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 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 can 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.

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 [21], effect systems have been used for purposes including region-based memory management [13], tracking exceptions [5, 28], communication behaviour [4] and atomicity [20] 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 studied the semantics of effect systems, with a focus on formally justifying such effect-dependent equational reasoning [6, 9, 12, 23, 30]. Our approach 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 $\Gamma \vdash e : ? e \tau e$ guarantees that the result of evaluating $e$ will be of type $\tau$ with side-effects at most $e$, under assumptions $\Gamma$ (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.

Shared-variable concurrency allows many more possible interactions. The environment now includes anything that may be running concurrently and, moreover, atomic steps of $e$ and its concurrent environment may be arbitrarily interleaved, so it no longer suffices 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.

This paper explores effect types as a lightweight interface language 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 smoothly 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 effect typing as a useful intermediate level of specification between conventional types (guaranteeing little about the behaviour of concurrent code) and richer, more complex, models and logics [31].

Our first contribution is a trace semantics for concurrent programs that explicitly describes possible interference by the environment. We extend Brookes semantics [15] 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, that is, the effect until finishing computation, which may allow “cleaning-up” some of the effects incurred during computation. Annotated function types $\tau_1 \xrightarrow{e_1} \tau_2$ also capture the effect during a call, $e_1$, the environmental interference, $e_2$, and the final effect, $e_3$. Rather than tracking effects 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.

Our second and main contribution is the soundness proof of a number of general program equivalences: Deadcode, Lambda-Hoist, Commuting, Duplication and Parallelization Rules. The parallelization rule, for example, describes when the parallel execution, $e_1;e_2$, of two programs, $e_1$ and $e_2$, can be approximated by their sequential execution $e_1;e_2$.

As our third contribution, we show that our denotational semantics can prove the equivalence of interesting programs, such as the soundness of an idealized Michael-Scott queue implementation. A longer account, with more examples and proofs, may be found in...
a companion technical report [7]. We start with some motivating examples:

**Equivalence modulo non-interference.** Our semantics justifies the equation (X := !X + 1; X := !X + 2) at the effect type unit & {chx} | e | e ∪ [rdx, wrx], 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 chx, 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 \( \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z} \), and consider the following functions:

\[
\text{readFst}() = p(!X).1 \\
\text{read_snd}() = p(!X).2 \\
\text{wrtFst} n = (\text{rec} \ y = \text{let} m = !X \text{ in} \\
\text{if} \ \text{cas}(X, m, p^{-1}(n, p(m).2)) \\
\text{then} () \ \text{else} \ \text{try} () ) \\
\text{wrt_snd} n = (\text{rec} \ y = \text{let} m = !X \text{ in} \\
\text{if} \ \text{cas}(X, m, p^{-1}(p(m).1, n)) \\
\text{then} () \ \text{else} \ \text{try} () )
\]

which multiplex two abstract integer references onto a single concrete one. Note that the write functions, wrtFst and wrt_snd, use compare-and-swap, 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 readFst followed by one to wrt_snd, 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 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 \( X_{\text{ver}} \) increments. We define

\[
\text{transact} f = \text{let rec try}() = \text{let} \ \text{val, ver} = \text{atomic}(!X_{\text{val}},!X_{\text{val}}) \\
\text{in let res = f(val) in if atomic(if \ (X_{\text{val}} = \text{ver} \ \text{then} \\
X_{\text{ver}} := \text{ver} + 1; \ X_{\text{val}} := \text{res}; \ \text{true else false}) \\
\text{then} () \ \text{else try}() )
\]

Under the assumption that \( f \) is a pure function (has effect type \( \mathbb{Z} \to \mathbb{Z} \) int for any \( e \)), we can show

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

at type unit & \( [rdx, wrx] \) | e | e ∪ [rdx, wrx] for any e not involving chaotic access, chx, to X. The environment effect e here may include reading and writing X, so concurrent calls to 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.

---

\(^1\) Being equal at a type means being may-indistinguishable for any observations which use the terms in question with that type.
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 \(h\), then, is a finite map from \(L\) 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 \(h[X\mapsto c]\) 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 \(\text{new}(v, v)\) yields a pair \((X, h')\) where \(X \in L\) is a fresh location and \(h' \in \mathbb{H}\) is \(h[X\mapsto v]\).

