Effect-Dependent Transformations for Concurrent Programs

Nick Benton  
Microsoft Research,  
Cambridge, UK  
nick@microsoft.com

Martin Hofmann  
LMU, Munich, Germany  
hofmann@ifi.lmu.de

Vivek Nigam  
UFPB, João Pessoa, Brazil  
vivek.nigam@gmail.com

ABSTRACT

We describe a denotational semantics for an abstract effect system for a higher-order, shared-variable concurrent language. The semantics validates general effect-based program equivalences, including sufficient conditions for replacing sequential composition with parallel composition. Effect annotations refer to abstract locations, specified by contracts, rather than physical footprints, allowing us to also show soundness of some transformations involving fine-grained concurrent data structures, such as Michael-Scott queues.

We build on a trace-based semantics for first-order programs due to Brookes. By moving from concrete to abstract locations, and adding type refinements capturing possible side-effects of both expressions and their environments, we can validate many equivalences that do not hold in an unrefined model. Refined types are interpreted using a game-based logical relation over sets of traces.

CSC Concepts

- Theory of computation  
- Type structures; Denotational semantics; Program analysis;

Keywords

Type and effect systems, concurrency, logical relations, parametricity, program transformation

1. INTRODUCTION

Type-and-effect systems refine conventional types with safe upper bounds on the possible side-effects of expression evaluation. Introduced by Gifford and Lucassen [21], uses of effect systems include region-based memory management [12], tracking exceptions [28, 27], communication behaviour [4] and atomicity [20] for concurrent programs, and information flow [13].

A major reason for tracking effects is to justify program transformations, most obviously in optimizing compilation [10]. 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 read and write are suitably disjoint. Several groups have studied semantics of effect systems and formal justification of effect-dependent transformations [23, 8, 5, 11, 30]. Our approach is to interpret effect-refined types using a logical relation over the semantics of an 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. This semantic approach decouples the meaning of refined types from any syntactic rules: one may establish that a term has a type using different 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 Γ ⊢ e : τ & ε guarantees that the result of evaluating e will have 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 just initial states and parameter values, and final states and results, of e itself and its free variables. All those interaction points are visible in the term and are governed by specific annotations appearing in the typing judgement.

Shared-variable concurrency allows more possible interactions. The environment now includes anything that may be running concurrently and, moreover, atomic steps of e and its environment may be interleaved, so it no longer suffices to just consider initial and final states. This leads to 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.

This paper explores effect types as a lightweight interfaces for modular reasoning about equivalence and refinement under environmental assumptions, e.g. for safely transforming sequential composition into parallelism. We show how the relational approach to effects scales to concurrency, allowing us to control interference and prove non-trivial equivalences, extending (somewhat to our surprise) to the correctness of some fine-grained algorithms. But functional correctness of particular tricky examples is not our main focus. We are interested in effects as useful intermediate specifications, between conventional types (guaranteeing little about the beh-
haviour of concurrent code) and richer, more complex, models and logics [31].

We first give a trace semantics for concurrent programs that explicitly describes possible interference by the environment. We extend Brookes semantics [14] to a higher-order language, and then refine it by affect 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, which may allow “cleaning-up” some of the effects occurring during computation. Annotated function types \( \tau_1 \xrightarrow{e_1,e_2} \tau_2 \) also capture the effect during a call, \( e_1 \), the environmental interference, \( e_2 \), and the final effect, \( e_3 \). Rather than tracking effects on 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 [5]. Each abstract location has its own notion of both equality and legal mutation. Write effects, for example, need only be flagged when several locations, or parts of locations [5]. Each abstract

We show the soundness of a number of generic equivalence, including a parallelization rule that describes when the parallel execution, \( e_1 || e_2 \), of two programs, \( e_1 \) and \( e_2 \), can be approximated by their sequential execution \( e_1; e_2 \).

Finally, we show that our semantics captures equivalences of interesting programs, including an idealized Michael-Scott queue and its atomic version. A longer account, with more examples and proofs, may be found in a companion technical report [6]. We start with some motivating examples:

**Equivalence modulo non-interference.** Our semantics justifies the equation \( (X := !X + 1; X := !X + 1) \rightarrow (X := !X + 1; X := !X + 2) \) at the effect type unit \& \{ch\( X \) \} \mid e \mid e \cup \{rd\( X \), wr\( X \)\}, provided that the effect, \( e \), of the concurrent environment does not involve \( X \). This says that the two commands are equivalent with return type unit,\(^1\) exhibit the effect ch\( X \), signifying concurrent or ‘chaotic’ access to \( X \) along the way, and have an overall end-to-end effect of \( e \) plus reading and writing \( X \).

**Overlapping references.** Let \( p, p^{-1} \) implement a bijection \( Z \rightarrow Z \times Z \), and consider the following functions:

\[
\begin{align*}
\text{readFst} & (p(!X).1) \\
\text{readSnd} & (p(!X).2) \\
\text{wrFst} n & = (\text{rec} \text{ try} = \text{let} m = !X \text{ in} \\
& \quad \text{if cas}(X, m, p^{-1}(n, p(m), 2)) \\
& \quad \text{then} () \text{ else} \text{ try} () \\
\text{wrSnd} n & = (\text{rec} \text{ try} = \text{let} m = !X \text{ in} \\
& \quad \text{if cas}(X, m, p^{-1}(p(m).1, n)) \\
& \quad \text{then} () \text{ else} \text{ try} ()
\end{align*}
\]

which multiplex two abstract integer references onto a single concrete one. Note that the write functions, \( \text{wrFst} \) and \( \text{wrSnd} \), use compare-and-swap, \( \text{cas} \), to atomically update the value of the reference.

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 \( \text{wrFst} \) followed by one to \( \text{wrSnd} \), 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 \( \text{atomic}(\cdot) \). 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 \( X \) in terms of two concrete ones, \( X_{\text{ver}} \) and \( X_{\text{val}} \), governed by a specification that says \( X_{\text{val}} \) may only change when \( X_{\text{ver}} \) increases. We define

\[
\begin{align*}
\text{transact } f & = \text{let rec try} () = \\
& \quad \text{let} (v, ver) = \text{atomic}((!X_{\text{val}}, !X_{\text{ver}})) \\
& \quad \text{in let} res = f(v, ver) \text{ if atomic}(v) \text{ then} \text{ let} v, ver = res; \text{ true else false} \\
& \quad \text{then} () \text{ else} \text{ try} ()
\end{align*}
\]

Under the assumption that \( f \) is a pure function (has effect \( \text{type} \text{ int} \xrightarrow{\varepsilon} \text{ int} \) for any \( \varepsilon \)), we can show

\[
\begin{align*}
\text{transact } f & = \text{atomic}(X_{\text{val}} := f(!X_{\text{val}}); X_{\text{val}} := !X_{\text{ver}} + 1)
\end{align*}
\]

at type unit \& \{rd\( X \), wr\( X \)\} \mid \varepsilon \mid \varepsilon \cup \{rd\( X \), wr\( X \)\} for any \( \varepsilon \) not including chaotic access, ch\( X \), to \( X \). The environment effect \( \varepsilon \) here may include reading and writing \( X \), so concurrent calls to \( \text{transact} \) are linearizable.

**Michael-Scott queue.** The Michael-Scott Queue [26] (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 enqueued at the end, involving a traversal that is done without locking. Once the end, \( p \), of the list is found, the program atomically

![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 enqueue operation is illustrated by the dotted arrow and the box with the element \( n_{j+1} \) (in blue), while the dequeue operation is illustrated by the dot dashed head pointer (in red).](image-url)
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 null
  else atomic(if !p.next = null then
    !atomic(if !p.next = null then
      lp.next := ref(x, null); true else false)
  then () else try (lp.next)
  else try (lp.next)) \head

mem x = (rec find l =
  if l = null then false else
  if !l.ele = x then true else false
  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.

