Abstraction Functions as Types

Modular Verification of Cost and Behavior in Dependent Type Theory

Harrison Grodin, Runming Li, and Robert Harper

POPL 2026

Carnegie Mellon University

motivation

record $\textsc{PreQueue}$ **where**

X : **Type**

empty : $1 \to X$

enqueue : $\mathbb{N} \to X \to X$

dequeue : $X \to \mathbb{N} \times X$

$LQ : \textsc{PreQueue}$

$LQ.X := \textsc{List}\ \mathbb{N}$

$LQ.\text{empty}\ () := []$

$LQ.\text{enqueue}\ n\ l := l +\!\!+ [n]$

$LQ.\text{dequeue}\ [] := (0, [])$

$LQ.\text{dequeue}\ (n :: l) := (n, l)$

$LQ : \textsc{PreQueue}$

$LQ.X := \textsc{List}\ \mathbb{N}$

$LQ.\mathrm{empty}\ () := []$

$LQ.\mathrm{enqueue}\ n\ l := l +\!\!+ [n]$

$LQ.\mathrm{dequeue}\ [] := (0, [])$

$LQ.\mathrm{dequeue}\ (n :: l) := (n, l)$

$BQ : \textsc{PreQueue}$

$BQ.X := \textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}$

$BQ.\mathrm{empty}\ () := ([], [])$

$BQ.\mathrm{enqueue}\ n\ (l_1, l_2) := (n :: l_1, l_2)$

$BQ.\mathrm{dequeue}\ (l_1, n :: l_2) := n, (l_1, l_2)$

$BQ.\mathrm{dequeue}\ (l_1, []) := \ldots\ \mathit{reverse}\ l_1 \ldots$

When a programmer makes use of an abstract data object,
he is concerned only with the behavior
which that object exhibits. . .

Liskov and Zilles (1974)

Modular verification?

$$c_1\ c_2 : (Q : \textsc{PreQueue}) \to \mathbb{N} \times Q.X$$

$$c_1\ Q \coloneqq Q.\mathsf{empty}\ () \triangleright Q.\mathsf{enqueue}\ 1 \triangleright Q.\mathsf{enqueue}\ 2 \triangleright Q.\mathsf{dequeue}$$

$$c_2\ Q \coloneqq (1, Q.\mathsf{empty}\ () \triangleright Q.\mathsf{enqueue}\ 2)$$

Modular verification?

$$c_1\ c_2 : (Q : \textsc{PreQueue}) \to \mathbb{N} \times Q.X$$
$$c_1\ Q := Q.\text{empty}\ () \triangleright Q.\text{enqueue}\ 1 \triangleright Q.\text{enqueue}\ 2 \triangleright Q.\text{dequeue}$$
$$c_2\ Q := (1, Q.\text{empty}\ () \triangleright Q.\text{enqueue}\ 2)$$

Theorem?

For all queue implementations $Q : \textsc{PreQueue}$, $c_1\ Q = c_2\ Q$.

Modular verification?

$$c_1 \ c_2 : (Q : \textsc{PreQueue}) \to \mathbb{N} \times Q.X$$
$$c_1 \ Q := Q.\text{empty} \ () \triangleright Q.\text{enqueue} \ 1 \triangleright Q.\text{enqueue} \ 2 \triangleright Q.\text{dequeue}$$
$$c_2 \ Q := (1, Q.\text{empty} \ () \triangleright Q.\text{enqueue} \ 2)$$

Theorem?

For all queue implementations $Q : \textsc{PreQueue}$, $c_1 \ Q = c_2 \ Q$.

Example

Indeed, $c_1 \ LQ = (1, [2]) = c_2 \ LQ$.

Modular verification?

$$c_1\ c_2 : (Q : \textsc{PreQueue}) \to \mathbb{N} \times Q.X$$
$$c_1\ Q := Q.\mathsf{empty}\ () \triangleright Q.\mathsf{enqueue}\ 1 \triangleright Q.\mathsf{enqueue}\ 2 \triangleright Q.\mathsf{dequeue}$$
$$c_2\ Q := (1, Q.\mathsf{empty}\ () \triangleright Q.\mathsf{enqueue}\ 2)$$

Theorem?

For all queue implementations $Q : \textsc{PreQueue}$, $c_1\ Q = c_2\ Q$.

Example

Indeed, $c_1\ LQ = (1, [2]) = c_2\ LQ$.

Counterexample

Alas, $c_1\ BQ = (1, ([], [2])) \neq (1, ([2], [])) = c_2\ BQ$.

Semantic modularity

Observation

Efficient implementations rarely satisfy verification-level properties.

For example, implementing dictionaries as balanced trees, union is not

- associative,
- commutative,
- ...

because the exact tree shape will not be the same.

Semantic modularity

Observation

Efficient implementations rarely satisfy verification-level properties.

For example, implementing dictionaries as balanced trees, union is not

- associative,
- commutative,
- ...

because the exact tree shape will not be the same.

