Abstract Effects and Concurrency

Nick Benton[1], Martin Hofmann[2], and Vivek Nigam[3]

Microsoft Research, Cambridge[1], LMU, Munich[2], and UFPB, João Pessoa[3]
nick@microsoft.com[1], hofmann@ifi.lmu.de[2], vivek.nigam@gmail.com[3]

Abstract. We describe a denotational semantics for an abstract effect system for a higher-order, shared-variable concurrent programming language. We prove the soundness of a number of general effect-based program equivalences, including a parallelization equation that specifies sufficient conditions for replacing sequential composition with parallel composition. We also exploit the structure given by effect annotations and abstract locations to show the soundness of some operations on fine-grained concurrent data structures, such as Michael-Scott queues, that allow concurrent access to different parts of mutable data structures.
Our semantics is based on refining a trace-based semantics for first-order programs due to Brookes. By moving from concrete to abstract locations, and adding type refinements that capture the possible side-effects of both expressions and their concurrent environments, we are able to validate many equivalences that do not hold in an unrefined model. The meanings of types are expressed using a game-based logical relation over sets of traces. Two programs e_1 and e_2 are logically related if one is able to solve a two-player game: for any trace with result value v_1 in the semantics of e_1 (challenge) that the player presents, the opponent can present an (response) equivalent trace in the semantics e_2 with a logically related result value v_2 and vice-versa.

1 Introduction

Type-and-effect systems refine conventional types with extra static information capturing a safe upper bound on the possible side-effects of expression evaluation. Since their introduction by Gifford and Lucassen [18], effect systems have been used for many purposes, including region-based memory management [13], tracking exceptions [24,8], communication behaviour [6] and atomicity [17] for concurrent programs, and information flow [14].

A major reason for tracking effects is to justify program transformations, most obviously in optimizing compilation [11]. For example, one may remove computations whose results are unused, *provided* that they are sufficiently pure, or commute two state-manipulating computations, *provided* that the locations they may read and write are suitably disjoint. Several groups have recently studied the semantics of effect systems, with a focus on formally justifying such effect-dependent equational reasoning [19,10,9,12,27]. A common approach, which we follow here, is to interpret effect-refined types using a logical relation over the (denotational or operational) semantics of the unrefined (or untyped) language, simultaneously identifying both the subset of

computations that have a particular effect type and a coarser notion of equivalence (or approximation) on that subset. Such a semantic approach decouples the meaning of effect-refined types from particular syntactic rules: one may establish that a term has a type using various more or less approximate inference systems, or by detailed semantic reasoning.

For sequential computations with global state, denotational models already provide significant abstraction. For example, the denotations of `skip` and `X++;X--` are typically equal, so it is immediate that the second is semantically pure. More generally, the meaning of a judgement $\Gamma \vdash e : \tau \& \varepsilon$ guarantees that the result of evaluating e will be of type τ with side-effects at most ε, under assumptions Γ (a 'rely' condition), on the behaviour of e's free variables. The possible interaction points between e and its environment are restricted to initial states and parameter values, and final states and results, of e itself and its explicitly-listed free variables. Furthermore, all those interaction points are visible in the term and are governed by specific annotations appearing in the typing judgement.

For shared-variable concurrency, there are many more possible interactions. An expression's environment now also includes anything that may be running concurrently and, moreover, atomic steps of e and its concurrent environment may be arbitrarily interleaved, so it is no longer sufficient to just consider initial and final states. A priori, this leads to far fewer equations between programs. For example, `X++;X--` may be distinguished from `skip` by being run concurrently with a command that reads or writes `X`. But few programs do anything useful in the presence of unconstrained interference, so we need ways to describe and control it. Traditionally, interference is controlled via locking; an extra mechanism that requires a further programming discipline, not described or enforced by ordinary type systems, to be followed. More exciting still are fine-grained, optimistic algorithms, which rely on custom protocols being followed by multiple threads with concurrent access to a shared data structure. Such algorithms can significantly outperform ones based on coarse-grained locking, but are notoriously challenging to get right, and still harder to verify.

There is a huge literature on reasoning about shared-variable concurrency, from type systems ensuring race-freedom of programs with locks [1] to sophisticated semantic models for reasoning about refinement of fine-grained concurrent datastructures [28]. This paper explores the use of effect types as a straightforward, lightweight interface language for modular reasoning about equivalence and refinement, e.g. for safely transforming sequential composition into parallelism. We show how the semantics of a simple effect system scales smoothly to the concurrent setting, allowing us to control interference and prove non-trivial equivalences, extending (somewhat to our surprise) to the correctness of some fine-grained algorithms.

There are three main ingredients. The first is our earlier account of store effects in terms of preservation of sets of binary relations on the heap [10]. The second is a trace semantics for concurrent programs, due to Brookes [15], which explicitly describes possible interference by the environment. We extend Brookes's semantics to a higher-order language and then refine it by a relationally-formulated effect system that separately tracks the store effects of an expression during evaluation, the effects of transitions by the environment, and the end-to-end effect. The third ingredient is the notion of abstract

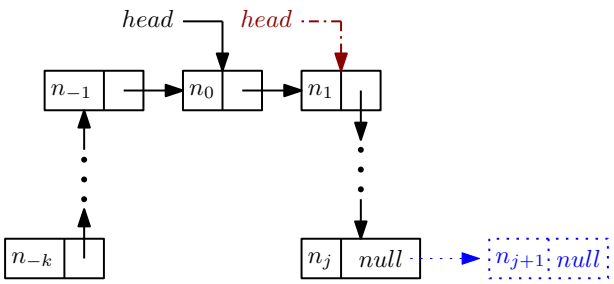

Fig. 1. Illustration of a Michael-Scott Queue. The list resulting from the pointer to the element n_0 (the *head* pointer with the continuous arrow in black) contains the list of elements $[n_1, \ldots, n_j]$. The enqueueing operation is illustrated by the dotted arrow and the box with the element n_{j+1} (in blue), while the dequeueing operation is illustrated by the dot dashed head pointer (in red).

```
dequeue () = let rec try () =
               let n0 =!head in if !n0.1 = null then null else
                 let n1 =!n0.2 in if cas(!head, n0, n1) then !n1.1 else try ()
             in try ()

enqueue(x) = let c = ref(x, null) in
               let rec try (p) = if !p.2 = null then cas(!p.2, null, c) else try (!p.2)
             in try (!head)
```

Fig. 2. Enqueue and Dequeue programs for a Michael-Scott Queue at location *head*.

location [9]. Rather than tracking effects at the level of individual concrete heap cells, we view the heap as a set of abstract data structures, each of which may span several locations, or parts of locations. Each abstract location has its own notion of equality, and its own notion of legal mutation. Write effects, for example, need only be flagged when the equivalence class of an abstract location may change.

Equivalence modulo non-interference: Our semantics will justify the typed equation

$$\vdash (X := !X + 1; X := !X + 1) = (X := !X + 2) : T(\texttt{unit}, \{co_X\}, \varepsilon, \varepsilon \cup \{rd_X, wr_X\})$$

which says that the two commands are equivalent with return type `unit`, an effect along the way (co_X) of accessing X and an overall effect of ε plus reading and writing X, *provided* that the effect, ε, of the concurrent environment does not involve X.

Overlapping References: Let p, p^{-1} implement a bijection $\mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$, and consider the following functions:

```
readFst () = p(!X).1
readSnd () = p(!X).2
wrtFst n = let rec try () = let m =!X in let (x, y)= p(m) in let m′ = p^{-1}(n, y) in
             if cas(X, m, m′) then () else try ()
           in try ()
wrtSnd n = let rec try () = let m =!X in let (x, y)= p(m) in let m′ = p^{-1}(x, n) in
             if cas(X, m, m′) then () else try ()
           in try ()
```

which multiplexes two abstract integer references onto a single concrete one. The write functions, wrtFst and wrtSnd, use compare-and-swap, `cas`, to atomically update the value of the reference. Atomicity is needed to prevent updates to one abstract location overwriting updates to the other by different threads. We can show that a program, t_1, that only reads and/or writes into one abstract integer reference can safely be executed in parallel with another program, t_2, that only reads and/or writes into the other integer reference, although t_1 and t_2 read and write into the same concrete location, which looks like a race. The behavior of $t_1 \| t_2$ is the same as $t_1; t_2$ or $t_2; t_1$.

Michael-Scott Queue: The Michael-Scott Queue [22] (MSQ) is a fine grained concurrent data structure, allowing threads to access and modify different parts of a queue safely and simultaneously. We present a version like that of Turon et al [28], which is an idealized version of the MSQ, without a tail pointer.

An MSQ maintains a pointer *head* to a non-empty linked list as depicted in Figure 1. The first node, the node containing the element n_0 in the figure, is not an element of the queue, but is a "sentinel". Hence the queue in the figure holds $[n_1, \ldots, n_j]$.

The enqueue and dequeue operations are defined in Figure 2 and illustrated in Figure 1. Elements are dequeued from the beginning of the linked list, and enqueued at the end, which involves a traversal that is done without locking. Once the end, p, of the linked list is found, the program attempts to insert the new element by using compare-and-swap. This is necessary because other programs may have enqueued elements to the end of the list, meaning that p is no longer the end of the list.

The dequeue operation should move the *head* pointer from the current sentinel, n_0, to the following element n_1. However, as other programs may also be attempting to dequeue an element, we use compare-and-swap to atomically update the *head* pointer if *head* still points to the same sentinel. Notice that the dequeued elements can still reach the sentinel of the queue. (In Figure 1, these are the nodes containing $n_{-k}, \ldots, n_{-1}$.) This is necessary because there might be other (slower) threads that want to enqueue an element and are still searching for the end of the list by traversing the portion of the queue that has already been dequeued.

We show that the fine-grained enqueue and dequeue of Figure 2 are equivalent to their atomic versions, `atomic`(enqueue) and `atomic`(dequeue), where all operations are performed in a single step.[1] This means that the MSQ fine-grained concurrent data structure behaves as a synchronized queue, implemented for instance using locks.

2 Syntax

In this section we define the syntax of our untyped metalanguage.

Syntax The syntax of untyped values and computations is:

$$
\begin{array}{rl}
v & ::= x \mid () \mid c \mid (v_1, v_2) \mid v.1 \mid v.2 \mid \texttt{rec}\ f\ x = t \\
t & ::= v \mid \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \mid v_1\ v_2 \mid \texttt{if}\ v\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \\
& \quad \mid !v \mid v_1 := v_2 \mid \texttt{ref}(v) \mid t_1 \| t_2 \mid \texttt{await}\ t_1\ \texttt{then}\ t_2
\end{array}
$$

[1] These atomic versions are equivalent to `atomic`(·)-wrapped sequential implementations, without the `cas`(,, s).

Here, x ranges over variables and constants c over constant symbols, each of which has an associated semantic interpretation $[\![c]\!] \in \mathbb{V}$ (see Section 3 below for the definition of $\mathbb{V}$); these include numerals, booleans, arithmetic operations, test functions to tell whether a value is an integer, a function, a pair, or a reference, equality test for simple values, etc. $\texttt{rec}\, f\, x = t$ defines a recursive function with body t and recursive calls made via f; we use $\lambda x.t$ as syntactic sugar in the case when $f \notin fv(t)$. Next, $!v$ (reading) returns the contents of location v, $v_1 := v_2$ (writing) updates location v_1 with value v_2, and $\texttt{ref}(v)$ (allocating) returns a fresh location initialized with v. The metatheory is simplified by using "let-normal form", in which the only elimination for computations is let, though we sometimes nest computations as shorthand for let-expanded versions in examples.

$t_1 \| t_2$ is evaluated by running t_1 and t_2 in parallel until they produce values v_1 and v_2, the result is then (v_1, v_2). Parallel evaluation is carried out by arbitrary interleaving with the understanding that assignments to, lookups from, and allocations of locations are atomic; the evaluation of nested expressions is, however, in general not atomic. To introduce further atomicity the $\texttt{await}\ t_1\ \texttt{then}\ t_2$ construct [15], which repeatedly, and atomically, evaluates t_1 until it returns $\texttt{true}$ at which point t_2 is evaluated atomically, without allowing any intermediate intervention of the environment. Typically, t_1 is a simple check for the current value of a reference.[2]

The await construct allows other concurrency primitives to be expressed. We define $\texttt{atomic}(t)$ as $\texttt{await}\ \mathit{true}\ \texttt{then}\ t$, which ensures that the expression t is executed in one step. Moreover, the compare and swap operation, $\texttt{cas}$, can be defined using $\texttt{atomic}$ as follows:

$$\texttt{cas}(X, v, t) = \texttt{atomic}\Big(\texttt{if}\ !X = v\ \texttt{then}\ X := t; \texttt{true}\ \texttt{else}\ \texttt{false}\Big)$$

We omit the standard operational semantics of this language, which can be obtained as a straightforward generalisation of the operational semantics given by Brookes [15]. Instead, we immediately move on to a denotational semantics which extends Brookes' trace semantics [15] with general recursion and higher-order functions.

3 Denotational Model

A *predomain* is an ω-cpo, *i.e.*, a partial order with suprema of ascending chains. A *domain* is a predomain with a least element, $\bot$. Recall that $f : A \to A'$ is *continuous* if it is monotone $x \leq y \Rightarrow f(x) \leq f(y)$ and preserves suprema of chains, *i.e.*, $f(\sup_i x_i) = \sup_i f(x_i)$. Any set is a predomain with the discrete order (flat predomain). If X is a set and A a predomain then any $f : X \to A$ is continuous. We denote a partial (continuous) function from set (predomain) A to set (predomain) B by $f : A \rightharpoonup B$. If A, B are predomains the cartesian product $A \times B$ and the set of continuous functions $A \to B$ form themselves predomains (with the obvious componentwise and pointwise orders) and make the category of predomains cartesian closed. Likewise, the partial continuous functions $A \rightharpoonup B$ between predomains A, B form a domain.

[2] Having atomic access to the whole store is rather a strong primitive, but we are not discussing full abstraction here. Rather we show particular equivalences, which are all the more robust if they hold in the presence of slightly unrealistically powerful contexts.

If $P \subseteq A$ and $Q \subseteq B$ are subsets of predomains A and B we define $P \times Q \subseteq A \times B$ and $P \to Q \subseteq A \to B$ in the usual way. We may write $f : P \to Q$ for $f \in P \to Q$.