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 atomic(enqueue) and atomic(dequeue), their atomic versions 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 int \(\Rightarrow (\text{int} \times \text{MSQ}) \times \text{MSQ} \Rightarrow \text{bool}\) provided the environment does not access the MSQ chaotically, i.e., \(e_κ \text{MSQ} \notin e_2\). This typing denotes that mem has the effect of reading the MSQ during both execution and as overall effect. With more assumptions on the environment effects \(e_2\), 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 unit \(\Rightarrow (\text{unit} \times \text{MSQ} \times \text{MSQ} \Rightarrow \text{unit}\). During execution, reset both reads and writes the MSQ, but we can show semantically that its overall effect is only the environmental effect \(e_2\) plus writing the MSQ: there is no overall read effect. Again, from the typing (and assumptions on \(e_2\)), one obtains equations involving reset without further semantic reasoning.

2. SYNTAX

We work with a metalanguage for concurrent, stateful computations and higher-order functions. Parallel computations communicate via a shared heap mapping dynamically allocated locations to structured values, which include pointers. For simplicity, 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 \(X_1, X_2, \ldots\) and a set \(\mathbb{V}\) of storeable "R-values", which include integers, booleans, locations, and tuples \((v_1, \ldots, v_n)\) of R-values. We assume that it is possible to tell of which form a value is and to project its components in case it is a tuple. A heap \(h \in H\), then, is a finite map from \(\mathbb{L}\) to \(\mathbb{V}\), written \(\{(X_1, c_1), (X_2, c_2), \ldots, (X_n, c_n)\}\), specifying that the value stored in location \(X_i\) is \(c_i\). We write \(\text{dom}(h)\) for the domain of \(h\) and write \(h[X\mapsto v]\) for the heap that agrees with \(h\) except that it maps \(X\) to \(c\). We also assume that \(\text{new}(h, v)\) yields a pair \((X, h')\) where \(X \in \mathbb{L}\) is a fresh location and \(h' \in H\) is \(h[X\mapsto v]\).

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

\[
 v ::= \quad x \mid (v_1, v_2) \mid v \mid c \mid \text{rec } f \ x = t
 e ::= \quad v \mid \text{let } x = e_1 \text{ in } e_2 \mid v \text{ if } e_1 \text{ else } e_2
\]

\[
 | v \mid v_1 := v_2 \mid \text{ref}(v) \mid e_1 \mid e_2 \mid \text{atomic}(e)
\]

Here, \(x\) ranges over variables, \(v_i\) over R-values, and \(c\) over built-in functions, including 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 \(F_1(n, n') = n + n'\) for integers \(n, n'\).

The construct \(\text{rec } f \ x = e\) 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, \(lv\) (reading) returns the contents of location \(v\), \(v_1 := v_2\) (writing) updates location \(v_1\) with value \(v_2\), and \(\text{ref}(v)\) (allocating) returns a fresh location initialized with \(v\). The metatheory is simplified by using "let-normal form", where the only elimination for computations is \(\text{let}\), though we nest computations as a shorthand in examples.

The construct \(\text{let } x = e_1 \text{ in } e_2\) 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. The command \(\text{atomic}(e)\) evaluates \(e\) in one step, without any environmental interference. One can then define a (more realistic) compare-and-swap operation \(\text{cas}(X, v_1, v_2)\) as \(\text{atomic}(\text{if } X = v_1 \text{ then } X := v_2; \text{true else false})\) this atomically both checks if location \(X\) contains \(v_1\) and, if so, replaces it with \(v_2\) and returns \(true\); otherwise the location is unchanged and the returned value is \(false\).

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 sketch a denotational semantics for our metalanguage based on Brookes’ trace semantics [14]. Fuller details, including a proof of adequacy with respect to an interleaving operational semantics, are in the technical report [6].

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 3.1 (Traces).** A trace is a finite sequence of the form \((h_1, k_1)(h_2, k_2) \cdots (h_n, k_n)\) where for \(1 \leq j \leq i \leq n\), we have \(h_i, k_i \in \mathbb{H}\) and \(\text{dom}(h_i) \subseteq \text{dom}(h_j), \text{dom}(h_i) \subseteq \text{dom}(k_i), \text{dom}(k_j) \subseteq \text{dom}(h_i), \text{dom}(k_i) \subseteq \text{dom}(k_j)\). We write \(\text{Tr}\) for the set of traces.

A trace of the form \(u(h, h) v\) where \(t = uv\) is said to arise from \(t\) by stuttering. A trace of the form \(u(h, k) v\) where \(t = u(h, q)(q, k)v\) is said to arise from \(t\) by mumbling. If \(t = (h_1, k_1)(h_2, k_2)(h_3, k_3)\), say, then \((h_1, k_1)(h_2, k_2)(h_3, k_3)\) arises from \(t\) by stuttering. If \(k_1 = h_2\), then the trace \((h_1, k_2)(h_3, k_3)\) arises from \(t\) by mumbling. A set of traces \(U\) is closed under stuttering and mumbling if whenever \(t'\) arises from \(t \in U\) by stuttering or mumbling then \(t' \in U\).

Brookes [14] 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 3.2 (Trace Monad).** Let \(A\) be a predomain (\(\omega\)-cpo, not necessarily with bottom). 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:

- \([\text{SE}\text{M}]:\) if \(t'\) arises from \(t\) by stuttering or mumbling and \((t, a) \in U\) then \((t', a) \in U\).
- \([\text{Down}]:\) if \((t, a_1) \in U\) and \(a_2 \leq a_1\) then \((t, a_2) \in U\).
- \([\text{Sup}]:\) if \((a_i)\) is a chain in \(A\) and \((t, a_i) \in U\) for all \(i\) then \((t, \text{sup}_A a_i) \in U\).

The elements of \(TA\) are partially ordered by inclusion.

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 \(t = (h_1, k_1) \cdots (h_n, k_n)\), then there could be \(n\) interactions with the environment with heaps \(h_1, \ldots, h_n\) being “played” by the environment and “answered” with heaps \(k_1, \ldots, k_n\) by the computation. This particular computation then ends with final value \(a\).

For example, the semantics of \(X ::= X + 1; X ::= X + 1\) contains many traces, including the following, where we write \([n]\) for the heap in which \(X\) has value \(n\):

- \([([10],12)],12\)
- \([([10],[11])((15],[16]),16]\)
- \([([10],[11])((15],[16])((17],[17]),17]\)
- \([([10],[11])((15],[16])((17],[17]),16]\)

Axiom \([\text{S}\&\text{M}]\) 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 [29], but do not affect observational equivalence. As non-termination is modelled by the empty set, we are working with an angelic ‘may semantics’ [17].

The semantics of \(X ::= 0;\) if \(X = 0\) then 0 else diverge, for example, is the same as that of \(X ::= 0; 0\) and contains \(([10],[0]),0\), but also say, \(([([10],[0])],([34],[34]),0\)), via stuttering. 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 \(X\).

We 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.

\[
e_1 \equiv \text{if } t \text{ then } X := 0; \text{true else } X := 0; \text{false}
\]

\[
e_2 \equiv X := 0; ?
\]

Both \(e_1\) and \(e_2\) admit the same traces, namely \(((x,0), \text{true})\) and \(((x,0), \text{false})\) and stuttering variants thereof. In 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 [1], forward-backward simulation [25], or speculation [31] in reasoning.

Axioms \([\text{Down}]\) and \([\text{Sup}]\) are known from the Hoare powdomain [29]. Additional nondeterministic outcomes that are less defined than existing ones are not recorded in the semantics.

