Effect-Dependent Transformations for Concurrent Programs

Nick Benton

Facebook, London, UK

Martin Hofmann

LMU, Munich, Germany

Vivek Nigam

fortiss, Munich

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. Effect annotations are relative to abstract locations specified by contracts rather than physical footprints allowing us in particular to show the soundness of some transformations involving 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 of e_2 with a logically related result value v_2.

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 [19], effect systems have been used for

Email addresses: `nick.benton@gmail.com` (Nick Benton), `hofmann@ifi.lmu.de` (Martin Hofmann), `vivek.nigam@gmail.com` (Vivek Nigam)

many purposes, including region-based memory management [13], tracking exceptions [27, 26], communication behaviour [5] and atomicity [18] 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 [21, 9, 6, 12, 29]. 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. Fine-grained, optimistic algorithms, which rely on custom protocols being followed by multiple threads with concurrent access to a shared data structure, can significantly outperform ones based on coarse-grained locking, but are notoriously challenging to write and verify.

There is a huge literature on 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 [31]. This paper explores 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.

We build on 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 semantically-formulated effect system that separately tracks: (1) the store effects of an expression during evaluation; (2) the assumed effects of transitions by the environment; and (3) the overall end-to-end effect. 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 [6]. 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. Both typing and refinement judgements may be established by a combination of generic type-based rules and semantic reasoning in the model.

This paper is an extended archival version of [10] which has been presented at PPDP 2016. In addition to the conference version this paper has more detail about the higher-order version of Brookes' trace semantics (Section 3), more examples, in particular the one on loop parallelization, and detailed proofs of the main results on soundness of the logical relation and general reasoning principles (Theorem 7.7) and on canonical program equivalences (Theorem 9.1).

We begin with some motivating examples.

Equivalence modulo non-interference.. Our semantics justifies the equation $(X := {!X} + 1; X := {!X} + 1) = (X := {!X} + 2)$ *at* the effect type $\texttt{unit} \,\&\, \{co_X\} \mid \varepsilon \mid \varepsilon {\cup} \{rd_X, wr_X\}$, *provided* that the effect, ε, of the concurrent environment does not involve X. This says that the two commands are equivalent with return type $\texttt{unit}$,[1] exhibit the effect co_X, signifying concurrent or 'chaotic' access to X along the way, and have an overall end-to-end effect of ε plus reading and writing X.

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

$$
\begin{array}{l}
\mathsf{readFst}\ () = p({!X}).1 \\
\mathsf{readSnd}\ () = p({!X}).2 \\
\mathsf{wrtFst}\ n = (\texttt{rec}\ \mathsf{try}\ \_ = \texttt{let}\ m = {!X}\ \texttt{in} \\
\qquad\qquad \texttt{if}\ \mathsf{cas}(X, m, p^{-1}(n, p(m).2)) \\
\qquad\qquad \texttt{then}\ ()\ \texttt{else}\ \mathsf{try}\ ()\ )() \\
\mathsf{wrtSnd}\ n = (\texttt{rec}\ \mathsf{try}\ \_ = \texttt{let}\ m = {!X}\ \texttt{in} \\
\qquad\qquad \texttt{if}\ \mathsf{cas}(X, m, p^{-1}(p(m).1, n)) \\
\qquad\qquad \texttt{then}\ ()\ \texttt{else}\ \mathsf{try}\ ()\ )()
\end{array}
$$

which multiplex two abstract integer references onto a single concrete one. Note that the write functions, wrtFst and wrtSnd, use compare-and-swap, cas, to atomically

[1] Being equal at a type means being may-indistinguishable for any observations which use the terms at that type.

update the value of the reference. More precisely as follows

$$\mathtt{cas}(X, v_1, v_2) = \mathtt{atomic}(\mathtt{if}\ !X = v_1\ \mathtt{then}\ X := v_2; \mathtt{true}\ \mathtt{else}\ \mathtt{false})$$

where `atomic` enforces the atomic evaluation of the argument expression.

Our generic rules (Figure 5) then say that a program, e_1, that only reads and/or writes one abstract reference can be commuted, or executed in parallel, with another program, e_2, that only reads and/or writes into a different reference. This lets one use types to, say, justify parallelizing a call to wrtFst followed by one to wrtSnd, even though they read and write the same concrete location, which looks like a race.

Version numbers:. One can isolate a transaction that reads and then writes a piece of state simply by enclosing the whole thing in $\mathtt{atomic}(\cdot)$. A more concurrent alternative adds a monotonic version number to the data. A transaction then works on a private copy, only committing its changes back (and incrementing the version) if the current version number is the same as that of the original copy. We can define an abstract integer reference $\mathfrak{X}$ in terms of two concrete ones, X_{ver} and X_{val}, governed by a specification that says $!X_{\text{val}}$ may only change when $!X_{\text{ver}}$ increases. We define

$$\begin{array}{l}
\mathsf{transact}\ f = \mathtt{let\ rec}\ \mathsf{try}() = \\
\mathtt{let}\ (val, ver) = \mathtt{atomic}((!X_{\text{val}}, !X_{\text{ver}})) \\
\ \mathtt{in\ let}\ res = f(val)\ \mathtt{in\ if}\ \mathtt{atomic}(\mathtt{if}\ !X_{\text{ver}} = ver\ \mathtt{then} \\
\quad X_{\text{ver}} := ver + 1;\ X_{\text{val}} := res;\ \mathtt{true\ else\ false}) \\
\ \mathtt{then}\ ()\ \mathtt{else}\ \mathsf{try}() \\
\mathtt{in}\ \mathsf{try}()
\end{array}$$

Under the assumption that f is a pure function (has effect type $\mathtt{int} \xrightarrow[\varepsilon]{\emptyset \mid \varepsilon} \mathtt{int}$ for any ε), we can show

$$\mathsf{transact}\ f = \mathtt{atomic}(X_{\text{val}} := f(!X_{\text{val}}); X_{\text{ver}} :=!X_{\text{ver}} + 1)$$

at type $\mathtt{unit}\&\{rd_{\mathfrak{X}}, wr_{\mathfrak{X}}\} \mid \varepsilon \mid \varepsilon \cup \{rd_{\mathfrak{X}}, wr_{\mathfrak{X}}\}$ for any ε not including chaotic access, $co_{\mathfrak{X}}$, to $\mathfrak{X}$. The environment effect ε here *may* include reading and writing $\mathfrak{X}$, so concurrent calls to transact are linearizable.

Loop Parallelization:. Our next example is inspired by a loop unrolling optimization [30]. Assume given a linked list of integers pointed by *head*. Consider the fol-

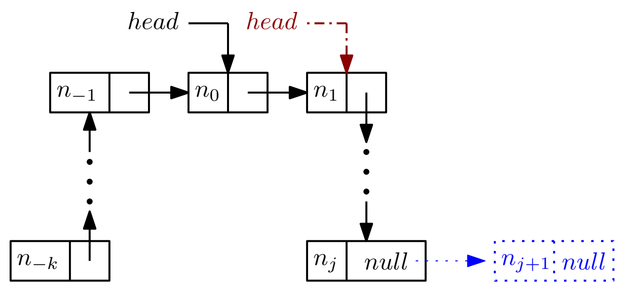

Figure 1: Illustration of a Michael-Scott Queue. The list resulting from the pointer to the element n_0 (the *head* pointer with the continuous arrow in black) contains the list of elements $[n_1, \ldots, n_j]$. The enqueueing operation is illustrated by the dotted arrow and the box with the element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

lowing functions:

```
map f =      let rec applyf n =
               n.ele := f(n.ele); if n.next = null then ()
               else applyf (n.next)
             in if !head = null then () else applyf (!head)
map2Par f =  let rec applyf2 n =
               n.ele := f(n.ele) || n.next.ele := f(n.next.ele);
               if n.next.next = null then ()
               else if n.next.next.next = null then
                   n.next.next.ele = f(n.next.next.ele)
                 else applyf2 (n.next.next)
             in if !head = null then ()
                else if !head.next = null then
                  !head.next.ele := f(!head.next.ele)
                else applyf2 (!head)
```

The function map simply applies a pure function f to each element of the list, each element per iteration. The function map2Par, on the other hand, applies f to two consecutive elements of the list in parallel, potentially allowing one to exploit multiple cores. Our effect-based reasoning will soundly transform map into map2Par (under the assumption that the environment does not interfere with the list).

Michael-Scott queue.. The Michael-Scott Queue [25] (MSQ) is a fine grained concurrent data structure, allowing threads to access and modify different parts of a queue safely and simultaneously. We present an idealized version like that of Turon et al [31], which omits a tail pointer.

An MSQ maintains a pointer *head* to a non-empty linked list as depicted in Figure 1. The first node, that 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 the diagram to the right. Elements are dequeued from the beginning of the list, and en-

```
dequeue () =   (rec try () = let n0 =!head in
                if !n0.next = null then null
                else let n1 =!n0.next in
                  if cas(!head, n0, n1) then !n1.ele
                  else try ()) ()
enqueue(x) =   (rec try (p) =
                if !p.next = null then
                  if atomic(if !p.next = null then
                   !p.next := ref(x, null); true else false)
                  then () else try (!p.next)
                else try (!p.next)) !head
mem x =        (rec find l =
               if l = null then false else
                  if !l.ele = x then true else
                  find !l.next) !head.next
reset () =     (rec deqAll () =
                if dequeue () = null then ()
                  else deqAll ()) ()
```

Figure 2: Enqueue, Dequeue, Membership, and Reset programs for a Michael-Scott Queue at location *head*.

queued at the end, involving a traversal that is done without locking. Once the end, p, of the list is found, the program atomically attempts to insert the new element. This operation has to be atomic because other programs may have enqueued elements to the end of the list, meaning that p is no longer the end of the list.

We prove that the enqueue and dequeue of Figure 2 are equivalent to their atomic versions `atomic`(enqueue) and `atomic`(dequeue), which perform all operations in a single step, at a type that allows the environment to be concurrently reading and writing the queue. So the fine-grained MSQ behaves like a synchronized queue, as might also be implemented using locks.

We can also show that mem is equivalent to its atomic version `atomic`(mem) at type $\texttt{int} \xrightarrow[\varepsilon_2]{\emptyset \mid \varepsilon_2, rd_{MSQ}} \texttt{bool}$ provided the environment does not access the MSQ chaotically, *i.e.*, $co_{MSQ} \notin \varepsilon_2$. This typing denotes that mem has the effect of reading the MSQ, both during execution and as overall effect. With more assumptions on the environment effects ε_2, namely, that it does not enqueue nor dequeue MSQ, mem may participate in many of the equations we prove sound, *e.g.*, commuting, deadcode.

Similarly, reset is equivalent to `atomic`(reset) at the type $\texttt{unit} \xrightarrow[\varepsilon_2]{rd_{MSQ} wr_{MSQ} \mid \varepsilon_2, wr_{MSQ}} \texttt{unit}$. During execution, reset both reads and writes the MSQ, but we can show semantically that its overall effect is only the environmental effect ε_2 plus writing the MSQ; there is no overall read effect. Again, from the typing (and assumptions on ε_2), one obtains equations involving reset without further semantic reasoning.

2. Syntax

In this section we define the syntax of a metalanguage for concurrent, stateful computations and higher-order functions. Communication between parallel computations is via a shared heap mapping dynamically allocated locations to structured values, which include pointers. To keep the model simple, we do not allow functions to be stored in the heap (no higher-order store).

Memory model. We assume a countably infinite set $\mathbb{L}$ of physical locations $X_1, \ldots, X_n, \ldots$ and a set $\mathbb{VB}$ of "R-values" that can be stored in those references including integers, booleans, locations, and tuples of R-values, written $(v_1, \ldots, v_n)$. We assume that it is possible to tell of which form a value is and to retrieve its components in case it is a tuple. A heap h, then, is a *finite map* from $\mathbb{L}$ to $\mathbb{VB}$, written $\{(X_1, \mathsf{c}_1), (X_2, \mathsf{c}_2), \ldots, (X_n, \mathsf{c}_n)\}$, specifying that the value stored in location X_i is c_i. We write $\mathrm{dom}(\mathsf{h})$ for the domain of h and 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 $\mathsf{h}[X \mapsto v]$.

Syntax of expressions. The syntax of untyped values and computations is:

$$
\begin{array}{rcl}
v & ::= & x \mid (v_1, v_2) \mid v_r \mid c \mid \mathtt{rec}\ f\ x = t \\
e & ::= & v \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid v_1\ v_2 \mid \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \\
 & & \mid !v \mid v_1 := v_2 \mid \mathtt{ref}(v) \mid e_1 \| e_2 \mid \mathtt{atomic}(e)
\end{array}
$$

Here, x ranges over variables, v_r over R-values, and c over built-in functions, which include arithmetic, testing whether a value is an integer, function, pair or reference, equality on simple values, etc. Each c has a corresponding semantic partial function F_c, so for example $F_+(n, n') = n + n'$ for integers n, n'.

The construct $\mathtt{rec}\ f\ x = e$ defines a recursive function with body e and recursive calls made via f; we use $\lambda x.e$ as syntactic sugar in the case when f is not free in e. Next, $!v$ (reading) returns the contents of location v, $v_1 := v_2$ (writing) updates location v_1 with value v_2, and $\mathtt{ref}(v)$ (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. We emphasize that the use of let-normal form is merely a convenience not reducing expressivity in any way. For example, we view $v_1 := v_2$ as short-hand for $\mathtt{let}\ x_1 = v_1\ \mathtt{in}\ \mathtt{let}\ x_2 = v_2\ \mathtt{in}\ x_1 := x_2$ thus writing the evaluation order explicitly.

The construct $e_1 \| e_2$ is evaluated by arbitrarily interleaving evaluation steps of e_1 and e_2 until each has produced a value, say v_1 and v_2; the result is then (v_1, v_2). Assignment, dereferencing and allocation are atomic, but evaluation of nested expressions is generally not. To enforce atomicity, $\mathtt{atomic}(e)$ evaluates an arbitrary e in one step, without any environmental interference.

We define the free variables, $FV(e)$, of a term, closed terms, and the substitution $e[v/x]$ of v for x in e, in the usual way. Locations may occur in terms, but the type system will constrain their use.

3. Denotational Model

We now describe a denotational semantics for our metalanguage based on Brookes' trace semantics [15]. In the technical report [7] we give some more detail and in particular a proof of adequacy with respect to an interleaving operational semantics, which we elide here since it is not germane to the topic of this article.

3.1. Preliminaries

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.

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 $f_x(a) = f(x, a)$. As usual, f_x^i is the i-th iteration of f_x. 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. Thinking of U as a predicate on the elements of A, we have that $f^{\ddagger}(x)$ satisfies U provided that f_x preserves and U is admissible and an $\bot \in U$. This principle is known as Scott induction. 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 partial function $f : A \rightharpoonup A$ is a *retract* if $f(a) \leq a$ and $f(f(a)) = f(a)$ hold for all $a \in A$. In short: $f \leq id_A$ and $f \circ f \leq f$. If, in addition, f has a finite image then f is called a *deflation* [3]. Note that if f is a retract then $\mathrm{dom}(f) = \mathrm{Img}(f)$ and if $a \in \mathrm{Img}(f)$ then $a = f(a)$. We also note that if a is in the image of a deflation then a is compact.