A subset $U \subseteq A$ is *admissible* if whenever $(a_i)_i$ is an ascending chain in A such that $a_i \in U$ for all i, then $\sup_i a_i \in U$, too. If $f : X \times A \to A$ is continuous and A is a domain then one defines $f^\ddagger(x) = \sup_i f_x^i(\bot)$ with $f_x(a) = f(x, a)$. One has, $f(x, f^\ddagger(x)) = f^\ddagger(x)$ and if $U \subseteq A$ is admissible and contains $\bot$ and $f : X \times U \to U$ then $f^\ddagger : X \to U$, too. An element d of a predomain A is *compact* if whenever $d \leq \sup_i a_i$ then $d \leq a_i$ for some i. E.g. in the domain of partial functions from $\mathbb{N}$ to $\mathbb{N}$ the compact elements are precisely the finite ones. A continuous function $f : A \to A$ is a deflation (also known as retract) if $f(a) \leq a$ and $f(f(a)) = f(a)$ holds for all $a \in A$. In short: $f \leq id_A$ and $f; f \leq f$.

Heaps We assume a countable set $\mathbb{L}$ of physical locations $X_1, \ldots, X_n, \ldots$ and a set $\mathbb{V}_b$ of "R-values" that can be stored in those references including integers, written $int(n)$ for some $n \in \mathbb{Z}$, booleans, written $bool(b)$ for $b \in \mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$ and tuples of R-values, written $(v_1, \ldots, v_n)$. We assume that it is possible to tell whether a value is of that form and in this case to retrieve the components. A heap h, then, is a *finite map* from $\mathbb{L}$ to $\mathbb{V}_b$, often written as $[[X_1, \mathsf{c}_1], [X_2, \mathsf{c}_2], \ldots, [X_n, \mathsf{c}_n]]$, specifying that the value stored in physical location X_i is c_i. We write $\mathrm{dom}(\mathsf{h})$ for the domain of h. Finally, we write $\mathsf{h}[X \mapsto \mathsf{c}]$ for the heap that agrees with h except that it gives the variable X the value c. The set of heaps is denoted by $\mathbb{H}$. We also assume that $new(\mathsf{h}, v)$ yields a pair (X, h') where $X \in \mathbb{L}$ is a fresh location and $\mathsf{h}' \in \mathbb{H}$ is $\mathsf{h}[X \mapsto v]$.

If A is a predomain we define the state monad as usual by $SA = \mathbb{H} \to \mathbb{H} \times A$. It is well known that this defines a strong monad on the category of predomains.

3.1 Traces

We use traces to model terminating runs of concurrent computation. A trace records such a run as a sequence of pairs of heaps each representing pre- and post-state of a single atomic action. The semantics of a program then is a (typically very large) set of traces which provides for all possible environment interactions. It can be likened to a graph of a function which also contains argument-value pairs for each possible argument.

Definition 1 (Traces). *A trace is a finite sequence of pairs* (h, k) *where* $\mathsf{h}, \mathsf{k} \in \mathbb{H}$. *We write Tr for the set of traces.*

Let t be a trace. A trace of the form $u\,(\mathsf{h}, \mathsf{h})\,v$ where $t = uv$ is said to arise from t by stuttering. A trace of the form $u(\mathsf{h}, \mathsf{k})v$ where $t = u(\mathsf{h}, \mathsf{q})(\mathsf{q}, \mathsf{k})v$ is said to arise from t by mumbling. For example, if $t = (\mathsf{h}_1, \mathsf{k}_1)(\mathsf{h}_2, \mathsf{k}_2)(\mathsf{h}_3, \mathsf{k}_3)$ then $(\mathsf{h}_1, \mathsf{k}_1)(\mathsf{h}, \mathsf{h})(\mathsf{h}_2, \mathsf{k}_2)(\mathsf{h}_3, \mathsf{k}_3)$ arises from t by stuttering. In the special case where $\mathsf{k}_1 = \mathsf{h}_2$ the trace $(\mathsf{h}_1, \mathsf{k}_2)(\mathsf{h}_3, \mathsf{k}_3)$ arises by mumbling. A set of traces U is closed under stuttering and mumbling if whenever t' arises from t by stuttering or mumbling and $t \in U$ then $t' \in U$, too.

We recall that Brookes [15] interprets while-programs with parallel composition as sets of traces closed under stuttering and mumbling and shows that this semantics is adequate and fully abstract for may-equivalence when all variables are global and of integer type. The following definition extends this semantics with result values drawn from an

arbitrary predomain and thus permits an extension of Brookes's model to higher-order functions and general recursion. As one might expect, the extension to higher-order is no longer fully abstract, but remains adequate.

Definition 2 (Trace Monad). *Let A be a predomain. A domain TA is defined as follows. The elements of TA are sets U of pairs (t, a) where t is a trace and $a \in A$ such that the following properties are satisfied:*

- *[S&M]: if t' arises from t by stuttering or mumbling and $(t, a) \in U$ then $(t', a) \in U$.*
- *[Down]: if $(t, a_1) \in U$ and $a_2 \leq a_1$ then $(t, a_2) \in U$.*
- *[Sup]: if $(a_i)_i$ is a chain in A and $(t, a_i) \in U$ for all i then $(t, \sup_i a_i) \in U$.*

The elements of TA are partially ordered by inclusion.

Lemma 1. *If A is a predomain then TA is a domain.*

Proof. The supremum of a chain $(U_i)_i$ in TA is the closure under [Sup] of the union $\bigcup_i U_i$. It contains all pairs (t, a) such that there exists i_0 and a chain $(a_i)_i$ with supremum a such that $(t, a_i) \in U_{i_0+i}$.

An element U of TA represents the possible outcomes of a nondeterministic, interactive computation with final result in A. Thus, if $(t, a) \in U$ for $t = (\mathsf{h}_1, \mathsf{k}_1) \ldots (\mathsf{h}_n, \mathsf{k}_n)$ then this means that there could be n interactions with the environment with heaps $\mathsf{h}_1, \ldots, \mathsf{h}_n$ being "played" by the environment and "answered" with heaps $\mathsf{k}_1, \ldots, \mathsf{k}_n$ by the computation. After that, this particular computation ends and a is the final result value.

For example, the semantics of $X :=!X + 1; X :=!X + 1; !X$ will contain many traces including the following, where we write $[n]$ for the heap in which X has value n:

$(([10], [12]), 12)$,
$(([10], [11])([15], [16]), 16)$,
$(([10], [11])([17], [17])([15], [16]), 16)$

Axiom [S&M] is taken from Brookes. It ensures that the semantics does not distinguish between late and early choice [28] and related phenomena which are reflected, e.g., in resumption semantics [25], but do not affect observational equivalence. Notice that a non-terminating computation is represented by the empty set and thus is invisible if it *may* happen, but does not necessarily do so ("may semantics" [23]). For example, the semantics of a program like $X := 0; \texttt{if } X{=}0 \texttt{ then } 0 \texttt{ else diverge}$ will be the same as that of $X := 0; 0$ and contain, for example $(([10], [0]), 0)$ but also (stuttering) $((([10], [0]), ([34], [34])), 0)$. Note that it is not possible to tell from a trace whether an external update of X has happened before or after the reading of X.

Let us also illustrate how traces iron out some intensional differences that show up when concurrency is modelled using transition systems or resumptions. Consider the following two programs where ? denotes a nondeterministically chosen boolean value.

$$t_1 \equiv \texttt{if ? then } X := 0; \texttt{true else } X := 0; \texttt{false} \qquad \text{and} \qquad t_2 \equiv X := 0;\ \texttt{?}$$

Both t_1 and t_2 admit the same traces, namely $(([x], [0]), \texttt{true})$ and $(([x], [0]), \texttt{false})$ and stuttering variants thereof.

In a semantic model based on transition systems or resumptions and bisimulation these would be distinguished and special mechanisms such as history and prophecy

variables [2], forward-backward simulation [21], or speculation [28] must be installed to recover useful reasoning principles.[3]

Axioms [Down] and [Sup] are known from the Hoare powerdomain [25] for angelic nondeterminism. Recall that the Hoare powerdomain PA contains the subsets of A which are downclosed ([Down]) and closed under suprema of chains ([Sup]). Such subsets are also known as Scott-closed sets. Thus, TA is the restriction of $P(Tr \times A)$ to the sets closed under stuttering and mumbling. Axiom [Down] ensures that the ordering is indeed a partial order and not merely a preorder. It is also closely related to "may semantics". Additional nondeterministic outcomes that are less defined than existing ones are not recorded in the semantics. Axiom [Sup] ensures that the embedding of values as singletons is continuous.

Definition 3. *If $U \subseteq Tr \times A$ then we denote $U^\dagger$ the least subset of TA containing U, i.e. $U^\dagger$ is the closure of under mumbling and stuttering [S& M], [Down], [Sup].*

Definition 4. *Let A, B be predomains. We define the following continuous functions*

$$\eta : A \to TA \qquad \eta(a) := (\{((\mathsf{h},\mathsf{h}),a) \mid \mathsf{h} \in \mathbb{H}\})^\dagger \in TA$$
$$ap : (A \to TB) \times TA \to TB \qquad ap(f,g) := (\{(uv,b) \mid (u,a) \in g \wedge (v,b) \in f(a)\})^\dagger$$

Proposition 1. *The functions η and ap endow TA with the structure of a strong monad.*

A partial function $c : \mathbb{H} \rightharpoonup \mathbb{H} \times A$ (an element of the state monad SA) can be (continuously) transformed into an element $\mathit{fromstate}(c)$ by

$$\mathit{fromstate} : SA \to TA \qquad \mathit{fromstate}(c) := \{((\mathsf{h},\mathsf{k}),a) \mid c(\mathsf{h}) = (\mathsf{k},a)\}^\dagger$$

If t_1, t_2, t_3 are traces, we write $\mathit{inter}(t_1,t_2,t_3)$ to mean that t_3 can be obtained by interleaving t_1 and t_2 in some way, i.e., t_3 is contained in the shuffle of t_1 and t_2. In order to model parallel composition we introduce the following helper function

$$\| \;:\; TA \times TB \to T(A \times B) \quad U \parallel V := \{(t_3,(a,b)) \mid \mathit{inter}(t_1,t_2,t_3),(t_1,a) \in U,(t_2,b) \in V\}^\dagger$$

If $b \in T\mathbb{B}$ and $c \in TA$ we define a computation $\mathit{await}(b,c) \in TA$ by

$$\mathit{await} : T\mathbb{B} \times TA \to TA \quad \mathit{await}(U,V) := \{((\mathsf{h},\mathsf{k}),v) \mid ((\mathsf{h},\mathsf{h}_1),\mathtt{true}) \in U,((\mathsf{h},\mathsf{k}),v) \in V\}^\dagger$$

This function will serve as the interpretation of the await construct. For it to be implementable with reasonable effort, one will in practice restrict U to correspond to a boolean expression on heaps whose execution is atomic and does not modify the heap in any way. Semantically, however, the above definition makes perfect sense and allows us to avoid the introduction of a separate semantic category of values. Notice that due to mumbling $((\mathsf{h},\mathsf{k}),v) \in V$ iff there exists an element $((\mathsf{h}_1,\mathsf{h}_2)(\mathsf{h}_2,\mathsf{h}_3)\ldots(\mathsf{h}_{n-2},\mathsf{h}_{n-1})(\mathsf{h}_{n-1},\mathsf{h}_n),v) \in V$ where $\mathsf{h} = \mathsf{h}_1$ and $\mathsf{h}_n = \mathsf{k}$. The presence of such an element, however, models an atomic execution of the computation represented by V.

[3] We do not have an argument that trace semantics forever removes the need for such mechanisms, merely that particular examples that require them in other models can be done without in the trace model.

3.2 Semantic values

The predomain of values $\mathbb{V}$ comprises R-values, tuples of values, and continuous functions from values to elements of $T\mathbb{V}$, i.e. the predomain $\mathbb{V}$ is defined as the least solution of the following domain equation. $\mathbb{V} \simeq \mathbb{V}_b + (\mathbb{V} \to T\mathbb{V}) + \mathbb{V}^*$. We tend to identify the summands of the right hand side with subsets of $\mathbb{V}$ but may use tags like $fun(f) \in \mathbb{V}$ when $f : \mathbb{V} \to T\mathbb{V}$ to avoid ambiguities. We will refer to the elements of $T\mathbb{V}$ as *computations*.

We fix an increasing sequence of *finite* sets $(V_i)_i$ such that $\mathbb{V}_b = \bigcup_i V_i$ and then obtain the following families of continuous functions $p_i : \mathbb{V} \to \mathbb{V}$ and $q_i : T\mathbb{V} \to T\mathbb{V}$:

$$p_i(v) = v \text{ if } v \in V_i \quad p_i(v) \text{ undef. otherwise.} \quad q_0(U) = \emptyset \quad p_i(v_1, \ldots, v_n) = (p_i(v_1), \ldots, p_i(v_n))$$
$$q_{i+1}(U) = \{(t, p_i(v)) \mid (t, v) \in U\} \quad p_i(g) = fun(q_i; g; p_i) \text{ if } g : \mathbb{V} \to T\mathbb{V}$$

The p_i and q_i each form an increasing chain of deflations with *finite image* and it follows from the standard solution theory of recursive domain equations [26,5,3] that $\sup_i p_i = id_{\mathbb{V}}$ and $\sup_i q_i = id_{T\mathbb{V}}$. Moreover, the elements of the form $p_i(a)$ and $q_i(U)$ are compact and thus every $v \in \mathbb{V}$ can be written as the supremum of a chain of compact elements, namely $p_i(v)$. The same goes for computations. Thus, $\mathbb{V}$ and $T\mathbb{V}$ are in fact *Scott* (pre)domains [3].

Definition 5. *Let P be a subset of a predomain A. We define $Adm(P)$ as the least admissible superset of P. Concretely, $a \in Adm(P)$ iff there exists a chain $(a_i)_i$ such that $a_i \in P$ for all i and $a = \sup_i a_i$.*

Lemma 2. *Let A, B be predomains and $P \subset A$, $Q \subset B$. We have $Adm(P) \times Adm(Q) = Adm(P \times Q)$.*

Proof. The $\supseteq$ direction is obvious. For $\subseteq$ suppose that $a \in Adm(P)$ and $b \in Adm(Q)$ so that $a = \sup_i a_i$ and $b = \sup_i b_i$ with $a_i \in P$ and $b_i \in Q$. We have $(a_i, b_i) \in P \times Q$ so $(a, b) \in Adm(P \times Q)$.

Corollary 1. *If $f : A_1 \times \cdots \times A_n$ is continuous; $P_i \subseteq A_i$ are arbitrary subsets and $Q \subseteq B$ is admissible then $f : P_1 \times \cdots \times P_n \to Q$ implies $f : Adm(P_1) \times \cdots \times Adm(P_n) \to Q$.*

Lemma 3. *Let A, B be predomains and let $(p_i)_i$ be a chain of deflations on B such that $p_i(b)$ is compact for each i and $\sup_i p_i = id$ and $b \in Q$ implies $p_i(b) \in Q$ for all i.*

Then $P \to Adm(Q) = Adm(P \to Q)$.