**Definition 3.3.** If \(U \subseteq \text{Tr} \times A\) then \(U^\dagger\) is the least subset of TA containing \(U\), i.e. \(U^\dagger\) is the closure of \(U\) under \([\text{SE}\text{M}],[\text{Down}],[\text{Sup}].\)

**Definition 3.4.** Let \(A,B\) be predomains. We define the continuous functions \(\text{rtn} : A \rightarrow TA\) and \(\text{bnd} : (A \rightarrow TB) \rightarrow TA \times TB\) by:

\[
\text{rtn}(a) := \{((h, h), a) \mid h \in \mathbb{H}\}\dagger
\]

\[
\text{bnd}(f, g) := \{(uv, b) \mid (u, a) \in g \land (v, b) \in f(a)\}\dagger
\]

These endow \(TA\) with the structure of a strong monad. A partial function \(c : \mathbb{H} \rightarrow \mathbb{H} \times A\) (an element of the state monad \(SA\)) can be (continuously) transformed into an element from\(\text{state}(c)\), where from\(\text{state} : SA \rightarrow TA\) is defined by from\(\text{state}(c) := \{((h,k),a) \mid c(h) = (k,a)\}\dagger\). If \(t_1,t_2,t_3\) are traces, we write \(\text{inter}(t_1,t_2,t_3)\) to mean that \(t_3\) can be obtained by interleaving \(t_1\) and \(t_2\) in some way, i.e., \(t_3\) is contained in the shuffle of \(t_1\) and \(t_2\). In order to model parallel composition we introduce the following helper function

\[
| : TA \times TB \rightarrow T(A \times B)
\]

\[
U \mid V = \{(t_3, (a,b)) \mid \text{inter}(t_1, t_2, t_3), (t_1, a) \in U, (t_2, b) \in V\}\dagger
\]

The continuous map \(a : TA \rightarrow TA\) is defined by \(a(U) = \{(h, k), v) \mid (h, k), v \in U\}\dagger.\) Notice that due to mumbling \((h, k), v) \in U\} iff there exists an element of the form:

\[
((h_1, h_2)(h_3, h_4)\cdots(h_{n-2}, h_{n-1})(h_{n-1}, h_n), v) \in U
\]

where \(h = h_1\) and \(h_n = h\). Such an element models an atomic execution of the computation represented by \(U\).

### 3.1 Semantic values

The predomain \(V\) of values is the least solution of

\[
V \equiv \mathbb{VB} + (V \rightarrow TV) + V^*.
\]

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

There are (canonical) families of definitions \(p_i : V \rightarrow V\) and \(q_i : TV \rightarrow TV\) such that that \(p_i\) and \(q_i\) are ascending chains converging to the identity. A consequence is that \(V\) and \(TV\) are bifinite (equivalently SFP) predomains [2] and as such also Scott predomains. These technicalities
help with the compatibility of the admissible closure of logical predicates and simplify reasoning in general; they are discussed in more detail in the technical report [6].

The semantics of values \[\llbracket v \rrbracket \in V \rightarrow V\] and terms \([t] \in V \rightarrow TV\] are given by the recursive clauses in Figure 3. Environments, \(\rho\), are properly tuples of values; we abuse notation slightly by treating them as maps from variables, \(x\), to values, \(v\), (and write \(\rho[x=v]\) for functional update) to avoid mentioning an explicit context in which untyped terms are well-formed.

4. ABSTRACT LOCATIONS

We simplify and extend our previous notion of abstract locations [5]. These allow complicated data structures that span several concrete locations, or only parts of them, to be 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. Local 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 [3], though one difference is that abstract locations do not require concrete footprints.

In the presence of concurrency, we actually 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 make while an operation on the abstract location is ongoing, but temporarily preempted.

Definition 4.1 (Concurrent Abstract Location). A concurrent abstract location \(l\) comprises:

1. A partial equivalence relation \(\sim\) on \(H\) modeling the “semantic equivalence” on the bits of the store that \(l\) uses. If \(h \sim h'\) then the same computation started on \(h\) and \(h'\), respectively, will yield related or even equal results.

2. A partial equivalence relation \(\overset{1}{\sim}\) on \(H\) refining \(\sim\) and modeling the “strict equivalence” on the bits of the store that \(l\) uses. If a concurrent computation on \(l\) has reached \(h\) and is preempted, then another computation may replace \(h\) with \(h'\) where \(h \overset{1}{\sim} h'\) and then the original computation on \(l\) may resume on \(h'\) without the final result being compromised.

3. A transitive (and reflexive on the support of \(\overset{1}{\sim}\)) relation \(\overset{2}{\rightarrow}\) 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 \(h \overset{2}{\rightarrow} h_1\) then \(h_1\) might arise by writing to \(l\) in \(h\) and all possible writes are specified by \(\overset{2}{\rightarrow}\). We call \(\overset{2}{\rightarrow}\) the step relation of \(l\).

These data must satisfy the following conditions where \(h : l\) stands for \(h \overset{1}{\sim} h\).

1. If \(h : l\) then \(h \overset{1}{\sim} h\); if \(h \overset{1}{\sim} h_1\) then \(h : l\) and \(h_1 : l\).

If \(h \overset{2}{\rightarrow} h_1\) and at the same time \(h \overset{1}{\sim} h_1\), then we say that \(h_1\) arises from \(h\) by a silent move in \(l\). Our semantic framework will permit silent moves at all times.

We now describe abstract locations corresponding to our earlier motivating examples.

**Single integer.** Our simplest example is the following abstract location, parametric in a concrete location \(X\):

\[
\begin{align*}
\text{int}_X(h) &\iff \exists n. h(X) = \text{int}(n) \\
\text{int}_X(h) &\iff \exists h'(X) = \text{int}(n) \\
\text{int}_X(h) &\iff \exists h'(X) \\
\end{align*}
\]

Two heaps are semantically equivalent w.r.t. \(\text{int}(X)\) if the values stored in \(X\) are equal integers; the step relation requires all other concrete locations to be unchanged. We may write \(\text{rd}_X, \text{wr}_X, \text{ch}_X\) for \(\text{rd}_{\text{int}(X)}, \text{wr}_{\text{int}(X)}, \text{ch}_{\text{int}(X)}\).

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

\[
\begin{align*}
\text{fst}_X(h) &\iff \exists a_1, a_2, a'_1, a'_2 \in \mathbb{Z}. h(X) = p^{-1}(a_1, a_2) \land \ h'(X) = p^{-1}(a'_1, a'_2) \land a_1 = a'_1 \\
\text{fst}_X(h) &\iff h \overset{1}{\sim} h' \\
\end{align*}
\]

The semantic (and strict) equivalence of \(\text{fst}(X)\) (respectively, \(\text{snd}(X)\)) 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 \(\text{fst}(X)\) (respectively, \(\text{snd}(X)\)) 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 \(X\) consists of two concrete locations \(X_{\text{val}}\) and \(X_{\text{ver}}\), and its relations are:

\[
\begin{align*}
\text{x}_X &\iff h(X_{\text{val}}) = h'(X_{\text{val}}) \\
\text{x}_X &\iff h \overset{1}{\sim} h' \\
\text{h}_X &\iff \forall X'. \ X' \notin \{X_{\text{ver}}, X_{\text{val}}\}. h(X') = h_1(X') \land h : X \land h(X_{\text{ver}}) = h_1(X_{\text{ver}}) \land [h(X_{\text{val}}) \neq h_1(X_{\text{val}}) \Rightarrow h(X_{\text{ver}}) < h_1(X_{\text{ver}})]
\end{align*}
\]

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 decrease, and increases if the value changes.

**Michael-Scott queue.** For concrete location \(X\) we introduce a concurrent abstract location \(\text{msq}(X)\) first informally as follows: we have \(h \overset{1}{\sim} \text{msq}(X)\) if both \(h\) and \(h'\) contain a well-formed MSQ rooted at \(X\) and these queues contain the same entries in the same order. But they may use different locations for the nodes and have different garbage tails.