Syntactic vs. Semantic Modularity

In the interest of practicality, simple programming languages include tools for modularity using syntactic approximations (e.g., existential types).

For verification, we need a semantic notion of modularity.

How can we reconcile

modularity with verification?

abstraction functions

Abstraction Functions

For [proving correctness of BQ.enqueue*]... define the relationship between the abstract space [LQ.X] in which [LQ*.enqueue*] is written, and the space [BQ.X] of the concrete representation... by giving a function [α : BQ.X $\to$ LQ.X]...*

Hoare (1972)

Abstraction Functions

*For [proving correctness of BQ.*enqueue*]... define the relationship between the abstract space [LQ.X] in which [LQ.*enqueue*] is written, and the space [BQ.X] of the concrete representation... by giving a function [α : BQ.X $\rightarrow$ LQ.X]...*

Hoare (1972)

$$
\begin{array}{ccc}
1 & \xrightarrow{BQ.\mathsf{empty}} & BQ.X \\
\Vert & & \downarrow{\alpha} \\
1 & \xrightarrow[LQ.\mathsf{empty}]{} & LQ.X
\end{array}
\qquad
\begin{array}{ccc}
BQ.X & \xrightarrow{BQ.\mathsf{enqueue}\ n} & BQ.X \\
{\alpha}\downarrow & & \downarrow{\alpha} \\
LQ.X & \xrightarrow[LQ.\mathsf{enqueue}\ n]{} & LQ.X
\end{array}
\qquad
\begin{array}{ccc}
BQ.X & \xrightarrow{BQ.\mathsf{dequeue}} & \mathbb{N} \times BQ.X \\
{\alpha}\downarrow & & \downarrow{\mathbb{N}\times\alpha} \\
LQ.X & \xrightarrow[LQ.\mathsf{dequeue}]{} & \mathbb{N} \times LQ.X
\end{array}
$$

Abstraction Functions

For [proving correctness of BQ.enqueue]... define the relationship between the abstract space [LQ.X] in which [LQ.enqueue] is written, and the space [BQ.X] of the concrete representation... by giving a function [α : BQ.X $\to$ LQ.X]...

Hoare (1972)

$$
\begin{array}{ccc}
1 & \xrightarrow{BQ.\mathrm{empty}} & BQ.X \\
\| & & \downarrow \alpha \\
1 & \xrightarrow[LQ.\mathrm{empty}]{} & LQ.X
\end{array}
\qquad
\begin{array}{ccc}
BQ.X & \xrightarrow{BQ.\mathrm{enqueue}\ n} & BQ.X \\
\alpha \downarrow & & \downarrow \alpha \\
LQ.X & \xrightarrow[LQ.\mathrm{enqueue}\ n]{} & LQ.X
\end{array}
\qquad
\begin{array}{ccc}
BQ.X & \xrightarrow{BQ.\mathrm{dequeue}} & \mathbb{N} \times BQ.X \\
\alpha \downarrow & & \downarrow \mathbb{N}\times\alpha \\
LQ.X & \xrightarrow[LQ.\mathrm{dequeue}]{} & \mathbb{N} \times LQ.X
\end{array}
$$

$$\alpha : BQ.X \to LQ.X$$

$$\alpha(l_1, l_2) \coloneqq l_2 +\!\!+\ rev(l_1)$$

Verification, up to abstraction

$$c_1 \ c_2 : (Q : \textsc{PreQueue}) \to \mathbb{N} \times Q.X$$
$$c_1 \ Q := Q.\text{empty} \ () \triangleright Q.\text{enqueue} \ 1 \triangleright Q.\text{enqueue} \ 2 \triangleright Q.\text{dequeue}$$
$$c_2 \ Q := (1, Q.\text{empty} \ () \triangleright Q.\text{enqueue} \ 2)$$

Remark

Even though $c_1 \ BQ = (1, \underline{([], [2])}) \neq (1, \underline{([2], [])}) = c_2 \ BQ$,

$$\alpha([], [2]) = [2] = \alpha([2], []).$$

Verification, up to abstraction

$$c_1 \;\; c_2 : (Q : \textsc{PreQueue}) \to \mathbb{N} \times Q.X$$
$$c_1 \;\; Q := Q.\mathsf{empty}\ () \triangleright Q.\mathsf{enqueue}\ 1 \triangleright Q.\mathsf{enqueue}\ 2 \triangleright Q.\mathsf{dequeue}$$
$$c_2 \;\; Q := (1, Q.\mathsf{empty}\ () \triangleright Q.\mathsf{enqueue}\ 2)$$

Remark

Even though $c_1 \; BQ = (1, \underline{([], [2])}) \neq (1, \underline{([2], [])}) = c_2 \; BQ$,

$$\alpha([], [2]) = [2] = \alpha([2], []).$$

Observation

Client-side verification of BQ happens at the level of LQ using α.