Proof. The $\supseteq$ direction is again obvious. For $\subseteq$ suppose that $f \in P \to Adm(Q)$ and chose for each $a \in A$ a chain $(b_{i,a})_i$. such that $a \in P$ implies $(b_{i,a})_i \in Q$ and $\sup_i b_{i,a} = f(a)$.

We now claim that for each j and $a \in P$ we have $p_j(f(a)) \in Q$. Indeed, the chain $(p_j(b_{i,a}))_j$ converges against $p_j(f(a))$, but since $p_j(f(a))$ is compact there must exist j such that $p_j(b_{i,a}) = p_j(f(a))$. Thus, $p_j(f(a)) \in Q$. It follows that the functions $f; p_i$ whose supremum is f are in $P \to Q$ and so $f \in Adm(P \to Q)$ as required.

An extract of the semantics of values $[\![v]\!] \in \mathbb{V} \to \mathbb{V}$ and terms $[\![t]\!] \in \mathbb{V} \to T\mathbb{V}$ are given by the recursive clauses in Figure 3. The notation $\eta(x)$ stands for the i-th projection from $\eta \in \mathbb{V}$ if x is x_i and $\eta[x \mapsto v]$ (functionally) updates the i-th slot in η when $x = x_i$.

$$
\begin{aligned}
[\![\texttt{let}\ x{=}e_1\ \texttt{in}\ e_2]\!]\eta\ \mathsf{h} &= \{(t_1 t_2, v) \mid (t_1, u) \in \lceil\!\lceil e_1 \rceil\!\rceil\eta, (t_2, v) \in \lceil\!\lceil e_2 \rceil\!\rceil\eta[x{\mapsto}u]\}^{\ddagger} \\
[\![!v]\!]\eta &= \mathit{fromstate}(\lambda\mathsf{h}.(\mathsf{h}, \mathsf{h}(X))), \text{ when } \lceil\!\lceil v \rceil\!\rceil\eta = \mathit{loc}(X) \\
[\![v_1 := v_2]\!]\eta\ \mathsf{h} &= \mathit{fromstate}((\mathsf{h}[X{\mapsto}\lceil\!\lceil v_2 \rceil\!\rceil\eta], \mathit{int}(0))), \text{ if } \lceil\!\lceil v_1 \rceil\!\rceil\eta = \mathit{loc}(X) \\
[\![\texttt{ref}(v)]\!]\eta\ \mathsf{h} &= \mathit{fromstate}(\lambda\mathsf{h}.\mathit{new}(\mathsf{h}, \lceil\!\lceil v \rceil\!\rceil\eta)) \\
[\![\texttt{await}\ t_1\ \texttt{then}\ t_2]\!]\eta &= \mathit{await}([\![t_1]\!]\eta, [\![t_2]\!]\eta) \\
[\![t_1 \parallel t_2]\!]\eta &= [\![t_1]\!]\eta \parallel [\![t_2]\!]\eta \\
\lceil\!\lceil v \rceil\!\rceil\eta &= 0, \text{otherwise} \\
[\![t]\!]\eta\ \mathsf{h} &= \emptyset, \text{otherwise}
\end{aligned}
$$

Fig. 3. Semantics of untyped meta language (extract)

Initial heap The underlying semantics of the untyped language supports dynamic allocation of concrete references. However, for simplicity, we do not currently treat dynamic allocation of *abstract* locations in our logical relation, and therefore assume we are given an initial heap $\mathsf{h}_{\mathit{init}}$ in which appropriate datastructures for abstract locations are already allocated. For example, in the case of the MSQ we require a queue to be already set up in the initial heap. During the computation, such a structure it may evolve and grow via allocation of concrete locations, but we cannot install a new queue during the computation. (We can, of course, introduce a constant that does exactly that, but the framework given here would be unable to reason about it.) This restriction could be lifted by using a Kripke logical relation, as in our earlier work [9], but we refrain from introducing those extra complexities here.

4 Abstract Locations

We build on the concept of abstract locations defined by Benton, Hofmann, and Nigam [9]. These allow more complicated data structures which use several concrete locations or only parts of them to be a regarded as a single "location" that can be written to and read from. Essentially, an abstract location is given by a partial equivalence relation on heaps modelling well-formedness and equality together with a transitive relation modelling allowed modifications of the abstract location. Abstract locations then allow certain commands that modify the physical heap to be treated as read-only or even pure if they respect the contracts. Abstract location are related to *islands* [4] which also allow one to specify heap allocated data structures and use transition systems for that purpose. An important difference is that abstract locations do not require physical footprints in the form of sets of concrete locations.

Due to the absence of dynamic allocation at the level of abstract locations we can slightly simplify the original definition from [9] here.

Definition 6 (Abstract Location). *An* abstract location $\mathfrak{l}$ *consists of the following data:*

- *a partial equivalence relation* $\overset{\mathfrak{l}}{\sim}$ *on* $\mathbb{H}$ *modeling the "semantic equality" on the bits of the store that* $\mathfrak{l}$ *uses. We refer to* $\overset{\mathfrak{l}}{\sim}$ *as the* rely relation *of* $\mathfrak{l}$.
- *a transitive relation* $\overset{\mathfrak{l}}{\to}$ *modeling how exactly the heap my change upon writing the abstract location and in particular what bits of the store such writes leave intact. In other words, if* $\mathsf{h} \overset{\mathfrak{l}}{\to} \mathsf{h}_1$ *then* h_1 *might arise by writing to* $\mathfrak{l}$ *in* h *and all possible writes are specified by* $\overset{\mathfrak{l}}{\to}$*. We refer to* $\overset{\mathfrak{l}}{\to}$ *as the* guarantee relation *of* $\mathfrak{l}$.

subject to the following conditions where $\mathsf{h} : \mathfrak{I}$ *stands for* $\mathsf{h} \overset{\mathfrak{I}}{\sim} \mathsf{h}$.

1. *if* $\mathsf{h} : \mathfrak{I}$ *then* $\mathsf{h} \overset{\mathfrak{I}}{\to} \mathsf{h}$;
2. *if* $\mathsf{h} \overset{\mathfrak{I}}{\to} \mathsf{h}_1$ *then* $\mathsf{h} : \mathfrak{I}$ *and* $\mathsf{h}_1 : \mathfrak{I}$;
3. $\overset{\mathfrak{I}}{\sim};\overset{\mathfrak{I}}{\to}\subseteq\overset{\mathfrak{I}}{\to};\overset{\mathfrak{I}}{\sim}$, *that is, if* $\mathsf{h} \overset{\mathfrak{I}}{\sim} \mathsf{h}'$ *and* $\mathsf{h} \overset{\mathfrak{I}}{\to} \mathsf{h}_1$, *then there exists* h'_1 *such that* $\mathsf{h}' \overset{\mathfrak{I}}{\to} \mathsf{h}'_1$ *and* $\mathsf{h}_1 \overset{\mathfrak{I}}{\sim} \mathsf{h}'_1$.

We now introduce some examples of abstract locations. The first two, for single integers and pairs of integers, are basically the same as in [9]. Later, we will introduce a third example of abstract locations for Michael-Scott queues.

Single Integer For our simplest example, consider the following abstract location parametric with respect to concrete location X as follows:

$$\mathsf{h} \overset{\mathfrak{int}(X)}{\sim} \mathsf{h}' \iff \exists n.\mathsf{h}(X) = \mathit{int}(n) \wedge \mathsf{h}'(X) = \mathit{int}(n)$$
$$\mathsf{h} \xrightarrow{\mathfrak{int}(X)} \mathsf{h}_1 \iff \mathsf{h} : \mathfrak{int}^R(X), \mathsf{h}_1 : \mathfrak{int}^R(X) \text{ and } \forall X' \in \mathbb{L}.X' \neq X \Rightarrow \mathsf{h}(X') = \mathsf{h}_1(X)$$

Two heaps are in the rely relation if the values stored in X are integers and equal; the guarantee requires all other concrete locations to be unchanged.

We sometimes abuse notation in effect annotations by referring to abstract locations holding single integers by their corresponding concrete locations, writing rd_X, wr_X, co_X for $rd_{\mathfrak{int}(X)}, wr_{\mathfrak{int}(X)}, co_{\mathfrak{int}(X)}$.

Overlapping references Recall the overlapping references example introduced earlier. Let X be the concrete location encoding a pair of values. We define abstract locations $\mathfrak{fst}$ and $\mathfrak{snd}$ by:

$$\mathsf{h} \overset{\mathfrak{fst}(X)}{\sim} \mathsf{h}' \iff \exists a_1, a_2.\mathsf{h}(X) = (a_1, a_2) \wedge \mathsf{h}'(X) = (a'_1, a'_2) \wedge a_1 = a'_1$$
$$\mathsf{h} \overset{\mathfrak{snd}(X)}{\sim} \mathsf{h}' \iff \exists a_1, a_2.\mathsf{h}(X) = (a_1, a_2) \wedge \mathsf{h}'(X) = (a'_1, a'_2) \wedge a_2 = a'_2$$
$$\mathsf{h} \xrightarrow{\mathfrak{fst}(X)} \mathsf{h}_1 \iff \mathsf{h} : \mathfrak{fst}(X), \mathsf{h}_1 : \mathfrak{fst}(X) \text{ and } (\forall X' \neq X.\mathsf{h}(X) = \mathsf{h}_1(X')) \wedge (\forall a_1, a_2.\mathsf{h}(X) = (a_1, a_2) \wedge \mathsf{h}_1(X) = (a'_1, a'_2) \Rightarrow a_2 = a'_2)$$
$$\mathsf{h} \xrightarrow{\mathfrak{snd}(X)} \mathsf{h}_1 \iff \mathsf{h} : \mathfrak{snd}(X), \mathsf{h}_1 : \mathfrak{snd}(X) \text{ and } (\forall X' \neq X.\mathsf{h}(X) = \mathsf{h}_1(X')) \wedge (\forall a_1, a_2.\mathsf{h}(X) = (a_1, a_2) \wedge \mathsf{h}_1(X) = (a'_1, a'_2) \Rightarrow a_1 = a'_1)$$

The rely of $\mathfrak{fst}(X)$ (respectively, $\mathfrak{snd}(X)$) specifies that two heaps h and h' are equivalent whenever they both store a pair of values in X and the first projections (respectively, second projection) of these pairs are the same. The guarantee of $\mathfrak{fst}(X)$ (respectively, $\mathfrak{snd}(X)$) specifies that it keeps all other locations alone and does not change the second projection (respectively, first projection) of the pair stored at location X.

4.1 Refinement of abstract locations

In the concurrent setting we will often need two abstract locations providing two different views on the same data structure; an internal one where equality is finer, yet contracts are weaker, and more transitions are possible.

Consider, for instance a set data structure that caches the size in an extra integer field that must be kept consistent. We model this as an abstract location whose $\sim$-relation insists that the size field is correctly set and that the data structure itself is of the right shape. It will equate different instances of the data structure as long as they are rooted at the same entry point and have the same content *qua* set. An operation that replaces an instance by an equivalent one may be counted as pure or at least read only.

Assuming that the updates to this data structure are not implemented atomically, i.e. allow interaction by the environment, then clearly, we can neither allow the environment to silently modify the data structure under our hands, nor can we guarantee to the environment that the data structure will be in consistent state at all times.

Thus, we use another abstract location whose $\sim$-part only imposes those invariants that are needed to be maintained even during a computation and only equates heaps that are truly indistinguishable even by a concurrent operation in the middle of its operation. Typically, this will mean that the set data structures will have to be identical even in their physical layout and differences are tolerated only in those parts of the heap that are not affected at all by the operations in question.

Generalising this example leads to a notion of refinement between locations.

Definition 7 (Refinement of locations). *We say that abstract location* $\mathfrak{l}$ *refines abstract location* $\mathfrak{k}$*, written* $\mathfrak{l} \leq \mathfrak{k}$ *if the following hold.*

- $\mathsf{h} : \mathfrak{k}$ *implies* $\mathsf{h} : \mathfrak{l}$
- $\mathsf{h} \xrightarrow{\mathfrak{k}} \mathsf{h}'$ *implies* $\mathsf{h} \xrightarrow{\mathfrak{l}} \mathsf{h}'$
- $\mathsf{h} : \mathfrak{k}$, $\mathsf{h}' : \mathfrak{k}$, $\mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}'$ *implies* $\mathsf{h} \overset{\mathfrak{k}}{\sim} \mathsf{h}'$
- $\mathsf{h} \xrightarrow{\mathfrak{l}} \mathsf{h}'$, $\mathsf{h} \overset{\mathfrak{k}}{\sim} \mathsf{h}'$ *implies* $\mathsf{h} \xrightarrow{\mathfrak{k}} \mathsf{h}'$

4.2 Worlds

We will group the abstract locations used to describe a program into a *world*. In this paper we do not model dynamic evolution of worlds; all abstract locations ever used must be set up upfront. While dynamic allocation may happen to increase a data structure modelled by an abstract location, e.g. in the Michael-Scott Queue example, no new such datastructures can appear. It is possible, however, to extend our work in this direction by using (proof-relevant) Kriple logical relations [9,4].

Definition 8 (world). *A* world *is a set of abstract locations.*

The relation $\mathsf{h} \models \mathsf{w}$ *(heap* h *satisfies world* w*) is defined as the largest relation such that* $\mathsf{h} \models \mathsf{w}$ *implies*

- $\mathsf{h} : \mathfrak{l}$ *for all* $\mathfrak{l} \in \mathsf{w}$*;*
- *if* $\mathfrak{l} \in \mathsf{w}$ *and* $\mathsf{h} \xrightarrow{\mathfrak{l}} \mathsf{h}_1$ *then* $\mathsf{h} \overset{\mathfrak{l}'}{\sim} \mathsf{h}_1$ *holds for all* $\mathfrak{l}' \in \mathsf{w}$ *with* $\mathfrak{l}' \neq \mathfrak{l}$ *and* $\mathsf{h}_1 \models \mathsf{w}$.

We also write $\mathsf{h} \sim_{\mathsf{w}} \mathsf{h}'$ *to mean that* $\mathsf{h}, \mathsf{h}' \models \mathsf{w}$ *and* $\mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}'$ *for all* $\mathfrak{l} \in \mathsf{w}$.

The original account of abstract locations [9] also has a notion of independence of locations which facilitate reasoning in the presence of dynamic allocation, and in particular permitted relocation of abstract locations. Since we are not currently treating dynamic allocation of abstract locations, we can avoid this notion here.