The relation \(h \overset{1}{\sim} \text{msq}(X)\) 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 may relocate the queue or remove the garbage tail, which would be allowed if we merely required that such operations do not change the \(\overset{1}{\sim}_{\text{MSQ}(X)}\)-class.
The relation $\text{map}(X)$, finally, is defined as the transitive closure of the actions 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 $\text{head}$, $\text{next}$, $\text{elem}$ using some layout convention, e.g. $v.\text{head} = v.1$, etc. We then define

$$h, X \xrightarrow{\text{next}} X' \iff X' \text{ can be reached from } X \text{ in } h$$

by following a chain of next pointers

We use $\text{List}(X, h, (X_0, \ldots, X_n), (v_1, \ldots, v_n))$ to mean that $h(X)$ points to a linked list with nodes $X_0, \ldots, X_n$ and entries $v_1, \ldots, v_n$. The first node $X_0$ acts as a sentinel and its $\text{elem}$ component is ignored. Formally:

$$h(X).\text{head} = X_0 \quad h(X_i).\text{elem} = v_i \text{ for } 1 \leq i \leq n \quad h(X_i).\text{next} = X_{i+1} \text{ for } 0 \leq i \leq n - 1 \quad h(X_n).\text{next} = \text{null}$$

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

$$fp(X, h) = \{X' \mid X \xrightarrow{\text{next}} X' \lor X' \xrightarrow{\text{next}} X\}$$

Write $\text{snoc}(h, h', X, v)$ to mean that $h'$ arises from $h$ by attaching a new node containing $v$ at the end of the list pointed to by $h$. So $\text{List}(X, h, (X_0, \ldots, X_n), (v_1, \ldots, v_n))$ implies $\exists X_{n+1} \notin \text{dom}(h).\text{List}(X, h', (X_0, \ldots, X_n, X_{n+1}), (v_1, \ldots, v_n))$. We omit the obvious frame conditions. Then

$$h \xrightarrow{\text{map}(X)} h' \iff \exists \tilde{X}, \tilde{X}', \tilde{v}, \text{List}(X, h, \tilde{X}, \tilde{v}, \tilde{v}) \land \text{List}(X, h', \tilde{X}', \tilde{v})$$

$h \xrightarrow{\text{map}(X)} h'$ $\iff h \xrightarrow{\text{snoc}(X)} h' \land \forall X' \in fp(X, h).h'(X') = h'(X')$

$h \xrightarrow{\text{map}(X)} h_1$ $\iff h : \text{snoc}(X) \land h_1 : \text{map}(X) \land \text{step}^*(h, h_1)$

$\text{step}(h, h_1)$ $\iff \forall X' \neq X.h(X') = h_1(X') \land (h_1(X) = h(X), next \lor \exists v.\text{snoc}(h, h_1, X, v))$

In these examples, the only silent moves are identities. But datastructures such as collections that reorganize during lookups, or which use late initialization [5] do involve non-trivial silent moves.

### 4.1 Worlds

We group the abstract locations used by a program into a world. Here, all abstract locations must be established up front. Concrete locations may be dynamically allocated to grow an abstract location, as in the MSQ example, but worlds themselves do not evolve. We have previously shown [5, 3] how proof-relevant Kripke logical relations can account for dynamic allocation of abstract locations, but leave the combination of those with concurrency for future work.

**Definition 4.2 (World).** A world is a set of abstract locations.

The relation $h \models w$ (heap $h$ satisfies world $w$) is the largest relation such that $h \models w$ implies

- $h : t$ for all $t \in \mathbb{W}$;
- if $t \in \mathbb{W}$ and $h \models t$, then $h \not\models h'$ holds for all $h' \in \mathbb{W}$ with $h' \neq h$ and $h_1 \models w$.

Note that if $w$ contains two “interfering” abstract locations, e.g. has both an integer location and a boolean location placed at the same physical location, there will be no heap $h$ such that $h \models w$. We assume a fixed current world $w$ which may appear in definitions without being notationally reflected. (See Assumption 1 later.)

### 5. EFFECTS

The elementary effects are $\text{rd}_l$ (reading from $l$), $\text{wr}_l$ (writing to $l$), and $\text{ch}_l$ (chaotic access), for each abstract location $l$. An effect, ranged over by $\varepsilon$, is a set of elementary effects.

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 $\text{wr}_X$ effect, but $e_1$ and $e_2$ are distinguishable by contexts that assume the $\text{wr}_X$-effect. Thus, $e_1$ and $e_2$ are not equal "at type" $\text{wr}_X$. At type $\text{ch}_X$ they are, however, equal, because a context that copes with this effect may not assume that both produce equal results.

So $e_1$ is a ‘don’t care’ effect, requiring the environment not to look at a particular location during a concurrent computation. For example, we can show that $X := !X + 1 ; X := \top X + 1$ is equivalent to $X := !X + 2$ “at type” $\text{unit} \land \text{ch}_X \in \varepsilon \lor \in \text{rsd} \lor \text{wr}_X$, where $\varepsilon$ is any effect such that $X \not\models \text{loc}(\varepsilon)$. 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. The $\text{ch}_X$ effect is required because $X$ may be different during the computations. However, once the programs are finished, the value of $X$ will be the same in both cases, so the end-to-end effect need not include $\text{ch}_X$. The $\text{ch}$ effects are akin to the private regions from [11], but seem more permissive.

We use the notation $\text{rsd}(\varepsilon), \text{wr}(\varepsilon), \text{ch}(\varepsilon)$ to refer to the abstract locations $l$ for which $\varepsilon$ contains $\text{rd}_l, \text{wr}_l$, and $\text{ch}_l$, respectively. We write $\text{loc}(\varepsilon) := \text{rsd}(\varepsilon) \cup \text{wr}(\varepsilon) \cup \text{ch}(\varepsilon)$.

Our semantics of effects follows the relational style [8, 11]. Intuitively, two computations are related at $\text{rd}_X$ if they produce related results when run in states that have related values for $X$. Should the starting states differ on the value of $X$, then their behavior is unconstrained. They are related at $\text{wr}_X$ if either they leave the $X$ unchanged or they write related values to $X$, i.e., the values of $X$ are equal at the end. If they are related at $\text{ch}_X$, then arbitrary modifications of $X$ are allowed.
Definition 5.1. An effect \( e \) is well-formed (with respect to the current world) if \( \text{locs}(e) \subseteq \text{w} \) and \( \text{rds}(e) \cap \text{chs}(e) = \emptyset \) and \( \text{chs}(e) \subseteq \text{wrs}(e) \). An effect specification is a triple \((e_1, e_2, e_3)\) of well-formed effects such that \( e_2 \subseteq e_3 \).

A specification \((e_1, e_2, e_3)\) approximates the behavior of a computation \( e \) as follows: \( e_1 \) summarizes side effects that may occur during the execution of \( e \) (corresponding to a guarantee condition in the rely-guarantee formalism [16]); \( e_2 \) summarizes effects of the interacting environment that \( e \) can tolerate while still functioning as expected (a rely condition). Finally, \( e_3 \) 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 \( e_3 \) 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 \( e_1 \), on the other hand, do not necessarily show up in \( e_3 \), for a computation might be able to clean up those effects prior to returning a final result.

The requirement that \( \text{rds}(e) \cap \text{chs}(e) = \emptyset \) is owed to the fact that all effects should preserve their own precondition; the precondition of \( \text{rd} \) is agreement on \( I \), which is not preserved by \( \text{ch} \). The requirement \( \text{chs}(e) \subseteq \text{wrs}(e) \) reflects that \( ch \) includes \( wr \) as a special case.

Consider computations \( e_1 = X := !X + 1; X := !X + 1 \) and \( e_2 = X := !X + 2 \). Let \( \text{e}_X \) stand for \( \{\text{rd}_X, \text{w}_X\} \). Each of the two computations can be assigned the effect \((e_X, \emptyset, e_X)\), but they are distinguishable at that effect typing. Let \( e \) be \( \text{if } X = 1 \text{ then diverge, } \) which has effect specification \((\emptyset, e_X, e_X)\). Assuming that \( e_1 = e_2 \) at type \((e_X, \emptyset, e_X)\), then from our parallel congruence rule (in Figure 5) we could derive that \( e_1 \upharpoonright e = e_2 \upharpoonright e \) at effect type \((e_X, e_X, e_X)\), which is clearly not true. Under the looser specification \((\text{ch}_X, \emptyset, e_X)\), however, \( e_1 \) and \( e_2 \) are indistinguishable, and our semantics is able to validate this equivalence, see Example 7.6.