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

\[
\begin{align*}
v &::= x | (v_1, v_2) | v_1 | c | \text{rec } f = t \\
e &::= \text{let } x = e_1 \text{ in } e_2 | v_1 | \text{if } f \text{ then } e_1 \text{ else } e_2 | ![v] | v_1 := v_2 | \text{ref}(v) | e_1[e_2] | \text{atomic}(e)
\end{align*}
\]

Here, \(x\) ranges over variables, \(v\) 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 \(F_{\text{let}}(n, n') = n + n'\) for integers \(n, n'\).

The construct \(\text{rec } f = e\) 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 \(v_2\) with value \(v_1\), 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 shorthand for let-expanded versions in examples.

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, \(\text{atomic}(e)\) evaluates an arbitrary \(e\) in one step, without any environmental interference. One can then define a (more realistic) compare-and-swap operation \(\text{cas}(X, v_1, v_2)\):

\[
\text{cas}(X, v_1, v_2) = \text{atomic}(\text{if } X = v_1 \text{ then } X := v_2; \text{true else false})
\]

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, \(\text{FV}(e)\), 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 [15]. Fuller details, including a proof of adequacy with respect to a small-step operational semantics using interleaving, can be found in [7].

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 \(h_1(k_1), h_2(k_2), \ldots, h_n(k_n)\) where for \(1 \leq j \leq n\), we have \(h_i, k_i \in \mathbb{H}\) and \(\text{dom}(h_i) \subseteq \text{dom}(h_{i+1}), \text{dom}(k_i) \subseteq \text{dom}(k_{i+1})\). We write \(T\) for the set of traces.

Let \(t\) be a trace. A trace of the form \(\text{u}(h, h', v)\) where \(t = uv\) is said to arise from \(t\) by stuttering. A trace of the form \(\text{u}(h, k, v)\) where \(t = uh, q(k, v)\) is said to arise from \(t\) by mumbling. For example, if \(t = (h_1, k_1)(h_2, k_2)(h_3, k_3)\), then \((h_1, k_1)(h_2, k_2)(h_3, k_3)\) arises from \(t\) by stuttering and \(h_2, k_2\) 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 3.2 (Trace Monad).** Let \(A\) be a predomain (\(\omega\)-cpo not necessarily with bottom). Elements of the domain \(\mathcal{T} A\) 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 \((t', a') \in U\).
- \([\text{Down}]\): if \((t, a_1) \in U\) and \(a_2 \leq a_1\) then \((t, a_2) \in U\).
- \([\text{Sup}]\): if \((a, a)\) is a chain in \(A\) and \((t, a) \in U\) for all \(i\) then \((t, a, a) \in U\).

The elements of \(\mathcal{T} A\) are partially ordered by inclusion.

An element \(U\) of \(\mathcal{T} A\) represents the possible outcomes of a nondeterministic, interactive computation with final result in \(A\). Thus, if \((t, a) \in U\) for \(t = (h_1, k_1) \ldots (h_n, k_n)\) 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. After that, this particular computation ends and \(a\) is the final result value.

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

\[
\begin{align*}
&([10], [11], [15], [16]), 12 \\
&([10], [11], [15], [16]), 16 \\
&([10], [11], [15], [16]), [17, 17]), 17 \\
&([10], [11], [15], [16]), [17, 17]), 16 \\
&([10], [11], [15], [16]), [17, 17]), 16
\end{align*}
\]

Axioms [S&M] are taken from Brooks. 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. Note that non-termination is modelled by the empty set, so we are working with an angelic ‘may semantics’ [27]. For example, the semantics

---

Image: Figure 2. Enqueue, Dequeue, Membership, and Reset programs for a Michael-Scott Queue at location head.
of $X := 0$; if $X = 0$ then 0 else diverge is the same as that of $X := 0$ 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 $? \in \text{arity}$ denotes a nondeterministically chosen boolean value.

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

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

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 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 [1], forward-backward simulation [25], or speculation [31] in reasoning.