Nevertheless, intuitively, we want the abstract locations in a world to be independent from each other, e.g., it makes no sense to have a w specifying both an integer location

and a boolean location placed at the same physical location. In this case there will be no heap h such that $\mathsf{h} \models \mathsf{w}$, but this will be ruled out by our requirement on the initial heap below.

Internal and external world We will henceforth *fix* two worlds, the *internal world* w_i and the external world w_e. The external world contains abstract locations that are acted upon by whole computations, i.e. they describe the contracts and allowed state changes that constrain the end-to-end behaviour of concurrent computations. The internal world, on the other hand, contains abstract locations that are acted upon by steps of computations, i.e. they describe the contracts and allowed state changes that constrain the behaviour of individual atomic steps.

In many cases, the internal and external world will be equal. This is in particular the case if all our data structures are to be accessed atomically. As explained above, there are, however, important examples of concurrent data structures where they differ.

We require a bijection between the internal world and the external world; if $\mathfrak{l} \in \mathsf{w}_e$ then $\mathfrak{l}_i \in \mathsf{w}_i$ is the corresponding location in the internal world; conversely $\mathfrak{l}_e \in \mathsf{w}_e$ corresponds to $\mathfrak{l} \in \mathsf{w}_i$. We require that $\mathfrak{l}_i \leq \mathfrak{l}_e$ holds for all $\mathfrak{l} \in \mathsf{w}_e$, or equivalently, $\mathfrak{l} \leq \mathfrak{l}_e$ for all $\mathfrak{l} \in \mathsf{w}_i$. Alternatively, we could have used a single set of location *identifiers* and then associated to each of those two abstract locations one refining the other.

We require that the initial heap h_{init} satisfy both the external and the internal world ($\mathsf{h}_{init} \models \mathsf{w}_i$ and $\mathsf{h}_{init} \models \mathsf{w}_e$). Note that this excludes in particular the case where one of the worlds contains overlapping locations, e.g. two integer references at the same concrete location for then no satisfying heap exists at all.

The following is clear from the definitions.

Lemma 4. *If* $\mathsf{h} \models \mathsf{w}_e$ *then* $\mathsf{h} \models \mathsf{w}_i$.

Lemma 5. *If* $\mathsf{h} \models \mathsf{w}_i$ *and* $\mathsf{h} \xrightarrow{\mathfrak{l}} \mathsf{h}'$ *for some* $\mathfrak{l} \in \mathsf{w}_e$ *then* $\mathsf{h}' \models \mathsf{w}_i$.

Michael-Scott Queues We specify the internal, $\mathfrak{msq}_i(X)$, and external, $\mathfrak{msq}_e(X)$, abstract locations of for the Michael-Scott queue. Intuitively, the layout of the queue in a heap will be preserved for the internal location, while it will not be relevant for the external location. We represent pointers *head*, *next*, *elem* using some layout convention, e.g. $v.head = v.1$, etc. We then define

$$\mathsf{h}, X \xrightarrow{next} X' \iff X' \text{ can be reached from } X \text{ in } \mathsf{h} \text{by following a chain of next pointers}$$

Moreover, we use $List(X, \mathsf{h}, (X_0, \ldots, X_n), (v_1 \ldots, v_n))$ to signal that $\mathsf{h}(X)$ points to a linked list with nodes $X_0, \ldots, X_n$ and entries $v_1, \ldots v_n$; formally:

$$\begin{array}{ll} \mathsf{h}(X).head = X_0 & \mathsf{h}(X_i).elem = v_i \text{ for } i = 1, \ldots, n \\ \mathsf{h}(X_i).next = X_{i+1} \text{ for } i = 0, \ldots, n-1 & \mathsf{h}(X_n).next = null \end{array}$$

For a given list of elements, $\boldsymbol{a} = (a_0, \ldots, a_n)$, denoted by $\boldsymbol{a}$, we define $H(\boldsymbol{a}) = a_0$ and $T(\boldsymbol{a}) = (a_1, \ldots, a_n)$. Moreover, $\boldsymbol{a}, a_{n+1}$ denotes the list $(a_0, \ldots, a_n, a_{n+1})$.

The external abstract location only looks at the elements in the list and not at the layout of the list. In particular, two heaps are considered equivalent in the external

abstract location when X points to linked lists that contain the same lists, as specified below:

$$\mathsf{h} \overset{\mathfrak{msq}_e(X)}{\sim} \mathsf{h}' \iff \exists \boldsymbol{X}.\exists \boldsymbol{v}.\exists \boldsymbol{X'}.\exists \boldsymbol{v'}.\mathit{List}(X,\mathsf{h},\boldsymbol{X},\boldsymbol{v}) \wedge \mathit{List}(X,\mathsf{h}',\boldsymbol{X'},\boldsymbol{v'}) \wedge\ (\boldsymbol{v}=\boldsymbol{v'})$$

On the other hand, the internal abstract location also looks at the list layout, that is, the concrete locations that form the linked list, including the elements that have been dequeued, as specified below by the last conjunction of the internal location rely relation.

$$\begin{aligned}\mathsf{h} \overset{\mathfrak{msq}_i(X)}{\sim} \mathsf{h}' \iff\ & \exists \boldsymbol{X}.\exists \boldsymbol{v}.\exists \boldsymbol{X'}.\exists \boldsymbol{v'}.\mathit{List}(X,\mathsf{h},\boldsymbol{X},\boldsymbol{v}) \wedge \mathit{List}(X,\mathsf{h}',\boldsymbol{X'},\boldsymbol{v'}) \wedge \\ & (\boldsymbol{v}=\boldsymbol{v'}) \wedge (\boldsymbol{X}=\boldsymbol{X'}) \wedge\ \forall X'.\mathsf{h}(X') \overset{next}{\to} H(\boldsymbol{X}) \iff \mathsf{h}'(X') \overset{next}{\to} H(\boldsymbol{X'})\end{aligned}$$

The operations on a Michael-Scott Queue are specified as follows where the layout of the queue is modified as illustrated by Figure 1. The relations$\mathsf{h} \xrightarrow{\mathfrak{msq}_e(X)} \mathsf{h}_1$ and $\mathsf{h} \xrightarrow{\mathfrak{msq}_i(X)} \mathsf{h}_1$ are identical and given as the transitive closure of the following:

$$\begin{aligned}&\lambda \mathsf{h},\mathsf{h}_1.\mathsf{h} : \mathfrak{msq}_i(X) \wedge \mathsf{h}_1 : \mathfrak{msq}_i(X) \wedge \\ &\quad \mathsf{h}=\mathsf{h}_1 \vee \exists \boldsymbol{X}.\exists \boldsymbol{v}.\mathit{List}(X,\mathsf{h},\boldsymbol{X},\boldsymbol{v}) \wedge \forall X'.\mathsf{h}(X') \overset{next}{\to} H(\boldsymbol{X}) \iff \mathsf{h}_1(X') \overset{next}{\to} H(\boldsymbol{X}) \wedge \\ &\qquad \mathit{List}(X,\mathsf{h}_1,T(\boldsymbol{X}),T(\boldsymbol{v}))\ \texttt{// Dequeue} \\ &\qquad\quad \vee \exists X_{n+1}.\exists v_{n+1}.\mathit{List}(X,\mathsf{h}_1,(\boldsymbol{X},X_{n+1}),(\boldsymbol{v},v_{n+1}))] \qquad \texttt{// or Enqueue}\end{aligned}$$

5 Effects

For each abstract location $\mathfrak{l}$ we have three elementary effects $rd_{\mathfrak{l}}$ (reading from $\mathfrak{l}$), $wr_{\mathfrak{l}}$ (writing to $\mathfrak{l}$), and $co_{\mathfrak{l}}$ (chaotic or concurrent access). The chaotic access is similar to writing, but allows writes that are not in sync. For example, $e_1 = X := 1$ and $e_2 = X := 2$ both have individually the wr_X effect, but e_1 and e_2 are distinguishable with a context that assumes the wr_X-effect. Thus, e_1 and e_2 are not equal "at type" wr_X. At type co_X they are, however, equal, because a context that copes with this effect may not assume that both produce equal results.

We use the $co_{\mathfrak{l}}$ effect to tell the environment not to look at a particular location during a concurrent computation. For example, we will be able to show that $X := !X + 1; X := !X + 1$ is equivalent to $X := !X + 2$ "at type" $(co_X, \emptyset, \{rd_X, wr_X\})$. This means that the two computations are indistinguishable by environments that do not read, let alone modify X during the computation and assume regular read-write access once it is completed.

It would be possible to replace the co-effect using a special set of private locations akin to the private regions from [12], but as we will now see, $co_{\mathfrak{l}}$ can be modelled with the same relational mechanism as the other effects so treating it as an effect streamlines the development.

We use the notation $\mathrm{rds}(\varepsilon)$, $\mathrm{wrs}(\varepsilon)$, $\mathrm{cos}(\varepsilon)$ to refer to the abstract locations $\mathfrak{l}$ for which ε contains $rd_{\mathfrak{l}}$, $wr_{\mathfrak{l}}$, and $co_{\mathfrak{l}}$, respectively. We write $\mathrm{locs}(\varepsilon) := \mathrm{rds}(\varepsilon) \cup \mathrm{wrs}(\varepsilon) \cup \mathrm{cos}(\varepsilon)$. We also write ε^C for ε with each $wr_{\mathfrak{l}}$ in ε replaced by $co_{\mathfrak{l}}$.

Definition 9. *An effect ε is well-formed w.r.t. a world* w*, written* $\mathsf{w} \vdash \varepsilon$*, if it only mentions locations in* w *and for no location* $\mathfrak{l}$ *does it contain both $rd_{\mathfrak{l}}$ and $co_{\mathfrak{l}}$. Moreover, if ε contains $co_{\mathfrak{l}}$ then it also contains $wr_{\mathfrak{l}}$.*

An effect specification is a triple $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ where

- $\mathsf{w}_i \vdash \varepsilon_1$, $\mathsf{w}_i \vdash \varepsilon_2$, $\mathsf{w}_e \vdash \varepsilon_3$
- *elementary effects of the form $co_{\mathfrak{l}}$ appear only in ε_1 and ε_2*
- *if $wr_{\mathfrak{l}}$ appears in ε_2 then $wr_{\mathfrak{l}_e}$ appears in ε_3*
- *if $rd_{\mathfrak{l}}$ appears in ε_2 then $rd_{\mathfrak{l}_e}$ appears in ε_3 and* $\forall \mathsf{h}, \mathsf{h}'. \mathsf{h} \overset{\mathfrak{l}_e}{\sim} \mathsf{h}' \Rightarrow \mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}'$.

An effect specification $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ approximates the behaviour of a computation t in the following way: the effect ε_1 summarizes side effects that may occur during the execution of t (corresponding to a guarantee condition in the rely-guarantee formalism); the effect ε_2 summarizes effects of the interacting environment that t can tolerate while still functioning as expected (corresponding to a rely condition). Finally, ε_3 summarizes the side effects that may occur between start and completion of t. All the effects that the environment might introduce must be recorded in ε_3 because they are not under "our" control and might happen at any time even as the very last thing before the final result is returned. The effects flagged in ε_1, on the other hand, do not necessarily show up in ε_3, for a computation might be able to clean up those effects prior to returning the final result.

Consider the computations $t_1 = X := !X + 1; X := !X + 1$ and $t_2 = X := !X + 2$. Let ε_X stand for $\{rd_X, wr_X\}$ and analogously ε_Y. Each of the two computations can be assigned the effect $(\varepsilon_X, \varepsilon_Y, \varepsilon_X \cup \varepsilon_Y)$, but they are distinguishable at that effect typing. Under the looser specification $(\{co_{\varepsilon_X}\}, \varepsilon_Y, \varepsilon_X \cup \varepsilon_Y)$, however, they are indistinguishable, and our semantics is able to validate this equivalence, see Example 1.

Notations. If $\mathsf{w}_e \vdash \varepsilon$ we write ε_i for the effect obtained by replacing each location $\mathfrak{l}$ in ε by the corresponding $\mathfrak{l}_i$.

For any well-formed effects $\varepsilon, \varepsilon'$ we use the notation $\varepsilon \perp \varepsilon'$ to mean that $\mathrm{rds}(\varepsilon) \cap \mathrm{wrs}(\varepsilon') = \mathrm{rds}(\varepsilon') \cap \mathrm{wrs}(\varepsilon) = \mathrm{wrs}(\varepsilon) \cap \mathrm{wrs}(\varepsilon') = \emptyset$. Note that this implies in particular $\mathrm{cos}(\varepsilon) \cap \mathrm{rds}\varepsilon' = \emptyset$, etc. Intuitively, two programs exhibiting effects ε and ε', respectively, commute with each other.

6 Typing rules

Types are given by the grammar

$$\tau ::= \mathtt{unit} \mid \mathtt{int} \mid \mathtt{bool} \mid A \mid \tau_1 \times \tau_2 \mid \tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \,|\, \varepsilon_3} \tau_2$$

where A ranges over user-specified abstract types. They will typically include reference types such as `intref` and also types like lists, sets, and even objects. In $\tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \,|\, \varepsilon_3} \tau_2$ the triple of effects $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ must be an effect specification.

We use two judgments:

- $\Gamma \vdash v \leq v' : \tau$ specifying that values v and v' have type τ and that v approximates v';
- $\Gamma \vdash t \leq t' : \tau \;\&\; \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$ specifies that the programs t and t' under the context Γ have type τ, with the effect specification $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ specifying, respectively, the effects during execution, the effects of the interacting environment and the start and completion effects. Moreover, t approximates t' at this specification.

We assume an ambient set of *axioms* each having the form (v, v', τ) where v, v' are values in the metalanguage and τ is a type meaning that v and v' are claimed to be of type τ and that v approximates v'. This must then be proved "manually" using the semantics rather than using the rules.

We can also define a typing judgement $\Gamma \vdash v : \tau$ to stand for $\Gamma \vdash v \leq v : \tau$ and can then derive familiar typing rules. To save space we do not give these derived rules explicitly.

While most of the rules are standard, the effect rule for `await` and the parallel composition require a bit of explanation.

Since `await` t_1 `then` t_2 is executed in a single step if t_1 is evaluated to `true`, we can discount the environmental interaction described by ε_2.

The parallel composition rule states that two programs t_1 and t_2 can be composed when their internal effects are not conflicting in the sense that the internal effects of one program appears as environment interaction effects of the other program. Note the relationship to the parallel composition rule of rely-guarantee [16].

Rule (Sem) makes available all kinds of semantic program transformations, including commuting conversions for let and if, fixpoint unrolling, and beta and eta equalities.