A intuitive effect specification for the program \( !X \) is \text{int} \& \text{rd}_X | e | e, \text{rd}_X. However, it can also be assigned the effect \text{int} \& \emptyset | e | e, \text{rd}_X. Some effect specifications seem not to be needed in practice. The important ones are those \((e_1, e_2, e_3)\) that do not have read effects in \( e_1 \cup e_2 \).

We write \( e \upharpoonright e \) for \( e \) with all read effects removed and each \( wr \) in \( e \) replaced by \( ch \). We sometimes write \( \text{rd}_X, \text{w}_X, \text{ch}_X \) for \( \text{rd}_{\text{int}(X)}, \text{w}_{\text{int}(X)}, \text{ch}_{\text{int}(X)} \). Note that if \( e \upharpoonright e \) is a well-formed effect, then \( \text{rds}(e) \cap \text{wrs}(e) \cup \text{chs}(e) \) = \( \emptyset \). We use this observation to simplify some side conditions, abbreviating \( \{ch, wr\} \) by just \( ch \) in examples, so the chaotic effect silently implies the write effect.

Notations: For well-formed effects \( e, e' \) we write \( e \perp e' \) to mean \( \text{rds}(e) \cap \text{wrs}(e') = \text{rds}(e') \cap \text{wrs}(e) = \text{wrs}(e') \cap \text{wrs}(e) = \emptyset \). Note that this implies \( \text{chs}(e) \cap \text{chs}(e') = \emptyset \). We write \( h \sim \text{rd}(e) \) to mean \( h \sim \text{rd}(e) \) for each \( l \in \text{rds}(e) \). We write \( \sim \text{rd} \) for the transitive closure of \( \cup \{ \text{rds}(e) \uparrow \text{rds}(e) \} \). Thus, \( \sim \text{rd} \) allows steps by locations recorded as writing in \( e \) and silent steps by all locations in the current world. We define \( e_1 \cup e_2 \), appearing in the parallel congruence rule, by \( e_{1} \cup e_{2} = (e_{1} \cup e_{2}) \{ \text{wr}_{X}, \text{w}_{Y} \notin e_{1} \cap e_{2} \} \{ \text{ch}_{X}, \text{ch}_{Y} \notin e_{1} \cap e_{2} \} \).

6. TYPING AND CONGRUENCE RULES

Types are given by the grammar

\[
\tau ::= \text{unit} | \text{int} | \text{bool} | A | \tau_1 \times \tau_2 | \tau_1 \vdash e_1 \vdash e_3 \rightarrow \tau_2
\]

where \( A \) ranges over user-specified abstract types. They will typically include reference types such as \text{intref} and also types like lists, sets, and even objects. In \( \tau_1 \vdash e_1 \vdash \tau_2 \) the triple of effects \((e_1, e_2, e_3)\) must be an effect specification.

We use two judgments:

- \( \Gamma \vdash v \leq v' : \tau \) specifying that values \( v \) and \( v' \) have type \( \tau \) and that \( v \) approximates \( v' \).
- \( \Gamma \vdash e \leq e' : \tau \& e_1 \vdash e_2 \vdash e_3 \) specifying that the programs \( e \) and \( e' \) under the context \( \Gamma \) have type \( \tau \), with the effect specification \((e_1, e_2, e_3)\) 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 of the form \((v, v', \tau)\) where \( v, v' \) are values and \( \tau \) is a type, meaning that \( v \) and \( v' \) are claimed to be of type \( \tau \) and that \( v \) approximates \( v' \). These must be proved “manually” using the semantics, as they generally depend on the subtleties of particular abstract locations, but useful equational consequences can then be established by generic type-based rules.

We also define typing judgements \( \Gamma \vdash v : \tau \) and \( \Gamma \vdash e : \tau \& e_1 \vdash e_2 \vdash e_3 \) simply to be abbreviations for the ‘diagonal’ part of the inequational judgements, i.e. they hold when \( \Gamma \vdash v \leq v : \tau \) and \( \Gamma \vdash e \leq e : \tau \& e_1 \vdash e_2 \vdash e_3 \) can be derived from the rules from Figure 6.

We will justify all the rules semantically using a logical relation (Section 7) and conclude their soundness w.r.t. typed observational approximation and equivalence (Section 8). But we first sketch the intuition behind some of the rules.

The parallel composition rule states that \( e_1 \) and \( e_2 \) can be composed when their internal effects are not conflicting, in the sense that the internal effects of one appear as environment interaction effects of the other. Note the relationship to the parallel composition rule of the rely-guarantee formalism [16]. Also note that the effects of \( e_1 \) and \( e_2 \) are not required to be independent from each other as they are in the parallelization 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 \( \text{w}_X \in e_1 \) but cleans up this modification by eventually restoring the old value of \( X \). This would be reflected by \( \text{w}_X \notin e \cup e' \cup e_2 \). In that case, we would not expect to see \( \text{w}_X \) in the end-to-end effect of the parallel composition and that is precisely what \( \sqcup \) achieves.

The rules labelled \( \text{Sem} \) make available program transformations that are valid on the level of the \text{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 [11]. The other ones are concurrent version of analogous rules for sequential composition that have been analysed in previous work [8, 7, 30, 5] 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 composition 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 obvi-
ous against chaotic access, hence the \((-C)^C\) strengthenings
in the premise.

The reason for the appearance of \((-C)^C\) 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 \(e_1\) so long
as they are cleaned up by the time \(e_1\) completes and the
intervening environment does not notice (modelled by the
conditions \(e_1 \perp \varepsilon\) and final effect \(e_0 = e^{C}_C \cup \emptyset\)). In
the conclusion the transient effect \(e_1\) shows up again, but \((-C)^C\)
- 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.

The following definitions provide the semantics of effects.

**Definition 6.1 (Tiling).** Assume \(w \vdash \varepsilon\). Then we
write \([e](h, h', h_1, h_1')\) to mean that (i) \(h \models w \Rightarrow h \xrightarrow{\varepsilon} h_1\)
and (ii) \(h' \models w \Rightarrow h' \xrightarrow{\varepsilon} h_1'\) and (iii) \(h \xrightarrow{\mathrm{rd}(e)} h'\) and \(l \in \mathrm{wrs}(e) \) imply \(h = h_1 = h_1'\).

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 \(\varepsilon\), and if \(h, h'\) agree on the reads of \(\varepsilon\) then written locations will either be identically (equiva-

If the step relations of all abstract locations commute, then tiling admits an alternative characterisation in terms of
preservation of binary relations [8]. The above, more oper-

**Lemma 6.2.** Suppose that \(w \vdash \varepsilon\), \(w \vdash e_1\), \(w \vdash e_2\). The
following hold whenever well-formed.
1. \([e](h, h', h_1, h_1')\) and \([\varepsilon](h_1, h_1', h_2, h_2')\) imply \([e](h, h', h_2, h_2')\)
2. \([\varepsilon](h, h', h')\)
3. If \(e_1 \subseteq e_2\) then \([e_1](h, h', h_1, h_1') \Rightarrow [e_2](h, h', h_1, h_1')\)
4. \([e](h, h', h_1, h_1') \Rightarrow [e^C](h, h', h_1, h_1')\)
5. If \([\varepsilon](h, h', k, k')\) and \(h \xrightarrow{\mathrm{rd}(e)} h'\) then \(k \xrightarrow{\mathrm{rd}(e)} k'\). (This relies on \(\mathrm{rds}(e) \cap \mathrm{cns}(e) = \emptyset\).)
6. Suppose \([e](h, h', h_1, h_1')\). If \(h \models w\) then \(h_1 \models w\); if \(h' \models w\) then \(h_1' \models w\).