We define the usual state monad on predomains, by taking $SA = \mathbb{H} \rightharpoonup \mathbb{H} \times A$. As we seen, Scott induction applies to admissible predicates only which motivates the following definition:

Definition 3.1. *Let P be a subset of a predomain A. Then $Adm(P)$ is 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 $a = \sup_i a_i$.*

We will often find ourselves in the situation of wanting to show some property P, but (since we want to use Scott induction) are only able to prove $Adm(P)$. The following lemma says intuitively that if we know $x_1 \in Adm(P_1) \ldots x_n \in Adm(P_n)$ then we can actually assume $x_1 \in P_1 \ldots x_n \in P_n$ so long as the end result ("Q") is admissible and the x_is are used in a continuous fashion.

Lemma 3.2. *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 $f : Adm(P_1) \times \cdots \times Adm(P_n) \to Q$.*

The following lemma has a similar purpose. It asserts that under mild condition on the pre-domains involved, in order to show that some continuous function $Adm(P \to Q)$ it suffices to show that it is in $P \to Adm(Q)$.

Lemma 3.3. *Let A, B be predomains and let $(p_i)_i$ be a chain of retracts on B such that $p_i(b)$ is compact for each i and $\sup_i p_i = id_B$ and $b \in Q$ implies $p_i(b) \in Q$ for all i. Then $P{\to}Adm(Q) = Adm(P \to Q)$.*

3.2. Traces

A trace models a terminating run of a concurrent computation as a sequence of pairs of heaps, each representing pre- and post-state of one or more atomic actions. The semantics of a program then is a (typically large) set of traces (and final values), accounting for all possible environment interactions.

Definition 3.4 (Traces)**.** *A trace is a finite sequence of the form $(\mathsf{h}_1, \mathsf{k}_1)(\mathsf{h}_2, \mathsf{k}_2)\cdots(\mathsf{h}_n, \mathsf{k}_n)$ where for $1 \le j \le i \le n$, we have $\mathsf{h}_i, \mathsf{k}_i \in \mathbb{H}$ and $\mathrm{dom}(\mathsf{h}_j) \subseteq \mathrm{dom}(\mathsf{h}_i), \mathrm{dom}(\mathsf{h}_j) \subseteq \mathrm{dom}(\mathsf{k}_i), \mathrm{dom}(\mathsf{k}_j) \subseteq \mathrm{dom}(\mathsf{h}_i), \mathrm{dom}(\mathsf{k}_j) \subseteq \mathrm{dom}(\mathsf{k}_i)$. 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 case where $\mathsf{k}_1 = \mathsf{h}_2$ the trace $(\mathsf{h}_1, \mathsf{k}_2)(\mathsf{h}_3, \mathsf{k}_3)$ arises from t 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.

Brookes [15] gives a fully-abstract semantics for while-programs with parallel composition using sets of traces closed under stuttering and mumbling. We here extend his semantics to higher-order functions and general recursion.

Definition 3.5 (Trace Monad)**.** *Let A be a predomain. Elements of the domain 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 $a_2 \le a_1$ 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 3.6. *If A is a predomain then TA is a domain.*

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 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 $X := !X + 1; X := !X + 1; !X$ contains many traces, including the following, where we write $[n]$ for the heap in which X has value n:

$$
\begin{array}{l}
(([10],[12]),12),\\
(([10],[11])([15],[16]),16),\\
(([10],[11])([15],[16])([17,17]),17),\\
(([10],[11])([15],[16])([17,17]),16),\\
(([10],[11])([17],[17])([15],[16]),16),\ldots
\end{array}
$$

Axiom [S&M] is taken from Brookes. It ensures that the semantics does not distinguish between late and early choice [31] and related phenomena which are reflected, e.g., in resumption semantics [28], but do not affect observational equivalence. Note that non-termination is modelled by the empty set, so we are working with an 'angelic' notion of equivalence ('may semantics' [17]). For example, the semantics of $X := 0;$ `if` $X{=}0$ `then` 0 `else diverge` is the same as that of $X := 0; 0$ and contains, 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.

$$
\begin{array}{lcl}
e_1 & \equiv & \texttt{if}\ ?\ \texttt{then}\ X := 0; \texttt{true}\ \texttt{else}\ X := 0; \texttt{false}\\
e_2 & \equiv & X := 0;\ ?
\end{array}
$$

Both e_1 and e_2 admit the same traces, namely $(([x],[0]),\texttt{true})$ and $(([x],[0]),\texttt{false})$ and stuttering variants thereof. In semantic models based on transition systems or resumptions and bisimulation, these are distinguished, which necessitates the use of special mechanisms such as history and prophecy variables [2], forward-backward simulation [24], or speculation [31] in reasoning.

Axioms [Down] and [Sup] are known from the Hoare powerdomain [28]. 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. Additional nondeterministic outcomes that are less defined than existing ones are not recorded in the semantics.

Definition 3.7. *If $U \subseteq Tr \times A$ then $U^\dagger$ is the least subset of TA containing U, i.e. $U^\dagger$ is the closure of U under [S& M], [Down], [Sup].*

Definition 3.8. *Let A, B be a predomains. We define the continuous functions $rtn : A \to TA$ and $bnd : (A \to TB) \times TA \to TB$ by:*

$$
\begin{array}{l}
rtn(a) := (\{((\mathsf{h},\mathsf{h}),a) \mid \mathsf{h} \in \mathbb{H}\})^\dagger\\
bnd(f,g) := (\{(uv,b) \mid (u,a) \in g \wedge (v,b) \in f(a)\})^\dagger
\end{array}
$$

These endow TA with the structure of a strong monad. The continuous function $\mathit{fromstate} : SA \to TA$ is defined by:

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

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

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

The continuous map $\mathit{at} : TA \to TA$ is defined by:

$$\mathit{at}(U) := \{((\mathsf{h},\mathsf{k}),v) \mid ((\mathsf{h},\mathsf{k}),v) \in U\}^\dagger$$

Notice that due to mumbling $((\mathsf{h},\mathsf{k}),v) \in U$ iff there exists an element

$$((\mathsf{h}_1,\mathsf{h}_2)(\mathsf{h}_2,\mathsf{h}_3)(\mathsf{h}_{n-2},\mathsf{h}_{n-1})(\mathsf{h}_{n-1},\mathsf{h}_n),v) \in U$$

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 U.

3.3. Semantic values

The predomain $\mathbb{V}$ of untyped values is the least solution of the following domain equation:

$$\mathbb{V} \simeq \mathbb{VB} + (\mathbb{V} \to T\mathbb{V}) + \mathbb{V}^*.$$

That is, values are either R-values, continuous functions from values to computations ($T\mathbb{V}$), or tuples of values. We tend to identify the summands of the right hand side with subsets of $\mathbb{V}$ but may use tags like $\mathit{fun}(f) \in \mathbb{V}$ when $f : \mathbb{V} \to T\mathbb{V}$ to avoid ambiguity.

We have families of deflations $p_i : \mathbb{V} \multimap \mathbb{V}$ and $q_i : T\mathbb{V} \to T\mathbb{V}$, referred to as canonical deflations, so that $(p_i)_i$ and $(q_i)_i$ are ascending chains converging to the identity. The definition is entirely standard and may be found in the technical report [7]. It shows in particular that $\mathbb{V}$ and $T\mathbb{V}$ are *bifinite* (equivalently SFP) (pre-)domains [3] and as such also Scott (pre-) domains. The presence of these deflations allows us to apply Lemma 3.3 and simplifies reasoning in general.

The semantics of values $\lceil\!\lceil v \rceil\!\rceil \in \mathbb{V} \to \mathbb{V}$ and terms $[\![t]\!] \in \mathbb{V} \to T\mathbb{V}$ are given by the recursive clauses in Figure 3. Environments, ρ, are properly tuples of values; we abuse notation slightly by treating them as maps from variables, x, to values, v, (and write $\rho[x{\mapsto}v]$ for functional update) to avoid mentioning an explicit context in which untyped terms are well-formed. The last clause applies to semantically ill-typed programs, for example:

$$[\![\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]\rho$$

when $\lceil\!\lceil v \rceil\!\rceil\rho$ does not return a boolean value, but, *e.g.*, a number or a location.

$$
\begin{array}{rcl}
\lceil\!\lceil x \rceil\!\rceil\rho & = & \rho(x) \\
\lceil\!\lceil v_r \rceil\!\rceil\rho & = & v_r \\
\lceil\!\lceil (v_1, v_2) \rceil\!\rceil\rho & = & (\lceil\!\lceil v_1 \rceil\!\rceil\rho, \lceil\!\lceil v_2 \rceil\!\rceil\rho) \\
\lceil\!\lceil v.i \rceil\!\rceil\rho & = & d_i \text{ if } i = 1, 2, \lceil\!\lceil v \rceil\!\rceil\rho = (d_1, d_2) \\
\lceil\!\lceil c \rceil\!\rceil\rho & = & \mathit{fun}(f) \\
 & & \text{where } f(v) = \mathit{rtn}(F_c(v)) \text{ if } F_c(v) \text{ is defined} \\
 & & \text{and } f(v) = \emptyset\text{, otherwise.} \\
\lceil\!\lceil \mathtt{rec}\ f\ x = e \rceil\!\rceil\rho & = & \mathit{fun}(g^{\ddagger}(\rho)) \\
 & & \text{where } g(\rho, u) = \lambda d.\lceil\!\lceil e \rceil\!\rceil\rho[f \mapsto u, x \mapsto d] \\
\lceil\!\lceil v \rceil\!\rceil\rho & = & 0\text{, otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
\llbracket v \rrbracket\rho & = & \mathit{rtn}(\lceil\!\lceil v \rceil\!\rceil\rho) \\
\llbracket \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \rrbracket\rho & = & \mathit{bnd}(\lambda d.\llbracket e_2 \rrbracket\rho[x \mapsto d], \llbracket e_1 \rrbracket\rho) \\
\llbracket v_1\ v_2 \rrbracket\rho & = & \lceil\!\lceil v_1 \rceil\!\rceil\rho(\lceil\!\lceil v_2 \rceil\!\rceil\rho) \\
\llbracket \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rrbracket\rho & = & \llbracket e_1 \rrbracket\rho\text{, if } \lceil\!\lceil v \rceil\!\rceil\rho = \mathtt{true} \\
\llbracket \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rrbracket\rho & = & \llbracket e_2 \rrbracket\rho\text{, if } \lceil\!\lceil v \rceil\!\rceil\rho = \mathtt{false} \\
\llbracket !v \rrbracket\rho & = & \mathit{fromstate}(\lambda \mathsf{h}.(\mathsf{h}, \mathsf{h}(X)))\text{, when } \lceil\!\lceil v \rceil\!\rceil\rho = X \\
\llbracket v_1 := v_2 \rrbracket\rho & = & \mathit{fromstate}(\lambda \mathsf{h}.(\mathsf{h}[X \mapsto \lceil\!\lceil v_2 \rceil\!\rceil\rho], ()))\text{, if } \lceil\!\lceil v_1 \rceil\!\rceil\rho = X \\
\llbracket \mathtt{ref}(v) \rrbracket\rho & = & \mathit{fromstate}(\lambda \mathsf{h}.\mathit{new}(\mathsf{h}, \lceil\!\lceil v \rceil\!\rceil\rho)) \\
\llbracket \mathtt{atomic}(e) \rrbracket\rho & = & \mathit{at}(\llbracket e \rrbracket)\rho \\
\llbracket e_1 \| e_2 \rrbracket\rho & = & \llbracket e_1 \rrbracket\rho \mid \llbracket e_2 \rrbracket\rho \\
\llbracket e \rrbracket\rho & = & \emptyset\text{, otherwise}
\end{array}
$$

Figure 3: Denotational semantics

4. Abstract Locations

We build on the concept of abstract locations defined by Benton et al [6]. These allow complicated data structures that span 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 locations 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 in the present paper, we can slightly simplify the original definition [6], dropping those axioms that involve the interaction with dynamic allocation.[2] On the other hand, in the presence of concurrency, we need *two* partial equivalence relations: one that models semantic equivalence and well-formedness and a finer one that constrains the heap modifications that other concurrent computations that are independent of the given abstract locations are allowed to do *while* an operation on the abstract location is ongoing, but temporarily preempted.

Definition 4.1 (Concurrent Abstract Location)**.** *A* concurrent abstract location ł *consists of the following data:*

(1) a partial equivalence relation $\overset{\text{ł}}{\sim}$ *on* $\mathbb{H}$ *modeling the "semantic equivalence" on the bits of the store that* ł *uses. If* $\mathsf{h} \overset{\text{ł}}{\sim} \mathsf{h}'$ *then the same computation started on* h *and* h'*, respectively, will yield related or even equal results.*

(2) a partial equivalence relation $\overset{\text{ł}}{=}$ *on* $\mathbb{H}$ *refining* $\overset{\text{ł}}{\sim}$ *and modeling the "strict equivalence" on the bits of the store that* ł *uses. If a concurrent computation on* ł *has reached* h *and is preempted, then another computation may replace* h *with* h' *where* $\mathsf{h} \overset{\text{ł}}{=} \mathsf{h}'$ *and then the original computation on* ł *may resume on* h' *without the final result being compromised.*

(3) a transitive (and reflexive on the support of $\overset{\text{ł}}{\sim}$*) relation* $\overset{\text{ł}}{\to}$ *modeling how exactly the heap may change upon writing the abstract location and in particular what bits of the store such writes leave intact. In other words, if* $\mathsf{h} \overset{\text{ł}}{\to} \mathsf{h}_1$ *then* h_1 *might arise by writing to* ł *in* h *and all possible writes are specified by* $\overset{\text{ł}}{\to}$*. We call* $\overset{\text{ł}}{\to}$ *the* step relation *of* ł.

In addition, we require the following conditions where $\mathsf{h} : \text{ł}$ *stands for* $\mathsf{h} \overset{\text{ł}}{\sim} \mathsf{h}$.

1. If $\mathsf{h} : \text{ł}$ *then* $\mathsf{h} \overset{\text{ł}}{=} \mathsf{h}$*;*

[2]Though our examples do all satisify these axioms, leaving the way open to a future extension with dynamically allocation of abstract locations and concurrency.

2. *if* $\mathsf{h} \xrightarrow{\mathfrak{l}} \mathsf{h}_1$ *then* $\mathsf{h} : \mathfrak{l}$ *and* $\mathsf{h}_1 : \mathfrak{l}$.

If $\mathsf{h} \xrightarrow{\mathfrak{l}} \mathsf{h}_1$ *and at the same time* $\mathsf{h} \stackrel{\mathfrak{l}}{=} \mathsf{h}_1$, *then we say that* h_1 *arises from* h *by a* silent move *in* $\mathfrak{l}$. *Our semantic framework will permit silent moves at all times.*

We now introduce some examples of abstract locations.