We also include two examples of genuinely effect-dependent (in)equalities: the parallelization and the commuting computations rules. We expect to be able to justify further rules of this kind using our methods in future.

7 Typed observational equivalence

Definition 10 (Observational equivalence). *Let v, v' be value expressions where $\vdash v : \tau$ and $\vdash v' : \tau$. We say that v observationally approximates v' at type τ if for all f such that $\vdash f : \tau \xrightarrow{\varepsilon} \mathtt{int}$ ("observations") it is the case that if $((\mathsf{h}_{init}, \mathsf{k}), n) \in [\![f\ v]\!]$ for $v \in \mathbb{Z}$ and starting from h_{init} then $((\mathsf{h}_{init}, \mathsf{k}'), n) \in [\![f\ v']\!]$ for some k'. We write $\vdash v \leq_{obs} v'$ in this case. We say that v and v' are observationally equivalent at type τ, written $\vdash v =_{obs} v'$ if both $\vdash v \leq_{obs} v' : \tau$ and $\vdash v' \leq_{obs} v : \tau$.*

This means that for every test harness f we build around v and v', no matter how complicated it is and whatever environments it sets up to run concurrently with v and v' it is the case that each terminating computation of v (in the environment installed by f) can be matched by a terminating computation with the same result by v' in the same environment. It is important, however, that the environment be well typed, thus will respect the contracts set up by the type τ. E.g. if τ is a functional type expecting, say, a pure function as argument then, by the typing restriction, the environment f cannot suddenly feed v and v' a side-effecting function as input.

$$\frac{}{\Gamma \vdash \mathtt{true} \le \mathtt{true} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{false} \le \mathtt{false} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash n \le n : \mathtt{int}} \qquad \frac{}{\Gamma, x : \tau \vdash x \le x : \tau}$$

$$\frac{\Gamma \vdash v \le v' : \tau}{\Gamma \vdash v \le v' : \tau \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3} \qquad \frac{\Gamma \vdash v \le v' : \tau_1 \times \tau_2}{\Gamma \vdash v.i \le v'.i : \tau_i} \qquad \frac{\Gamma \vdash v_i \le v_i' : \tau_1 \; i = 1 \ldots n}{\Gamma \vdash (v_1, \ldots, v_n) \le (v_1', \ldots, v_n') : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash v_1 \le v_1' : \tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2 \quad \Gamma \vdash v_2 \le v_2' : \tau_1}{\Gamma \vdash v_1\ v_2 \le v_1'\ v_2' : \tau_2 \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3} \qquad \frac{\Gamma \vdash v \le v' : \mathtt{int} \quad \Gamma \vdash t_1 \le t_1' : \tau \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad \Gamma \vdash t_2 \le t_2' : \tau \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}{\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \le \mathtt{if}\ v'\ \mathtt{then}\ t_1'\ \mathtt{else}\ t_2' : \tau \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}$$

$$\frac{\Gamma \vdash t_1 \le t_1' : \tau_1 \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad \Gamma, x{:}\tau_1 \vdash t_2 \le t_2' : \tau_2 \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}{\Gamma \vdash \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 \le \mathtt{let}\ x = t_1'\ \mathtt{in}\ t_2' : \tau_2 \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3} \qquad \frac{\Gamma, f{:}\tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2, x{:}\tau_1 \vdash t \le t' : \tau_2 \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}{\Gamma \vdash \mathtt{rec}\ f\ x = t \le \mathtt{rec}\ f\ x = t' : \tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2}$$

$$\frac{\Gamma \vdash t_1 \le t_1' : \mathtt{bool} \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad \Gamma \vdash t_2 \le t_2' : \tau \,\&\, \varepsilon_1 \mid \emptyset \mid \varepsilon_3}{\Gamma \vdash \mathtt{await}\ t_1\ \mathtt{then}\ t_2 \le \mathtt{await}\ t_1'\ \mathtt{then}\ t_2' : \tau \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3} \qquad \frac{\Gamma \vdash t_1 \le t_1' : \tau_1 \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad \Gamma \vdash t_2 \le t_2' : \tau_2 \,\&\, \varepsilon_2 \mid \varepsilon_1 \mid \varepsilon_3}{\Gamma \vdash t_1 \| t_2 \le t_1' \| t_2' : \tau_1 \times \tau_2 \,\&\, \varepsilon_1 \cup \varepsilon_2 \mid \varepsilon_1 \cap \varepsilon_2 \mid \varepsilon_3}$$

$$\frac{\Gamma \vdash t \le t' : \tau \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad \varepsilon_1 \subseteq \varepsilon_1' \quad \varepsilon_2' \subseteq \varepsilon_2 \quad \varepsilon_3 \subseteq \varepsilon_3'}{\Gamma \vdash t \le t' : \tau \,\&\, \varepsilon_1' \mid \varepsilon_2' \mid \varepsilon_3'} \qquad \frac{[\![t]\!] \le [\![t']\!]}{\Gamma \vdash t \le t' : \tau \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}\ \text{Sem}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \qquad \Gamma \vdash t_2 : \tau_2 \,\&\, \varepsilon_1' \mid \varepsilon_2 \mid \varepsilon_3' \qquad \varepsilon_1 \perp \varepsilon_1' \quad \varepsilon_2 \perp \varepsilon_1 \quad \varepsilon_2 \perp \varepsilon_1' \quad \varepsilon_3 \perp \varepsilon_3'}{\Gamma \vdash (\mathtt{let}\ x = t_1\ \mathtt{in}\ \mathtt{let}\ y = t_2\ \mathtt{in}\ (x,y)) = (\mathtt{let}\ y = t_2\ \mathtt{in}\ \mathtt{let}\ x = t_1\ \mathtt{in}\ (x,y)) : \tau_1 \times \tau_2 \,\&\, (\varepsilon_1 \cup \varepsilon_1')^C \mid \varepsilon_2 \mid \varepsilon_3 \cup \varepsilon_3'}\ \text{Commuting}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \tau_1 \,\&\, \varepsilon_1^C \mid \varepsilon_2^C \mid \varepsilon_3 \qquad \Gamma \vdash t_2 : \tau_2 \,\&\, \varepsilon_1'^C \mid \varepsilon_2^C \mid \varepsilon_3' \\ \varepsilon_1^C \perp \varepsilon_2^C \quad \varepsilon_1'^C \perp \varepsilon_2^C \quad \varepsilon_1^C \perp \varepsilon_1'^C \quad \varepsilon_3 \perp \varepsilon_3'\end{array}}{\Gamma \vdash t_1 \| t_2 = (\mathtt{let}\ x = t_1\ \mathtt{in}\ \mathtt{let}\ y = t_2\ \mathtt{in}\ (x,y)) : \tau_1 \times \tau_2 \,\&\, \varepsilon_1^C \cup \varepsilon_1'^C \mid \varepsilon_2^C \mid \varepsilon_3 \cup \varepsilon_3'}\ \text{Parallelization}$$

Fig. 4. Typing and inequational theory

7.1 Heap Relations

Definition 11. *For each elementary effect ε we define a set of heap relations $\mathcal{R}(\varepsilon)$ as follows.*

- *if $rd_\mathrm{I} \in \varepsilon$ then $\mathsf{h}\,R\,\mathsf{h}'$ implies $\mathsf{h} \overset{\mathrm{I}}{\sim} \mathsf{h}'$;*
- *if $wr_\mathrm{I} \in \varepsilon$ then $\mathsf{h}\,R\,\mathsf{h}'$ and $\mathsf{h}_1 \overset{\mathrm{I}}{\sim} \mathsf{h}_1'$, $\mathsf{h} \xrightarrow{\mathrm{I}} \mathsf{h}_1$, $\mathsf{h}' \xrightarrow{\mathrm{I}} \mathsf{h}_1'$ together imply $\mathsf{h}_1\,R\,\mathsf{h}_1'$;*
- *if $co_\mathrm{I} \in \varepsilon$ then $\mathsf{h}\,R\,\mathsf{h}'$ and $\mathsf{h} \xrightarrow{\mathrm{I}} \mathsf{h}_1$, $\mathsf{h}' \xrightarrow{\mathrm{I}} \mathsf{h}_1'$ together imply $\mathsf{h}_1\,R\,\mathsf{h}_1'$;*

Now, if $\mathsf{w} \vdash \varepsilon$ for $\mathsf{w} = \mathsf{w}_i$ or $\mathsf{w} = \mathsf{w}_e$ we define a set of relations $\mathcal{R}_\mathsf{w}(\varepsilon)$ by $R \in \mathcal{R}_\mathsf{w}(\varepsilon)$ if

- *if $\mathsf{h}\,R\,\mathsf{h}'$ then both $\mathsf{h} \models \mathsf{w}$ and $\mathsf{h}' \models \mathsf{w}$ and $\mathsf{h} \models \mathsf{w}_i$ and $\mathsf{h}' \models \mathsf{w}_i$ (if $\mathsf{w} = \mathsf{w}_i$ then the second part is vacuous);*
- *if $\mathsf{h} \sim_\mathsf{w} \mathsf{h}_1$ and $\mathsf{h} \xrightarrow{\mathrm{I}} \mathsf{h}_1$ and $\mathsf{h}' \xrightarrow{\mathrm{I}} \mathsf{h}_1'$ for some $\mathrm{I} \in \mathsf{w}$ and $\mathsf{h}' \sim \mathsf{h}_1'$ and $\mathsf{h}\,R\,\mathsf{h}'$ then $\mathsf{h}_1\,R\,\mathsf{h}_1'$;*
- *for each elementary effect $\varepsilon_0 \in \varepsilon$ we have $R \in \mathcal{R}(\varepsilon_0)$.*

In [9] heap relations were required to be completely oblivious against $\sim_\mathsf{w}$. Here this obliviousness is restricted to actual moves since this streamlines some arguments.

Lemma 6. *Suppose that* $\mathsf{w} \vdash \varepsilon$ *and define*

$$(\mathsf{h}\, R_\varepsilon \mathsf{h}') \iff \mathsf{h} \models \mathsf{w} \wedge \mathsf{h}' \models \mathsf{w} \wedge \forall \mathfrak{l} \in \mathsf{w}. rd_\mathfrak{l} \in \varepsilon \Rightarrow \mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}'$$

Then R_ε *is the smallest relation in* $\mathcal{R}_\mathsf{w}(\varepsilon)$ *and whenever* $\mathsf{h} \models \mathsf{w}$ *then* $\mathsf{h} R_\varepsilon \mathsf{h}$ *holds.*

Proof. The second part is clear. It is also clear that $R_\varepsilon \in \mathcal{R}_\mathsf{w}(wr_\mathfrak{l})$ for all $\mathfrak{l} \in \mathsf{w}$. Finally, if $rd_\mathfrak{l} \notin \varepsilon$ then $R_\varepsilon \in \mathcal{R}_\mathsf{w}(co_\mathfrak{l})$. But $\mathsf{w} \vdash \varepsilon$ implies that $rd_\mathfrak{l} \notin \varepsilon$ whenever $co_\mathfrak{l} \in \varepsilon$. This shows that $R_\varepsilon \in \mathcal{R}_\mathsf{w}(\varepsilon)$. It is clear that it is smaller than any other such relation.

Definition 12 (Tiling). *Let* $\mathsf{w} \models \varepsilon$ *for* $\mathsf{w} = \mathsf{w}_i, \mathsf{w}_e$. *For heaps* $\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1'$ *we define the tiling relation* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$ *by:*

$$[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1') \iff \forall R \in \mathcal{R}_\mathsf{w}(\varepsilon). \mathsf{h}\, R\, \mathsf{h}' \Rightarrow \mathsf{h}_1\, R\, \mathsf{h}_1'$$

Lemma 7. *Suppose that* $\mathsf{w} \vdash \varepsilon$, $\mathsf{w} \vdash \varepsilon_1$, $\mathsf{w} \vdash \varepsilon_2$. *The following hold whenever well-formed.*

1. *If* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$ *and* $[\varepsilon](\mathsf{h}_1, \mathsf{h}_1', \mathsf{h}_2, \mathsf{h}_2')$ *then* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_2, \mathsf{h}_2')$*;*
2. $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}, \mathsf{h}')$
3. *If* $\varepsilon_1 \subseteq \varepsilon_2$ *then* $[\varepsilon_1](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1') \Rightarrow [\varepsilon_2](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$
4. *Suppose that* $\mathsf{h}\, R_\varepsilon \mathsf{h}'$ *and* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$. *For all* $\mathfrak{l} \in \mathsf{w} \setminus \mathrm{wrs}(\varepsilon) \cup \mathrm{cos}(\varepsilon)$ *we have* $\mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}_1$ *and* $\mathsf{h}' \overset{\mathfrak{l}}{\sim} \mathsf{h}_1'$.
5. *Suppose that* $\mathsf{h}\, R_\varepsilon \mathsf{h}'$ *and* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$. *For* $\mathfrak{l} \in \mathrm{wrs}(\varepsilon)$ *we have* $\mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}_1$ *and* $\mathsf{h}' \overset{\mathfrak{l}}{\sim} \mathsf{h}_1'$ *(not written) or* $\mathsf{h}_1 \overset{\mathfrak{l}}{\sim} \mathsf{h}_1'$ *(identically written).*
6. *Suppose that* $R \in \mathcal{R}(\varepsilon)$ *and* $\mathsf{h}\, R\, \mathsf{h}'$ *and* $(\mathsf{h}, \mathsf{h}_1) \in (\bigcup_{\mathfrak{l} \in \mathrm{wrs}(\varepsilon) \cup \mathrm{cos}(\varepsilon)} \overset{\mathfrak{l}}{\to} \cup \bigcup_{\mathfrak{l} \in \mathsf{w}} (\overset{\mathfrak{l}}{\hookrightarrow} \cap \sim_\mathsf{w}))^*$. *Then there exists* h_1' *such that* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$.

Proof. The first three are direct. For the fourth one define $R = \{(\mathsf{h}_0, \mathsf{h}_0') \mid \mathsf{h}_0 R_\varepsilon \mathsf{h}_0' \wedge \mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}_1 \wedge \mathsf{h}' \overset{\mathfrak{l}}{\sim} \mathsf{h}_1'\}$. It is clear that $R \in \mathcal{R}_\mathsf{w}(\varepsilon)$ so the claim follows. For the fourth one we use the relation $R = \{(\mathsf{h}_0, \mathsf{h}_0') \mid \mathsf{h}_0 R_\varepsilon \mathsf{h}_0' \wedge \mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}_1 \wedge \mathsf{h}' \overset{\mathfrak{l}}{\sim} \mathsf{h}_1' \vee \mathsf{h}_1 \overset{\mathfrak{l}}{\sim} \mathsf{h}_1'\}$.

