Phase Distinctions in Type Theory

Robert Harper
Carnegie Mellon University

(Joint work with Jon Sterling, Yue Niu, and Harrison Grodin)

December 2021

Acknowledgments

Thank you to David and Tim for the kind invitation!

Please see cited papers for references and discussion of related work.

This work was supported in part by AFOSR under grants MURI FA9550-15-1-0053 and FA9550-19-1-0216 (Tristan Nguyen, program manager) and in part by NSF under award number CCF-1901381. 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 or the NSF.

The Original Phase Distinction

Most PL's distinguish two phases of processing:

- Compile-time: parsing, type checking, compilation.
- Run-time: (compilation and) execution, including effects.

The distinction is fundamental to well-established development practices!

- Separate development.
- Interfaces to libraries.
- Stability under change.

Static vs Dynamic in PL's

The phase distinction is expressed by distinguishing static from dynamic levels.

Static part: kinds classify constructors

- Type classifies types.
- Closed under products, functions, singletons.

Dynamic part: types classify code.

- Products, sums, functions.
- Control and storage effects.

Dynamic part depends on static part, but static depends only on static.

Modules Are Mixed-Phase

Program modules consolidate static and dynamic parts.

```
signature QUEUE = sig
  type elt
  type t
  val emp : t
  val ins : elt * t ⇀ t
  val rem : t ⇀ (elt * t) option
end

structure QL :> QUEUE = struct
  type elt = bool
  type t = elt list
  val emp = nil
  val ins = cons
  fun rem nil = NONE
    | let val (x,q') = rev q in SOME (x, rev q')
end
```

Modules and Phases

Types such as $\texttt{QL.elt} \rightharpoonup \texttt{QL.t}$ threaten the phase distinction!

- Structure `QL` has static and dynamic components.
- What, then, is type equality?

Moggi addressed this concern analytically:

- All modules decompose into static and dynamic parts.
- Maps between modules inherently respect phase separation.

For example, $q{:}\texttt{QUEUE} \vdash M : \texttt{QUEUE}$ separates into two parts:

- M^{st} defines types `elt` and `t` in terms of $q^{st}.\texttt{elt}$ and $q^{st}.\texttt{t}$.
- M^{dy} defines values `emp`, etc in terms of types and values in q.

Sharing 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 anticipatory abstraction over shared components.
- Supports off-the-shelf re-use.

But what do sharing specifications mean?

Types for program modules

Dreyer, Rossberg and Russo: full-scale analytic account in their F'ing Modules.

- Phase separation into System Fω.
- Rich module structure, including abstraction and sharing.

Here we consider a synthetic account.

- Modules come first: everything is a module.
- Phase are isolated declaratively.
- Type equality is phase-sensitive.

ModTT: A Type System for Modules

Following MacQueen, start with dependent types:

- A universe of core language types and programs.
- Dependent products $x{:}\sigma_1 \times \sigma_2$ for hierarchy.
- Dependent functions $x{:}\sigma_1 \rightarrow \sigma_2$ for parameterization.

Extend dependent types with

- A lax account of abstraction and effects.
- A modal account of the phase distinction.
- Extension types for sharing [cf cubical type theories].

Polymorphism arises from modules abstracting over the universe.

SML was the first full-scale dependently typed programming language!

Structure of ModTT

Basic judgments:

$\Gamma \vdash \sigma \; sig$	signature
$\Gamma \vdash \sigma \equiv \sigma'$	signature equality
$\Gamma \vdash V : \sigma$	module value
$\Gamma \vdash V \equiv V' : \sigma$	module value equality
$\Gamma \vdash M \div \sigma$	module computation
$\Gamma \vdash M \equiv M' \div \sigma$	module computation equality