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

$$\begin{array}{lcl}
\mathsf{h} \stackrel{\mathfrak{int}(X)}{\sim} \mathsf{h}' & \iff & \exists n. \mathsf{h}(X) = int(n) \wedge \mathsf{h}'(X) = int(n) \\
\mathsf{h} \stackrel{\mathfrak{int}(X)}{=} \mathsf{h}' & \iff & \mathsf{h} \stackrel{\mathfrak{int}(X)}{\sim} \mathsf{h}' \\
\mathsf{h} \xrightarrow{\mathfrak{int}(X)} \mathsf{h}_1 & \iff & \\
\multicolumn{3}{l}{\qquad \mathsf{h} : \mathfrak{int}(X), \mathsf{h}_1 : \mathfrak{int}(X) \text{ and } \forall X' \in \mathbb{L}. X' \neq X \Rightarrow \mathsf{h}(X') = \mathsf{h}_1(X)}
\end{array}$$

Two heaps are semantically equivalent (w.r.t. $\mathfrak{int}(X)$ that is) if the values stored in X are integers and equal; the step relation requires all other concrete locations to be unchanged.

We will sometimes abuse notation and write rd_X, wr_X, co_X for $rd_{\mathfrak{int}(X)}, wr_{\mathfrak{int}(X)}, co_{\mathfrak{int}(X)}$.

Overlapping references. Let X be a concrete location encoding a pair of integer values using a bijection p. We define the abstract location $\mathfrak{fst}(X)$ as below. We omit $\mathfrak{snd}(X)$ which is similar, but only looks at the second projection, instead of the first.

$$\begin{array}{l}
\mathsf{h} \stackrel{\mathfrak{fst}(X)}{\sim} \mathsf{h}' \iff \exists a_1 a_2 a_1' a_2' \in \mathbb{Z}. \mathsf{h}(X) = p^{-1}(a_1, a_2) \wedge \\
\qquad\qquad\qquad \mathsf{h}'(X) = p^{-1}(a_1', a_2') \wedge a_1 = a_1' \\
\mathsf{h} \stackrel{\mathfrak{fst}(X)}{=} \mathsf{h}' \iff \mathsf{h} \stackrel{\mathfrak{fst}(X)}{\sim} \mathsf{h}' \\
\mathsf{h} \xrightarrow{\mathfrak{fst}(X)} \mathsf{h}_1 \iff \mathsf{h} : \mathfrak{fst}(X), \mathsf{h}_1 : \mathfrak{fst}(X) \text{ and} \\
\quad (\forall X' \neq X. \mathsf{h}(X') = \mathsf{h}_1(X')) \wedge (\forall a_1 a_2 a_1' a_2' \in \mathbb{Z}. \mathsf{h}(X) = p^{-1}(a_1, a_2) \wedge \\
\qquad\qquad \mathsf{h}_1(X) = p^{-1}(a_1', a_2') \Rightarrow a_2 = a_2')
\end{array}$$

The semantic (and strict) equivalence 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 step relation 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.

Version Numbers. The abstract location $\mathfrak{X}$ consists of two concrete locations X_{Val} and X_{Ver} and its relations are specified as follows:

$$\begin{array}{lcl}
\mathsf{h} \stackrel{\mathfrak{X}}{\sim} \mathsf{h}' & \iff & \mathsf{h}(X_{Val}) = \mathsf{h}'(X_{Val}) \\
\mathsf{h} \stackrel{\mathfrak{X}}{=} \mathsf{h}' & \iff & \mathsf{h} \stackrel{\mathfrak{X}}{\sim} \mathsf{h}' \\
\mathsf{h} \xrightarrow{\mathfrak{X}} \mathsf{h}_1 & \iff & \forall X' \notin \{X_{Ver}, X_{Val}\}. \mathsf{h}(X') = \mathsf{h}_1(X') \ \wedge \\
& & \mathsf{h} : \mathfrak{X} \wedge \mathsf{h}_1 : \mathfrak{X} \wedge \mathsf{h}(X_{Ver}) <= \mathsf{h}_1(X_{Ver}) \ \wedge \\
& & [\mathsf{h}(X_{Val}) \neq \mathsf{h}_1(X_{Val}) \Rightarrow \mathsf{h}(X_{Ver}) < \mathsf{h}_1(X_{Ver})]
\end{array}$$

Two heaps are semantically equivalent if they have the same value (independent of the version number). The step relation specifies that the version number does not descrease and it increases if the value changes.

Loop Parallelization. For a concrete location X, we introduce two concurrent abstract locations $\mathfrak{listeven}(X)$ and $\mathfrak{listodd}(X)$, which only look, respectively, at the elements in the even and odd positions of the linked list pointed to by X. Formally, let $L(X, \mathsf{h})$ denote that $\mathsf{h}(X)$ points to a well formed linked list of integers of length $L(X, \mathsf{h}).len$ and locations $L(X, \mathsf{h}).locs$ and that $L(X, \mathsf{h})[i]$ is the i^{th} node of the list for $1 \leq i \leq L(X, \mathsf{h}).len$. The relations for $\mathfrak{listeven}(X)$ are as below. We omit the relations for $\mathfrak{listodd}(X)$, which are similar.

$$
\begin{array}{lll}
\mathsf{h} \overset{\mathfrak{listeven}(X)}{\sim} \mathsf{h}' & \iff & L(X,\mathsf{h}) \wedge L(X,\mathsf{h}') \wedge L(X,\mathsf{h}).len = L(X,\mathsf{h}').len \wedge \\
 & & L(X,\mathsf{h})[2i] = L(X,\mathsf{h}')[2i] \\
 & & \text{for } 0 \leq i \leq \lfloor L(X,\mathsf{h}).len/2 \rfloor \\
\mathsf{h} \overset{\mathfrak{listeven}(X)}{=} \mathsf{h}' & \iff & \mathsf{h} \overset{\mathfrak{listeven}(X)}{\sim} \mathsf{h}' \\
\mathsf{h} \xrightarrow{\mathfrak{listeven}(X)} \mathsf{h}_1 & \iff & \mathsf{h} : \mathfrak{listeven}(X) \wedge \mathsf{h}_1 : \mathfrak{listeven}(X) \wedge \\
 & & L(X,\mathsf{h}).len = L(X,\mathsf{h}_1).len \\
 & & \text{for } 0 \leq i \leq \lfloor L(X,\mathsf{h}).len/2 \rfloor \\
 & & \quad L(X,\mathsf{h})[2i+1] = L(X,\mathsf{h}_1)[2i+1] \wedge \\
 & & \quad L(X,\mathsf{h})[2i].next = L(X,\mathsf{h}_1)[2i].next \wedge \\
 & & \forall X' \notin L(X,\mathsf{h}).locs. \mathsf{h}(X') = \mathsf{h}_1(X')
\end{array}
$$

The step relation $\mathsf{h} \xrightarrow{\mathfrak{listeven}(X)} \mathsf{h}_1$ specifies that $\mathsf{h} : \mathfrak{listeven}(X)$ and that h_1 arises from h by possibly modifying the list entries at even positions leaving everything else alone.

Michael-Scott queue. For concrete location X we introduce a concurrent abstract location $\mathfrak{msq}(X)$ first informally as follows: we have $\mathsf{h} \overset{\mathfrak{msq}(X)}{\sim} \mathsf{h}'$ if both h and h' contain a well-formed MSQ rooted at X and these queues contain the same entries in the same order. They may, however, use different locations for the nodes and also have different garbage tails.

The relation $\mathsf{h} \overset{\mathfrak{msq}(X)}{=} \mathsf{h}'$ asserts that h and h' are identical on the part reachable and co-reachable from X via *next* pointers. This means that while an MSQ operation is working on the queue no concurrent operation working elsewhere is allowed to relocate the queue or remove the garbage trail which would be the case if we merely required that such operations do not change the $\overset{MSQ(X)}{\sim}$-class.

The relation $\xrightarrow{\mathfrak{msq}(X)}$, finally, is defined as the transitive closure of the actions of operations on the MSQ: adding nodes at the tail and moving nodes from the head to the garbage tail.

We now give a formal definition. We represent pointers *head*, *next*, *elem* using some layout convention, e.g. $v.head = v.1$, etc. We then define

$$
\begin{array}{lll}
\mathsf{h}, X \overset{next}{\to} X' & \iff & X' \text{ can be reached from } X \text{ in } \mathsf{h} \\
 & & \text{by following a chain of next pointers}
\end{array}
$$

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 $v_1, \ldots v_n$. Note that the first node X_0 acts as a sentinel and its *elem* component is ignored. Formally:

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

We define $fp(X, \mathsf{h})$ as the set of locations reachable and co-reachable from X via *next*, formally:

$$fp(X, \mathsf{h}) = \{X' \mid X \overset{next}{\to} X' \vee X' \overset{next}{\to} X\}$$

Finally, we define $snoc(\mathsf{h}, \mathsf{h}', X, v)$ to mean that h' arises from h by attaching a new node containing v at the end of the list pointed to by X in h. Thus, in particular, $List(X, \mathsf{h}, (X_0, \ldots, X_n), (v_1 \ldots, v_n))$ implies $List(X, \mathsf{h}', (X_0, \ldots, X_n, X_{n+1}), (v_1 \ldots, v_n, v))$ for some $X_{n+1} \notin \mathrm{dom}(\mathsf{h})$. We omit the obvious frame conditions. We now define

$$\begin{array}{lcl} \mathsf{h} \overset{\mathfrak{msq}(X)}{\sim} \mathsf{h}' & \Longleftrightarrow & \exists \vec{X}\, \vec{X'}\, \exists \vec{v}. List(X, \mathsf{h}, \vec{X}, \vec{v}) \wedge List(X, \mathsf{h}', \vec{X'}, \vec{v}) \\ \mathsf{h} \overset{\mathfrak{msq}(X)}{=} \mathsf{h}' & \Longleftrightarrow & \mathsf{h} \overset{\mathfrak{msq}(X)}{\sim} \mathsf{h}' \wedge \forall X' \in fp(X, \mathsf{h}). \mathsf{h}(X') = \mathsf{h}'(X') \\ \mathsf{h} \xrightarrow{\mathfrak{msq}(X)} \mathsf{h}_1 & \Longleftrightarrow & \mathsf{h} : \mathfrak{msq}(X) \wedge \mathsf{h}_1 : \mathfrak{msq}(X) \wedge step^*(\mathsf{h}, \mathsf{h}_1) \\ step(\mathsf{h}, \mathsf{h}_1) & \Longleftrightarrow & \forall X' \neq X. \mathsf{h}(X') = \mathsf{h}_1(X') \wedge \\ & & [\mathsf{h}_1(X) = \mathsf{h}(X).next \vee \exists v. snoc(\mathsf{h}, \mathsf{h}_1, X, v)] \end{array}$$

In all of these examples, the only silent moves are identity moves. This is not so in the examples from [6] which contained data-structures that would reorganize during lookups and also patterns like late initialisation.

4.1. 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 allocation of concrete locations 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) Kripke logical relations [6, 4].

Definition 4.2 (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* $\mathsf{h} \overset{\mathfrak{l}}{\to} \mathsf{h}_1$ *then* $\mathsf{h} \overset{\mathfrak{l}'}{=} \mathsf{h}_1$ *holds for all* $\mathfrak{l}' \in \mathsf{w}$ *with* $\mathfrak{l}' \neq \mathfrak{l}$ *and* $\mathsf{h}_1 \models \mathsf{w}$.

The original account of abstract locations [6] also has a notion of independence of locations which facilitates 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.

We remark that if our world w contains two obviously "dependent" abstract locations, e.g. has both an integer location and a boolean location placed at the same physical location, then there will be no heap h such that $\mathsf{h} \models \mathsf{w}$.

We assume a fixed *current* world w which may appear in definitions without being notationally reflected. See also Assumption 1.

5. Effects

For each abstract location ł we have three elementary effects $rd_{\text{ł}}$ (reading from ł), $wr_{\text{ł}}$ (writing to ł), and $co_{\text{ł}}$ (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 $co_{\text{ł}}$ 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" $\texttt{unit}\ \&\ co_X \mid \varepsilon \mid \varepsilon \cup \{rd_X, wr_X\}$ whenever $X \notin \mathrm{locs}(\varepsilon)$. 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 alternatively be possible to replace the co-effect using a special set of private locations akin to the private regions from [12].

We use the notation $\mathrm{rds}(\varepsilon)$, $\mathrm{wrs}(\varepsilon)$, $\mathrm{cos}(\varepsilon)$ to refer to the abstract locations ł for which ε contains $rd_{\text{ł}}$, $wr_{\text{ł}}$, and $co_{\text{ł}}$, 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 all read effects removed and each $wr_{\text{ł}}$ in ε replaced by $co_{\text{ł}}$.

Definition 5.1. *An effect ε is well-formed (with respect to the current world) if* $\mathrm{locs}(\varepsilon) \subseteq \mathsf{w}$ *and* $\mathrm{rds}(\varepsilon) \cap \mathrm{cos}(\varepsilon) = \emptyset$ *and* $\mathrm{cos}(\varepsilon) \subseteq \mathrm{wrs}(\varepsilon)$*. An effect specification is a triple* $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ *of well-formed effects such that* $\varepsilon_2 \subseteq \varepsilon_3$*.*

An effect specification $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ approximates the behaviour of a computation e in the following way: the effect ε_1 summarizes side effects that may occur during the execution of e (corresponding to a guarantee condition in the rely-guarantee formalism [16]); the effect ε_2 summarizes effects of the interacting environment that e 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 e. 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. The requirement that $\mathrm{rds}(\varepsilon) \cap \mathrm{cos}(\varepsilon) = \emptyset$ is owed to the fact that all effects should preserve their own precondition, however the precondition of $rd_{\text{ł}}$ is agreement on ł which is not preserved by $co_{\text{ł}}$. The requirement $\mathrm{cos}(\varepsilon) \subseteq \mathrm{wrs}\varepsilon$ reflects the fact that $\mathrm{cos}(\text{ł})$ includes $wr_{\text{ł}}$ as a special case.

Note that if $\varepsilon^C \cup \varepsilon_1$ is a (well-formed) effect, then it is the case that $\mathrm{rds}(\varepsilon_1) \cap (\mathrm{wrs}(\varepsilon) \cup \mathrm{cos}(\varepsilon)) = \emptyset$. We will use this observation to simplify some side conditions.

In our concrete examples, we abbreviate $\{co_{\text{ł}}\} \cup \{wr_{\text{ł}}\}$ by just $co_{\text{ł}}$, in other words, the chaotic effect silently implies the write effect.

Consider the computations $e_1 = X := !X + 1; X := !X + 1$ and $e_2 = X := !X + 2$. 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 7.5.

Finally, consider the program $e = !X$ that simply reads a location storing an integer. We can show that this program has type $\mathbb{Z} \ \& \ \emptyset \mid \varepsilon \mid \varepsilon, rd_X$, where the read effect on X is only in the global effects.

Notations.. 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. We write $\mathsf{h} \overset{\mathrm{rds}(\varepsilon)}{\sim} \mathsf{h}'$ to mean $\mathsf{h} \overset{\text{ł}}{\sim} \mathsf{h}'$ for each $\text{ł} \in \mathrm{rds}(\varepsilon)$. We write $\xrightarrow{\varepsilon}$ for the transitive closure of $\bigcup_{\text{ł} \in \mathrm{wrs}(\varepsilon)} \xrightarrow{\text{ł}} \cup \bigcup_{\text{ł} \in \mathsf{w}} \xrightarrow{\text{ł}} \cap \overset{\text{ł}}{=}$. Thus, $\xrightarrow{\varepsilon}$ allows steps by locations recorded as writing in ε and silent steps by all locations in the current world.