Build $\left(\begin{array}{c} BQ.X \\ \big\downarrow{\scriptstyle \alpha} \\ LQ.X \end{array} \right)$ **into a type.**

$$\begin{CD} BQ.X \\ @VV{\alpha}V \\ LQ.X \end{CD} \qquad \begin{CD} 1 @>{BQ.\mathsf{empty}}>> BQ.X \\ @| @VV{\alpha}V \\ 1 @>>{LQ.\mathsf{empty}}> LQ.X \end{CD} \qquad \begin{CD} BQ.X @>{BQ.\mathsf{enqueue}\ n}>> BQ.X \\ @V{\alpha}VV @VV{\alpha}V \\ LQ.X @>>{LQ.\mathsf{enqueue}\ n}> LQ.X \end{CD}$$

$$BLQ.X \qquad 1 \xrightarrow{BLQ.\mathsf{empty}} BLQ.X \qquad BLQ.X \xrightarrow{BLQ.\mathsf{enqueue}\ n} BLQ.X$$

the abstract phase

The abstract phase

Definition

The abstract phase is a proposition, **abs**.

When **abs** holds (i.e., is in the context), we are looking at the abstract interface.

The abstract phase

Definition

The abstract phase is a proposition, **abs**.

When **abs** holds (i.e., is in the context), we are looking at the abstract interface.

Goal

Abstractly, want $BLQ.X = LQ.X$.

$$\begin{array}{c} \mathsf{BQ}.X \\ \downarrow{\scriptstyle \alpha} \\ \mathsf{LQ}.X \end{array}
\qquad\qquad
\begin{array}{ccc} 1 & \xrightarrow{\mathsf{BQ}.\mathsf{empty}} & \mathsf{BQ}.X \\ \| & & \downarrow{\scriptstyle \alpha} \\ 1 & \xrightarrow[\mathsf{LQ}.\mathsf{empty}]{} & \mathsf{LQ}.X \end{array}
\qquad\qquad
\begin{array}{ccc} \mathsf{BQ}.X & \xrightarrow{\mathsf{BQ}.\mathsf{enqueue}\ n} & \mathsf{BQ}.X \\ {\scriptstyle \alpha}\downarrow & & \downarrow{\scriptstyle \alpha} \\ \mathsf{LQ}.X & \xrightarrow[\mathsf{LQ}.\mathsf{enqueue}\ n]{} & \mathsf{LQ}.X \end{array}$$

Modalities and gluing

Definition

Definable using **abs**,

- the *concrete modality* $\bullet$ marks data as private (available for efficiency), and
- the *abstract modality* $\circ$ marks data as public (available for verification).

Modalities and gluing

Definition

Definable using **abs**,

- the *concrete modality* $\bullet$ marks data as private (available for efficiency), and
- the *abstract modality* $\circ$ marks data as public (available for verification).

Definition (gluing)

$$BLQ.X \coloneqq \{(b_\bullet, l_\circ) : \bullet(BQ.X) \times \circ(LQ.X) \mid \mathrm{map}_\bullet(\eta_\circ \circ \alpha)(b_\bullet) = \eta_\bullet(l_\circ)\}$$

Modalities and gluing

Definition

Definable using **abs**,

- the *concrete modality* $\bullet$ marks data as private (available for efficiency), and
- the *abstract modality* $\circ$ marks data as public (available for verification).

Definition (gluing)

$$BLQ.X := \{(b_\bullet, l_\circ) : \bullet(BQ.X) \times \circ(LQ.X) \mid \mathrm{map}_\bullet(\eta_\circ \circ \alpha)(b_\bullet) = \eta_\bullet(l_\circ)\}$$

Lemma

Abstractly, $\bullet(BQ.X) = 1$ *and* $\circ(LQ.X) = LQ.X$.

Modalities and gluing

Definition

Definable using **abs**,

- the *concrete modality* $\bullet$ marks data as private (available for efficiency), and
- the *abstract modality* $\circ$ marks data as public (available for verification).

Definition (gluing)

$$BLQ.X := \{(b_\bullet, l_\circ) : \bullet(BQ.X) \times \circ(LQ.X) \mid \mathrm{map}_\bullet(\eta_\circ \circ \alpha)(b_\bullet) = \eta_\bullet(l_\circ)\}$$

Lemma

Abstractly, $\bullet(BQ.X) = 1$ *and* $\circ(LQ.X) = LQ.X$.

Theorem

Abstractly, $BLQ.X = \{((), l) : 1 \times LQ.X \mid () = ()\} = LQ.X$.