Structure of ModTT

Signatures:

$$\frac{}{\Gamma \vdash \mathsf{type}\ \mathit{sig}} \qquad \frac{\Gamma \vdash \tau : \mathsf{type}}{\Gamma \vdash \mathsf{val}(\tau)\ \mathit{sig}} \qquad \frac{\Gamma \vdash \sigma\ \mathit{sig}}{\Gamma \vdash \Diamond\sigma\ \mathit{sig}}$$

$$\frac{\Gamma \vdash \sigma_0\ \mathit{sig} \qquad \Gamma, x{:}\sigma_0 \vdash \sigma_1\ \mathit{sig}}{\Gamma \vdash x : \sigma_0 \times \sigma_1\ \mathit{sig}} \qquad \frac{\Gamma \vdash \sigma_0\ \mathit{sig} \qquad \Gamma, x{:}\sigma_0 \vdash \sigma_1\ \mathit{sig}}{\Gamma \vdash x : \sigma_0 \rightarrow \sigma_1\ \mathit{sig}}$$

Structure of ModTT

Module computations and encapsulation:

$$\frac{\Gamma \vdash M \div \sigma}{\Gamma \vdash \{M\} : \Diamond\sigma} \qquad \frac{\Gamma \vdash V : \sigma}{\Gamma \vdash \mathsf{ret}(V) \div \sigma}$$

$$\frac{\Gamma \vdash V : \Diamond\sigma \qquad \Gamma, X : \sigma \vdash M \div \sigma'}{\Gamma \vdash X \leftarrow V; M \div \sigma'}$$

Sealing is a *pro forma* effect. For $M : \sigma$,

$$M :> \sigma \triangleq X \leftarrow \{M\}; \mathsf{ret}(X)$$

"Generativity": multiple bind's induce distinct values with distinct type components.

Modal Formulation of Phases

Propositional signature specifies static phase:

$$\frac{}{\Gamma \vdash \text{🔓}_{\mathsf{st}}\ \mathit{sig}} \qquad\qquad \frac{\Gamma \vdash V, V' : \text{🔓}_{\mathsf{st}}}{\Gamma \vdash V \equiv V' : \text{🔓}_{\mathsf{st}}}$$

Static equivalence of signatures:

$$\Gamma, \text{🔓}_{\mathsf{st}} \vdash \sigma \equiv \sigma'$$

Type checking respects static equivalence:

$$\frac{\Gamma \vdash V : \sigma \qquad \Gamma, \text{🔓}_{\mathsf{st}} \vdash \sigma \equiv \sigma'}{\Gamma \vdash V : \sigma'}$$

(and similarly for computations)

The static phase identifies expressions:

$$\frac{\Gamma \vdash \tau : \mathsf{type} \qquad \Gamma \vdash \text{🔓}_{\mathsf{st}}}{\Gamma \vdash * : \mathsf{val}(\tau)} \qquad\qquad \frac{\Gamma \vdash e : \mathsf{val}(\tau) \qquad \Gamma \vdash \text{🔓}_{\mathsf{st}}}{\Gamma \vdash e \equiv * : \mathsf{val}(\tau)}$$

and module computations:

$$\frac{\Gamma \vdash \sigma \ \mathit{sig} \qquad \Gamma \vdash \text{🔓}_{\mathsf{st}}}{\Gamma \vdash * \div \sigma} \qquad\qquad \frac{\Gamma \vdash M \div \sigma \qquad \Gamma \vdash \text{🔓}_{\mathsf{st}}}{\Gamma \vdash M \equiv * \div \sigma}$$

Necessary for static type checking, a key design parameter.

Static Extent

Sharing is accounted for by static extent signatures:

FORMATION
$$\frac{\Gamma \vdash \sigma \; sig \qquad \Gamma, \text{🔓}_{\mathsf{st}} \vdash V : \sigma}{\Gamma \vdash \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow V\} \; sig}$$

INTRODUCTION
$$\frac{\Gamma \vdash U : \sigma \qquad \Gamma, \text{🔓}_{\mathsf{st}} \vdash U \equiv V : \sigma}{\Gamma \vdash U : \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow V\}}$$

ELIMINATION
$$\frac{\Gamma \vdash U : \{\sigma \mid \text{🔓}_{\mathsf{st}} \hookrightarrow V\}}{\Gamma \vdash U : \sigma \qquad \Gamma, \text{🔓}_{\mathsf{st}} \vdash U \equiv V : \sigma}$$

SML sharing is encodable in terms of static extent.