**7. LOGICAL RELATION**

**Definition 7.1 (Specifications).** A value specification
is a relation \(E \subseteq \forall \times \forall\) such that

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

Similarly, a computation specification is a relation \(Q \subseteq TV \times TV\) such that \(\leq; Q \leq \subseteq Q\) and \(Q\) is admissible qua relation 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. Admis-

**Definition 7.2.** If \(E \subseteq \forall \times \forall\) and \(Q \subseteq TV \times TV\) then
the relation \(E \rightarrow Q \subseteq \forall \times \forall\) is defined by

\[f E \rightarrow Q f' \iff \forall x \ x' \ (x E x') \Rightarrow (f(x) Q f'(x'))\]

In particular, for \(f E \rightarrow Q f'\) 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 \(E \rightarrow Q\).

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 intuitive proofs.

**Definition 7.4.** Let \(E \subseteq \forall \times \forall\) be a value specification
and \((e_1, e_2, e_3)\) an effect specification. We define the
relations \(T_0(E, e_1, e_2, e_3)\) and \(T(E, e_1, e_2, e_3)\) between sets of
trace-value pairs, i.e. on \(P(Tr \times Values)\):

\[(U, U') \in T_0(E, e_1, e_2, e_3)\] if and only if

\[
\forall \left(((h_1, k_1), \ldots, (h_n, k_n), a) \in U h_1 \models w \Rightarrow
\begin{align*}
\forall h_1' h_1'' \models w & \Rightarrow h_1' \xrightarrow{\mathrm{rd}(e_3)} h_1'' \\
\exists k'_1 = [e_1](h_1', k_1' , k_1'') & = \forall h_2, h_2' \models e_2((k_1', k_1'', k_2', k_2'')) \Rightarrow \exists k'_2 = [e_2](h_2', k_2' , k_2'') \land \forall h_3, h_3' \models e_3((k_2', k_2'', k_3', k_3'')) \\
\cdots & \\
\exists k'_n = [e_n](k_1'' , k_2'' , \ldots, k_n'') & \land \forall a' \in U : ((a, a') \in U) \land ((k_1'' , k_2'' , \ldots, k_n'') = a' )
\end{align*}
\]

We define the relation \(T(E, e_1, e_2, e_3) \subseteq TV \times TV\) as the
least admissible superset of \(T_0\).

**Remark 7.5.** Taking the admissible closure is necessary
for the validity of the fixpoint rule. The technical report [6]
explains how the underlying preconditions being \(\text{SFP}\) allows
these admissible closures to be safely ‘ignored’ in proofs.

The game-theoretic view of \(T_0(E, e_1, e_2, e_3)\) may be un-
derstood as follows. Given \(U, U' \in TV\) we can consider a game
between a proponent (who believes \((U, U') \in TV)\) and an
opponent who believes otherwise. The game begins by the
opponent selecting an element \(((h_1, k_1), \ldots, (h_n, k_n), a) \in U\)
and \(h_1 \models w\), the pilot trace, and a start heap \(h_1' \models w\) such that \(h_1 \xrightarrow{\mathrm{rd}(e_3)} h_1''\) to begin a trace in \(U'\). Then, the proponent answers with a matching heap \(k_1'\) so that \([e_1](h_1', k_1', k_1'')\).

If \(h_1 \xrightarrow{\mathrm{rd}(e_3)} h_1''\) does not hold, proponent does not need
to ensure that writes are in sync. The opponent then plays
a heap \(h_2\) so that \([e_2](k_1', k_2', h_2')\). At this point, it is in
the proponents interest to make sure that \(k_1' \xrightarrow{\mathrm{rd}(e_2)} k_1'\) for
otherwise opponent may make “funny” moves.

Then proponent plays heap \(k_2\) such that \([e_1](h_2, k_2, h_2', k_2')\),
etc until proponent has played \(k_2'\) so that \([e_1](h_n, k_n', k_n'')\).
After that final heap has been played, it is checked that
\([e_3](h, h', k, k')\) holds. If not, proponent loses. If yes, then
proponent must also play a value \(a'\) and it is then checked whether or not \(((h_i, k_i), \ldots, (h_n, k_n), a) \in U\) and \((a, a')\).
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 \((U, U') \in T_0(E, e_1, e_2, e_3)\) iff
the proponent has a winning strategy for that game.

Remark that by Lemma 6.2(6) well-formedness of heaps
w.r.t. the ambient world is a global invariant which we can
henceforth assume. We now illustrate the game with a few
examples.
Γ ⊢ true ≤ true : bool  Γ ⊢ false ≤ false : bool
Γ ⊢ e₁ ≤ e₂ : τ & e₁ | e₂ | e₃  Γ ⊢ e₁ ≤ e₂ : τ & e₁ | e₂ | e₃
Γ ⊢ e₁ ≤ e₃ : τ & e₁ | e₂ | e₃  Γ ⊢ n ≤ n : int  Γ ⊢ x : τ & x ≤ x : τ
Γ ⊢ v ≤ v' : τ  Γ ⊢ v ≤ v' : τ & e₁ | e₂ | e₃  Γ ⊢ (v₁, v₂) ≤ (v₁', v₂') : τ₁ × τ₂
Γ ⊢ n. e ≤ e' : τ  Γ ⊢ v ≤ v' : τ & e₁ | e₂ | e₃  Γ ⊢ (v₁, v₂) ≤ (v₁', v₂') : τ₁ × τ₂

Figure 4: Typing and congruence rules

Example 7.6. Consider again the programs e₁ = (X := !X + 1; Y := !X + 1) and e₂ = (X := !X + 2). Let I = \text{int}(X) be the abstract location for a single integer stored at X (see Section 4). Let E = [\text{unit}] = \{()\}, ()\} be the value specification for the unit type.

We show that ([\text{ch}_1], [\text{ch}_2]) \in T(E, \{\text{ch}_1, \epsilon, \epsilon \cup \{\text{rd}_1, \text{wr}_1\}\} under the assumption that \{\text{ch}_1\} \perp \epsilon, that is, when the environment does not read or write X. This condition is clearly necessary, for e₁ and e₂ can be distinguished by an environment that reads or writes X.

Let us now prove the claim when \{\text{ch}_1\} \perp \epsilon. The opponent picks a pilot trace in the semantics of e₁, for example, ((h₁, k₁)(h₂, k₂), () where h₁(X) = n and k₁(X) = n + 1 and h₂(X) = n′ and k₂(X) = n′ + 1. The other possible traces are stuttering or mismalling variants of this one and do not present additional difficulties. The opponent also chooses a heap h₃ such that h₁ \sim h₃, i.e., h₃(X) = n. Now the proponent will choose to stutter for the time being and thus selects k₃ \vDash h₃. Indeed, [(\epsilon)(h₁, h₃, k₁, k₃)] holds, so this is legal. The opponent now presents h₄ such that [\epsilon](k₁, k₃, h₂, h₄). By the assumption on \epsilon we know that n′ = h₂(X) = k₁(X) = n + 1 and also h₄(X) = k₄(X) = n. The proponent now answers with h₅ := ((h₃, h₄)|X→n+2). It follows that [\epsilon](h₃, h₄, k₂, k₃) and also [\epsilon](h₅, h₄, k₁, k₃). Finally, by stuttering (h₅, h₄, h₅|X→n+2) \in [e₂] so that proponent wins the game.