We define the notation $\varepsilon_1 \sqcup \varepsilon_2$ which appears in the parallel congruence rule by

$$\varepsilon_1 \sqcup \varepsilon_2 = \varepsilon_1 \cup \varepsilon_2 \setminus \{wr_\ell \mid wr_\ell \notin \varepsilon_1 \cap \varepsilon_2\} \setminus \{co_\ell \mid co_\ell \notin \varepsilon_1 \cap \varepsilon_2\}$$

6. Typing and congruence rules

Types are given by the grammar

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

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 $\tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2$ 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',

- $\Gamma \vdash e \leq e' : \tau \ \& \ \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$ specifying that the programs e and e' 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, e approximates e' 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 assume that whenever (v, v', τ) an axiom, then so are (v, v, τ) and (v', v', τ).

We also define typing judgements $\Gamma \vdash v : \tau$ and $\Gamma \vdash e : \tau \ \& \ \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$ which denote the special case when $\Gamma \vdash v \leq v : \tau$ and $\Gamma \vdash e \leq e : \tau \ \& \ \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$ can be derived from the rules from Figure 6. We do not formulate explicit typing rules to save space.

The plan is to justify all the rules semantically using a logical relation (Section 7) and to then conclude their soundness w.r.t. typed observational appoximation and equivalence (Section 8).

The parallel composition rule states that two programs e_1 and e_2 can be composed when their internal effects are not conflicting in the sense that the internal effects of one program appear as environment interaction effects of the other program. Note the relationship to the parallel composition rule of the rely-guarantee formalism [16]. Also note that the effects of computations e_1 and e_2 are not required to be independent from each other as we do in the parallization rule further down.

The appearance of the $\sqcup$-operation deserves special mention. It might be, for example, that e_1 modifies X on the way, thus $wr_X \in \varepsilon_1$ but cleans up this modification by eventually restoring the old value of X. This would be reflected by $wr_X \notin \varepsilon \cup \varepsilon' \cup \varepsilon_2$. In that case, we would not expect to see wr_X in the end-to-end effect of the parallel composition and that is precisely what $\sqcup$ achieves.

The rules labelled (Sem) make available all kinds of program transformations that are valid on the level of the *untyped* denotational semantics, including commuting conversions for let and if, fixpoint unrolling, and beta and eta equalities.

Finally, we have several effect-dependent (in)equalities: the parallelization rule generalises a similar rule from [12]. The other ones are concurrent version of analogous rules for sequential computation that have been analysed in previous work [9, 8, 29, 6] and are at the basis of all kinds of compiler optimizations. The side conditions on the effects are rather subtle and much less obvious than those found in a sequential setting. The parallelization rule is similar to the parallel congruence rule in that it requires the participating computations to mutually tolerate each other. This time, however, since the two computations being compared will do rather different things temporarily they must be oblivious against chaotic access, hence the $(-)^C$ strengthenings in the premise.

The reason for the appearance of $(-)^C$ in the other rules is similar. The rule for pure lambda hoist seems unusual and will thus be explained in more detail. First, the computation e_1 to be hoisted may indeed have side effects ε_1 so long as they are cleaned up by the time e_1 completes and the intervening environment does not notice (modelled by the conditions $\varepsilon_1 \perp \varepsilon$ and final effect $\varepsilon^C = \varepsilon^C \cup \emptyset$). In the conclusion the transient effect ε_1 shows up again, but $(-)^C$-ed since it only appears in different sides. Also in the other rules like commuting etc. it is the case that the familiar side conditions on applicability only affect the end-to-end effects whereas the transient effects are merely required not to interfere with the environment.

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

$$\frac{}{\Gamma, x : \tau \vdash x \le x : \tau} \qquad \frac{\Gamma \vdash v \le v' : \tau}{\Gamma \vdash v \le v' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3} \qquad \frac{\Gamma \vdash v \le v' : \tau_1 \times \tau_2}{\Gamma \vdash v.i \le v'.i : \tau_i}$$

$$\frac{\Gamma \vdash e_1 \le e_2 : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad \Gamma \vdash e_1 \le e_2 : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}{\Gamma \vdash e_1 \le e_3 : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}$$

$$\frac{\Gamma \vdash v_i \le v_i' : \tau_1 \ i = 1, 2}{\Gamma \vdash (v_1, v_2) \le (v_1', v_2') : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash v_1 \le v_1' : \tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2 \quad \Gamma \vdash v_2 \le v_2' : \tau_1}{\Gamma \vdash v_1\ v_2 \le v_1'\ v_2' : \tau_2 \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}$$

$$\frac{\begin{array}{c}\Gamma \vdash v \le v' : \mathtt{bool} \\ \Gamma \vdash e_1 \le e_1' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \ \ \Gamma \vdash e_2 \le e_2' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3\end{array}}{\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \le\ \mathtt{if}\ v'\ \mathtt{then}\ e_1'\ \mathtt{else}\ e_2' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \le e_1' : \tau_1 \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \\ \Gamma, x{:}\tau_1 \vdash e_2 \le e_2' : \tau_2 \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3\end{array}}{\Gamma \vdash \mathtt{let}\ x{=}e_1\ \mathtt{in}\ e_2 \le \mathtt{let}\ x{=}e_1'\ \mathtt{in}\ e_2' : \tau_2 \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}$$

$$\frac{\Gamma \vdash e_1 \le e_1' : \tau_1 \mathbin{\&} \varepsilon_1 \mid \varepsilon \cup \varepsilon_2 \mid \varepsilon \cup \varepsilon_2 \cup \varepsilon' \quad \Gamma \vdash e_2 \le e_2' : \tau_2 \mathbin{\&} \varepsilon_2 \mid \varepsilon \cup \varepsilon_1 \mid \varepsilon \cup \varepsilon_1 \cup \varepsilon'}{\Gamma \vdash e_1 \| e_2 \le e_1' \| e_2' : \tau_1 \times \tau_2 \mathbin{\&} \varepsilon_1 \cup \varepsilon_2 \mid \varepsilon \mid \varepsilon \cup \varepsilon' \cup (\varepsilon_1 \sqcup \varepsilon_2)}$$

$$\frac{\Gamma \vdash e_1 \le e_2 : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad [\![e_1]\!] = [\![e_1']\!] \quad [\![e_2]\!] = [\![e_2']\!]}{\Gamma \vdash e_1' \le e_2' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}\ \text{Sem}$$

$$\frac{\Gamma, f{:}\tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2, x{:}\tau_1 \vdash e \le e' : \tau_2 \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}{\Gamma \vdash \mathtt{rec}\ f\ x = e \le \mathtt{rec}\ f\ x = e' : \tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2} \qquad \frac{(v, v', \tau)\ \text{an axiom}}{\Gamma \vdash v \le v' : \tau}\ \text{Ax}$$

$$\frac{\Gamma \vdash e \le e' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad \varepsilon_1 \subseteq \varepsilon_1' \quad \varepsilon_2' \subseteq \varepsilon_2 \quad \varepsilon_3 \subseteq \varepsilon_3'}{\Gamma \vdash e \le e' : \tau \mathbin{\&} \varepsilon_1' \mid \varepsilon_2' \mid \varepsilon_3'}$$

$$\frac{\Gamma \vdash e \le e' : \tau \mathbin{\&} \varepsilon_1 \mid \emptyset \mid \varepsilon_3}{\Gamma \vdash \mathtt{atomic}(e) \le \mathtt{atomic}(e') : \tau \mathbin{\&} \varepsilon_3 \mid \varepsilon_2 \mid \varepsilon_2 \cup \varepsilon_3}\ \text{Atom}$$

Figure 4: Typing and congruence rules

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau_1 \ \& \ \varepsilon_1 \mid \varepsilon^C \cup \varepsilon_2^C \mid \varepsilon^C \cup \varepsilon_2^C \cup \varepsilon_1' \\ \Gamma \vdash e_2 : \tau_2 \ \& \ \varepsilon_2 \mid \varepsilon^C \cup \varepsilon_1^C \mid \varepsilon^C \cup \varepsilon_1^C \cup \varepsilon_2' \\ \varepsilon_1 \perp \varepsilon_2 \quad \varepsilon_1 \perp \varepsilon \quad \varepsilon_2 \perp \varepsilon\end{array}}{\Gamma \vdash e_1 \| e_2 \leq (\texttt{let}\ x = e_1\ \texttt{in}\ \texttt{let}\ y = e_2\ \texttt{in}\ (x,y)) : \tau_1 \times \tau_2 \ \& \ \varepsilon_1^C \cup \varepsilon_2^C \mid \varepsilon \mid \varepsilon \cup \varepsilon_1' \cup \varepsilon_2'} \text{ Parallelization}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau_1 \ \& \ \varepsilon_1 \mid \varepsilon^C \mid \varepsilon^C \cup \varepsilon_1' \\ \Gamma \vdash e_2 : \tau_2 \ \& \ \varepsilon_2 \mid \varepsilon^C \mid \varepsilon^C \cup \varepsilon_2' \\ \varepsilon_1' \perp \varepsilon_2' \quad \varepsilon_1 \perp \varepsilon \quad \varepsilon_2 \perp \varepsilon\end{array}}{\begin{array}{l}\Gamma \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ \texttt{let}\ y = e_2\ \texttt{in}\ (x,y)) = \\ \qquad (\texttt{let}\ y = e_2\ \texttt{in}\ \texttt{let}\ x = e_1\ \texttt{in}\ (x,y)) : \tau_1 \times \tau_2 \ \& \ \varepsilon_1^C \cup \varepsilon_2^C \mid \varepsilon \mid \varepsilon \cup \varepsilon_1' \cup \varepsilon_2'\end{array}} \text{ Commuting}$$

$$\frac{\Gamma \vdash e : \tau \ \& \ \varepsilon_1 \mid \varepsilon_2^C \mid \varepsilon_2^C \cup \varepsilon' \quad \mathrm{rds}(\varepsilon') \cap \mathrm{wrs}(\varepsilon') = \emptyset \quad \varepsilon_2 \perp \varepsilon_1}{\begin{array}{l}\Gamma \vdash (\texttt{let}\ x = e\ \texttt{in}\ (x,x)) \leq \\ \qquad (\texttt{let}\ x = e\ \texttt{in}\ \texttt{let}\ y = e\ \texttt{in}\ (x,y))) : \tau \times \tau \ \& \ \varepsilon_1^C \mid \varepsilon_2 \mid \varepsilon_2 \cup \varepsilon'\end{array}} \text{ Duplicated}$$

$$\frac{(v, v', \tau) \text{ an axiom}}{\Gamma \vdash v \leq v' : \tau} \text{ Ax}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau_1 \ \& \ \varepsilon_1 \mid \varepsilon^C \mid \varepsilon^C \\ \Gamma, x : \tau_3, y : \tau_1 \vdash e_2 : \tau_2 \ \& \ \varepsilon_2 \mid \varepsilon \mid \varepsilon \cup \varepsilon_2 \\ \varepsilon \perp \varepsilon_1\end{array}}{\Gamma \vdash \texttt{let}\ y = e_1\ \texttt{in}\ \lambda x.e_2 \leq \lambda x.\texttt{let}\ y = e_1\ \texttt{in}\ e_2 : \tau_3 \xrightarrow[\varepsilon]{\varepsilon_1^C \cup \varepsilon_2 \mid \varepsilon \cup \varepsilon_3} \tau_2 \ \& \ \varepsilon_1^C \mid \varepsilon \mid \varepsilon} \text{ Lambda Hoist}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau_1 \ \& \ \varepsilon_1 \mid \varepsilon^C \mid \varepsilon^C \cup \varepsilon_1' \\ \Gamma \vdash e_2 : \tau_2 \ \& \ \varepsilon_2 \mid \varepsilon \mid \varepsilon_2' \\ \varepsilon_1 \perp \varepsilon \\ \mathrm{wrs}(\varepsilon_1') = \emptyset\end{array}}{\Gamma \vdash e_2 \leq (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \tau_2 \ \& \ \varepsilon_1^C \cup \varepsilon_2 \mid \varepsilon \mid \varepsilon \cup \varepsilon_2'} \text{ Deadcode}$$

Figure 5: Effect-dependent transformations.

The following definitions provide the semantics of our effect annotations.

Definition 6.1 (Tiling)**.** *Let* $\mathsf{w} \vdash \varepsilon$. *We write* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}'_1)$ *to mean that (i)* $\mathsf{h} \models \mathsf{w} \Rightarrow \mathsf{h} \xrightarrow{\varepsilon} \mathsf{h}_1$ *and (ii)* $\mathsf{h}' \models \mathsf{w} \Rightarrow \mathsf{h}' \xrightarrow{\varepsilon} \mathsf{h}'_1$ *and (iii)* $\mathsf{h} \overset{\mathrm{rds}(\varepsilon)}{\sim} \mathsf{h}'$ *and* $\text{ł} \in \mathrm{wrs}(\varepsilon) \setminus \cos(\varepsilon)$ *imply* $(\mathsf{h} \overset{\text{ł}}{=} \mathsf{h}_1 \wedge \mathsf{h}' \overset{\text{ł}}{=} \mathsf{h}'_1) \vee \mathsf{h}_1 \overset{\text{ł}}{\sim} \mathsf{h}'_1$.

Thus, assuming semantic consistency of heaps, h and h' evolve to h_1 and h'_1 according to the modifying (writing or chaotic) locations in ε, and if h, h' agree on the reads of ε then written locations will either be identicallly modified or left alone.

If the step relations of all abstract locations commute with each other then tiling admits an alternative characterisation in terms of preservation of binary relations [9]. The present more operational version is inspired by the treatment of effects in [12].

Lemma 6.2. *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. $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}'_1) \Rightarrow [\varepsilon^C](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}'_1)$
5. *If* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{k}, \mathsf{k}')$ *and* $\mathsf{h} \overset{\mathrm{rds}(\varepsilon)}{\sim} \mathsf{h}'$ *then* $\mathsf{k} \overset{\mathrm{rds}(\varepsilon)}{\sim} \mathsf{k}'$. *(this relies on* $\mathrm{rds}(\varepsilon) \cap \cos(\varepsilon) = \emptyset$*.)*
6. *Suppose* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}'_1)$. *If* $\mathsf{h} \models \mathsf{w}$ *then* $\mathsf{h}_1 \models \mathsf{w}$*; if* $\mathsf{h}' \models \mathsf{w}$ *then* $\mathsf{h}'_1 \models \mathsf{w}$.

7. Logical Relation

Definition 7.1 (Specifications)**.** *A value specification is a relation* $E \subseteq \mathbb{V} \times \mathbb{V}$ *such that*

- *if* $x_1 \leq x$ *and* $y \leq y_1$ *and* $x\,E\,y$ *then* $x_1\,E\,y_1$ *(in short thus* $\leq; E; \leq \subseteq E$*);*
- *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 an admissible subset of* $\mathbb{V} \times \mathbb{V}$*;*
- *if* $x\,E\,y$ *then* $p_i(x)\,E\,p_i(y)$ *for each i, i.e. E is closed under the canonical deflations.*

Similarly, a computation specification is an admissible subset of $T\mathbb{V} \times T\mathbb{V}$ *such that the relation* $Q \subseteq T\mathbb{V} \times T\mathbb{V}$*,* $\leq; Q; \leq \subseteq Q$ *and Q is closed under the canonical deflations* q_i*.*

The requirement $\leq; E; \leq \,\subseteq E$ ensures smooth interaction with the down-closure built into our trace monad. Admissibility is needed for the soundness of recursion and closure under the canonical deflations, finally is needed so that Lemma 3.3 can be applied.