Phased Modalities

The static open induces open and closed modalities:

- Purely static: $\bigcirc_{\mathsf{st}}(\sigma) = \unlock_{\mathsf{st}} \to \sigma$.
- Purely dynamic: $\bullet_{\mathsf{st}}(\sigma) = \sigma \vee \unlock_{\mathsf{st}}$ (pushout of projections)

Thus, modules of the closed signature has trivial static part:

$$\bigcirc_{\mathsf{st}}(\bullet_{\mathsf{st}}(\sigma)) \cong 1$$

Think of a module as indexed over its static aspect.

- Static aspect isolates index (type components).
- Dynamic aspect “hides” static aspect by “shifting” it to the dynamic.

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: If $e : \forall t.\sigma$, then for all τ, τ' and all $R : \tau \leftrightarrow \tau'$,

$$e[\tau] =_\sigma e[\tau'] \quad (\mathit{rel.}\ t \mapsto R)$$

Consequently, no client can distinguish corresponding implementations of an ADT:

- Define correspondence relation between implementation types.
- Show that the operations preserve this relation.

In short, an application of (binary, heterogeneous) Tait computability.

Relational Parametricity

Reynolds worked analytically with System F:

- Function types a la Tait: $R_{\tau_1 \to \tau_2} = R_{\tau_1} \to R_{\tau_2}$.
- Polymorphic types a la Girard: quantify over admissible types.

Extending Reynolds to ModTT poses challenges:

- Universe permits types as outputs, not just inputs.
- Mixed-phase dependent types.
- Extent types, static equivalence.
- Modality for effects.

Generalize parametricity relations to parametricity structures.

- Proof-relevant, for the universe (classifier of classifiers).
- Binary, heterogeous, as in Reynolds.
- Phase-separated, in two senses!

Logical Relations as Types

A synthetic account of parametricity, ParamTT.

- All types are parametricity structures.
- Itself a type system for phase-separated modules.
- Phase distinction between syntax and semantics.

Syntactic phase, $\unlock_{\mathsf{syn}}$.

- Purely syntactic: $\bigcirc_{\mathsf{syn}}(\sigma) = \unlock_{\mathsf{syn}} \to \sigma$.
- Purely semantic: $\bullet_{\mathsf{syn}}(\sigma) = \sigma \vee \unlock_{\mathsf{syn}}$.

Implicitly binarized a la Wadler with left and right parts:

- $\unlock_{\mathsf{syn}} = \unlock_{\mathsf{syn/l}} \vee \unlock_{\mathsf{syn/r}}$
- $\unlock_{\mathsf{syn/l}} \wedge \unlock_{\mathsf{syn/r}} = \bot$.

Interpretation of Signatures and Modules

Signatures are interpreted as

- A syntactic signature, together with
- A parametricity structure for its elements.

$$
\begin{aligned}
\mathit{Sig} &: \{\mathscr{U} \mid 🔓_{\mathsf{syn}} \hookrightarrow \mathit{Sig}\} \\
&\cong \sigma : \mathit{Sig} \times \{\mathscr{U} \mid 🔓_{\mathsf{syn}} \hookrightarrow \sigma\}
\end{aligned}
$$

Correspondingly, modules of a signature extract that structure:

$$
\begin{aligned}
\mathit{Mod} &: \{\mathit{Sig} \to \mathscr{U} \mid 🔓_{\mathsf{syn}} \hookrightarrow \mathit{Mod}\} \\
\mathit{Mod}(\sigma, \sigma^*) &\triangleq \sigma^*
\end{aligned}
$$

Dependent Functions

For $\sigma_0 : \mathit{Sig}$ and $\sigma_1 : \sigma_0 \to \mathrm{Sig}$,

$$\begin{aligned}\sigma_\pi^* &: \{\mathscr{U} \mid \unlock_{\mathsf{syn}} \hookrightarrow \mathit{Mod}(\sigma_\pi)\} \\ &\cong x : \mathit{Mod}(\sigma_0) \to \mathit{Mod}(\sigma_1(x))\end{aligned}$$

where

$$\sigma_\pi \triangleq x : \mathit{Mod}(\sigma_0) \to \sigma_1(x)$$

(the syntactic function signature, under $\unlock_{\mathsf{syn}}$)

Provides interpretation of abstraction and application.