Example 7.7. Consider e₁ = (X := !X + 1)|Y := !Y + 1) and e₂ = (X := !X + 1; Y := !Y + 1). We show ([e₁], [e₂]) \in T(E, \{\text{ch}_X, \text{ch}_Y\}, \epsilon, \epsilon \cup \{\text{rd}_X, \text{rd}_Y, \text{wr}_X, \text{wr}_Y\}) provided \epsilon 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 [e₁], for example, the trace ([n₁|n₂], [n₁|n₂ + 1]|([n₁|n₂ + 1], [n₁ + 1|n₂ + 1]|()(), where [n₁|n₂] denotes a heap where X and Y store n₁ and n₂, respectively. Notice that in this trace, Y is incremented before X and since \epsilon 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 h₁ that agrees with the initial heap [n₁|n₂] on the reads of \epsilon∪\{rd_X, rd_Y, wr_X, wr_Y\}. Thus, h₁ should be of the form [n₁|n₂].

We now play the move ([n₁|n₂], [n₁ + 1|n₂ + 1]). This is a valid move as [\epsilon](h₁, h₁, h₂, h₃)|[n₁|n₂, [n₁ + 1, n₁|n₂ + 1, [n₁ + 1|n₂]). The environment moves returning [n₁ + 1|n₂] as it does not read nor modify X and Y. We can now match the trace above by playing ([n₁ + 1|n₂], [n₁ + 1|n₂ + 1]) and returning (), winning the game.

The following is one of our main technical results, and shows that the computation specifications T(\ldots) can indeed serve as the basis for a logical relation. We just show here the soundness proof for the parallel congruence rule. The missing proofs appear in the technical report [6].

Theorem 7.8. The following hold whenever well-formed.

1. If (U, U') \in T(E, e₁, e₂, e₃) then (q₁(U), q₂(U')) \in T(E, e₁, e₂).
2. T(E, e₁, e₂, e₃) is a computation specification.
3. If (U, U') \in T(E, e₁, e₂, e₃) then (U', U') \in T(E, e₁, e₂, e₃).
4. If (a, a') \in E then rtn(a), rtn(a') is in T(E, e₁, e₂, e₃).
Moreover, we have an environment move which forms the tile \([e']([k_0],n_0+1,h_1,n_2+1)]\). So the tile \([e \cup e_1]([h_1],h_2',n_1+1,n_2+1)]\) 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:

\[
(h'_m, k'_m) \cdots (h'_n, k'_n)
\]
8. 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 \( \tau \) if for all \( f \) such that \( \vdash f : \tau \overset{e_1|e_3}{\rightarrow} \text{int} \) (“observations”) it is the case that if \( (h_{\text{init}}, k), n) \in \{ f v \} \) for \( v \in \mathbb{Z} \) and starting from \( h_{\text{init}} \) then \( (h_{\text{init}}, k'), n) \in \{ f v' \} \) for some \( k' \). We write \( \vdash v \leq_{\text{obs}} v' \) in this case. We say that \( v \) and \( v' \) are observationally equivalent at type \( \tau \), written \( \vdash v =_{\text{obs}} v' \) if both \( \vdash v \leq_{\text{obs}} v' : \tau \) and \( \vdash v' \leq_{\text{obs}} v : \tau \).

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 \( \tau \). E.g. if \( \tau \) 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.

Observational approximation extends canonically to open terms by lambda abstracting free variables (and adding a dummy abstraction in the case of closed terms) [5].

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 \( \vdash v \leq_{\text{obs}} v' : \tau \).

Proof. If \( \vdash f : \tau \overset{e_1|e_3}{\rightarrow} \text{int} \) then by Thm 7.9 we have

\[
([f], [f]) \in [\tau] \overset{e_1|e_3}{\rightarrow} \text{int},
\]

so

\[
([f v], [f v']) \in T([\text{int}], e_1, e_2, e_3)^+.
\]

Let \( ((h_{\text{init}}, k), v) \in [f v] \). We have \( h_{\text{init}} = w \) and thus in particular \( h_{\text{init}} = w^{\text{rdt}(e_2)|\text{rdr}(e_1)} \). Hence there exist a matching heap \( k' \) and a value \( v' \) such that \( ((h_{\text{init}}, k'), v') \in [f v'] \) and \( v = v' \in \mathbb{Z} \).

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 \( e_2 \) form Example 7.6 we find that \( \lambda_{\epsilon}.e_1 =_{\text{obs}} \lambda_{\epsilon}.e_2 \) at type \( \text{unit} \overset{e_1|\text{val}(v_1, u_1)}{\rightarrow} \text{unit} \) whenever \( X \) does not appear in \( \epsilon \).

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 [8, 30], but the side conditions on the environmental interaction are now rather less obvious. We also note that some equations now only hold in one direction, i.e. become inequations. This is in particular the case for duplicated computations. Suppose that \( ? \) is a computation that nondeterministically chooses a boolean value and let \( e := \text{let } x = ? \text{ in } (x, x) \). Then, even though \( ? \) does not read nor write any location we only have \( e \leq (? , ?) \), but not \((? , ?) \leq e \) for \((? , ?) \) admits the result \( \text{true, false} \) 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 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 \( \varepsilon_1 \) and \( \varepsilon_2 \) in the premises are replaced by \( \varepsilon_1' \) and \( \varepsilon_2' \) 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 \( \varepsilon_1 \) and \( \varepsilon_2 \) can be modified in any way (see Example 7.7). However, in the global effect, we can still guarantee the effects \( \varepsilon_1' \) and \( \varepsilon_2' \) because of the \( \perp \)-conditions. This intuition appears directly in the soundness proofs.

Theorem 9.1. The following hold whenever well-formed.

- Commuting If \( (U_1, U'_1) \in T(E_1, e_1, e_2', e_2, e_1', e_2') \) and \( (U_2, U'_2) \in T(E_2, e_2, e_2', e_2, e_1', e_2') \) and \( e_1 \perp e_2 \) then

\[
\begin{align*}
& \left\{ \left( (t_1, v_1), (v_1, v_2) \right) \mid (t_1, v_1) \in U_1, (t_2, v_2) \in U_2 \right\}, \\
& \left\{ \left( t_1', v_1', v_2' \right) \mid (t_1', v_1', v_2') \in U'_1, (t_2', v_2') \in U'_2 \right\}
\end{align*}
\]

in \( T(E_1 \times E_2, e_1, e_2, e_1', e_2', e_1, e_2') \).

- Duplicated Given \( (U, U') \in T(E, e_1, e_2, e_3, e_2') \) with \( \text{rds}(e') \cap \text{wrs}(e') = \emptyset \) and \( e_2 \perp e_1 \), we have

\[
\begin{align*}
& \{ (t, v, v') \mid (t, v) \in U \}, \\
& \{ (t_1', v_1', v_2') \mid (t_1', v_1', v_2') \in U'_1 \} \in T(U, e_1, e_2, e_2')
\end{align*}
\]

- Pure Let \( (U, U') \in T(E, e_1, e_2, e_3, e_3') \), such that \( e_1 \perp e_2 \).

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 \( q_i = w \) and value \( v \), then \( \text{rtn}(v) \in U' \in T(E, e_1, e_2, e_2') \).

- Dead Suppose that \( (U, U') \in T(\text{unit}, e_1, e_2, e_3, e_1') \), where \( \text{wrs}(e_1') = \emptyset \) and \( e_1 \perp e_2 \). Then \( \{ (\text{rtn}(v)) \} \in T(\text{unit}, e_1', e_2, e_1') \).

- Parallelization If \( (U_1, U'_1) \in T(E_1, e_1, e_2, e_2', e_2, e_1', e_2', e_2, e_1') \) and \( (U_2, U'_2) \in T(E_2, e_2, e_2', e_2', e_2, e_1', e_2', e_2, e_1') \) and \( e_1 \perp e_2 \) and \( e_1 \perp e_2 \), then

\[
\begin{align*}
& \left\{ (t_1, (v_1', v_2')) \mid (t_1, v_1') \in U_1, (t_2, v_2') \in U_2 \right\}, \\
& \left\{ (t_1', v_1', v_2') \mid (t_1', v_1', v_2') \in U'_1, (t_2', v_2') \in U'_2 \right\}
\end{align*}
\]

in \( T(E_1 \times E_2, e_1, e_2, e_1', e_2', e_2, e_1', e_2') \).