Axioms [Down] and [Sup] are known from the Hoare powerdomain [29]. Additional non-deterministic outcomes that are less defined than existing ones are not recorded in the semantics.

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

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

\[
\text{rtm}(a) := \{((h\cdot h), a) \mid h \in H\}^l
\]

\[
\text{bnd}(f, g) := \{((u\cdot v), (u, a), e \in g \land (v, b) \in f(a))\}^l
\]

These endow $TA$ with the structure of a strong monad. A partial function $f : H \rightarrow H \times A$ (an element of the state monad $SA$) can be (continuously) transformed into an element $\text{fromstate}(e)$, where $\text{fromstate} : SA \rightarrow TA$ is defined by $\text{fromstate}(e) := \{((h, k), a) \mid c(h) = (k, a))^l$. If $t_1, t_2, t_3$ are traces, we write $\text{inter}(t_1, t_2, t_3)$ to mean that $t_1$ 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

\[
\mid : TA \times TB \rightarrow TA \times B
\]

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

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

\[
((h_1, h_2)(h_2, h_3), \ldots, (h_n-2, h_n-1)(h_n-1, h_n), v) \in U
\]

where $h = h_1$ and $h_0 = h$. The presence of such an element models an atomic execution of the computation represented by $U$.

**3.1 Semantic values**

The predomain $V$ of untyped values is the least solution of

\[
\begin{align*}
V & \equiv \forall B + (V \rightarrow TV) + V^*.
\end{align*}
\]

That is, 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. The definition is entirely standard and may be found in [7]. 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 (see Lemmas A.1 and A.2 in the Appendix).

The semantics of values $\llbracket v \rrbracket \in V \rightarrow V$ and terms $\llbracket f \rrbracket \in V \rightarrow TV$ 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 $\rho[x \mapsto v]$ 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 [6]. 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. 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 $l$ consists of the following data:

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 $\equiv$ 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 $h \equiv h'$ 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 $\equiv$) relation $\rightarrow$, 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 $h \equiv h_1$ then $h_1$ might arise by writing to $l$ in $h$ and all possible writes are specified by $\rightarrow$. We call $\rightarrow$ the step relation of $l$.

These data must satisfy the following conditions where $h : l$ stands for $h \sim h$

1. If $h : l$ then $h \equiv h$;

2. if $h \rightarrow h_1$ then $h : l$ and $h_1 : l$.

If $h \equiv h_1$ and at the same time $h \equiv h_1$, 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 suitable for our previous motivating examples.

**Single integer.** Our simplest example is the following abstract location, parametric with respect to concrete location $X$:

\[
\begin{align*}
&\text{h} \quad \equiv \quad \exists n. h(X) = \text{int}(n) \land h'(X) = \text{int}(n) \\
&h \quad \equiv \quad h' \\
&h \quad \equiv \quad \text{int}(X), h_1 : \text{int}(X) \land \forall X' \in \mathbb{L}, X' \neq X \Rightarrow h(X') = h_1(X')
\end{align*}
\]
Two heaps are semantically equivalent (w.r.t. \(\text{int}(X)\) that is) if the values stored in \(X\) are integers and equal; the step relation requires all other concrete locations to be unchanged. We sometimes 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.

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

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 \(X_{\text{val}}\), and its relations are:

\[
\begin{align*}
\text{h} \sim \text{h}' & \iff h(X_{\text{val}}) = h'(X_{\text{val}}) \\
\text{h} \equiv \text{h}' & \iff h \sim h' \\
\text{h} \sim \text{h}' & \iff (\forall X'. h(X') \neq h'(X')) \land h(X) = h'(X) \land h(X_{\text{val}}) = h'(X_{\text{val}}) \\
\end{align*}
\]

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 \(\text{h} \sim \text{h}'\) 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 \(\text{h} \sim \text{h}'\) 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 able 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 \(X^{\text{STG}}\)-class.

The relation \(\text{next}(s)\) 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 \(\text{head}, \text{next}, \text{elem}\) using some layout convention, e.g. \(v.\text{head} = v.1\), etc. We then define

\[
\begin{align*}
\text{h}(X) \text{.next} = X' \iff X' \text{ can be reached from } h(X) \\
\end{align*}
\]

by following a chain of next pointers.