Definition 7.2. *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).*

Lemma 7.3. *If E and Q are specifications so is $E \to Q$.*

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 7.4. *Let $E \subseteq \mathbb{V} \times \mathbb{V}$ be a value specification and $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ 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 $\mathcal{P}(Tr \times Values)$:*

$(U, U') \in T_0(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ if and only if

$$\left[\begin{array}{l} \forall((\mathsf{h}_1, \mathsf{k}_1)\dots(\mathsf{h}_n, \mathsf{k}_n), a) \in U. \mathsf{h}_1 \models \mathsf{w} \Rightarrow \\ \quad \forall \mathsf{h}'_1 . \mathsf{h}'_1 \models \mathsf{w} \Rightarrow \mathsf{h}_1 \overset{\mathrm{rds}(\varepsilon_3)}{\sim} \mathsf{h}'_1 \Rightarrow \\ \qquad \exists \mathsf{k}'_1 . [\varepsilon_1](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{k}_1, \mathsf{k}'_1) \wedge \forall \mathsf{h}'_2 . [\varepsilon_2](\mathsf{k}_1, \mathsf{k}'_1, \mathsf{h}_2, \mathsf{h}'_2) \Rightarrow \\ \qquad\quad \exists \mathsf{k}'_2 . [\varepsilon_1](\mathsf{h}_2, \mathsf{h}'_2, \mathsf{k}_2, \mathsf{k}'_2) \wedge \forall \mathsf{h}'_3 . [\varepsilon_2](\mathsf{k}_2, \mathsf{k}'_2, \mathsf{h}_3, \mathsf{h}'_3) \Rightarrow \\ \qquad\qquad \dots \\ \qquad\qquad \exists \mathsf{k}'_n . [\varepsilon_1](\mathsf{h}_n, \mathsf{k}_n, \mathsf{h}'_n, \mathsf{k}'_n) \wedge \ [\varepsilon_3](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{k}_n, \mathsf{k}'_n) \wedge \\ \qquad\qquad \exists a' \in \mathbb{V}.(a, a') \in E \wedge ((\mathsf{h}'_1, \mathsf{k}'_1)\dots(\mathsf{h}'_n, \mathsf{k}'_n), a') \in U' \end{array}\right]$$

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 $((\mathsf{h}_1, \mathsf{k}_1)\dots(\mathsf{h}_n, \mathsf{k}_n), a) \in U$ and $\mathsf{h}_1 \models \mathsf{w}$, the *pilot trace* and a start heap $\mathsf{h}'_1 \models \mathsf{w}$ such that $\mathsf{h}_1 \overset{\mathrm{rds}(\varepsilon_3)}{\sim} \mathsf{h}'_1$ to begin a trace in U'. Then, the proponent answers with a matching heap k'_1 so that $[\varepsilon_1](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{k}_1, \mathsf{k}'_1)$. If $\mathsf{h}_1 \overset{\mathrm{rds}(\varepsilon_1)}{\sim} \mathsf{h}'_1$ does not hold, proponent does not need to ensure that writes are in sync. The opponent then plays a heap h'_2 so that $[\varepsilon_2](\mathsf{k}_1, \mathsf{k}'_1, \mathsf{h}_2, \mathsf{h}'_2)$. At this point, it is in the proponents interest to make sure that $\mathsf{k}_1 \overset{\mathrm{rds}(\varepsilon_2)}{\sim} \mathsf{k}'_1$ for otherwise opponent may make "funny" moves.

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 $[\varepsilon_1](\mathsf{h}_n, \mathsf{h}'_n, \mathsf{k}_n, \mathsf{k}'_n)$. After that final heap has been played, it is checked that $[\varepsilon_3](\mathsf{h}, \mathsf{h}', \mathsf{k}_n, \mathsf{k}'_n)$ holds. If not, proponent loses. If yes, then proponent must also play a value a' and it is then checked whether or not $((\mathsf{h}'_1, \mathsf{k}'_1)\dots(\mathsf{h}'_n, \mathsf{k}'_n), a') \in U'$ 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.

We notice that by Lemma 6.2(6) well-formedness of heaps w.r.t. the ambient world is a global invariant which allows us to refrain form explicitly assuming and asserting it in subsequent proofs and statements.

We now illustrate the game with a few examples.

Example 7.5. Consider the following programs:

$$e_1 = (X := !X + 1; X := !X + 1) \quad \text{and} \qquad e_2 = (X := !X + 2).$$

Let $ł = \mathrm{int}(X)$ be the abstract location for a single integer stored at X (see Section 4). Let $E = [\![\texttt{unit}]\!] = \{((), ())\}$ be the value specification for the unit type.

We show that $([\![e_1]\!], [\![e_2]\!]) \in T(E, \{co_ł\}, \varepsilon, \varepsilon \cup \{rd_ł, wr_ł\}\}$ under the assumption that $\{co_ł\} \perp \varepsilon$, that is, when the environment does not read nor write X. This condition is clearly necessary, for e_1 and e_2 can be distinguished by an environment allowed to read or write X.

Let us now prove the claim when $\{co_ł\} \perp \varepsilon$. The opponent picks a pilot trace in the semantics of e_1, for example, $((\mathsf{h}_1, \mathsf{k}_1)(\mathsf{h}_2, \mathsf{k}_2), ())$ where $\mathsf{h}_1(X) = n$ and $\mathsf{k}_1(X) = n + 1$ and $\mathsf{h}_2(X) = n'$ and $\mathsf{k}_2(X) = n' + 1$. The other possible traces are stuttering or mumbling variants of this one and do not present additional difficulties. The opponent also chooses a heap h'_1 such that $\mathsf{h}_1 \overset{ł}{\sim} \mathsf{h}'_1$, i.e., $\mathsf{h}'_1(X) = n$. Now the proponent will choose to stutter for the time being and thus selects $\mathsf{k}'_1 := \mathsf{h}'_1$. Indeed, $[co_ł](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{k}_1, \mathsf{k}'_1)$ holds, so this is legal. The opponent now presents h'_2 such that $[\varepsilon](\mathsf{k}_1, \mathsf{k}'_1, \mathsf{h}_2, \mathsf{h}'_2)$. By the assumption on ε we know that $n' = \mathsf{h}_2(X) = \mathsf{k}_1(X) = n+1$ and also $\mathsf{h}'_2(X) = \mathsf{k}'_1(X) = n$. The proponent now answers with $\mathsf{k}'_2 := \mathsf{h}'_2[X{\mapsto}n+2]$. It follows that $[co_ł](\mathsf{h}_2, \mathsf{h}'_2, \mathsf{k}_2, \mathsf{k}'_2)$ and also $[rd_ł, wr_ł](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{k}_2, \mathsf{k}'_2)$. Finally, by stuttering $(\mathsf{h}'_1, \mathsf{h}'_1)(\mathsf{h}'_2, \mathsf{h}'_2[X{\mapsto}n+2]) \in [\![e_2]\!]$ so that proponent wins the game.

Example 7.6. Consider the following programs e_1 and e_2:

$$(X := !X + 1 \| Y := !Y + 1) \quad \text{and} \quad (X := !X + 1; Y := !Y + 1).$$

We show $([\![e_1]\!], [\![e_2]\!]) \in T(E, \{co_X, co_Y\}, \varepsilon, \varepsilon \cup \{rd_X, rd_Y, wr_X, wr_Y\})$, provided ε does not read nor modify X and Y. This equivalence could be deduced syntactically using our parallelization equation shown in Figure 5. For illustrative purpose, however, we describe its semantic proof using a game.

The opponent picks a pilot trace in $[\![e_1]\!]$, for example, the trace $([n_1|n_2], [n_1|n_2 + 1])([n_1|n_2+1], [n_1+1|n_2+1])((), ())$, where $[n_X|n_Y]$ denotes a heap where X and Y store n_X and n_Y, respectively. Notice that in this trace, Y is incremented before X and since ε does not read nor modify X and Y, the environment move does not change the values in X nor Y. We are also given an initial heap h'_1 that agrees with the initial heap $[n_1|n_2]$ on the reads of $\varepsilon \cup \{rd_X, rd_Y, wr_X, wr_Y\}$. Thus, h'_1 should be of the form $[n_1|n_2]$.

We now play the move $([n_1|n_2], [n_1 + 1|n_2])$. This is a valid move in the game as $[co_X, co_Y]([n_1|n_2], [n_1|n_2], [n_1|n_2 + 1], [n_1 + 1|n_2])$. The environment moves returning $[n_1 + 1|n_2]$ as it does not read nor modify X and Y. We can now match the trace above by playing $([n_1 + 1|n_2], [n_1 + 1|n_2 + 1])$ and returning $((), ())$, winnning the game.

The following is one of 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.

Theorem 7.7. *The following hold whenever well-formed.*

1. *If* $(U, U') \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *then* $(q_i(U), q_i(U')) \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$.
2. $T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *is a computation specification.*
3. *If* $(U, U') \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *then* $(U^\dagger, U'^\dagger) \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$.
4. *If* $(a, a') \in E$ *then* $(rtn(a), rtn(a'))$ *is in* $T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$.
5. *Suppose that* $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ *is an effect specification where* $\varepsilon_1 \cup \varepsilon_2 \subseteq \varepsilon_3$. *Suppose that whenever* $\mathsf{h} \overset{\mathrm{rds}(\varepsilon_1)}{\sim} \mathsf{h}'$ *and* $c(\mathsf{h}) = (\mathsf{h}_1, a)$ *then there exist* (h'_1, a') *such that* $c'(\mathsf{h}') = (\mathsf{h}'_1, a')$ *and* $[\varepsilon_1](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}'_1)$ *and* aEa'. *We then have for any* ε_2, $(\mathit{fromstate}(c), \mathit{fromstate}(c')) \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$.

6. *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*
$$(bnd(f, U), bnd(f', U')) \in T(E_2, \varepsilon_1, \varepsilon_2, \varepsilon_3)$$
7. *If* $(U_1, U_1') \in T(E_1, \varepsilon_1, \varepsilon \cup \varepsilon_2, \varepsilon \cup \varepsilon_2 \cup \varepsilon')$ *and* $(U_2, U_2') \in T(E_2, \varepsilon_2, \varepsilon \cup \varepsilon_1, \varepsilon \cup \varepsilon_1 \cup \varepsilon')$ *then*
$$(U_1 \mid U_1', U_2 \mid U_2') \in T(E_1 \times E_2, \varepsilon_1 \cup \varepsilon_2, \varepsilon, \varepsilon \cup \varepsilon' \cup (\varepsilon_1 \sqcup \varepsilon_2))$$
8. $(U, U') \in T(E, \varepsilon_1, \emptyset, \varepsilon_3) \Rightarrow (at(U), at(U')) \in T(\varepsilon_3, \varepsilon_2, \varepsilon_2 \cup \varepsilon_3)$.

Proof. In each case, using Corollary 3.2 and Lemma 3.3 (for case 6), 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 $(U, U') \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ 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 This is an easy consequence from 1.

Ad 3 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 6.2) 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 $[\varepsilon_3](\mathsf{h}_1, \mathsf{h}_1', \mathsf{h}, \mathsf{h}')$ knowing $[\varepsilon_3](\mathsf{h}_1, \mathsf{h}_1', \mathsf{k}_n, \mathsf{k}_n')$ where $u = (\mathsf{h}_1, \mathsf{k}_1) \dots (\mathsf{h}_n, \mathsf{k}_n)$ and $u' = (\mathsf{h}_1', \mathsf{k}_1') \dots (\mathsf{h}_n', \mathsf{k}_n')$ is the matching trace. We have $[\varepsilon_2](\mathsf{k}_n, \mathsf{k}_n', \mathsf{h}, \mathsf{h}')$ for otherwise opponent's playing h' would have been illegal. Since, by assumption $\varepsilon_2 \subseteq \varepsilon_3$, we can conclude $[\varepsilon_3](\mathsf{k}_n, \mathsf{k}_n', \mathsf{h}, \mathsf{h}')$ and then $[\varepsilon_3](\mathsf{h}_1, \mathsf{h}_1', \mathsf{h}, \mathsf{h}')$ by Lemma 6.2(3&1).

If (t, a) arises by mumbling then we must have $t = u(\mathsf{h}_1, \mathsf{h}_3)v$ and $t_1 = u(\mathsf{h}_1, \mathsf{h}_2)(\mathsf{h}_2, \mathsf{h}_3)v$. 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 6.2) 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 aEa_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 4. Suppose aEa'. By 3 which we have just proved we only need to match elements of the form $((\mathsf{h}, \mathsf{h})a)$. The opponent plays h' where $\mathsf{h} \overset{\mathrm{rds}(\varepsilon_3)}{\sim} \mathsf{h}'$ and we answer with h' itself and a'. This is always a legal move (Lemma 6.2) and aEa', so we win the game.

Ad 5. 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' where $\mathsf{h} \overset{\varepsilon_3}{\sim} \mathsf{h}'$. The assumption gives (h_1', a') such that $c'(\mathsf{h}') = (\mathsf{h}_1', a')$ and $[\varepsilon_1](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$ and aEa'. We thus play h_1' and a' and indeed $[\varepsilon_{1/3}](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$ and aEa' hold so this is a winning move.

Ad 6. Suppose $(f, f') \in E_1 \to T(E_2, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ and $(U, U') \in T(E_1, \varepsilon_1, \varepsilon_2, \varepsilon_3)$. Suppose that $(uv, b) \in ap(f, U)$ where $(u, a) \in U$ and (v, b) in $f(a)$ (note that we can ignore the †-closure). We need to produce a trace $(u'v', b') \in ap(f', U')$ such that $(u', a') \in U'$ and (v', b') in $f'(a')$ and bE_2b'. Assume that:

$$u = (\mathsf{h}_1, \mathsf{k}_1) \cdots (\mathsf{h}_n, \mathsf{k}_n) \text{ and } v = (\mathsf{h}_{n+1}, \mathsf{k}_{n+1}) \cdots (\mathsf{h}_{n+m}, \mathsf{k}_{n+m})$$

We are given a heap h_1', such that $\mathsf{h}_1 \overset{\mathrm{rds}(\varepsilon_3)}{\sim} \mathsf{h}_1'$. We can use the strategy S_1 from $(U, U') \in T(E_1, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ for (u, a). We play according to S_1 to work off the u-part. This results in a matching trace $u' \in U'$:

$$u' = (\mathsf{h}_1', \mathsf{k}_1') \cdots (\mathsf{h}_n', \mathsf{k}_n')$$

where $[\varepsilon_3](\mathsf{h}_1, \mathsf{h}_1', \mathsf{k}_n, \mathsf{k}_n')$ and $(a, a') \in E_2$. We get $(f(a), f(a')) \in T(E_2, \varepsilon_1, \varepsilon_2, \varepsilon_3)$. Now, we are given a heap h_{n+1}' that is an environment move forming the tile

$$[\varepsilon_2](\mathsf{k}_n, \mathsf{k}_n', \mathsf{h}_{n+1} \mathsf{h}_{n+1}')$$

From the fact that $\varepsilon_2 \subseteq \varepsilon_3$ and Lemma 6.2(5) we can conclude $\mathsf{h}_{n+1} \overset{\mathrm{rds}(\varepsilon_3)}{\sim} \mathsf{h}_{n+1}'$.