Programming with a phased implementation type

$$BLQ.\mathrm{empty} : 1 \to BLQ.X$$

Programming with a phased implementation type

$$BLQ.\mathsf{empty} : 1 \to \{(b_\bullet, l_\circ) : \bullet(BQ.X) \times \circ(LQ.X) \mid \mathsf{map}_\bullet(\eta_\circ \circ \alpha)(b_\bullet) = \eta_\bullet(l_\circ)\}$$

Programming with a phased implementation type

$$BLQ.\mathsf{empty} : 1 \to \{(b_\bullet, l_\circ) : \bullet(BQ.X) \times \circ(LQ.X) \mid \mathsf{map}_\bullet(\eta_\circ \circ \alpha)(b_\bullet) = \eta_\bullet(l_\circ)\}$$

$$BLQ.\mathsf{empty}\ () := (\eta_\bullet(BQ.\mathsf{empty}\ ()), \eta_\circ(LQ.\mathsf{empty}\ ()))$$

Programming with a phased implementation type

$$BLQ.\mathsf{empty} : 1 \to \{(b_\bullet, l_\circ) : \bullet(BQ.X) \times \circ(LQ.X) \mid \mathsf{map}_\bullet(\eta_\circ \circ \alpha)(b_\bullet) = \eta_\bullet(l_\circ)\}$$

$$BLQ.\mathsf{empty}\ () := (\eta_\bullet(BQ.\mathsf{empty}\ ()), \eta_\circ(LQ.\mathsf{empty}\ ()))$$

To show

$$\mathsf{map}_\bullet(\eta_\circ \circ \alpha)(\eta_\bullet(BQ.\mathsf{empty}\ ())) = \eta_\bullet(\eta_\circ(LQ.\mathsf{empty}\ ()))$$

Programming with a phased implementation type

$$BLQ.\mathsf{empty} : 1 \to \{(b_\bullet, l_\circ) : \bullet(BQ.X) \times \circ(LQ.X) \mid \mathsf{map}_\bullet(\eta_\circ \circ \alpha)(b_\bullet) = \eta_\bullet(l_\circ)\}$$

$$BLQ.\mathsf{empty}\ () := (\eta_\bullet(BQ.\mathsf{empty}\ ()), \eta_\circ(LQ.\mathsf{empty}\ ()))$$

To show

$$\eta_\bullet(\eta_\circ(\alpha(BQ.\mathsf{empty}\ ()))) = \eta_\bullet(\eta_\circ(LQ.\mathsf{empty}\ ()))$$

Programming with a phased implementation type

$$BLQ.\mathsf{empty} : 1 \to \{(b_\bullet, l_\circ) : \bullet(BQ.X) \times \circ(LQ.X) \mid \mathsf{map}_\bullet(\eta_\circ \circ \alpha)(b_\bullet) = \eta_\bullet(l_\circ)\}$$

$$BLQ.\mathsf{empty}\ () := (\eta_\bullet(BQ.\mathsf{empty}\ ()), \eta_\circ(LQ.\mathsf{empty}\ ()))$$

To show

$$\eta_\bullet(\eta_\circ(\alpha(BQ.\mathsf{empty}\ ()))) = \eta_\bullet(\eta_\circ(LQ.\mathsf{empty}\ ()))$$

it suffices to show that $\alpha(BQ.\mathsf{empty}\ ()) = LQ.\mathsf{empty}\ ()$:

$$\begin{array}{ccc} 1 & \xrightarrow{BQ.\mathsf{empty}} & BQ.X \\ \| & & \downarrow{\scriptstyle\alpha} \\ 1 & \xrightarrow[LQ.\mathsf{empty}]{} & LQ.X \end{array}$$

noninterference and modularity

"The principle of non-interference": . . . the correct working. . . can be established by taking. . . into account [the] exterior specification only, and not the particulars of [the] interior construction.

Dijkstra (1965)

Noninterference

Definition (queue specification type)

$$\textsc{Queue} := \{Q : \textsc{PreQueue} \mid \textbf{abs} \to (Q = LQ)\}$$

Noninterference

Definition (queue specification type)

$$\textsc{Queue} := \{Q : \textsc{PreQueue} \mid \textbf{abs} \to (Q = LQ)\}$$

Example

$BLQ : \textsc{Queue}$, because abstractly, $BLQ = LQ$.

Noninterference

Definition (queue specification type)

$$\textsc{Queue} := \{Q : \textsc{PreQueue} \mid \textbf{abs} \to (Q = LQ)\}$$

Example

$BLQ : \textsc{Queue}$, because abstractly, $BLQ = LQ$.

$$c_1\ c_2 : (Q : \textsc{Queue}) \to \mathbb{N} \times Q.X$$
$$c_1\ Q := Q.\text{empty}\ () \triangleright Q.\text{enqueue}\ 1 \triangleright Q.\text{enqueue}\ 2 \triangleright Q.\text{dequeue}$$
$$c_2\ Q := (1, Q.\text{empty}\ () \triangleright Q.\text{enqueue}\ 2)$$

Noninterference

Definition (queue specification type)

$$\textsc{Queue} := \{Q : \textsc{PreQueue} \mid \textbf{abs} \to (Q = LQ)\}$$

Example

$BLQ : \textsc{Queue}$, because abstractly, $BLQ = LQ$.

$$c_1\ c_2 : (Q : \textsc{Queue}) \to \mathbb{N} \times Q.X$$
$$c_1\ Q := Q.\text{empty}\ () \triangleright Q.\text{enqueue}\ 1 \triangleright Q.\text{enqueue}\ 2 \triangleright Q.\text{dequeue}$$
$$c_2\ Q := (1, Q.\text{empty}\ () \triangleright Q.\text{enqueue}\ 2)$$

Theorem

For all $Q : \textsc{Queue}$, *have abstractly,* $c_1(Q) = c_1(LQ) = c_2(LQ) = c_2(Q)$.

cost analysis

You cannot have interchangeable modules unless these modules share similar complexity behavior… Complexity assertions have to be part of the interface.