For the last item we repeatedly use the third clause of Def. 6.

Lemma 8. *If* $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ *is an effect specification then* $\mathcal{R}_{\mathsf{w}_e}(\varepsilon_3) \subseteq \mathcal{R}_{\mathsf{w}_i}(\varepsilon_2)$.

8 Logical Relation

Definition 13 (Specifications). *A value specification is a relation* $E \subseteq \mathbb{V} \times \mathbb{V}$ *such that*

- *if* $x\, E\, y$ *then* $x\, E\, x$ *and* $y\, E\, y$*;*
- *if* $x_1 \leq x$ *and* $y \leq y_1$ *and* $x\, E\, y$ *then* $x_1\, E\, y_1$*;*
- *if* $(x_i)_i$ *and* $(y_i)_i$ *are chains such that* $x_i\, E\, y_i$ *then* $\sup_i x_i\, E\, \sup_i y_i$*, i.e., E is admissible qua relation;*
- *if* $x\, E\, y$ *then* $p_i(x)\, E\, p_i(y)$ *for each i, i.e. E is closed under the canonical projections.*

Similarly, a computation specification is a relation $Q \subseteq T\mathbb{V} \times T\mathbb{V}$ *such that* $x\, Q\, y$ *implies* $x\, Q\, x$ *and* $y\, Q\, y$ *and* $\leq; Q; \leq \subseteq Q$ *and Q is admissible qua relation and Q is closed under the canonical projections* q_i.

Definition 14. *If $E \subseteq \mathbb{V} \times \mathbb{V}$ and $Q \subseteq T\mathbb{V} \times T\mathbb{V}$ then the relation $E{\to}Q \subseteq \mathbb{V} \times \mathbb{V}$ is defined by*

$$f\,E{\to}Q\,f' \iff \forall x\,x'.(x\,E\,x') \Rightarrow (f(x)\,Q\,f'(x'))$$

In particular, for $f\,E{\to}Q\,f'$ to hold, both f, f' must be functions (and not elements of base type or tuples).

The following is direct.

Lemma 9. *If E and Q are specifications so is $E{\to}Q$.*

Definition 15. *Let $E \subseteq \mathbb{V} \times \mathbb{V}$ and $\mathsf{w} \vdash \varepsilon$. We define an admissible subset $S_{\mathsf{w}}(E,\varepsilon) \subseteq (S\,\mathbb{V})^2$ by*

$$(c,c') \in S_{\mathsf{w}}(E,\varepsilon) \iff \left[\begin{array}{c} \forall R \in \mathcal{R}_{\mathsf{w}}(\varepsilon).\forall \mathsf{h},\mathsf{h}'.(\mathsf{h}\,R\,\mathsf{h}') \Rightarrow (c(\mathsf{h})\downarrow \iff c'(\mathsf{h}')\downarrow) \wedge \forall \mathsf{h}_1,\mathsf{h}'_1,a,a'. \\ c(\mathsf{h}) = (\mathsf{h}_1,a) \wedge c'(\mathsf{h}') = (\mathsf{h}'_1,a') \Rightarrow (\mathsf{h}\,R\,\mathsf{h}'_1) \wedge (a\,E\,a') \end{array}\right]$$

This definition is a natural generalisation of the logical relation in [10] from sets to domains and from concrete locations to abstract locations.

The following is the crucial definition of this paper; it gives a semantic counterpart to observational approximation and, due to its game-theoretic flavour, allows for very intuitive proofs.

Definition 16. *Let $E \subseteq \mathbb{V} \times \mathbb{V}$ and $(\varepsilon_1,\varepsilon_2,\varepsilon_3)$ be an effect specification. We define the relations $T_0(E,\varepsilon_1,\varepsilon_2,\varepsilon_3)$ and $T(E,\varepsilon_1,\varepsilon_2,\varepsilon_3)$ between sets of trace-value pairs, i.e. on $T\mathbb{V} \subseteq \mathcal{P}(Tr \times Values)$, as follows.*

$$(U,U') \in T_0(E,\varepsilon_1,\varepsilon_2,\varepsilon_3) \iff \left[\begin{array}{l} \forall((\mathsf{h}_1,\mathsf{k}_1)\ldots(\mathsf{h}_n,\mathsf{k}_n),a) \in U.\forall R \in \mathcal{R}_{\mathsf{w}_e}(\varepsilon_3).\Rightarrow \\ \quad \forall \mathsf{h}'_1.(\mathsf{h}_1\,R\,\mathsf{h}'_1) \Rightarrow \\ \quad\quad \exists \mathsf{k}'_1.[\varepsilon_1](\mathsf{h}_1,\mathsf{h}'_1,\mathsf{k}_1,\mathsf{k}'_1) \wedge \forall \mathsf{h}'_2.[\varepsilon_2](\mathsf{k}_1,\mathsf{k}'_1,\mathsf{h}_2,\mathsf{h}'_2) \Rightarrow \\ \quad\quad\quad \exists \mathsf{k}'_2.[\varepsilon_1](\mathsf{h}_2,\mathsf{h}'_2,\mathsf{k}_2,\mathsf{k}'_2) \wedge \forall \mathsf{h}'_3.[\varepsilon_2](\mathsf{k}_2,\mathsf{k}'_2,\mathsf{h}_3,\mathsf{h}'_3) \Rightarrow \\ \quad\quad\quad\quad \ldots \\ \quad\quad\quad\quad \exists \mathsf{k}'_n.[\varepsilon_1](\mathsf{h}_n,\mathsf{k}_n,\mathsf{h}'_n,\mathsf{k}'_n) \wedge\ (\mathsf{k}_n\,R\,\mathsf{k}'_n) \wedge \\ \quad\quad\quad\quad \exists a' \in \mathbb{V}.(a,a') \in E \wedge ((\mathsf{h}'_1,\mathsf{k}'_1)\ldots(\mathsf{h}'_n,\mathsf{k}'_n),a') \in U' \end{array}\right]$$

We define the relation $T(E,\varepsilon_1,\varepsilon_2,\varepsilon_3) \subseteq T\mathbb{V} \times T\mathbb{V}$ as the admissible closure of T_0, i.e. $Adm(T_0(E,\varepsilon_1,\varepsilon_2,\varepsilon_3))$.

The game-theoretic view of $T_0(E,\varepsilon_1,\varepsilon_2,\varepsilon_3)$ may be understood as follows. Given $U, U' \in T\mathbb{V}$ we can consider a game between a proponent (who believes $(U,U') \in T\mathbb{V}$) and an opponent who believes otherwise. The game begins by the opponent selecting an element $((\mathsf{h}_1,\mathsf{k}_1)\ldots(\mathsf{h}_n,\mathsf{k}_n),a) \in U$, the *pilot trace*, and a heap relation $R \in \mathcal{R}_{\mathsf{w}_e}(\varepsilon_3)$ and a matching start heap h'_1 so that $(\mathsf{h}_1\,R\,\mathsf{h}'_1)$. Then, the proponent answers with a matching heap k'_1 so that $[\varepsilon_1](\mathsf{h}_1,\mathsf{h}'_1,\mathsf{k}_1,\mathsf{k}'_1)$. The opponent then plays a heap h'_2 so that $[\varepsilon_2](\mathsf{k}_1,\mathsf{k}'_1,\mathsf{h}_2,\mathsf{h}'_2)$. Then, again, proponent plays a heap k'_2 such that $[\varepsilon_1](\mathsf{h}_2,\mathsf{h}'_2,\mathsf{k}_2,\mathsf{k}'_2)$ and so on until, proponent has played k'_n so that $[\varepsilon_1](\mathsf{h}_n,\mathsf{h}'_n,\mathsf{k}_n,\mathsf{k}'_n)$. At that point proponent must also play a value a' and it is then checked whether or not $((\mathsf{h}'_1,\mathsf{k}'_1)\ldots(\mathsf{h}'_n,\mathsf{k}'_n),a') \in U'$ and $(a\,E\,a')$. If this is the case or if at any one point in the game the opponent was unable to move because there exists no appropriate heap then the proponent has won the game. Otherwise the opponent wins and we have $(U,U') \in T_0(E,\varepsilon_1,\varepsilon_2,\varepsilon_3)$ iff the proponent has a winning strategy for that game.

The following is the main technical result of our paper and shows that the computation specifications $T(\dots)$ can indeed serve as the basis for a logical relation. Its proof can be found in the Appendix.

Theorem 1 (Main result). *The following hold whenever well-formed.*

1. *If* $(U, U') \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *then* $(q_i(U), q_i(U')) \in T(E, \varepsilon_1, \varepsilon_2)$.
2. *If* $(U, U') \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *then* $(U^\dagger, U'^\dagger) \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$*, too.*
3. *If* $(a, a') \in E$ *then* $(\eta(a), \eta(a'))$ *is in* $T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$.
4. *If* $(c, c') \in S_{\mathsf{w}_i}(E, \varepsilon_1) \cap S_{\mathsf{w}_e}(E, \varepsilon_3)$ *then* $(\mathit{fromstate}(c), \mathit{fromstate}(c')) \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *for any* ε_2.
5. *If* $(f, f') \in E_1 \to T(E_2, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *and* $(U, U') \in T(E_1, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *then* $(\mathit{ap}(f, U), \mathit{ap}(f', U')) \in T(E_2, \varepsilon_1, \varepsilon_2, \varepsilon_3)$.
6. *If* $(U_1, U_1') \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *and* $(U_2, U_2') \in T(E, \varepsilon_2, \varepsilon_1, \varepsilon_3)$ *then* $(U_1 \parallel U_1', U_2 \parallel U_2') \in T(E, \varepsilon_1 \cup \varepsilon_2, \varepsilon_1 \cap \varepsilon_2, \varepsilon_3)$.
7. *If* $(U_1, U_1') \in T(\mathbb{B}, \varepsilon_1, \emptyset, \varepsilon_3)$ *and* $(U_2, U_2') \in T(E, \varepsilon_1, \emptyset, \varepsilon_3)$ *then* $(\mathit{await}(U_1, U_2), \mathit{await}(U_1', U_2')) \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$.

8.1 Relationship to typed observational approximation

We will assign a value specification $[\![\tau]\!]$ to each refined type by putting

- $[\![\mathtt{int}]\!] = \{(v, v') \mid v = v' \in \mathbb{Z}\}$
- $[\![\tau_1 \times \tau_2]\!] = [\![\tau_1]\!] \times [\![\tau_2]\!]$
- $[\![\tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2]\!] = [\![\tau_1]\!] \to T([\![\tau_2]\!], \varepsilon_1, \varepsilon_2, \varepsilon_3)$

We omit the obvious definition of the other basic types and assume value specifications for user-specified types as given.

Theorem 2. *Suppose that* $\Gamma \vdash v : \tau$ *and* $\Gamma \vdash t : \tau \;\&\; \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$ *and assume that for each axiom* (v, v', τ) *it holds that* $(v, v) \in [\![\tau]\!]$ *and* $(v', v') \in [\![\tau]\!]$*. Then* $(\eta, \eta') \in [\![\Gamma]\!]$ *(interpreting a context as a cartesian product) implies* $(\lceil v \rceil \eta, \lceil v' \rceil \eta') \in [\![\tau]\!]$ *and* $([\![t]\!]\eta, [\![t]\!]\eta') \in T([\![\tau]\!], \varepsilon_1, \varepsilon_2, \varepsilon_3)$.

Proof. By induction on derivations. Most cases are already subsumed by Theorem 1. The typing rules regarding functions and recursion follow from the definitions and from the fact that all specifications are admissible.

Corollary 2. *Suppose that* $\vdash v : \tau$ *and* $\vdash v' : \tau$ *and that* $([\![v]\!], [\![v']\!]) \in [\![\tau]\!]$*. Then* $\vdash v \leq_{obs} v' : \tau$.

Proof. If $\vdash f : \tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \mathtt{int}$ then by the above Theorem we have $([\![f]\!], [\![f]\!]) \in [\![\tau \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \mathtt{int}]\!]$, so $([\![f\ v]\!], [\![f\ v']\!]) \in T([\![\mathtt{int}]\!], \varepsilon_1, \varepsilon_2, \varepsilon_3)$.

Let $((\mathsf{h}_{init}, \mathsf{k}), v) \in [\![f\ v]\!]$. We have $(\mathsf{h}_{init}, \mathsf{h}_{init}) \in R_{\varepsilon_3}$ by Lemma 6 and the assumption that $\mathsf{h}_{init} \models \mathsf{w}_e$. There must therefore exist a matching heap k' and a value v' such that $((\mathsf{h}_{init}, \mathsf{k}'), v') \in [\![f\ v']\!]$ and $v = v' \in \mathbb{Z}$.

We can thus use semantic reasoning as in the proof of Theorem 1 to deduce observational equivalences. Figure 4 summarises an inequational theory that can be proved sound in this fashion. We expect other effect-dependent equations such as those from [10] to hold, but have not yet finalized the side-conditions.

Lemma 10. *Let A, B, C be sets, $R \subseteq A \times A$, $S \subseteq B \times B$, $Q \subseteq C \times C$ be reflexive relations and $f, f' : A \times B \to C$ be functions. If $aRa' \wedge bSb' \Rightarrow f(a,b)Qf'(a',b')$ then $aR^*a' \wedge bS^*b' \Rightarrow f(a,b)Q^*f'(a',b')$ where $(-)^*$ denotes transitive closure.*

Theorem 3. *Suppose that for each type axiom (v, v', τ) one has $([\![v]\!], [\![v']\!]) \in [\![\tau]\!]$. Then, if $\vdash v \leq v' : \tau$ is derivable using the rules in Figure 4 then $\vdash v \leq_{obs} v' : \tau$.*

Proof (Sketch). We show by induction on derivations that whenever $\Gamma \vdash v \leq v' : \tau$ is derivable then $([\![v]\!], [\![v']\!])$ is in the transitive closure of the following relation:

$\{(f, f') \mid \forall(\eta, \eta') \in [\![\Gamma]\!].(f\eta, f'\eta') \in [\![\tau]\!]\}$

and an analogous assertion for computations. It is important that transitive closure is taken because specifications are not themselves required to be transitive. Lemma 10 above allows us to do each case as if no transitive closure had been applied. The congruence rules then follow from the previous results, in particular Theorem 1. For the effect-dependent equivalences we proceed in essentially the same fashion as in [27] or [10]. The cases of commuting computations and parallelization are detailed in Lemmas 11 and 12 in the appendix.