Proof. We here sketch the soundness proof for parallelization. More details, and proofs for the other transformations, appear in the technical report [6].

Assume w.l.o.g. that the pilot trace is \( (t, (v_1, v_2)) \) where \( \text{inter}(t_1, t_2, t) \) and \( (t_1, v_1) \in U_1 \). Just as in the commuting case we set up two side games \( U \) vs. \( U'_1 \) on \( t_1, v_1 \). Unlike that case, however, these games are running simultaneously and along with the main game. Moves by the environment in the main game are forwarded to the side game we are currently in, i.e., the one to which the current portion of \( t \) being played on belongs. At each change of control, we
switch between the two side games making last sequence of moves of the other game into a single environment move. It is here that the resilience against chaotic modification is needed. Once the play is over we then assert the claims about the end-to-end effect $\varepsilon \cup \varepsilon' \cup \varepsilon''$ location by location using the definition of tiling. □

**Theorem 9.2.** Suppose that $\Gamma \vdash v \leq v' : \tau$ and $\Gamma \vdash e \leq e' : \tau$ and $\varepsilon \perp \varepsilon_1 \perp \varepsilon_2 \perp \varepsilon_3$ and assume that for each axiom $(v, v', \tau)$ it holds that $(v, v') \in \tau^+$. Then $(\varepsilon : \eta, \eta') \in \Gamma^+$ (interpreting a context as a cartesian product) implies $(\nu[v], \nu[v'] : \eta') \in \tau^+$ and $(\nu[e] : \varepsilon, \varepsilon') \in T(\tau^+, \varepsilon_1, \varepsilon_2, \varepsilon_3)$.

**Proof Sketch.** In essence the proof is by induction on derivations of inequalities. However, we need to slightly strengthen the induction hypothesis. Define

$$
\Gamma \vdash \tau = \{ (f, f') \mid \forall (\eta, \eta') \in \Gamma, (f(\eta), f'(\eta')) \in \tau \} \\
\Gamma \vdash \sigma = \{ (f, f') \mid \forall (\eta, \eta') \in \Gamma, (f(\eta), f'(\eta')) \in \tau \}
$$

We now show by induction on derivations of $\Gamma \vdash v \leq v' : \tau$ implies $(\nu[v], \nu[v']) \in \Gamma^+$ and that $\Gamma \vdash e \leq e' : \tau$ and $\varepsilon \perp \varepsilon_1 \perp \varepsilon_2 \perp \varepsilon_3$ implies $(\nu[e], \nu[e']) \in \tau^+$ where $\varepsilon = \varepsilon_1 \cap \varepsilon_2 \cap \varepsilon_3$.

The various cases now follow from earlier results in a straightforward manner. We use Theorem 7.8 for the congruence rules and Theorem 9.1 for the effect-dependent transformations.

As a representative case we show the case where $e \equiv \lambda x. e_1 \equiv e_1 \text{ in } e_2$ and $e' \equiv \lambda x. e_1 \equiv e_2$. Inductively, we know $(\nu[e_1], \nu[e_2]) \in \Gamma \vdash \tau \& \tau(\varepsilon_2, \varepsilon_3)$ and $(\nu[e_1], \nu[e_2]) \in \Gamma \vdash \tau \& \tau(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ for some $\varepsilon_1, \varepsilon_2 > 0$. By Theorem 7.9, we also have $(\nu[e_1], \nu[e_2]) \in \Gamma \vdash \tau \& \tau(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ 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 7.8 (6) repeatedly ($n_1$ times) so as to conclude

$$
(\nu[e], \nu[e']) \in \Gamma \vdash \tau \& \tau(\varepsilon_1, \varepsilon_2, \varepsilon_3)
$$

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.

We now return to the examples 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 $\varepsilon$ does not read nor write $X$:

$$
\text{readFst} : \text{unit} \xrightarrow{\emptyset \uparrow \varepsilon, \text{ch}_2 \text{msq}(X), \text{rd}_{\text{fst}}(X)} \text{int} \\
\text{writeFst} : \text{int} \xrightarrow{\uparrow \varepsilon, \text{ch}_2 \text{msq}(X), \text{wr}_{\text{fst}}(X)} \text{unit}
$$

The analogous typings for $\text{readSnd}$ and $\text{writeSnd}$ are elided. We justify this typing semantically as described in Theorem 7.8. To illustrate how this is done, consider the function (writeFst 17). We show how the game is played against itself using the typing shown above. We start with a “pilot trace”, say: $(\emptyset(3), \emptyset(3), (17)(3), (17)(3), (17)(3))$ 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).

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

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

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

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 $e_1, \varepsilon_1 \cup \varepsilon_2, e_2 \cup \varepsilon_2$, $e_2, \varepsilon_1 \cup \varepsilon_1$, $e_1 \cup \varepsilon_2$, where $\varepsilon_1 \cap \varepsilon_2 = \varepsilon_2 \cap \varepsilon_1 = \emptyset$. 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.

**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

$$
(\text{queue}, \text{atomic}(\text{dequeue}), \text{unit} \xrightarrow{\text{MSQ}} \text{int})
$$

where $\text{MSQ} = \{ \text{rd}_{\text{msq}}(X), \text{wr}_{\text{msq}}(X) \}$. 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 $(h_1, k_1, \ldots, h_n, k_n) \rightarrow \pi$. (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 $(h_i, k_i)$ the element $a$ is dequeued and $h_i \rightarrow k_i$ holds for $j \neq i$. We can thus match this trace by a trace in the semantics of atomic dequeue ($\pi$) by stuttering until $i$:

$$(h'_1, h'_2, \ldots, h'_n) \rightarrow (h'_1, \ldots, h'_i, h'_n) \rightarrow (h'_1, h'_2, \ldots, h'_n) \rightarrow \pi$$

where $h_i$ and $h'_i$ 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 $h_i$ and $h'_i$ 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.

$$
\ldots, (h'_{i+1}, \ldots, h'_n) \rightarrow \pi
$$

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 deducce the equivalence of $op_1 \parallel \cdots \parallel op_n$ and atomic($op_1 \parallel \cdots \parallel op_n$) for $op$, being enqueues or dequeues, which effectively amounts to linearizability [19].
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. This 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.

Interesting as those proofs are, we include them only to demonstrate the scope of our semantics. The most important contribution is the theory of effect-dependent equivalences. The theory smoothly but considerably extends earlier such theories proposed in the sequential settings [8, 30]. Notably, in the presence of concurrency the rules for code duplication, motion, and deletion, which in the sequential realm are fairly intuitive, get nontrivial side conditions. The same is true for the – effect-dependent – parallel congruence rule. Such rules are presented and justified here for the first time.

There is much research on modelling and verification of concurrency and some of the broad ideas here, such as rely-guarantee [16], are widely used. The traditional focus was simple program logics, but there is a growing body of impressive work on equivalences, abstraction and refinement, building on earlier work on separation and encapsulated state in sequential settings. Abstract locations, with custom notions of equivalence and evolution, are like the islands of Ahmed et al [3], and recent work of Turon et al [31] on relational models for fine-grained concurrency develops richer abstractions, notably state transition systems expressing inter-thread protocols that can involve ownership transfer, as well as a treatment of refinement for concurrent ADTs. Similarly, the ‘RGSim’ relation of Liang et al. for proving concurrent refinements under contextual assumptions also has many similarities with our logical relation [24, Def.4]. The idea of abstract locations that can overlap in concrete storage whilst appearing independent to clients also appears in work on ‘fictional’ separation [22, 18].

Most previous work aims at proving particular, concrete equivalences and refinements. Sophisticated logics such as Turon et. al.’s CaReSL [31] can verify more complex fine-grained algorithms than our system. However, our work is not an implementation using multiple, potentially overlapping, real locations. That would involve working with two levels of code and we do not yet know if it would work.

11. REFERENCES

That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, pages 56–72, 2016.


