e \, R_{\tau_1 \to \tau_2} \, e'$ iff $e_1 \, R_{\tau_1} \, e_1'$ implies $e(e_1) \, R_{\tau_2} \, e'(e_1')$.

That is,

Relational Interpretation

The key to Reynolds' method is to interpret types as heterogenous binary relations.

- First, choose $R_t \subseteq \tau_\mathsf{L} \times \tau_\mathsf{R}$.
- Extend to $R_\tau \subseteq [\tau_\mathsf{L}/t]\tau \times [\tau_\mathsf{R}/t]\tau$ by the action of type constructors.

Importantly,

- $e \, R_{\mathsf{bool}} \, e'$ iff either $e \equiv \#\mathsf{t} \equiv e'$ or $e \equiv \#\mathsf{f} \equiv e'$.
- $e \, R_{\tau_1 \to \tau_2} \, e'$ iff $e_1 \, R_{\tau_1} \, e'_1$ implies $e(e_1) \, R_{\tau_2} \, e'(e'_1)$.

That is,

- Observable outcomes are identical.

Relational Interpretation

The key to Reynolds' method is to interpret types as heterogenous binary relations.

- First, choose $R_t \subseteq \tau_{\mathsf{L}} \times \tau_{\mathsf{R}}$.
- Extend to $R_\tau \subseteq [\tau_{\mathsf{L}}/t]\tau \times [\tau_{\mathsf{R}}/t]\tau$ by the action of type constructors.

Importantly,

- $e \, R_{\mathsf{bool}} \, e'$ iff either $e \equiv \#\mathsf{t} \equiv e'$ or $e \equiv \#\mathsf{f} \equiv e'$.
- $e \, R_{\tau_1 \to \tau_2} \, e'$ iff $e_1 \, R_{\tau_1} \, e_1'$ implies $e(e_1) \, R_{\tau_2} \, e'(e_1')$.

That is,

- Observable outcomes are identical.
- Functions preserve the correspondence.

Reynolds Had It Easy

Reynolds worked with System F in which

- There are no dependencies: types are separate from terms *a priori*.

Reynolds Had It Easy

Reynolds worked with System F in which

- There are no dependencies: types are separate from terms *a priori*.
- Types are never computed as outputs, only taken as inputs.

Reynolds Had It Easy

Reynolds worked with System F in which

- There are no dependencies: types are separate from terms *a priori*.
- Types are never computed as outputs, only taken as inputs.

But these ingredients are necessary for a module system!

Reynolds Had It Easy

Reynolds worked with System F in which

- There are no dependencies: types are separate from terms *a priori*.
- Types are never computed as outputs, only taken as inputs.

But these ingredients are necessary for a module system!

- The phase distinction must be considered explicitly.

Reynolds Had It Easy

Reynolds worked with System F in which

- There are no dependencies: types are separate from terms *a priori*.
- Types are never computed as outputs, only taken as inputs.

But these ingredients are necessary for a module system!

- The phase distinction must be considered explicitly.
- Proof-irrelevant relations must be generalized to proof-relevant families of types to account for the universe.

Logical Relations as Types

The interpretation is usefully structured as another module system!

- Representation independence proofs are structures.

Logical Relations as Types

The interpretation is usefully structured as another module system!

- Representation independence proofs are structures.
- Static/dynamic distinction carries over.

Logical Relations as Types

The interpretation is usefully structured as another module system!

- Representation independence proofs are structures.
- Static/dynamic distinction carries over.

A new phase distinction arises:

Logical Relations as Types

The interpretation is usefully structured as another module system!

- Representation independence proofs are structures.
- Static/dynamic distinction carries over.

A new phase distinction arises:

- Syntax, $\bigcirc_{\mathsf{syn}}(A) \cong \text{🔓}_{\mathsf{syn}} \to A$.

Logical Relations as Types

The interpretation is usefully structured as another module system!

- Representation independence proofs are structures.
- Static/dynamic distinction carries over.

A new phase distinction arises:

- Syntax, $\bigcirc_{\mathsf{syn}}(A) \cong 🔓_{\mathsf{syn}} \to A$.
- Semantics, $\bullet_{\mathsf{syn}}(A)$, its closed complement: $\bigcirc_{\mathsf{syn}}(\bullet_{\mathsf{syn}}(A)) \cong \mathbf{1}$.

Logical Relations as Types

The interpretation is usefully structured as another module system!

- Representation independence proofs are structures.
- Static/dynamic distinction carries over.

A new phase distinction arises:

- Syntax, $\bigcirc_{\mathsf{syn}}(A) \cong \text{🔓}_{\mathsf{syn}} \to A$.
- Semantics, $\bullet_{\mathsf{syn}}(A)$, its closed complement: $\bigcirc_{\mathsf{syn}}(\bullet_{\mathsf{syn}}(A)) \cong \mathbf{1}$.

Types in this larger setting exhibit both distinctions independently!

Intepretation of Types

Types are interpreted a la Reynolds:

$$||Ty|| \cong \sum_{\tau:\mathsf{Type}} \mathsf{El}(\tau) \to \mathsf{Prop}_{\bullet_{\mathsf{syn}\vee\mathsf{st}}}$$

That is,

Intepretation of Types

Types are interpreted a la Reynolds:

$$||Ty|| \cong \sum_{\tau:\mathsf{Type}} \mathsf{El}(\tau) \to \mathsf{Prop}_{\bullet_{\mathsf{syn}\vee\mathsf{st}}}$$

That is,

- A syntactic type, τ, and

Intepretation of Types

Types are interpreted a la Reynolds:

$$||Ty|| \cong \sum_{\tau:\mathsf{Type}} \mathsf{El}(\tau) \to \mathsf{Prop}_{\bullet_{\mathsf{syn} \vee \mathsf{st}}}$$

That is,

- A syntactic type, τ, and
- A semantic and dynamic proof-irrelevant predicate on its elements.