Stepanov (1995)

Cost analysis in dependent type theory

Calf

Calf is a dependent type theory for cost analysis with a monadic cost effect.

Cost is an effect: $\mathbf{charge}\langle c \rangle(-)$ records $c : \mathbb{C}$ units of cost.

Cost analysis in dependent type theory

Calf

Calf is a dependent type theory for cost analysis with a monadic cost effect.

Cost is an effect: $\mathbf{charge}\langle c\rangle(-)$ records $c : \mathbb{C}$ units of cost.

$$\frac{\Gamma \vdash v : X}{\Gamma \vdash \mathbf{ret}(v) : \mathbf{M}(X)} \qquad \frac{\Gamma \vdash c : \mathbb{C} \qquad \Gamma \vdash e : \mathbf{M}(X)}{\Gamma \vdash \mathbf{charge}\langle c\rangle(e) : \mathbf{M}(X)}$$

Cost analysis in dependent type theory

Calf

Calf is a dependent type theory for cost analysis with a monadic cost effect.

Cost is an effect: $\textbf{charge}\langle c \rangle(-)$ records $c : \mathbb{C}$ units of cost.

$$\frac{\Gamma \vdash v : X}{\Gamma \vdash \textbf{ret}(v) : \mathbf{M}(X)} \qquad \frac{\Gamma \vdash c : \mathbb{C} \qquad \Gamma \vdash e : \mathbf{M}(X)}{\Gamma \vdash \textbf{charge}\langle c \rangle(e) : \mathbf{M}(X)}$$

Decalf

Decalf extends Calf with *inequality of costs* (simple directed type theory).

$e \leq e'$ means e and e' compute the same data, but e takes less-or-equal cost.

record $\textsc{PreQueue}$ **where**

X : **Type**

empty : $1 \to \mathbf{M}(X)$

enqueue : $\mathbb{N} \to X \to \mathbf{M}(X)$

dequeue : $X \to \mathbf{M}(\mathbb{N} \times X)$

record $\textsc{PreQueue}$ **where**

$X : \mathbf{Type}$

$\text{empty} : 1 \to \mathbf{M}(X)$

$\text{enqueue} : \mathbb{N} \to X \to \mathbf{M}(X)$

$\text{dequeue} : X \to \mathbf{M}(\mathbb{N} \times X)$

$BQ : \textsc{PreQueue}$

$BQ.X \coloneqq \textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N}$

$BQ.\text{empty}\ () \coloneqq \mathbf{ret}([], [])$

$BQ.\text{enqueue}\ n\ (l_1, l_2) \coloneqq \mathbf{charge}\langle 1 \rangle(\mathbf{ret}(n :: l_1, l_2))$

$BQ.\text{dequeue}\ (l_1, n :: l_2) \coloneqq \mathbf{ret}(n, (l_1, l_2))$

$BQ.\text{dequeue}\ (l_1, []) \coloneqq \mathbf{charge}\langle |l_1| \rangle(\cdots \mathit{reverse}\ l_1 \cdots)$

$$LQ.\text{enqueue}\ n\ l \coloneqq \textbf{charge}\langle ? \rangle(\textbf{ret}(l +\!\!+ [n]))$$

$$LQ.\text{dequeue}\ [] \coloneqq \textbf{charge}\langle ? \rangle(\textbf{ret}())(0, [])$$

$$LQ.\text{dequeue}\ (n :: l) \coloneqq \textbf{charge}\langle ? \rangle(\textbf{ret}(n, l))$$

$$LQ.\mathrm{enqueue}\ n\ l := \mathbf{charge}\langle ? \rangle(\mathbf{ret}(l +\!\!+ [n]))$$

$$LQ.\mathrm{dequeue}\ [] := \mathbf{charge}\langle ? \rangle(\mathbf{ret}())(0, [])$$

$$LQ.\mathrm{dequeue}\ (n :: l) := \mathbf{charge}\langle ? \rangle(\mathbf{ret}(n, l))$$

$$\begin{array}{ccc}
BQ.X & \xrightarrow{BQ.\mathrm{enqueue}\ n} & \mathbf{M}(BQ.X) \\
\alpha \downarrow & ? & \downarrow \mathbf{M}(\alpha) \\
LQ.X & \xrightarrow[LQ.\mathrm{enqueue}\ n]{} & \mathbf{M}(LQ.X)
\end{array}$$