Core Language

Universe of core language types:

$$\begin{aligned} \mathrm{Type} : \{ \mathit{Sig} \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathit{Type} \} \\ \triangleq (\mathit{Type}, \mathit{Type}^*) \end{aligned}$$

The semantics of types is given by purely dynamic parametricity structures:

$$\begin{aligned} \mathit{Type}^* &: \{ \mathscr{U} \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathit{Type} \} \\ &\cong \tau : \mathit{Type} \times \{ \mathscr{U}_{\bullet_{\mathsf{st}}} \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathit{Val}(\tau) \} \end{aligned}$$

The parametricity structure for values is extracted from that of the universe:

$$\begin{aligned} \mathit{Val} : \{ \mathit{Type} \to \mathit{Sig} \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathit{Val} \} \\ \mathit{Val}(\tau, \tau^*) \triangleq (\mathit{Val}(\tau), \tau^*) \end{aligned}$$

Interpretation of Booleans

Booleans are interpreted as purely dynamic, purely semantic structure:

$$
\begin{aligned}
\mathit{Bool} &: \{\, \mathit{Type} \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathit{Bool} \} \\
&\triangleq (\mathit{Bool}, \mathit{Bool}^*)
\end{aligned}
$$

Booleans are observably either true or false:

$$
\begin{aligned}
\mathit{Bool}^* &: \{\mathscr{U} \mid \text{🔓}_{\mathsf{syn}} \hookrightarrow \mathit{Val}(\mathit{bool})\} \\
&\cong b : \mathit{Val}(\mathit{bool}) \times \bullet_{\mathsf{syn}} \bullet_{\mathsf{st}} (b^* : 2 \times \mathit{case}(b^*; \mathit{true}; \mathit{false}))
\end{aligned}
$$

Could also be given Reynolds-style by isolating the propositional parametricity structures (subterminal over each syntactic object).

Justifying ParamTT

Parametricity structures can be explained in terms of toposes:

- ParamTT is the internal language of a pre-sheaf topos.
- Syntactic phase distinction: glueing syntax to semantics.
- Static phase distinction: phase-separated sets.

All an instance of Sterling's Synthetic Tait Computability.

- Synthetic formulation of proof-relevant logical relations.
- Normalization for Cartesian cubical type theory.

Further Reading

Details, comparisons, and citations:

Jon Sterling and H., "Logical Relations as Types: Proof-Relevant Parametricity for Program Modules, " J. ACM v.68, n.6, October 2022.

Sterling's dissertation:

Jon M. Sterling, "First Steps in Synthetic Tait Computability." CMU SCS Ph.D. Thesis, October 2021.

Cost and Behavior in Type Theory

Dependent type theory expresses behavior of programs.

- Insertion sort: $\texttt{insertsort} : \texttt{seq} \to \texttt{seq}$
- Merge sort: $\texttt{mergesort} : \texttt{seq} \to \texttt{seq}$

Extensionally, these are equal:

$$\texttt{insertsort} \equiv \texttt{mergesort} : \texttt{seq} \to \texttt{seq}$$

Yet, they have different costs (number of comparisons on size n):

- $\texttt{insertsort} : \texttt{seq} \xrightarrow{n^2} \texttt{seq}$
- $\texttt{mergesort} : \texttt{seq} \xrightarrow{n \lg n} \texttt{seq}$

But equal things cannot have different properties. Phases to the rescue ....