Intepretation of Types

Types are interpreted a la Reynolds:

$$||\,Ty\,|| \cong \sum_{\tau:\mathsf{Type}} \mathsf{El}(\tau) \to \mathsf{Prop}_{\bullet_{\mathsf{syn}\vee\mathsf{st}}}$$

That is,

- A syntactic type, τ, and
- A semantic and dynamic proof-irrelevant predicate on its elements.

The elements of a type are those that satisfy the type's interpretation:

$$||El||(A, A^*) = \{\, M : El(A) \mid A^*(M) \,\}$$

Interpretation of Types

Booleans (observables) are interpreted discretely:

$$||Bool|| = \langle bool, \lambda b : El(bool).\bullet_{\mathsf{syn}\vee\mathsf{st}}(b \equiv true \vee b \equiv false)\rangle$$

Boolean constants validate the requirement:

$$||true|| = \langle true, \eta_{\bullet_{\mathsf{syn}\vee\mathsf{st}}}(inl(\star))\rangle$$
$$||false|| = \langle false, \eta_{\bullet_{\mathsf{syn}\vee\mathsf{st}}}(inr(\star))\rangle$$

Interpretation of Signatures

Signatures are interpreted as proof-relevant semantic families:

$$||Sig|| = \sum_{\sigma:Sig} Val(\sigma) \to U_{\bullet_{\mathsf{syn}}}$$

Access to their elements requires proof:

$$||Val|| = \lambda\langle\sigma, \sigma^*\rangle \in ||Sig||.\sum_{m:Val(\sigma)} \sigma^*(m)$$

Types as signatures are interpreted as proof-irrelevant predicates:

$$||Type : Sig|| = \langle Type, \lambda\tau : Val(Type).El(\tau) \to \mathrm{Prop}_{\bullet_{\mathsf{syn}\vee\mathsf{st}}}.$$

The Bigger Picture

All this is part of Sterling's program of Synthetic Tait Computability.

- Proof-relevant logical relations.

The Bigger Picture

All this is part of Sterling's program of Synthetic Tait Computability.

- Proof-relevant logical relations.
- Sheaf-theoretic formulation in terms of glueing.

The Bigger Picture

All this is part of Sterling's program of Synthetic Tait Computability.

- Proof-relevant logical relations.
- Sheaf-theoretic formulation in terms of glueing.
- Elegant proof of normalization for Cartesian cubical type theory.

The Bigger Picture

All this is part of Sterling's program of Synthetic Tait Computability.

- Proof-relevant logical relations.
- Sheaf-theoretic formulation in terms of glueing.
- Elegant proof of normalization for Cartesian cubical type theory.

See his forthcoming dissertation expected later this year!

Thank You!

Questions?

Correspondence Structure

A simulation over $\mathtt{Q}_{01} = [\blacksquare_{\mathsf{syn/l}} \hookrightarrow \mathtt{Q}_0, \blacksquare_{\mathsf{syn/r}} \hookrightarrow \mathtt{Q}_1]$ consists of the following data:

$$\begin{aligned} t &: \{\mathsf{Val}(\mathsf{type}) \mid \blacksquare_{\mathsf{syn}} \hookrightarrow \mathtt{Q}_{01}.\mathtt{t}\} \\ emp &: \{\mathsf{Val}(\langle\!|t|\!\rangle) \mid \blacksquare_{\mathsf{syn}} \hookrightarrow \mathtt{Q}_{01}.\mathtt{emp}\} \\ ins &: \{\mathsf{Val}(\langle\!|\mathsf{bool} * t \rightharpoonup t|\!\rangle) \mid \blacksquare_{\mathsf{syn}} \hookrightarrow \mathtt{Q}_{01}.\mathtt{ins}\} \\ rem &: \{\mathsf{Val}(\langle\!|t \rightharpoonup \mathsf{bool} * t|\!\rangle) \mid \blacksquare_{\mathsf{syn}} \hookrightarrow \mathtt{Q}_{01}.\mathtt{rem}\} \end{aligned}$$

$$\begin{aligned} &\mathsf{invariant} : \{\mathscr{U}^{\alpha}_{\bullet_{\mathsf{st}}} \mid \blacksquare_{\mathsf{syn}} \hookrightarrow \bullet_{\mathsf{st}}\bigcirc_{\mathsf{syn}}\mathsf{Val}(\mathtt{Q}_{01}.\mathtt{t})\} \\ &\mathsf{invariant} \cong \textstyle\sum_{q:\bigcirc_{\mathsf{syn}}\mathsf{Val}(\langle\!|\mathtt{Q}_{01}.\mathtt{t}|\!\rangle)} \bullet_{\mathsf{syn}}(\{\vec{x}, \vec{y}, \vec{z} : \bullet_{\mathsf{st}}(\mathsf{bits}) \mid \vec{x} = (\vec{y} + rev(\vec{z})) \land \ldots\}) \\ &\qquad \ldots = q = [\blacksquare_{\mathsf{syn/l}} \hookrightarrow \lceil\vec{x}\rceil \mid \blacksquare_{\mathsf{syn/r}} \hookrightarrow (\lceil\vec{y}\rceil, \lceil\vec{z}\rceil)] \end{aligned}$$