The (in)equational theory given is really only a first step. We expect stronger rules to be justifiable, in particular, versions of the effect-dependent equivalences that do allow a limited amount of environmental interaction:

Example 1. Consider the following programs:

$e_1 = (X := !X + 1; X := !X + 1) \qquad e_2 = (X := !X + 2)$

Let $\mathfrak{l} = \mathfrak{int}(X)$ be the abstract location for a single integer stored at X (see Section 4). We can show that $(e_1, e_2) \in T(\mathtt{unit}, \{co_{\mathfrak{l}}\}, \varepsilon, \varepsilon \cup \{rd_{\mathfrak{l}}, wr_{\mathfrak{l}}\})$ this equation holds if $\{co_{\mathfrak{l}}\} \perp \varepsilon$, that is, when the environment does not read nor write X, and does not hold otherwise, *e.g.*, if $rd_{\mathfrak{l}} \in \varepsilon$. We discuss how the proof works (in the former case) and where it fails (in the latter case).

For both cases, the opponent picks a relation $R \in \mathcal{R}(\varepsilon \cup \{rd_{\mathfrak{l}}, wr_{\mathfrak{l}}\})$ and a trace in the semantics of e_1, for example:

$(([X, n]\mathsf{h}, [X, n+1]\mathsf{h})([X, n+1]\mathsf{h}_1, [X, n+2]\mathsf{h}_1), ())$

The opponent also picks a heap h'_1 such that $\mathsf{h}[X{\mapsto}n]\ R\ \mathsf{h}'_1$. Thus h'_1 should be of the form $\mathsf{h}'[X{\mapsto}n]$. Now its the proponent's turn to find a k'_1 such that

$[co_{\mathfrak{l}}](\mathsf{h}[X{\mapsto}n], \mathsf{h}'[X{\mapsto}n], \mathsf{h}[X{\mapsto}n], \mathsf{k}'_1)$

Since $co_{\mathfrak{l}}$ relates any arbitrary writes to X, the proponent may chose $\mathsf{k}'_1 = \mathsf{h}'[X{\mapsto}n]$. Now it is opponent's turn to chose h'_2 such that:

$[\varepsilon](\mathsf{h}[X{\mapsto}n+1], \mathsf{h}'[X{\mapsto}n], \mathsf{h}_1[X{\mapsto}n+1], \mathsf{h}'_2)$

Here is where the opponent wins when $rd_{\mathfrak{l}} \in \varepsilon$. It is not the case that $\mathsf{h}[X{\mapsto}n+1]\ R'$ $\mathsf{h}'[X{\mapsto}n]$ for any $R' \in \mathcal{R}(\varepsilon)$ when $rd_{\mathfrak{l}} \in \varepsilon$ as the values in X are different. Therefore, the opponent is free to choose any heap h'_2, for example, $\mathsf{h}'_2 = \mathsf{h}''[X{\mapsto}n+1000]$. The proponent now loses the game as their only chance for chosing k'_2 such that

$[co_{\mathfrak{l}}](\mathsf{h}[X{\mapsto}n+1], \mathsf{h}''[X{\mapsto}n+1000], \mathsf{h}[X{\mapsto}n+2], \mathsf{k}'_2)$
and such that $\mathsf{h}_1[X{\mapsto}n+2]\ R\ \mathsf{k}'_2$ is to set X back to $n+2$. But then the trace:
$(\mathsf{h}'[X{\mapsto}n], \mathsf{h}'[X{\mapsto}n])(\mathsf{h}''[X{\mapsto}n+1000], \mathsf{h}''[X{\mapsto}n+2])$
does not belong to the semantics of e_2.

Notice that if ε does not interfere with $co_{\mathfrak{l}}$, then the opponent would necessary have to select a heap that does not change the value of X, that is, a heap of the form $\mathsf{h}''[X{\mapsto}n]$ and the proof would succeed.

Example 2. Let X and Y be two concrete locations and $\mathfrak{l}_1 = \mathfrak{int}(X)$ and $\mathfrak{l}_2 = \mathfrak{int}(Y)$ be the abstract locations for single integers stored in X and Y, respectively. Consider the following programs

$e_1 = X := !X + 1; X := !X - 1 \quad$ and $\quad e_2 = Y := !Y + 1; Y := !Y - 1$

We can show that:

$(e_1, \mathtt{skip}) \in T(\mathtt{unit}, \{co_{\mathfrak{l}_1}\}, \emptyset; \varepsilon_1) \qquad (e_2, \mathtt{skip}) \in T(\mathtt{unit}, \{co_{\mathfrak{l}_2}\}, \emptyset; \varepsilon_2)$

where $\varepsilon_1 = \{rd_{\mathfrak{l}_1}, wr_{\mathfrak{l}_1}\}$ and $\varepsilon_2 = \{rd_{\mathfrak{l}_2}, wr_{\mathfrak{l}_2}\}$. This means that there are winning strategies for the proponent for showing that for any t of e_1, there is an equivalent trace t' of `skip`. We omit the details.

We now return to the two examples that we discussed in Section 1 and demonstrate how to prove using our denotational semantics the properties that have been discussed informally.

Overlapping References With this example, we illustrate the parallelization rule. In particular, the functions declared in Section 1 have the following type, where ε does not read nor write X:

$$\mathsf{readFst} : \mathtt{unit} \xrightarrow[\varepsilon]{rd_{\mathsf{fst}(X)} \mid rd_{\mathsf{fst}(X)}} \mathtt{int} \quad \mathsf{writeFst} : \mathtt{int} \xrightarrow[\varepsilon]{wr_{\mathsf{fst}(X)} \mid wr_{\mathsf{fst}(X)}} \mathtt{unit}$$

$$\mathsf{readSnd} : \mathtt{unit} \xrightarrow[\varepsilon]{rd_{\mathsf{snd}(X)} \mid rd_{\mathsf{snd}(X)}} \mathtt{int} \quad \mathsf{writeSnd} : \mathtt{int} \xrightarrow[\varepsilon]{wr_{\mathsf{snd}(X)} \mid wr_{\mathsf{snd}(X)}} \mathtt{unit}$$

We justify this typing semantically as described in Theorem 1. To illustrate how this is done, consider the function writeSnd(17). We show how the game is played against itselt using the typing shown above. We start with a "pilot trace", say:

([2|3], [2|3]), ([2|17], [2|17]), (())

where $[x|y]$ denotes a store with $X = p(x, y)$ and other components left out for simplicity. The first step corresponds to our reading of X and in the second step – since there was no environment intervention – we write 17 into the first component.

We now start to play: Say that we start at the heap [13|12]. We answer [13|12]. If the environment does not change X, then we write 17 to its first component resulting in the following trace, which is possible for writeFst(17).

([13|12], [13|12]), ([13|12], [17|12]), (())

If, however, the environment plays [18|21] (a modification of both components of X has occurred), then we answer [17|21]. Again,

([13|12], [13|12]), ([18|21], [17|21]), (())

is a possible trace for writeFst(17). It is easy to check that there is a strategy that justifies the typing given above.

Now, consider a program, t_1, that only calls the functions readFst, writeFst, and another program, t_2, that only calls readSnd, writeSnd. Since the former functions have disjoint

effects to the latter ones, t_1 and t_2 will have effect specifications, respectively, of the form $(\varepsilon_1, \varepsilon, \varepsilon_1)$ and $(\varepsilon_2, \varepsilon, \varepsilon_2)$, where $\varepsilon_1 \cap \varepsilon_2 = \varepsilon_1 \cap \varepsilon = \varepsilon_2 \cap \varepsilon = \emptyset$. Thus we can use the parallelization rule shown in Figure 4 to conclude that the behavior of $t_1 \| t_2$ is the same as executing these programs sequentially, although they read and write to the same concrete location.

Michael-Scott Queue We now show that the functions enqueue and dequeue functions described in Section 1 for the Michael-Scott Queue have the same behavior of their atomic versions `atomic`(enqueue), `atomic`(dequeue). We show only the case for dequeue, as the case for enqueue is similar. In particular, we show that dequeue () has the same behavior of `atomic`(dequeue ()) at the following type

$T(\texttt{int}, MSQ_i, MSQ_i, MSQ_e)$

where $MSQ_i = \{rd_{\mathfrak{msq}_i(X)}, wr_{\mathfrak{msq}_i(X)}\}$ and $MSQ_e = \{rd_{\mathfrak{msq}_e(X)}, wr_{\mathfrak{msq}_e(X)}\}$. That is, at a type where the environment is allowed to read and write to the queue.

We only show dequeue () $\leq$ `atomic`(dequeue ()); the other direction being easy by stuttering.

We start the game where the opponent picks a trace of dequeue () and a relation in $R \in \mathcal{R}(MSQ_e)$: $(\mathsf{h}_0, \mathsf{k}_0)(\mathsf{h}_1, \mathsf{k}_1) \ldots (\mathsf{h}_i, \mathsf{k}_i) \ldots (\mathsf{h}_n, \mathsf{k}_n)$. and a final value a. Assume that only at the move $(\mathsf{h}_i, \mathsf{k}_i)$ one succeeds to dequeue an element. So clearly, $\mathsf{h}_j = \mathsf{k}_j$ for each $j \neq i$. We can match this trace a trace in the semantics of `atomic`(dequeue ()) by stuttering $(\mathsf{h}'_0, \mathsf{h}'_0)(\mathsf{h}'_1, \mathsf{h}'_1) \ldots (\mathsf{h}'_i, \mathsf{k}'_i) \ldots (\mathsf{h}'_n, \mathsf{h}'_n)$ where h_j and h'_j have exactly the same MSQ (even the layout). Notice that if the moves did not preserve the layout of the list, the opponent could have made any environment move and won the game (as in Example 1). Clearly, the same element is dequeued by the move $(\mathsf{h}'_i, \mathsf{k}'_i)$ and therefore the end value a' of this trace is the same as a.

9 Discussion

We have shown how a simple effect system for stateful computation and its relational semantics, combined with the notion of abstract locations, scales to a concurrent setting. The resulting type system provides a natural and useful degree of control over the otherwise anarchic possibilities for interference in shared variable languages, as demonstrated by the fact that we can delineate and prove the conditions for non-trivial contextual equivalences, including fine-grained data structures.

The primary goal of this line of work is not so much to find reasoning principles that support the most subtle equivalence arguments for particular programs, but rather to capture more generic properties of modules, expressed in terms of abstract locations and relatively simple effect annotations, that can be exploited by clients (including optimizing compilers) in external reasoning and transformations. But there are of course, particularly in view of the fact that we allow deeper reasoning to be used to establish that expressions can be assigned particular effect-refined types, very close connections with other work on richer program logics and models. Rely-guarantee reasoning is widely used in program logics for concurrency, including relational ones [20], whilst our abstract locations are very like the *islands* of Ahmed et al [4]. Recent work of Turon et al [28] on relational models for fine-grained concurrency introduces richer abstractions, notably state transition systems expressing inter-thread protocols that can involve

ownership transfer. These certainly allow the verification of more complex fine-grained algorithms than can be dealt with in our setting, and it would be natural to try defining an effect semantics over such a model. Indeed, one might reasonably hope that effects could provide something of a 'simplifying lens', with refined types capturing things that would otherwise be extra model structure or more complex invariants, such that the combination does not lead to further complexity. The use of Brookes's trace model (also used by, for example, Turon and Wand [29]) already seems to bring some simplification compared to transition systems or resumptions.

Birkedal et al [12] have also given a relational semantics for effects in a concurrent language. They only use simple type-based invariants on concrete locations, which are much less expressive than abstract locations, and only treat a very simple fine-grained example. On the other hand, their step-indexed operational relation does allow for higher-order store, and they do treat dynamic allocation via regions. Our logical relation seems (though this is obviously rather subjective) somewhat more straightforward than theirs, for example by not requiring public and private locations. But we note that our Definition 16, of the relations T_0 and T, does resemble, and serves the same purpose as, Birkedal et al's definition of 'safety' [12, Fig.6], as well as Liang et al's 'RGSim' relation [20, Def.4].

We have really only scratched the surface of effects for concurrency, and there are many directions for further work. Firstly, we expect further useful transformational rules to be justifiable in the model we already have. We should consider both higher-order store (though this may be challenging with our current style of model [7]) and dynamic allocation of abstract locations (following our earlier work in a sequential setting [9]). Finally, we have only considered an interleaving model of concurrency, and have as yet given no thought to how one might do something similar in the context of weak memory.

References

1. M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.
2. M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.
3. S. Abramsky and A. Jung. Domain theory, 1994. Online Lecture Notes, avaliable from CiteSeerX.
4. A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.
5. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, pages 575–631, 1993.
6. T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. World Scientific, 1999.
7. N. Benton, L. Beringer, M. Hofmann, and A. Kennedy. Relational semantics for effect-based program transformations: higher-order store. In *PPDP*, 2009.
8. N. Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *3rd ACM Workshop on Types in Language Design and Implementation (TLDI '07)*, 2007.
9. N. Benton, M. Hofmann, and V. Nigam. Abstract effects and proof-relevant logical relations. In *POPL*, pages 619–632, 2014.

10. N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations: Towards extensional semantics for effect analyses. In *APLAS*, volume 4279 of *LNCS*, 2006.
11. N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *ICFP*, 1998.
12. L. Birkedal, F. Sieczkowski, and J. Thamsborg. A concurrent logical relation. In P. Cégielski and A. Durand, editors, *CSL*, volume 16 of *LIPIcs*, pages 107–121. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
13. L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, 1996.
14. N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *15th European Symposium on Programming (ESOP '06)*, volume 3924 of *LNCS*. Springer, 2006.
15. S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.
16. J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput.*, 17(4):807–841, 2007.
17. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, 2003.
18. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *LISP and Functional Programming*, 1986.
19. O. Kammar and G. D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *POPL*, 2012.
20. H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 455–468, New York, NY, USA, 2012. ACM.
21. N. A. Lynch and F. W. Vaandrager. Forward and backward simulations, ii: Timing-based systems. *Inf. Comput.*, pages 1–25, 1996.
22. M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, May 1998.
23. R. D. Nicola and M. Hennessy. Testing equivalence for processes. In *ICALP*, pages 548–560, 1983.
24. F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of the 26 ACM Symposium on Principles of Programming Languages (POPL '99)*, 1999.
25. G. D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5(3):452–487, 1976.
26. M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982.
27. J. Thamsborg and L. Birkedal. A Kripke logical relation for effect-based program transformations. In *ICFP*, 2011.
28. A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 343–356. ACM, 2013.
29. A. J. Turon and M. Wand. A separation logic for refining concurrent objects. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, 2011.

A Proof of Theorem 1:

Proof. In each case, using Corollary 1 and Lemma 3 (for case 5), we can in fact assume w.l.o.g. that the assumed pairs are in $T_0(\dots)$ rather than $T(\dots)$.