Integrating Cost and Behavior

Calf = Cost-Aware Logical Framework.

- Extends Pedrot and Tabareau's ∂CBPV integrating effects and dependency.
- Sole effect: step counting for recording resource usage.

Phase $\unlock_{\mathsf{ext}}$ distinguishes intensional from extensional properties.

- $\texttt{insertsort} : \texttt{seq} \xrightarrow{n^2} \texttt{seq}$
- $\texttt{mergesort} : \texttt{seq} \xrightarrow{n \lg n} \texttt{seq}$
- $\unlock_{\mathsf{ext}} \vdash \texttt{insertsort} \equiv \texttt{mergesort} : \texttt{seq} \to \texttt{seq}$

Thus, intensional codifies algorithms, extensional codifies functions.

Expressing Cost

Calf is equipped with a writer monad for step counting

- step(e): increment step-count, then behave as e.
- $🔓_{\mathsf{ext}} \vdash \mathsf{step}(e) \equiv e : \tau$: disregard resource accounting.

Steps have no intrinsic meaning.

- Defined equationally, not via an operational interpretation.
- (Under development: realizability interpretation.)

Resources are abstract and problem-specific:

- Number of comparisons for `mergesort` and `insersort`.
- Number of modulus operations for `GCD`.
- Number of queue operations for `batched-queues`.

Open and Closed Modalities

The open modality, $\bigcirc_{\mathsf{ext}}(A)$, isolates behavior.

The closed modality, $\bullet_{\mathsf{ext}}(A)$, ensures non-interference:

- $\bigcirc_{\mathsf{ext}}(\bullet_{\mathsf{ext}}(\tau)) \cong \mathbf{1}$: no extensional component.
- Consequently, any $\bullet_{\mathsf{ext}}(A) \to \bigcirc_{\mathsf{ext}}(B)$ is constant.

In short behavior cannot depend on the step count.

- Counter type is $\bullet_{\mathsf{ext}}(N)$, for the sequential case.
- And is $\bullet_{\mathsf{ext}}(N) \times \bullet_{\mathsf{ext}}(N)$, for the parallel case.

Expressing Cost

Costs have an additive monoidal structure for work:

$$\mathrm{step}^0_X(e) = e \qquad\qquad \mathrm{step}^c_X(\mathrm{step}^d_X(e)) = \mathrm{step}^{c+d}(e)$$

Costs commute with computations:

$$\mathrm{bind}(step^c_{F(A)}(e); f) = \mathrm{step}^c_X(\mathrm{bind}(e; f))$$

$$\lambda x.\mathrm{step}^c_X(e) = \mathrm{step}^c_{A \to X}(\lambda x.e)$$

(Parallelism adds multiplicative monoid for span a la Blelloch)

Cost Bounds

The cost of a computation $\mathsf{hasCost}(B, e, c)$ is defined as

$$b : B \times (e =_{F(B)} \mathsf{step}^c(\mathsf{ret}(b)))$$

For $c : A \to N$, the type $a : A \xrightarrow{c} B$ is short for

$$f : (a : A \to B) \times (a : A \to \mathsf{hasCost}(B(a), f(a), c(a))).$$

Similarly, $\mathsf{isBounded}(B, e, c)$ specifies an upper bound:

$$c' : N \times \bigcirc_{\mathsf{ext}}(c \leq_N c') \times \mathsf{hasCost}(B, e, c')$$

Comparison is in extensional mode. Allows for using behavior to analyze cost.

A Nicely Closed World

Calf, as a dependent type theory, is limited to total functions.

- Type-specific induction/recursion.
- Awkward compared to general recursion.

How to express efficient algorithms in Calf?

- All algorithms are instrumented with a "clock" for recursive calls.
- If insufficient time is available, terminates with partial result.

The clock is an artificiality in the behavioral setting, but a natural here.

- Typical cost measures bound recursion depth.
- Use cost analysis to "set the clock."