We use \(\text{List}(X, h, (X_0, \ldots, X_n), (v_1, \ldots, v_n))\) to signal that \(h(X)\) 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 \(\text{elem}\) component is ignored. Formally:

\[
\begin{align*}
\text{h}(X).\text{head} = X_0 & \quad \text{h}(X).\text{elem} = v_i \text{ for } i = 1, \ldots, n \quad \text{h}(X).\text{next} = X_{i+1} \text{ for } i = 0, \ldots, n - 1 \quad \text{h}(X).\text{next} = \text{null} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{fp}(X, h) = \{ X' \mid \text{next}(X) \land X' \land X') \}
\end{align*}
\]

Finally, we define \(\text{sno}\) 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, \(\text{List}(X, h, (X_0, \ldots, X_n), (v_1, \ldots, v_n))\) implies \(\text{List}(X', h, (X_0, \ldots, X_n, X_{n+1}), (v_1, \ldots, v_n, v))\) for some \(X_{n+1} \notin \text{dom}(h)\). We omit the obvious frame conditions. We now define

\[
\begin{align*}
\text{h} \sim \text{h}' & \iff \exists X'. X'.\text{List}(h, X, \bar{X}, \bar{v}) \land \text{List}(h', X, \bar{X}, \bar{v}) \\
\text{h} \equiv \text{h}' & \iff h \sim h' \land \forall X'. h(X) = h'(X') \\
\text{h} \sim \text{h}' & \iff \exists X'. X'.\text{List}(h, X, \bar{X}, \bar{v}) \\
\end{align*}
\]

In these examples, the only silent moves are identities. But data-structures such as collections that reorganize during lookups, or which use late initialization [6] do involve non-trivial silent moves.

**4.1 Worlds**

We group the abstract locations used by a program into a world. In this paper, all these abstract locations must be established up front. While concrete locations may be dynamically allocated to grow an abstract location, as in the MSQ example, we do not allow worlds to evolve dynamically. We have previously shown [3, 6] 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 defined as the largest relation such that \(h \models w\) implies

- \(h \models \{\text{all } l \in w\}\)
- \(\text{if } l \in w \land h \models h_1 \text{ then } h \models h_1\) holds for all \(l' \in w\) with \(l' \neq 1\) and \(h_1 \models w\).
We remark that if our world \( w \) contains two “interfering” 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 \( h \models w \).

We assume a fixed current world \( w \) which may appear in definitions without being notionally reflected. (See Assumption 1 later.)

5. Effects

For each abstract location \( l \) we have three elementary effects, \( rl \), (reading from \( l \)), \( wr \), (writing to \( l \)), and \( ch \), (chaotic access). An effect, ranged over by \( e \), 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 \( wrx \) effect, but \( e_1 \) and \( e_2 \) are distinguishable by contexts that assume the \( wrx \) effect. Thus, \( e_1 \) and \( e_2 \) are not equal “at type” \( wrx \). At type \( chx \) they are, however, equal, because a context that copes with this effect may not assume that both produce equal results.

So \( ch \) 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 \( X := 1 + 1 ; X := 1 + 1 \) is equivalent to \( X := 1 + 2 \) “at type” \( unit \). & \( chx \), \( rds(e) \) is well-formed (with respect to the \( X \) locations), but useful equational consequences can then be established by generic type-based rules.

All the \( e \) carry the meaning that \( (rds(e) \cap \text{wr}(X) \cup \text{chs}(X)) = \emptyset \). We use this observation to simplify some side conditions. In our concrete example, we abbreviate \( \{ch \} \cup \{wr\} \) by just \( ch \), in other words, the chaotic effect silently implies the write effect.