$$LQ.\text{enqueue}\ n\ l := \mathbf{charge}\langle 1\rangle(\mathbf{ret}(l +\!\!+ [n]))$$

$$LQ.\text{dequeue}\ [] := \mathbf{charge}\langle ?\rangle(\mathbf{ret}())(0, [])$$

$$LQ.\text{dequeue}\ (n :: l) := \mathbf{charge}\langle ?\rangle(\mathbf{ret}(n, l))$$

$$\begin{array}{ccc}
BQ.X & \xrightarrow{BQ.\text{enqueue}\ n} & \mathbf{M}(BQ.X) \\
\alpha \downarrow & = & \downarrow \mathbf{M}(\alpha) \\
LQ.X & \xrightarrow[LQ.\text{enqueue}\ n]{} & \mathbf{M}(LQ.X)
\end{array}$$

$$LQ.\mathsf{enqueue}\ n\ l := \mathbf{charge}\langle 1 \rangle(\mathbf{ret}(l +\!\!+ [n]))$$
$$LQ.\mathsf{dequeue}\ [] := \mathbf{charge}\langle ? \rangle(\mathbf{ret}())(0, [])$$
$$LQ.\mathsf{dequeue}\ (n :: l) := \mathbf{charge}\langle ? \rangle(\mathbf{ret}(n, l))$$

$$\begin{array}{ccc} BQ.X & \xrightarrow{BQ.\mathsf{enqueue}\ n} & \mathbf{M}(BQ.X) \\ {\scriptstyle \alpha}\downarrow & = & \downarrow{\scriptstyle \mathbf{M}(\alpha)} \\ LQ.X & \xrightarrow[LQ.\mathsf{enqueue}\ n]{} & \mathbf{M}(LQ.X) \end{array} \qquad \begin{array}{ccc} BQ.X & \xrightarrow{BQ.\mathsf{dequeue}} & \mathbf{M}(\mathbb{N} \times BQ.X) \\ {\scriptstyle \alpha}\downarrow & ? & \downarrow{\scriptstyle \mathbf{M}(\mathbb{N}\times\alpha)} \\ LQ.X & \xrightarrow[LQ.\mathsf{dequeue}]{} & \mathbf{M}(\mathbb{N} \times LQ.X) \end{array}$$

$$LQ.\mathsf{enqueue}\ n\ l := \mathbf{charge}\langle 1\rangle(\mathbf{ret}(l ++ [n]))$$

$$LQ.\mathsf{dequeue}\ [] := \mathbf{charge}\langle 0\rangle(\mathbf{ret}())(0, [])$$

$$LQ.\mathsf{dequeue}\ (n :: l) := \mathbf{charge}\langle |n :: l|\rangle(\mathbf{ret}(n, l))$$

$$\begin{array}{ccc} BQ.X & \xrightarrow{BQ.\mathsf{enqueue}\ n} & \mathbf{M}(BQ.X) \\ \alpha\downarrow & = & \downarrow\mathbf{M}(\alpha) \\ LQ.X & \xrightarrow[LQ.\mathsf{enqueue}\ n]{} & \mathbf{M}(LQ.X) \end{array} \qquad \begin{array}{ccc} BQ.X & \xrightarrow{BQ.\mathsf{dequeue}} & \mathbf{M}(\mathbb{N} \times BQ.X) \\ \alpha\downarrow & \geq & \downarrow\mathbf{M}(\mathbb{N}\times\alpha) \\ LQ.X & \xrightarrow[LQ.\mathsf{dequeue}]{} & \mathbf{M}(\mathbb{N} \times LQ.X) \end{array}$$

$$LQ.\text{enqueue}\ n\ l := \mathbf{charge}\langle 1\rangle(\mathbf{ret}(l +\!\!+ [n]))$$
$$LQ.\text{dequeue}\ [\,] := \mathbf{charge}\langle 0\rangle(\mathbf{ret}())(0, [\,])$$
$$LQ.\text{dequeue}\ (n :: l) := \mathbf{charge}\langle |n :: l| \rangle(\mathbf{ret}(n, l))$$

$$\begin{array}{ccc} BQ.X & \xrightarrow{BQ.\text{enqueue}\ n} & \mathbf{M}(BQ.X) \\ \alpha\downarrow & = & \downarrow\mathbf{M}(\alpha) \\ LQ.X & \xrightarrow[LQ.\text{enqueue}\ n]{} & \mathbf{M}(LQ.X) \end{array} \qquad \begin{array}{ccc} BQ.X & \xrightarrow{BQ.\text{dequeue}} & \mathbf{M}(\mathbb{N} \times BQ.X) \\ \alpha\downarrow & \geq & \downarrow\mathbf{M}(\mathbb{N}\times\alpha) \\ LQ.X & \xrightarrow[LQ.\text{dequeue}]{} & \mathbf{M}(\mathbb{N} \times LQ.X) \end{array}$$

Observation

This is a common pattern: often, the true cost depends on private details!