Ad 1. Let $(t, a) \in q_i(U)$, i.e. $a = p_i(a_0)$ where $(t, a_0) \in U$. By down-closure ([Down]) we also have $(t, a) \in U$. We can now play the strategy guaranteed by the assumption $(U, U') \in T_0(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ which will yield (depending on the opponent's moves) a trace t' and a value a' such that $(t', a') \in U'$ and $(p_i(a), a') \in E$. Now, since E is a specification we get $(p_i(a), p_i(a')) \in E$ noting that p_i is idempotent. So, we modify the strategy so as to return $p_i(a')$ rather than a' and thus obtain a winning strategy asserting the desired conclusion.

Ad 2 Pick $(U, U') \in T_0(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$. Since $T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ is closed under suprema it suffices to show that $(q_j(U^\dagger), q_j(U'^\dagger)) \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ for each j. Fix such j and pick $(t, p_j(a)) \in q_j(U^\dagger)$, thus $(t, a) \in U^\dagger$.

By induction on the closure process we can assume w.l.o.g. that (t, a) arises from $(t_1, a) \in U$ by a single mumbling or stuttering step or that $(t, a_1) \in U$ for some $a_1 \geq a$ or else that $(t, a_i) \in U$ where $\sup_i a_i = a$.

In the former two cases fix a strategy for the original element of U. We will use this strategy to build a new one demonstrating that $(t, a) \in U'$, hence $(t, p_j(a)) \in q_j(U')$ as required.

If (t, a) arises by stuttering, so $t = u(\mathsf{h}, \mathsf{h})v$ and $t_1 = uv$ we play the strategy until u is worked off. If the opponent then produces a heap h' to match h we answer h'.

Now $[\varepsilon_1](\mathsf{h}, \mathsf{h}', \mathsf{h}, \mathsf{h}')$ is always true (Lemma 7) so this is a legal move. Thereafter, we continue just as in the original strategy. In the special case where v is empty, we must also show that $\mathsf{h}\, R \mathsf{h}'$ where $R \in \mathcal{R}_{\mathsf{w}_e}(\varepsilon_1, \varepsilon_2, \varepsilon_3)$. Let k, k' be the final states in the original trace t_1 and its match. We know that $\mathsf{k} R \mathsf{k}'$ and $[\varepsilon_2](\mathsf{k}, \mathsf{k}', \mathsf{h}, \mathsf{h}')$. The claim follows since $R_{\mathsf{w}_e}(\varepsilon_3) \subseteq R_{\mathsf{w}_i}(\varepsilon_2)$.

If (t, a) arises by mumbling then we must have $t_1 = u(\mathsf{h}_1, \mathsf{h}_2)(\mathsf{h}_2, \mathsf{h}_3)v$ and $t = u(\mathsf{h}_1, \mathsf{h}_3)v$. We play until the strategy has produced a match h'_2 for h_2. So far, the play has produced a trace u' matching u, and a state h'_1 so that $[\varepsilon_1](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{h}_2, \mathsf{h}'_2)$. Now, we can ask what the original strategy would produce if we gave it (temporarily assuming opponent's role) the state h'_2 as a match for h_2. Note that this is legal because $[\varepsilon_2](\mathsf{h}_2, \mathsf{h}'_2, \mathsf{h}_2, \mathsf{h}'_2)$. The strategy will then produce h'_3 such that $[\varepsilon_1](\mathsf{h}_2, \mathsf{h}'_2, \mathsf{h}_3, \mathsf{h}'_3)$ and our answer in the play on the new trace against the challenge h'_1 will be this very h'_3. Indeed, by composing tiles (Lemma 7) we have $[\varepsilon_1](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{h}_3, \mathsf{h}'_3)$ as required. Thereafter, the play continues according to the original strategy.

For down-closure, we play the strategy against (t, a_1) yielding a match $(t', a'_1) \in U'$ where $a_1 E a'_1$. That same strategy also wins against (t, a) because $a E a'_1$ since E is a value specification.

For closure under [Sup], finally, pick i so that $a_i \geq p_j(a)$ recalling that $a = \sup_i a_i$. Since we have a winning strategy for (t, a_i), we also have one (by down-closure which was already proved) for $(t, p_j(a))$ as required.

Ad 3. Suppose $a E a'$. By 2 which we have just proved we only need to match elements of the form $((\mathsf{h}, \mathsf{h})a)$. The opponent plays h' such that $\mathsf{h} R \mathsf{h}'$ for some $R \in \mathcal{R}_{\mathsf{w}_e}(\varepsilon_3)$.

We answer with h' itself and a'. This is a legal move since $[\varepsilon_1](\mathsf{h}, \mathsf{h}', \mathsf{h}, \mathsf{h}')$ whatever ε_1 is. Furthermore, $\mathsf{h}R\mathsf{h}'$ and aEa', so we win the game.

Ad 4. Suppose that $(c, c') \in S_{\mathsf{w}_i}(E, \varepsilon_1) \cap S_{\mathsf{w}_e}(E, \varepsilon_3)$. Again, we only need to match traces of the form $((\mathsf{h}, \mathsf{h}_1), a)$ where $c(\mathsf{h}) = (\mathsf{h}_1, a)$. In this case, suppose that the opponent plays h' with $\mathsf{h}R\mathsf{h}'$ for some $R \in \mathcal{R}_{\mathsf{w}_e}(\varepsilon_3)$. Since, $(c, c') \in S(E, \varepsilon)$ we know that $c'(\mathsf{h}') = (\mathsf{h}'_1, a')$ for some h'_1 and a' and $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}'_1)$ and also aEa'. Thus, h'_1 and a is a winning reply.

Ad 5. Suppose $(f, f') \in E_1 \to T_0(E_2, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ and $(U, U') \in T_0(E_1, \varepsilon_1, \varepsilon_2, \varepsilon_3)$. Suppose that $(uv, b) \in fU$ where $(u, a) \in U$ and (v, b) in $f(a)$ (note that we can ignore the †-closure). Choose a winning strategy S_1 for (u, a). We play according to S_1 to work off the u-part. This results in a matching trace u' (depending on opponent's moves, of course) and a value a' such that aE_1a'. We get $(f(a), f(a')) \in T_0(E_2, \varepsilon_1, \varepsilon_2, \varepsilon_3)$. This yields a winning strategy S_2 for (v, b). We can continue our play by using S_2 which yields a continuation v' of our trace and a final answer b'. It is then clear that $(u'v', b') \in ap(f', U')$ so this combination of strategies does indeed win.

Ad 6. Suppose w.l.o.g. that $(U_1, U'_1) \in T_0(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ and $(U_2, U'_2) \in T_0(E, \varepsilon_2, \varepsilon_1, \varepsilon_3)$ and let $(t, (a, b)) \in U_1 \parallel U_2$, thus $inter(t_1, t_2, t)$ (ignoring † by item 2) where $(t_1, a) \in U_1$ and $(t_2, b) \in U_2$. Let S_1, S_2 be corresponding winning strategies. The idea is to use S_1 when we are in t_1 and to use S_2 when we are in t_2. Supposing that t starts with a t_1 fragment we begin by playing according to S_1. Let $(\mathsf{h}, \mathsf{h}')$ be the initial heaps, so $\mathsf{h}R\mathsf{h}'$ where $R \in \mathcal{R}_{\mathsf{w}_e}(\varepsilon_3)$ is the external relation. Let $(\mathsf{h}_1, \mathsf{h}'_1)$ be the heaps reached at the end of this first period including a final move of the environment. Since the environment is constrained by $\varepsilon_1 \cap \varepsilon_2$ we can conclude $[\varepsilon_1](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}'_1)$. Now, since $(\varepsilon_2, \varepsilon_1, \varepsilon_3)$ is an effect specification, ε_1 is "subsumed" by ε_3 and we obtain $\mathsf{h}_1R\mathsf{h}'_1$. Thus, we are in a position to invoke S_2 on the subsequent t_2-fragment until we reach, after a final environment interaction the states $(\mathsf{h}_2, \mathsf{h}'_2)$ with $[\varepsilon_2](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{h}_2, \mathsf{h}'_2)$. Now, since S_1 tolerates an environment constrained by ε_2, we can tack this entire conversation on to the last environment move of S_1's play and let it resume from there. This then continues until all of t is worked off.

Ad 7. Let $((\mathsf{h}, \mathsf{k}), v)$ be a trace to match so that $((\mathsf{h}, \mathsf{h}_1), \mathtt{true}) \in U_1$ and $((\mathsf{h}, \mathsf{k}), v) \in V_1$. Furthermore, let h' and R be given. By assumption we get matching traces $((\mathsf{h}', \mathsf{h}'_1), \mathtt{true}) \in U'_1$ and $((\mathsf{h}', \mathsf{k}'), v') \in U'_2$. We answer by k', v'.

B Proof sketches for Commuting and Parallelization rules

Lemma 11. *Suppose that* $(U, U') \in T_0(A, (\varepsilon_1, \varepsilon_2, \varepsilon_3)$ *and that* $(V, V') \in T_0(A, (\varepsilon'_1, \varepsilon_2, \varepsilon'_3)$ *and that the effects* $\varepsilon_j, \varepsilon'_j$ *satisfy the side conditions of rule (Commuting), i.e.,* $\varepsilon_1 \perp \varepsilon'_1$, $\varepsilon_1 \perp \varepsilon_2$, ... *Define* $X = \{(uv, (a, b)) \mid (u, a) \in U, (v, b) \in V\}^\dagger$ *and* $Y = \{(vu, (a, b)) \mid (u, a) \in U', (v, b) \in V'\}^\dagger$. *Then* $(X, Y) \in T_0(A \times B, (\varepsilon_1 \cup \varepsilon'_1)^C, \varepsilon_2, \varepsilon_3 \cup \varepsilon'_3)$.

Proof (Sketch). Assume a trace $(uv, (a, b)) \in X$. By Theorem 1(2) we can assume $(u, a) \in U$ and $(v, b) \in V$. Suppose that u starts with $(\mathsf{h}_1, \mathsf{k}_1)(\mathsf{h}_2, \mathsf{k}_2) \dots$ and that v starts with $(\mathsf{h}_{100}, \mathsf{k}_{101}) \dots$. Also suppose that R and h'_1 have been proposed by the opponent. Using $(U, U') \in T_0(A, \varepsilon_1, \emptyset, \varepsilon_3)$ we find that $\mathsf{h}_{101}R_{\varepsilon'_3}\mathsf{h}'_1$ so that we can start an

aside game against the trace (v, b) beginning at $\mathsf{h}_{101}, \mathsf{h}'_1$ with us assuming the role of the environment.

Now we are set up to play against the pilot trace $(uv, (a, b))$. The side game tells us how make a V'-move from h'_1 to k'_1, say. We copy that move to the main game and this is legal since $[\varepsilon_1](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{k}_1, \mathsf{k}'_1)$ holds because condition on ε_1. If the environment now plays h'_2 then we can find a state h'_{102} such that $[\emptyset](\mathsf{k}_{101}, \mathsf{k}'_1, \mathsf{h}_{102}, \mathsf{h}'_{102})$ to play in the side game. This will produce an answer k'_2 which we play in the main game, and so on. This continues until the side game is used up at which point we install a side game against the original trace u noticing that the current state in the main game is R_{ε_3} related to h_1. We then use this new side game to advise our moves in the main game just as before until we are finished.

Lemma 12. *Suppose that* $(U, U') \in T_0(A, (\varepsilon_1, \varepsilon_2, \varepsilon_3)$ *and that* $(V, V') \in T_0(A, (\varepsilon'_1, \varepsilon_2, \varepsilon'_3)$ *and that the effects* $\varepsilon_j, \varepsilon'_j$ *satisfy the side conditions of rule (Parallelization). Define* $X = \{(w, (a, b)) \mid inter(u, v, w), (u, a) \in U, (v, b) \in V\}^\dagger$ *and* $Y = \{(uv, (a, b)) \mid (u, a) \in U', (v, b) \in V'\}^\dagger$. *Then* $(X, Y) \in T_0(A \times B, (\varepsilon_1 \cup \varepsilon'_1)^C, \varepsilon_2, \varepsilon_3 \cup \varepsilon'_3)$.

Proof (Sketch). Assume a pilot trace $(w, (a, b)) \in X$. By Theorem 1(2) we can assume $(u, a) \in U$ and $(v, b) \in V$ and $inter(u, v, w)$. Put $w = (h_1, k_1) \dots (h_n, k_n)$ where moves (h_i, k_i) belong to either (U, U') or (V, V'). Let $R \in \mathcal{R}(\varepsilon_3 \cup \varepsilon'_3)$ and let h'_1 be the starting heap where $(h_1, h'_1) \in R$. Now we should pick a k'_1 such that $[\varepsilon_1](h_1, h'_1, k_1, k'_1)$. Instead of performing all the pieces of U-moves in the pilot trace one by one, we make a big U'-move, that is, we chose a move $(h'_1, k'_1) \in U$, such that $k'_1 \overset{\mathfrak{l}}{\sim} k_n$ for all $\mathfrak{l} \in \mathrm{locs}(\varepsilon_3)$. Such a move is obtained by playing the game for (U, U') against the pilot trace and answering all moves by V or by the environment by stuttering moves. Then all the resulting U'-moves can be contracted into a single one by mumbling. Of course, this requires that—as requested—(U, U') are oblivious against Vs moves and the environmental moves even concurrently.

Now, we have an environment move to h'_2, but since its reads and writes are disjoint from $\varepsilon_3 \cup \varepsilon'_3$, it is the case that $k'_1 \overset{\mathfrak{l}}{\sim} h'_2$ for all $\mathfrak{l} \in \mathrm{locs}(\varepsilon_3 \cup \varepsilon'_3)$. We now make a big V'-move by chosing k'_2 as before with U', in such a way that $(h'_2, k'_2) \in V'$ and $k_n \overset{\mathfrak{l}}{\sim} k'_2$ for all $\mathfrak{l} \in \mathrm{locs}(\varepsilon'_3)$. We now fill the trace $(h1', k1')(h2', k2')$ with stuttering moves to obtain a trace that has the same length as the pilot trace. Thus,

$$(h'_1, k'_1)(h'_2, k'_2)(h'_3, h'_3)(h'_4, h'_4) \dots (h'_n, h'_n), (a', b') \text{ in } X$$

Since the environment's effects are disjoint from $\varepsilon_3 \cup \varepsilon'_3$, we have that $k'_2 \overset{\mathfrak{l}}{\sim} h'_n$ for all $\mathfrak{l} \in \mathrm{locs}(\varepsilon_3 \cup \varepsilon'_3)$. Thus, $(h_n, h'_n) \in R$.