Thus we can continue our play by using the strategy S_2 from

$$(f(a), f(a')) \in T(E_2, \varepsilon_1, \varepsilon_2, \varepsilon_3)$$

which yields a continuation v' of our trace and a final answer b'. It is then clear that $(u'v', b') \in bnd(f', U')$ so this combination of strategies does indeed win.

Ad 7. Suppose that $(U_1, U_1') \in T(E_1, \varepsilon_1, \varepsilon \cup \varepsilon_2, \varepsilon \cup \varepsilon_2 \cup \varepsilon')$ and $(U_2, U_2') \in T(E_2, \varepsilon_2, \varepsilon \cup \varepsilon_1, \varepsilon \cup \varepsilon_1 \cup \varepsilon')$ and let $(t, (a, b)) \in U_1 \mid U_2$, thus $inter(t_1, t_2, t)$ (ignoring † by item 3) 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 t be of the form:

$$\begin{aligned} t = \quad & (\mathsf{h}_1, \mathsf{k}_1) \cdots (\mathsf{h}_n, \mathsf{k}_n)(\mathsf{h}_{n+1}, \mathsf{k}_{n+1}) \cdots (\mathsf{h}_{n+m}, \mathsf{k}_{n+m}) \\ & (\mathsf{h}_{n+m+1}, \mathsf{k}_{n+m+1}) \cdots (\mathsf{h}_{n+m+k}, \mathsf{k}_{n+m+k}) \cdots (\mathsf{h}_p, \mathsf{k}_p) \end{aligned}$$

composed of pieces of the traces t_1 and t_2. Assume w.l.o.g. that the first piece $(\mathsf{h}_1, \mathsf{k}_1)\cdots \cdots(\mathsf{h}_n, \mathsf{k}_n)$ is a part of t_1. We are given a initial heap h'_1 such that $\mathsf{h} \overset{\mathrm{rds}(\varepsilon\cup\varepsilon'\cup(\varepsilon_1\sqcup\varepsilon_2))}{\sim} \mathsf{h}'$. Since $\mathrm{rds}(\varepsilon_1 \sqcup \varepsilon_2) = \mathrm{rds}(\varepsilon_1) \cup \mathrm{rds}(\varepsilon_2)$, we can apply strategy S_1 to guide us through the first part of the game, obtaining:

$$(\mathsf{h}'_1, \mathsf{k}'_1)\cdots(\mathsf{h}'_n, \mathsf{k}'_n)$$

Moreover, we have an environment move which forms the tile $[\varepsilon](\mathsf{k}_n, \mathsf{k}'_n, \mathsf{h}_{n+1}, \mathsf{h}_{n'+1})$. Thus, we have the tile $[\varepsilon \cup \varepsilon_1](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{h}_{n+1}, \mathsf{h}'_{n+1})$ which can be seen as an environment move for t_2. Therefore, we can use strategy S_2 for the U' and continue the game, obtaining the trace piece:

$$(\mathsf{h}'_{n+1}, \mathsf{k}'_{n+1})\cdots(\mathsf{h}'_{n+m}, \mathsf{k}'_{n+m})$$

Now, we can return to the S_1 game as the trace above is seen as an environment move for U. Alternating these strategies, we get a trace t which is in $(U \mid U')$. Let (a', b') be the final values reached at the end. It is clear that $[\varepsilon \cup \varepsilon' \cup \varepsilon_1 \cup \varepsilon_2](\mathsf{h}, \mathsf{h}', \mathsf{h}_p, \mathsf{h}'_p)$ and also aE_1a' and bE_2b'.

It remains to assert the stronger statement $[\varepsilon \cup \varepsilon' \cup (\varepsilon_1 \sqcup \varepsilon_2)](\mathsf{h}, \mathsf{h}', \mathsf{h}_p, \mathsf{h}'_p)$. To see this suppose that $wr_{\text{ł}} \in \varepsilon_1 \setminus \varepsilon_2 \setminus \varepsilon \setminus \varepsilon'$. Since the entire game can be viewed as an instance of the game U_1 vs U'_1 with interventions by U_2 vs. U'_2 regarded as environment interactions we have $[\varepsilon \cup \varepsilon_2 \cup \varepsilon'](\mathsf{h}, \mathsf{h}', \mathsf{h}_p, \mathsf{h}'_p)$ so that in fact $\mathsf{h} \overset{\text{ł}}{=} \mathsf{h}_p$ and $\mathsf{h}' \overset{\text{ł}}{=} \mathsf{h}'_p$. The case of $co_{\text{ł}}$ and ε_1,ε_2 interchanged is analogous.

Ad 8. This is direct from the definition of atomic and appealing on the fact that $(U, U') \in T(E, \varepsilon_1, \emptyset, \varepsilon_3)$. □

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

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

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

Assumption 1. *We henceforth adopt the following* soundness assumption *which must be established concretely for every concrete instance of our framework.*

- *The initial heap satisfies the current world:* $\mathsf{h}_{init} \models \mathsf{w}$.
- *Each axiom is type sound: whenever* (v, v', τ) *is an axiom then* $(v, v) \in [\![\tau]\!]$ *and* $(v', v') \in [\![\tau]\!]$.
- *Each axiom is inequationally sound: whenever* (v, v', τ) *is an axiom then* $(v, v') \in [\![\tau]\!]$.

Theorem 7.8. *Suppose that* $\Gamma \vdash v : \tau$ *and* $\Gamma \vdash e : \tau \ \&\ \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$. *Then* $(\eta, \eta') \in [\![\Gamma]\!]$ *(interpreting a context as a cartesian product) implies* $(\lceil\!\lceil v\rceil\!\rceil\eta, \lceil\!\lceil v\rceil\!\rceil\eta') \in [\![\tau]\!]$ *and* $([\![e]\!]\eta, [\![e]\!]\eta') \in T([\![\tau]\!], \varepsilon_1, \varepsilon_2, \varepsilon_3)$.

Proof. By induction on derivations. Most cases are already subsumed by Theorem 7.7. The typing rules regarding functions and recursion follow from the definitions and from the fact that all specifications are admissible. □

8. Typed observational approximation

Definition 8.1 (Observational approximation)**.** *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 $\vdash f : \tau \xrightarrow[\varepsilon]{\varepsilon_1 \,|\, \varepsilon_3} \mathtt{int}$ ("observations") it is the case that if $((\mathsf{h}_{init}, \mathsf{k}), n) \in [\![f\ v]\!]$ for $n \in \mathbb{Z}$ 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.

We remark that observational approximation extends canonically to open terms by lambda abstracting free variables (and adding a dummy abstraction in the case of closed terms) [6].

As usual, the logical relation is sound with respect to typed observational approximation and thus can be used to deduce nontrivial observational approximation relations. We state and prove the precise formulation of this result.

Theorem 8.2. *Let v, v' be closed values and suppose that $([\![v]\!], [\![v']\!]) \in [\![\tau]\!]$. Then $\vdash v \leq_{obs} v' : \tau$.*

Proof. If $\vdash f : \tau \xrightarrow[\varepsilon_2]{\varepsilon_1 \,|\, \varepsilon_3} \mathtt{int}$ then by Thm 7.8 we have $([\![f]\!], [\![f]\!]) \in [\![\tau \xrightarrow[\varepsilon_2]{\varepsilon_1 \,|\, \varepsilon_3} \mathtt{int}]\!]$, so $([\![f\ v]\!], [\![f\ v']\!]) \in T([\![\mathtt{int}]\!], \varepsilon_1, \varepsilon_2, \varepsilon_3)^+$.

Let $((\mathsf{h}_{init}, \mathsf{k}), n) \in [\![f\ v]\!]$. We have $\mathsf{h}_{init} \models \mathsf{w}$ and thus in particular $\mathsf{h}_{init} \overset{\mathrm{rds}(\varepsilon_3) \cup \mathrm{rds}(\varepsilon_1)}{\sim} \mathsf{h}_{init}$. There must therefore exist a matching heap k' and a value n' such that

$$((\mathsf{h}_{init}, \mathsf{k}'), n') \in [\![f\ v']\!] \text{ and } n = n' \in \mathbb{Z}$$

□

This means that the examples from earlier on give rise to valid transformations in the sense of observational approximation. For instance, for e_1 and e_2 from Example 7.5 we find that $\lambda\_.e_1 =_{obs} \lambda\_.e_2$ at type $\mathtt{unit} \xrightarrow[\varepsilon]{\{co_l\} \,|\, \varepsilon \cup \{rd_l, wr_l\}} \mathtt{unit}$ whenever X does not appear in ε.

9. Effect-dependent transformations

We will now establish the semantic soundness of the inequational theory of effect-dependent program transformations given in Figure 5. It includes concurrent versions of the effect-dependent equations from [9, 29], but the side conditions on the environmental interaction are by no means obvious. We also note that some equations now only hold in one direction thus become inequations. This is in particular the case for duplicated computations. Suppose that `?` is a computation that nondeterministically chooses a boolean value and let $e := \texttt{let}\ x = \texttt{?}\ \texttt{in}\ (x, x)$. Then, even though `?` does not read nor write any location we only have $e \le (?, ?)$, but not $(?, ?) \le e$ for $(?, ?)$ admits the result $(\texttt{true}, \texttt{false})$ but e does not. Furthermore, due to presence of nontermination the equations for dead code elimination and pure lambda hoist also hold in one direction only. It might be possible to restore both directions of said equations by introducing special effects for nondeterminism and nontermination; we have not explored this avenue. We concentrate on the individual effect-dependent transformations before summarising the foregoing results in the general soundness Theorem 9.2.

In many of the equations, co-effects play an important role. For example, in the commuting and parallelization equations, the internal effects ε_1 and ε_2 in the premises are replaced by ε_1^C and ε_2^C in the internal effects of the conclusion. This makes sense intuitively because the computations are run in a different order, so for the internal moves, the locations in ε_1 and ε_2 can be modified in any way (see Example 7.6). However, in the global effect, we can still guarantee the effects ε_1' and ε_2' because of the $\perp$-conditions. This intuition appears directly in the soundness proofs.

The following thus constitutes the second main technical result of our paper.

Theorem 9.1. *The following hold whenever well-formed.*

- ***Commuting*** *If* $(U_1, U_1') \in T(E_1, \varepsilon_1, \varepsilon^C, \varepsilon^C \cup \varepsilon_1')$ *and* $(U_2, U_2') \in T(E_2, \varepsilon_2, \varepsilon^C, \varepsilon^C \cup \varepsilon_2')$ *and* $\varepsilon_1 \perp \varepsilon$ *and* $\varepsilon_2 \perp \varepsilon$ *and* $\varepsilon_1' \perp \varepsilon_2'$ *then*
$$\begin{gathered}(\{(t_1 t_2, (v_1, v_2)) \mid (t_1, v_1) \in U_1, (t_2, v_2) \in U_2\}^\dagger,\\ \{(t_2' t_1', (v_1', v_2')) \mid (t_1', v_1') \in U_1', (t_2', v_2') \in U_2'\}^\dagger)\\ \in T(E_1 \times E_2, (\varepsilon_1 \cup \varepsilon_2)^C, \varepsilon, \varepsilon \cup \varepsilon_1' \cup \varepsilon_2')\end{gathered}$$
- ***Duplicated*** *If* $(U, U') \in T(E, \varepsilon_1, \varepsilon_2^C, \varepsilon_2^C \cup \varepsilon')$ *and* $\mathrm{rds}(\varepsilon') \cap \mathrm{wrs}(\varepsilon') = \emptyset$ *and* $\varepsilon_2 \perp \varepsilon_1$, *then*
$$\begin{gathered}(\{(t, (v, v)) \mid (t, v) \in U\}^\dagger, \{(t_1' t_2', (v_1', v_2')) \mid (t_1', v_1') \in U',\\ (t_2', v_2') \in U'\}^\dagger) \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_2 \cup \varepsilon')\end{gathered}$$
- ***Pure*** *Let* $(U, U') \in T(E, \varepsilon_1, \varepsilon_2^C, \varepsilon_2^C)$, *such that* $\varepsilon_1 \perp \varepsilon_2$. *If* $((q_1, k_1) \ldots (q_n, k_n), v) \in U$ *for some* arbitrary *trace* $t = (q_1, k_1) \ldots (q_n, k_n)$ *(with* $q_1 \models \mathtt{w}$*) and value* v, *then* $(rtn(v), U') \in T(E, \varepsilon_1^C, \varepsilon_2, \varepsilon_2)$*;*
- ***Dead*** *Suppose that* $(U, U') \in T(\texttt{unit}, \varepsilon_1, \varepsilon_2, \varepsilon_2 \cup \varepsilon_1')$, *where* $\mathrm{wrs}(\varepsilon_1') = \emptyset$ *and* $\varepsilon_1 \perp \varepsilon_2$. *Then* $(U, rtn(())) \in T(\texttt{unit}, \varepsilon_1^C, \varepsilon_2, \varepsilon_2 \cup \varepsilon_1')$.
- ***Parallelization*** *If* $(U_1, U_1') \in T(E_1, \varepsilon_1, \varepsilon^C \cup \varepsilon_2^C, \varepsilon^C \cup \varepsilon_2^C \cup \varepsilon_1')$ *and* $(U_2, U_2') \in T(E_2, \varepsilon_2, \varepsilon^C \cup \varepsilon_1^C, \varepsilon^C \cup \varepsilon_1^C \cup \varepsilon_2')$ *and* $\varepsilon_1 \perp \varepsilon_2$ *and* $\varepsilon_1 \perp \varepsilon$ *and* $\varepsilon_2 \perp \varepsilon$, *then*
$$\begin{gathered}(U_1 \| U_2, \{(t_1' t_2' (v_1', v_2')) \mid (t_1', v_1') \in U_1', (t_2', v_2') \in U_2'\}^\dagger) \in\\ T(E_1 \times E_2, \varepsilon_1^C \cup \varepsilon_2^C, \varepsilon, \varepsilon \cup \varepsilon_1' \cup \varepsilon_2')\end{gathered}$$

Proof. **Commuting.** By Theorem 7.7(3) we can assume our pilot trace t to be of the form:

$$(\mathsf{h}_1, \mathsf{k}_1)(\mathsf{h}_2, \mathsf{k}_2) \cdots (\mathsf{h}_n, \mathsf{k}_n) \quad (\mathsf{h}_{n+1}, \mathsf{k}_{n+1}) \cdots (\mathsf{h}_{n+m}, \mathsf{k}_{n+m})\ (a, b)$$

where

$$\begin{aligned} t_1 &= (\mathsf{h}_1, \mathsf{k}_1)(\mathsf{h}_2, \mathsf{k}_2) \cdots (\mathsf{h}_n, \mathsf{k}_n)\ v_1 \in U_1 \\ t_2 &= (\mathsf{h}_{n+1}, \mathsf{k}_{n+1}) \cdots (\mathsf{h}_{n+m}, \mathsf{k}_{n+m})\ v_2 \in U_2 \end{aligned}$$

We make similar use of Theorem 7.7(3) in the subsequent cases without explicit mention.