Calf Methodology

1. Instrument code for cost accounting,

$$\texttt{mod}_{instr}(x, y) = step(x \,\%\, y)$$

2. Define clocked version of algorithm,

$$\texttt{gcd}_{clocked} : \texttt{nat} \to (\texttt{nat} \times \texttt{nat}) \to \texttt{nat}.$$

$$\lambda(k).\lambda(x, y). \ldots \texttt{gcd}_{clocked}(k - 1)(y, \texttt{mod}_{instr}(y, \texttt{mod}_{instr}(x, y)))$$

3. Define cost recurrence by any means (cost of recurrence is irrelevant)

$$\texttt{gcd}_{depth}(x, y) : \texttt{nat} \times \texttt{nat} \to \texttt{nat}$$

4. Define complete algorithm:

$$\texttt{gcd}(x, y) = \texttt{gcd}_{clocked}(\texttt{gcd}_{depth}(x, y))(x, y)$$

Calf Methodology

The recurrence determines an upper bound on modulus operations:

$$\mathsf{isBounded}(N; \mathsf{gcd}(x, y); \mathsf{gcd}_{depth}(x, y))$$

The depth recurrence can be solved:

$$\mathsf{gcd}_{depth}(x, y) \leq \mathsf{Fib}^{-1}(x) + 1$$

Combining these,

$$\mathsf{isBounded}(N; \mathsf{gcd}(x, y); \mathsf{Fib}^{-1}(x, y))$$

These, and other (sequential and parallel) bounds, are fully mechanized in Agda.

Further Reading

For background, comparisons, citations, and full development:

Yue Niu, Jon M. Sterling, Harrison Grodin, and H, "A Cost-Aware Logical Framework." To appear, ACM SIGPLAN Symp. on Princ. of Prog. Lang. (POPL). Philadelphia, January 2022 (to appear).

Mechanization in Agda:

https://github.com/jonsterling/agda-calf

Watch for Niu's Ph.D., expected 2023!

Phase Distinctions Abound!

Information flow security (ongoing work, with Sterling and Stephanie Balzer).

- Public (vs private) phase, 🔓_{pub}.
- Public equivalence: all private computations are equated.
- Scales naturally to a lattice of levels.

Debugging vs delivery: $\text{🔓}_{deliver}$.

- Instrument code with profiling and tracing information (a la step counting).
- Active under debug phase, disregarded under $\text{🔓}_{deliver}$.
- Presented at ML Workshop,
  `https://www.cs.cmu.edu/~rwh/papers/multiphase/mlw.pdf`

Thank you!

Queue Signature

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

Queue Implementation: Lists

```
structure QL : QUEUE = struct
  type elt = bool
  type t = elt 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: Pair of Lists

```
structure QLL : QUEUE = struct
  type elt = bool
  type t = elt list * elt 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
```

Correspondence Structure

A simulation over $\mathtt{C} = [\unlock_{\mathsf{syn}/\mathsf{l}} \hookrightarrow \mathtt{QL}, \unlock_{\mathsf{syn}/\mathsf{r}} \hookrightarrow \mathtt{QLL}]$ consists of the following data:

$$\begin{aligned}
t &: \{\mathsf{Mod}(\mathsf{type}) \mid \unlock_{\mathsf{syn}} \hookrightarrow \mathtt{QC.t}\} \\
\mathit{emp} &: \{\mathsf{Mod}(\langle\!|t|\!\rangle) \mid \unlock_{\mathsf{syn}} \hookrightarrow \mathtt{QC.emp}\} \\
\mathit{ins} &: \{\mathsf{Mod}(\langle\!|\mathsf{bool} * t \rightharpoonup t|\!\rangle) \mid \unlock_{\mathsf{syn}} \hookrightarrow \mathtt{QC.ins}\} \\
\mathit{rem} &: \{\mathsf{Mod}(\langle\!|t \rightharpoonup \mathsf{bool} * t|\!\rangle) \mid \unlock_{\mathsf{syn}} \hookrightarrow \mathtt{QC.rem}\}
\end{aligned}$$

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