The sealing effect

$$\frac{\Gamma \vdash e : \mathbf{M}(X) \qquad \Gamma, \mathbf{abs} \vdash e_\circ : \mathbf{M}(X) \qquad \Gamma, \mathbf{abs} \vdash e \leq e_\circ}{\Gamma \vdash \mathbf{seal}(e;\ e_\circ) : \mathbf{M}(X)}$$

The sealing effect

$$\frac{\Gamma \vdash e : \mathbf{M}(X) \qquad \Gamma, \textbf{abs} \vdash e_\circ : \mathbf{M}(X) \qquad \Gamma, \textbf{abs} \vdash e \leq e_\circ}{\Gamma \vdash \textbf{seal}(e;\ e_\circ) : \mathbf{M}(X)}$$

Example

$$\begin{array}{ccc}
1 & \xrightarrow{\textbf{charge}\langle 2 \rangle(\textbf{ret}(\star))} & \mathbf{M}(1) \\
\| & \geq & \| \\
1 & \xrightarrow[\textbf{charge}\langle 3 \rangle(\textbf{ret}(\star))]{} & \mathbf{M}(1)
\end{array}$$

The sealing effect

$$\frac{\Gamma \vdash e : \mathbf{M}(X) \qquad \Gamma, \mathbf{abs} \vdash e_\circ : \mathbf{M}(X) \qquad \Gamma, \mathbf{abs} \vdash e \leq e_\circ}{\Gamma \vdash \mathbf{seal}(e;\ e_\circ) : \mathbf{M}(X)}$$

Example

$$\begin{array}{ccc} 1 & \xrightarrow{\mathbf{charge}\langle 2\rangle(\mathbf{ret}(\star))} & \mathbf{M}(1) \\ \| & \geq & \| \\ 1 & \xrightarrow[\mathbf{charge}\langle 3\rangle(\mathbf{ret}(\star))]{} & \mathbf{M}(1) \end{array}$$

$$\mathit{example} : 1 \to \mathbf{M}(1)$$

$$\mathit{example}\ () = \mathbf{seal}(\mathbf{charge}\langle 2\rangle(\mathbf{ret}(\star));\ \mathbf{charge}\langle 3\rangle(\mathbf{ret}(\star)))$$

$$\begin{array}{ccc} BQ.X & \xrightarrow{BQ.\mathsf{dequeue}} & \mathbf{M}(\mathbb{N} \times BQ.X) \\ {\scriptstyle \alpha}\big\downarrow & \geq & \big\downarrow{\scriptstyle \mathbf{M}(\mathbb{N}\times\alpha)} \\ LQ.X & \xrightarrow[LQ.\mathsf{dequeue}]{} & \mathbf{M}(\mathbb{N} \times LQ.X) \end{array}$$

$$
\begin{array}{ccc}
BQ.X & \xrightarrow{BQ.\mathsf{dequeue}} & \mathbf{M}(\mathbb{N} \times BQ.X) \\
\downarrow{\scriptstyle \alpha} & \geq & \downarrow{\scriptstyle \mathbf{M}(\mathbb{N}\times\alpha)} \\
LQ.X & \xrightarrow[LQ.\mathsf{dequeue}]{} & \mathbf{M}(\mathbb{N} \times LQ.X)
\end{array}
$$

$$BLQ.\mathsf{dequeue} : BLQ.X \to \mathbf{M}(\mathbb{N} \times BLQ.X)$$

$$BLQ.\mathsf{dequeue}\ (b_\bullet, l_\circ) \approx \mathbf{seal}(\mathsf{map}_\bullet(BQ.\mathsf{dequeue})(b_\bullet);\ \mathsf{map}_\circ(LQ.\mathsf{dequeue})(l_\circ))$$

see the paper for the real thing!

conclusion

Techniques Used

- univalence [Voevodsky]
- synthetic phase distinctions [Sterling and Harper]
- modalities [Rijke, Shulman, Spitters]
- Calf [Grodin, Niu, Sterling, Harper]

Techniques Used

- univalence [Voevodsky]
- synthetic phase distinctions [Sterling and Harper]
- modalities [Rijke, Shulman, Spitters]
- Calf [Grodin, Niu, Sterling, Harper]

Related Work

See the paper: we build on a long tradition!