We are also given a heap h'_1 such that

$$\mathsf{h}_1 \overset{\mathrm{rds}(\varepsilon \cup \varepsilon'_1 \cup \varepsilon'_2)}{\sim} \mathsf{h}'_1$$

Because $\varepsilon'_1 \perp \varepsilon'_2$, h_1 and h_{n+1} agree on the reads of ε'_2. Thus we can start a game U_2 vs. U'_2 using h'_1 and t_2. We forward all environment's moves from the main game to the side game and use the responses from the side game to answer in the main game. Suppose that the side game leads to the valid U_2-trace

$$(\mathsf{h}'_1, \mathsf{k}'_1)(\mathsf{h}'_2, \mathsf{k}'_2) \cdots (\mathsf{h}'_m, \mathsf{k}'_m)\ v'_2$$

where $v_2 E_2 v'_2$ and (1) $[\varepsilon^C \cup \varepsilon'_2](\mathsf{h}_{n+1}, \mathsf{h}'_1, \mathsf{k}_{n+m}, \mathsf{k}'_m)$. Notice that in the global game these are legal responses as $[\varepsilon_1^C \cup \varepsilon_2^C](\mathsf{h}_i, \mathsf{h}'_i, \mathsf{k}_i, \mathsf{k}'_i)$ for $1 \leq i \leq m$.

We now have an environment move $[\varepsilon](\mathsf{k}_m, \mathsf{k}'_m, \mathsf{h}_{m+1}, \mathsf{h}'_{m+1})$. Since $\varepsilon'_1 \perp \varepsilon$ and $\varepsilon'_2 \perp \varepsilon'_1$, the heaps h'_1 and h'_{m+1} agree in the reads of ε'_1. Therefore, we can run a game U_1 vs. U'_1 using h'_{m+1} and t_1, obtaining the trace:

$$(\mathsf{h}'_{m+1}, \mathsf{k}'_{m+1})(\mathsf{h}'_{m+2}, \mathsf{k}'_{m+2}) \cdots (\mathsf{h}'_{m+n}, \mathsf{k}'_{m+n})\ v'_1$$

where $v_1 E_1 v'_1$ and (2) $[\varepsilon^C \cup \varepsilon'_1](\mathsf{h}_1, \mathsf{h}'_{m+1}, \mathsf{k}_n, \mathsf{k}'_{m+n})$. The reasoning is similar to the use of the previous game.

Thus we have that $(v_1, v_2)(E_1 \times E_2)(v'_1, v'_2)$.

Now, we need to conclude that $[\varepsilon^C \cup \varepsilon'_1 \cup \varepsilon'_2](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{k}_{n+m}, \mathsf{k}'_{m+n})$. This follows from the fact that $\varepsilon'_1 \perp \varepsilon'_2$ and (1) and (2). In particular, from (1) and $\varepsilon'_1 \perp \varepsilon'_2$, we get that k_{m+n} and k'_{m+n} agree on the locations in ε'_2, while from (2), we get that k_{m+n} and k'_{m+n} agree on the locations in ε'_1. This finishes the proof.

Duplicated. Assume given a trace in U:

$$t = (\mathsf{h}_1, \mathsf{k}_1) \cdots (\mathsf{h}_n, \mathsf{k}_n)\ v$$

and a heap h'_1 such that $\mathsf{h}_1 \overset{\mathrm{rds}(\varepsilon_2 \cup \varepsilon')}{\sim} \mathsf{h}'_1$. Recall that $\mathrm{rds}(\varepsilon') \cap \mathrm{wrs}(\varepsilon') = \emptyset$ and moreover, since $\varepsilon_2^C \cup \varepsilon'$ is well formed, we also have $\mathrm{rds}(\varepsilon') \cap (\mathrm{wrs}(\varepsilon_2) \cup \mathrm{cos}(\varepsilon_2)) = \emptyset$. Thus h_1 and k_n agree on the reads of $\varepsilon' \cup \varepsilon_2^C$, i.e., the reads of ε'.

We start by simply stuttering:

$$t' = (\mathsf{h}'_1, \mathsf{h}'_1)(\mathsf{h}'_2, \mathsf{h}'_2) \cdots (\mathsf{h}'_n, ??).$$

leaving the final heap ?? yet to be determined. So far, this is a legal play in the main game because for $1 \leq i \leq n-1$, we have $[\varepsilon_1^C](\mathsf{h}_i, \mathsf{h}'_i, \mathsf{k}_i, \mathsf{h}'_i)$ and a chaotic effect on a location allows any changes to that location. Moreover, we may assume

$[\varepsilon_2](\mathsf{k}_i, \mathsf{h}_{i+1}, \mathsf{h}'_i, \mathsf{h}'_{i+1})$ for otherwise we would have won immediately. As a result, since $\varepsilon_1 \perp \varepsilon_2$, we inductively get $\mathsf{h}_i \overset{\mathrm{rds}(\varepsilon_2)}{\sim} \mathsf{h}'_i$ and, of course, $\mathsf{h}_i \overset{\mathrm{rds}(\varepsilon_2)}{\sim} \mathsf{k}_i$.

We will now play two side-games U vs. U' with pilot trace t so as to construct the missing heap "??". We first run a game starting at h'_n, where the environment moves are simply stutter moves. Recall that $\mathsf{h}_1 \overset{\mathrm{rds}(\varepsilon' \cup \cos(\varepsilon_2^C))}{\sim} \mathsf{h}'_n$ has already been asserted above. We thus obtain the following trace $t_1 \in U'$

$$t_1 = (\mathsf{h}'_n, \mathsf{q}_1)(\mathsf{q}_1, \mathsf{q}_2) \cdots (\mathsf{q}_{n-1}, \mathsf{q}_n)\ v'_1$$

where vEv'_1 and $[\varepsilon_2^C \cup \varepsilon'](\mathsf{h}_1, \mathsf{h}'_n, \mathsf{k}_n, \mathsf{q}_n)$. Notice that using stuttering environment moves is valid as $[\varepsilon_2^C](\mathsf{k}_i, \mathsf{q}_i, h_{i+1}, \mathsf{q}_i)$ for $1 \leq i \leq n-1$.

Since h_1 and k_n agree on the reads of ε' and q_n and k_n agree on $\mathrm{rds}(\varepsilon')$ from $[\varepsilon_2^C \cup \varepsilon'](\mathsf{h}_1, \mathsf{h}'_n, \mathsf{k}_n, \mathsf{q}_n)$, we can run the game U vs. U' again on q_n and t with stutter environment moves:

$$(\mathsf{q}_n, \mathsf{q}_{n+1})(\mathsf{q}_{n+1}, \mathsf{q}_{n+2}) \cdots (\mathsf{q}_{n+n-1}, \mathsf{q}_{n+n})\ v'_2$$

where vEv'_2 and $[\varepsilon_2^C \cup \varepsilon'](\mathsf{h}_1, \mathsf{q}_n, \mathsf{k}_n, \mathsf{q}_{n+n})$. Thus, $(v, v)(E \times E)(v'_1, v'_2)$. This trace is again valid for the same reasons above, namely ε_1^C allows any internal moves, and since $\varepsilon_1 \perp \varepsilon_2$, the environment moves are also legal.

We now put $?? := q_{n+n}$ which leads to a valid trace due to repeated mumbling. Finally, we shall show that $[\varepsilon_2 \cup \varepsilon'](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{k}_n, \mathsf{q}_{n+n})$ that is k_n and q_{n+n} agree on the reads of ε_2 and of ε':

- They agree on the reads of ε' because $[\varepsilon_2^C \cup \varepsilon'](\mathsf{h}_1, \mathsf{q}_n, \mathsf{k}_n, \mathsf{q}_{n+n})$ obtained from the game above;

- They agree on the reads of ε_2 because $\varepsilon_1 \perp \varepsilon_2$. The internal moves did not affect the locations read by ε_2.

Duplicated for result value `unit`: We can show that equality holds and not just $\leq$ when the result type is `unit`. The reverse direction is proved as follows: For a given pilot trace t of U, where e is executed twice, we can construct a trace t' in U' by first stuttering and then mimicking the second execution of e. Since the resulting type is `unit`, there values obtained in t are necessarily () which is also necessarily the same value obtained in the trace t'.

Pure. We start with a trace from $rtn(v)$, for example $(\mathsf{h}_1, \mathsf{h}_1), v$ and an arbitrary heap h'_1. We now consider the game involving U vs. U' on t, v and h'_1:

$$\begin{aligned} t &= (\mathsf{q}_1, \mathsf{k}_1)(\mathsf{q}_2, \mathsf{k}_2) \cdots (\mathsf{q}_n, \mathsf{k}_n), v \\ t' &= (\mathsf{h}'_1, \mathsf{k}'_1)(\mathsf{k}'_1, \mathsf{k}'_2) \cdots (\mathsf{k}'_{n-1}, \mathsf{k}'_n), v' \end{aligned}$$

We have that vEv' and $[\varepsilon_3](\mathsf{q}_1, \mathsf{h}'_1, \mathsf{k}_n, \mathsf{k}'_n)$. By mumbling, $(\mathsf{h}'_1, \mathsf{k}'_n) \in U'$. We can reply with k'_n in the main game.

Dead. Assume given a trace of the form:

$$(\mathsf{h}_1, \mathsf{k}_1) \cdots (\mathsf{h}_n, \mathsf{k}_n)\ v$$

and h'_1 such that $\mathsf{h}_1 \overset{\mathrm{rds}(\varepsilon_3)}{\sim} \mathsf{h}'_1$. We now initiate a side game U vs. U' on this trace and respond in the main game by stuttering. Thus, we obtain traces $(\mathsf{h}'_1, \mathsf{h}'_1) \cdots (\mathsf{h}'_n, \mathsf{h}'_n)$ () in the main game and $(\mathsf{h}'_1, \mathsf{k}'_1) \cdots (\mathsf{h}'_n, \mathsf{k}'_n)\ v'$ in the side game.

The main trace is in $rtn(())$. The side game tells us that $v = ()$ and that $\mathsf{h}_i \xrightarrow{\varepsilon_1} \mathsf{k}_i$ and therefore $[\varepsilon_1^C](\mathsf{h}_i, \mathsf{h}'_i, \mathsf{k}_i, \mathsf{h}'_i)$. It remains to show that $[\varepsilon \cup \varepsilon'_1 \cup \varepsilon'_2](\mathsf{h}_1, \mathsf{h}'_1, \mathsf{k}_n, \mathsf{k}'_n)$. This follows from the fact that ε_1 has only reads as h_i and k_i agree on all locations.

Parallelization.. We start with a trace in $U_1 \| U_2$. Assume that the trace is of the following form:

$$t_{1,1} t_{2,1} t_{1,2} t_{2,2} \ldots t_{1,n} t_{2,n}\ (v_1, v_2)$$

where each $t_{i,j}$ is a possibly empty sequence of moves of the form $(\mathsf{h}^1_{i,j}, \mathsf{k}^1_{i,j}) \cdots (\mathsf{h}^{m_{i,j}}_{i,j}, \mathsf{k}^{m_{i,j}}_{i,j})$ and

$$t_1 = t_{1,1} \cdots t_{1,n}\ v_1 \in U_1$$
$$t_2 = t_{2,1} \cdots t_{2,n}\ v_2 \in U_2$$

are traces from U_1 and U_2, respectively. We are also given a heap h'_1 such that $\mathsf{h}^1_{1,1} \overset{\mathrm{rds}(\varepsilon \cup \varepsilon'_1 \cup \varepsilon'_2)}{\sim} \mathsf{h}'_1$. We also have $\mathsf{h}^1_{1,1} \overset{\mathrm{rds}(\varepsilon^C \cup \varepsilon^C_2 \cup \varepsilon'_1)}{\sim} \mathsf{h}'_1$. We run a side game U_1 vs. U'_1 using h'_1 and t_1, yielding:

$$t'_{1,1} \cdots t'_{1,n}\ v'_1$$

Assume that $(\mathsf{h}'_1, \mathsf{k}'_1)$ and $(\mathsf{h}'_o, \mathsf{k}'_o)$ are, respectively, the first and last moves of this trace. We have $v_1 E_1 v'_1$ and (1) $[\varepsilon^C \cup \varepsilon^C_2 \cup \varepsilon'_1](\mathsf{h}^1_{1,1}, \mathsf{h}'_1, \mathsf{k}^m_{1,n}, \mathsf{k}'_o)$. Notice that these are legal moves in the global game as we have $[\varepsilon^C_1 \cup \varepsilon^C_2]$ tiles for the player moves and $[\varepsilon]$ times for the environment moves.

Now, assume there is an environment move $(\mathsf{k}_o, \mathsf{h}'_{o+1})$. Since $\varepsilon_1 \perp \varepsilon_2$ and $\varepsilon \perp \varepsilon_2$, the heaps $h^1_{1,1}$ and $h^1_{2,1}$ agree on the reads of ε'_2 and h'_1 and h'_{o+1} also agree on the reads of ε'_2. (Notice as well that $\mathrm{wrs}(\varepsilon_1) \cap \mathrm{rds}(\varepsilon'_2) = \emptyset$ as $\varepsilon^C \cup \varepsilon^C_1 \cup \varepsilon_2$ is a valid effect.) Therefore, we can invoke an U_2 game using h'_{o+1} and t_2, obtaining the trace:

$$t'_{2,1} \cdots t'_{2,n}\ v'_2$$

Assume that $(\mathsf{h}'_{o+1}, \mathsf{k}'_{o+1})$ and $(\mathsf{h}'_{o+p}, \mathsf{k}'_{o+p})$ are, respectively, the first and last moves of this trace. We have $v_2 E_2 v'_2$ and (2) $[\varepsilon^C \cup \varepsilon^C_1 \cup \varepsilon'_2](\mathsf{h}^1_{2,1}, \mathsf{h}'_{o+1}, \mathsf{k}^m_{2,n}, \mathsf{k}'_{o+p})$. For the same reasons as above, these are legal moves in the global game.

Therefore $(v_1, v_2)(E_1 \times E_2)(v'_1, v'_2)$.

We need now to prove that $[\varepsilon \cup \varepsilon'_1 \cup \varepsilon'_2](\mathsf{h}^1_{1,1}, \mathsf{h}'_1, \mathsf{k}^m_{2,n}, \mathsf{k}_{o+p})$. From (1) and $\varepsilon_1 \perp \varepsilon_2$ and $\varepsilon \perp \varepsilon_1$, we have that $\mathsf{k}^m_{2,n}$ and k_{o+p} agree on the locations of ε_1. Similarly, $\mathsf{k}^m_{2,n}$ and k_{o+p} agree on the locations of ε_2. Since there are only ε tiles and $\varepsilon \perp \varepsilon_1$ and $\varepsilon \perp \varepsilon_2$, $\mathsf{k}^m_{2,n}$ and k_{o+p} agree on the locations of ε. This finishes the proof.