Conclusion

- abstract models (like $LQ.X$) and abstraction functions (like α) can be built into types to realize verification-level properties: semantic modularity

Conclusion

- abstract models (like $LQ.X$) and abstraction functions (like α) can be built into types to realize verification-level properties: semantic modularity
- unobtrusive change: only postulate the phase proposition **abs**

Conclusion

- abstract models (like $LQ.X$) and abstraction functions (like α) can be built into types to realize verification-level properties: semantic modularity
- unobtrusive change: only postulate the phase proposition **abs**
- induces concrete $\bullet$ and abstract $\circ$ modalities and gluing, which are used to build

$$BLQ.X :\equiv \{(b_\bullet, l_\circ) : \bullet(BQ.X) \times \circ(LQ.X) \mid \mathsf{map}_\bullet(\eta_\circ \circ \alpha)(b_\bullet) = \eta_\bullet(l_\circ)\}$$

Conclusion

- abstract models (like $LQ.X$) and abstraction functions (like α) can be built into types to realize verification-level properties: semantic modularity
- unobtrusive change: only postulate the phase proposition **abs**
- induces concrete $\bullet$ and abstract $\circ$ modalities and gluing, which are used to build

$$BLQ.X := \{(b_\bullet, l_\circ) : \bullet(BQ.X) \times \circ(LQ.X) \mid \mathrm{map}_\bullet(\eta_\circ \circ \alpha)(b_\bullet) = \eta_\bullet(l_\circ)\}$$

- noninterference/modularity principles are rendered as theorems, enabling modular verification

Conclusion

- abstract models (like $LQ.X$) and abstraction functions (like α) can be built into types to realize verification-level properties: semantic modularity
- unobtrusive change: only postulate the phase proposition **abs**
- induces concrete $\bullet$ and abstract $\circ$ modalities and gluing, which are used to build

$$BLQ.X \coloneqq \{(b_\bullet, l_\circ) : \bullet(BQ.X) \times \circ(LQ.X) \mid \mathsf{map}_\bullet(\eta_\circ \circ \alpha)(b_\bullet) = \eta_\bullet(l_\circ)\}$$

- noninterference/modularity principles are rendered as theorems, enabling modular verification
- upper-bound cost specifications supported via a phased sealing effect

Bonus Slides

Semantics

Semantics (abstract)

Interpreting $[\![\textsf{abs}]\!] \coloneqq \top$, then $[\![BLQ.X]\!] = [\![LQ.X]\!]$.

Semantics (concrete)

Interpreting $[\![\textsf{abs}]\!] \coloneqq \bot$, then $[\![BLQ.X]\!] = [\![BQ.X]\!]$.

Semantics (presheaf/Kripke semantics on world poset $\{\textsf{abs} \vdash \top\}$)

Interpreting $[\![\textsf{abs}]\!] \coloneqq \left(\begin{array}{c} 0 \\ \downarrow \\ 1 \end{array}\right)$, then $[\![BLQ.X]\!] = \left(\begin{array}{c} \textsc{List}\ \mathbb{N} \times \textsc{List}\ \mathbb{N} \\ \downarrow \alpha \\ \textsc{List}\ \mathbb{N} \end{array}\right)$.

An incomplete list of related work

- Univalent representation independence [Angiuli, Cavallo, Mörtberg, and Zeuner]
- Verification of data structures [Nipkow et al.]
- Ghost code [Owicki and Gries; Filliâtre; Sterling]
- Algebraic specification [Sannella and Tarlecki] and views [Wadler]
- Existential types [Mitchell and Plotkin; Reynolds; Sterling] and Hoare logic [Hoare]

$$\begin{array}{c} \sum_{X:\mathbf{Type}} F(X) \\ \big\downarrow{\scriptstyle \mathsf{inj}} \\ \left(\sum_{X:\mathbf{Type}} F(X)\right) /\text{rep. ind.} \end{array} \qquad\qquad \begin{array}{ccc} \sum_{s:S} P(s) & \xrightarrow{\text{proof}} & \sum_{s:S} Q(s) \\ {\scriptstyle \mathsf{proj}_1}\big\downarrow & & \big\downarrow{\scriptstyle \mathsf{proj}_1} \\ S & \xrightarrow[f]{} & S \end{array}$$

$$\exists X . F(X) \qquad\qquad \{P\} f \{Q\}$$

The behavioral phase

Definition

The abstract phase [from Calf] is a proposition, **beh**.

When **beh** holds, we ignore cost:

$$e \leq e' \to e = e'$$

Corollary

Behaviorally, **charge**$\langle c \rangle(e) = e$.

Definition (queue specification type, with cost)

$$\textsc{Queue} := \{Q : \textsc{PreQueue} \mid \textbf{beh} \to (Q = LQ)\}$$

The real dequeue

$$BLQ.\text{dequeue} : BLQ.X \to \mathbf{M}(\mathbb{N} \times BLQ.X)$$

$$BLQ.\text{dequeue}\ (\eta_\bullet b, l_\circ) := \mathbf{seal}(impl;\ spec)$$

$$BLQ.\text{dequeue}\ (*\ \_, l_\circ) := spec$$

where

$$impl := \mathbf{let}\,\mathbf{ret}(n, b') = BQ.\text{dequeue}\ b\,\mathbf{in}\,\mathbf{ret}(n, (\eta_\bullet b', \eta_\circ(\alpha\ b')))$$

$$spec := \mathbf{let}\,\mathbf{ret}(n, l') = LQ.\text{dequeue}\ (l_\circ\ \_)\,\mathbf{in}\,\mathbf{ret}(n, (*\ \_, \eta_\circ l'))$$