□

Theorem 9.2. *Suppose that* $\Gamma \vdash v \le v' : \tau$ *and* $\Gamma \vdash e \le e' : \tau \ \&\ \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$ *and assume that for each axiom* (v, v', τ) *it holds that* $(v, v') \in [\![\tau]\!]^+$*. Then* $(\eta, \eta') \in [\![\Gamma]\!]^+$ *(interpreting a context as a cartesian product) implies* $(\lceil\!\lceil v \rceil\!\rceil\eta, \lceil\!\lceil v' \rceil\!\rceil\eta') \in [\![\tau]\!]^+$ *and* $([\![e]\!]\eta, [\![e']\!]\eta') \in T([\![\tau]\!], \varepsilon_1, \varepsilon_2, \varepsilon_3)^+$.

Sketch. In essence the proof is by induction on derivations of inequalities. However, we need to slightly strengthen the induction hypothesis as follows:

Define

$$
\begin{aligned}
[\![\Gamma \vdash \tau]\!] &= \{(f, f') \mid \forall(\eta, \eta') \in [\![\Gamma]\!].(f(\eta), f'(\eta')) \in [\![\tau]\!]\} \\
[\![\Gamma \vdash \tau\&(\varepsilon_1, \varepsilon_2, \varepsilon_3)]\!] &= \{(f, f') \mid \forall(\eta, \eta') \in [\![\Gamma]\!]. \\
&\qquad (f(\eta), f'(\eta')) \in T([\![\tau]\!], \varepsilon_1, \varepsilon_2, \varepsilon_3)\}
\end{aligned}
$$

We now show by induction on derivations that $\Gamma \vdash v \leq v' : \tau$ implies $([\![v]\!], [\![v']\!]) \in [\![\Gamma \vdash \tau]\!]^+$ and that $\Gamma \vdash e \leq e' : \tau \;\&\; \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$ implies $([\![e]\!], [\![e']\!]) \in [\![\Gamma \vdash \tau\&(\varepsilon_1, \varepsilon_2, \varepsilon_3)]\!]^+$.

The various cases now follow from earlier results in a straightforward manner. Namely, we use Theorem 7.7 for the congruence rules and Theorem 9.1 for the effect-dependent transformations.

As a representative case we show the case where $e \equiv \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ and $e' \equiv \mathtt{let}\ x = e'_1\ \mathtt{in}\ e'_2$. Inductively, we know $([\![e_1]\!], [\![e'_1]\!]) \in [\![\Gamma \vdash \tau_1\&(\varepsilon_1, \varepsilon_2, \varepsilon_3)]\!]^{n_1}$ and $([\![e_1]\!], [\![e'_1]\!]) \in [\![\Gamma, x{:}\tau_1 \vdash \tau\&(\varepsilon_1, \varepsilon_2, \varepsilon_3)]\!]^{n_2}$ for some $n_1, n_2 > 0$. By Theorem 7.8, we also have $([\![e_1]\!], [\![e_1]\!]) \in [\![\Gamma \vdash \tau_1\&(\varepsilon_1, \varepsilon_2, \varepsilon_3)]\!]$ and analogous statements for e'_1, e_2, e'_2. We can, therefore, assume, w.l.o.g. that $n_1 = n_2$ and then use Theorem 7.7 (6) repeatedly (n_1 times) so as to conclude $([\![e]\!], [\![e]\!]) \in [\![\Gamma \vdash \tau\&(\varepsilon_1, \varepsilon_2, \varepsilon_3)]\!]^{n_1}$.

The rules for dead code and pure lambda hoist rely on the cases "Dead" and "Pure" of Thm 9.1 in a slightly indirect way. We sketch the argument for pure lambda hoist. The pilot trace begins with a trace belonging to e_1 and yielding a value v for x. We can then invoke case "Pure" on subsequent occurrences of e_1 in the right hand side. □

Theorem 9.3. *Suppose that* $\vdash v : \tau$ *and* $\vdash v' : \tau$ *and that* $([\![v]\!], [\![v']\!]) \in [\![\tau]\!]^+$ *where* $(-)^+$ *denotes transitive closure. 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 Thm 7.8 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 $\mathsf{h}_{init} \models \mathsf{w}$ and thus in particular $\mathsf{h}_{init} \overset{\mathrm{rds}(\varepsilon_3)\cup\mathrm{rds}(\varepsilon_1)}{\sim} \mathsf{h}_{init}$. There must therefore exist a matching heap k' and a value v' such that

$$((\mathsf{h}_{init}, \mathsf{k}'), v') \in [\![f\ v']\!] \text{ and } v = v' \in \mathbb{Z}$$

□

We now return to the 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:

$$
\begin{aligned}
&\mathsf{readFst} : \mathtt{unit} \xrightarrow[\varepsilon^C, co_{\mathfrak{snd}(X)}]{\emptyset \mid \varepsilon^C, co_{\mathfrak{snd}(X)}, rd_{\mathfrak{fst}(X)}} \mathtt{int} \\
&\mathsf{writeFst} : \mathtt{int} \xrightarrow[\varepsilon^C, co_{\mathfrak{snd}(X)}]{wr_{\mathfrak{fst}(X)} \mid \varepsilon^C, co_{\mathfrak{snd}(X)}, wr_{\mathfrak{fst}(X)}} \mathtt{unit}
\end{aligned}
$$

The obvious and analogous typings for readSnd and writeSnd are elided. We justify this typing semantically as described in Theorem 7.7. To illustrate how this is done, consider the function (writeSnd 17). We show how the game is played against itself 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, e_1, that only calls readFst, writeFst, and another program, e_2, that only calls readSnd, writeSnd. Since the former functions have disjoint effects to the latter ones, e_1 and e_2 will have effect specifications, respectively, of the form $(\varepsilon_1, \varepsilon^C \cup \varepsilon_2^C, \varepsilon^C \cup \varepsilon_2^C \cup \varepsilon_1)$ and $(\varepsilon_2, \varepsilon^C \cup \varepsilon_1^C, \varepsilon^C \cup \varepsilon_1^C \cup \varepsilon_2)$, 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 5 to conclude that the behavior of $e_1 || e_2$ is the same as executing these programs sequentially, although they read and write to the same concrete location.

Loop Parallelization. We show that the function map is equivalent to map2Par. It is easy to see that the function map is equivalent to the program map2Seq, which is the program obtained from map2Par by replacing the underlined parallel operator '||' in map2Par by a sequential operator ';'. The proof goes simply by unfolding map.

We then proceed by showing map2Seq and map2Par are equivalent using our equations and the abstract locations $\mathfrak{listodd}(X)$ and $\mathfrak{listodd}(X)$ defined above. The piece of code that applies f first, namely $e_1 = n.ele := f(n.ele)$, has global effects $\varepsilon_1' = rd_{\mathfrak{listodd}(X)}, wr_{\mathfrak{listodd}(X)}$, while the second application, namely,

$$e_2 = n.next.ele := f(n.next.ele)$$

has effects $\varepsilon_2' = rd_{\mathfrak{listeven}(X)}, wr_{\mathfrak{listeven}(X)}$. Notice that $\varepsilon_1' \perp \varepsilon_2'$. Therefore, provided that the environment does not read nor modify the list, we can apply the parallelization equation to justify running e_1 and e_2 parallel is equivalent to running them in sequence.

Michael-Scott Queue. We now show that the enqueue and dequeue functions described in Section 1 for the Michael-Scott Queue have the same behavior as their atomic versions. We only show the case for dequeue, as the case for enqueue is similar. More precisely, we now justify the axiom

$$(\mathsf{dequeue}, \mathtt{atomic}(\mathsf{dequeue}), \mathtt{unit} \xrightarrow[MSQ]{MSQ\,|\,MSQ} \mathtt{int})$$

where $MSQ = \{rd_{\mathfrak{msq}(X)}, wr_{\mathfrak{msq}(X)}\}$. That is, they approximate each other at a type where the environment is allowed to operate on the queue as well. We also note that the converse of the axiom is obvious by stuttering and mumbling. After consuming a dummy argument () let the resulting pilot trace be $(\mathsf{h}_1, \mathsf{k}_1) \ldots (\mathsf{h}_i, \mathsf{k}_i) \ldots (\mathsf{h}_n, \mathsf{k}_n)a$ and h'_1 be the start heap to match. We can now assume that the passages from k_i to h_{i+1} are according to the protocol, i.e. $\mathsf{k}_i \xrightarrow{\mathfrak{msq}(X)} \mathsf{h}_{i+1}$. Namely, should this not be the case we are free to make arbitrary moves and still win the game by default of the environment player. Therefore, there must exist i such that in the move $(\mathsf{h}_i, \mathsf{k}_i)$ the element a is dequeued and $\mathsf{h}_j = \mathsf{k}_j$ holds for $j \neq i$. We can thus match this trace by a trace in the semantics of `atomic`($\mathsf{dequeue}$ ()) by stuttering until i:

$(\mathsf{h}'_1, \mathsf{h}'_1) \ldots (\mathsf{h}'_i, \ldots$

where h_j and h'_j have the same content, but not necessarily the exact same layout. Given the environment's allowed effects it is then clear that also h_i and h'_i have the same content, but not necessarily the same as h_1 and h'_1 because in the meantime other operations on the queue might have succeeded. We then dequeue the corresponding element from h'_i leading to k'_i and continue by stuttering.

$\ldots, \mathsf{k}'_i)(\mathsf{h}'_{i+1}, \mathsf{h}'_{i+1}) \ldots (\mathsf{h}'_n, \mathsf{h}'_n)a'$

It is now clear that this is a matching trace and that $a = a'$ so we are done.

Notice that the congruence rules now allow us to deduce the equivalence of $op_1 \parallel \cdots \parallel op_n$ and `atomic`(op_1) $\parallel \cdots \parallel$ `atomic`(op_n) for op_i being enqueues or dequeues, which effectively amounts to linearizability.

10. 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 [23], whilst our abstract locations are very like the *islands* of Ahmed et al [4]. Recent work of Turon et al [31] 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 [32]) already seems to bring some simplification compared to transition systems or resumptions.

Birkedal et al [12] have also studied relational semantics for effects in a concurrent language. The language considered there has dynamic allocation via regions and higher-order store, neither of which we have here. On the other hand, their invariants are based on simply-typed concrete locations and thus do not allow to capture effects at the level of whole datastructures as abstract locations do. As a result, the examples in [12] are of a simpler nature than ours. Furthermore, we offer a subtler parallelization rule, distinguish transient and end-to-end effects, and validate other effect-dependent equivalences like commuting, lambda hoist, deadcode and duplication. Our use of denotational methods and in particular the extension of Brookes' trace semantics to higher-order functions does result in a rather simpler and more intuitive definition of the logical relation by comparison with [12]. While some of the complications are due to the dynamic allocation and typed locations, others like the explicit step counting, the need for effect-instrumented operational semantics, and the separation of branches in the definition of safety are not. We thus see our work also as a proof-of-concept for denotational semantics in the realm of higher-order concurrent programming.

The 'RGSim' relation proposed by Liang *et al.* for proving concurrent refinements under contextual assumptions also has many similarities with our logical relation [23, Def.4]. The focus of that work is on proving particular equivalences and refinements, whereas we encapsulate general patterns of behaviour in a refined type system and can show the soundness of generic program transformations relying only on effect types (which combine smoothly with hand proofs of particular equivalences).

Since this work has been presented at PPDP 2016, Krogh-Jespersen et al [22] have proposed a system with similar goals as ours. It features higher-order store, i.e., the possibility of storing computations in the heap and not only flat values and pointer structures. In [10] we argued how our semantics-based approach can be extended to higher-order store as well, however, since the issue is mostly orthogonal we refrained from elaborating this path here in the context of concurrency. On the other hand, [22] has weaker rules than ours. Parallelization relies on essentially complete separation and it is even argued explicitly that parallelization comes down to "framing". In our work, and in [23], closer interaction is possible provided one establishes appropriate invariants in the style of rely-guarantee. Also, presumably due to lack of space, the classical effect-dependent rules such as duplication are not treated in [22] and few examples are given. A more detailed comparison should thus await an extended journal version of [22]. From a methodological point of view, [22] is rather different from the work presented here. Namely, equivalences are justified by a translation into the *unary* program logic Iris [20]. This approach has become popular in the last couple of years. Essentially, the idea is to compare the behaviour of two programs, i.e., both sides of an equivalence, by proving a statement in Hoare logic about one of them. The Hoare logic must for this purpose be augmented with special assertions allowing one to speak about steps of the other program. The big advantage of this approach is that the difficult soundness proof needs to be carried out only once and for a unary Hoare logic which is easier. Moreover, the unary Hoare logic, Iris, has been formalised in Coq. A possible disadvantage is that the encoding via a unary Hoare logic might be complicated and unwieldy. It is, however, a very interesting and potentially promising proposal. It would be interesting to see whether it can be used to justify the exact equational theory given in this paper. This would allow one to compare the approaches in a more direct way.

Besides that, there are many other directions for further work. Most importantly, we would like to add dynamic allocation of abstract locations following [6]. In addition to relieving us from having to set up all data structures in the initial heap this would, as we believe, also allow us to model and reason about lock-based protocols in an elegant way. Other possible extension include higher-order store as mentioned above and weak concurrency models. Somewhat further afield, it would be interesting to study ways of automatically inferring opportunities for applications of our equivalences to optimize programs and, relatedly, to use our theory to justify concrete compiler optimisations.

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 *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 340–353, 2009.

[5] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. World Scientific, 1999.

[6] N. Benton, M. Hofmann, and V. Nigam. Abstract effects and proof-relevant logical relations. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 619–632, 2014.

[7] N. Benton, M. Hofmann, and V. Nigam. Effect-dependent transformations for concurrent programs. *CoRR*, abs/1510.02419, 2015.

[8] N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations: higher-order store. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, pages 301–312, 2009.

[9] N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations. In *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*, pages 114–130, 2006.

[10] N. Benton, A. Kennedy, M. Hofmann, and V. Nigam. Counting successes: Effects and transformations for non-deterministic programs. In S. Lindley, C. McBride, P. W. Trinder, and D. Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 56–72. Springer, 2016.

[11] N. Benton, A. Kennedy, and G. Russell. Compiling standard ML to java bytecodes. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.*, pages 129–140, 1998.

[12] L. Birkedal, F. Sieczkowski, and J. Thamsborg. A concurrent logical relation. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, pages 107–121, 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] R. De Nicola and M. Hennessy. Testing equivalence for processes. In *Automata, Languages and Programming, 10th Colloquium, Barcelona, Spain, July 18-22, 1983, Proceedings*, pages 548–560, 1983.

[18] 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.

[19] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *LISP and Functional Programming*, 1986.

[20] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, pages 637–650. ACM, 2015.

[21] O. Kammar and G. D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 349–360, 2012.

[22] M. Krogh-Jespersen, K. Svendsen, and L. Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 218–231. ACM, 2017.

[23] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 455–468. ACM, 2012.

[24] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations, ii: Timing-based systems. *Inf. Comput.*, pages 1–25, 1996.

[25] 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.

[26] 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.

[27] 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.

[28] G. D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5(3):452–487, 1976.

[29] J. Thamsborg and L. Birkedal. A Kripke logical relation for effect-based program transformations. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 445–456. ACM, 2011.

[30] J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *POPL*, 2010.

[31] A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 343–356, 2013.

[32] A. J. Turon and M. Wand. A separation logic for refining concurrent objects. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 247–258. ACM, 2011.