Notations: For any well-formed effects \( e, e' \) we use the notation \( e \perp e' \) to mean that \( rds(e) \cap \text{wr}(e') = rds(e') \cap \text{wr}(e) = \emptyset \). Note that this implies in particular \( \text{chs}(e) \cap \text{rds}(e') = \emptyset \), etc. We write \( h \overset{(e)}{\leadsto} h' \) to mean \( h \overset{e}{\preceq} h' \) for each \( e \) in \( rds(e) \). We write \( \downarrow \) for the transitive closure of \( \cup \text{loc}(\downarrow) \). Thus, \( \downarrow \) allows steps by locations recorded as writing in \( e \) and silent steps by all locations in the current world. We define the notation \( e_1 \parallel e_2 \) which appears in the parallel congruence rule by \( e_1 \parallel e_2 = (e_1 \cup e_2) \setminus \{\text{wr}, \text{wr} \neq e_1 \cup e_2\} \setminus \{ch\} \). We use \( ch \) as abbreviations for the ‘diagonal’ part of \( \{rl, \text{wr}, \text{ch}\} \).

6. Typing and congruence rules

Types are given by the grammar

\[ \tau ::= \text{unit} | \int \mid \text{bool} | A | \tau_1 \times \tau_2 | \tau_1 \frac{e_1 \downarrow e_2}{\tau_2} \]

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

We use two judgments:

- \( \Gamma \vdash \nu \leq \nu' \vdash \tau \) specifying that values \( \nu \) and \( \nu' \) have type \( \tau \) and that \( \nu \) approximates \( \nu' \).
- \( \Gamma \vdash e \leq e' \vdash \tau \) and \( e_1 \mid e_2 \mid e_3 \) 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 each having the form \((\nu, \nu', \tau)\) where \( \nu, \nu' \) are values in the metalanguage and \( \tau \) is a type meaning that \( \nu \) and \( \nu' \) are claimed to be of type \( \tau \) and that \( \nu \) approximates \( \nu' \). These must be proved “manually” using the semantics, because they generally depend on the subtleties of particular abstract locations, but useful equational consequences can be then established by generic type-based rules.

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

In what follows, we will justify all the rules semantically using a logical relation (Section 7) and then 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 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 interaction (Section 7) and then conclude their soundness w.r.t. fixpoint unrolling, and beta and eta equalities.

The appearance of the \( \cup \) operation deserves special mention. It might be, for example, that \( e_1 \) modifies \( X \) on the way, thus \( wrs \in e_1 \) but cleans up this modification by eventually restoring the old value of \( X \). This would be reflected by \( wrs \notin e_\cup e_\cup e_2 \). In that case, we would not expect to see \( wrs_2 \) in the end-to-end effect of the parallel composition and that is precisely what \( \cup \) 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 analogues rules for sequential computation that have been analysed in previous work [6, 8, 9, 30] 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 \( e_1 \) so long as they are cleaned up by the time \( e_1 \) completes and the intervening environment does not notice (modelled by the conditions \( e_1 \parallel e \) and final effect \( e^f = e^f \cup \emptyset \)). In the conclusion the transient effect \( e_1 \) 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 computability 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).** Let \( w \vdash e \). We write \([e](h, h', h_1, h_1')\) to mean that (i) \( h \vdash w \Rightarrow h' \Downarrow h_1 \) and (ii) \( h' \vdash w \Rightarrow h' \Downarrow h_1' \) and (iii) \([\text{dfs}(e)](h_1, h_2, k_i, k'_i) \wedge [\text{dfs}(e)](k_i, k'_i, h_1, h_1') \Rightarrow [\text{dfs}(e)](h_1, h_2, k'_i, k'_i) \wedge [\text{dfs}(e)](k_i, k'_i, h_1, h_1') \Rightarrow \ldots \)

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 \( e \), and if \( h \) and \( h' \) agree on the reads of \( e \) then written locations will either be identical (equivalently) 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 above, more operational, version is inspired by that of Birksedal et al [12].

**Lemma 6.2.** Suppose that \( w \vdash e \), \( w \vdash e_1 \), \( w \vdash e_2 \). The following hold whenever well-formed.

1. If \([e](h, h', h_1, h_1') \) and \([e](h, h', h_2, h_2') \) then \([e](h, h', h_2, h_2') \).
2. If \([e](h, h', h, h') \) then \([e](h, h', h, h') \).
3. If \( e \equiv e \) \( [e](h, h', h_1, h_1') \Rightarrow [e](h, h', h_1, h_1') \).
4. If \([e](h, h', h_1, h_1') \Rightarrow [e](h', h_1, h_1') \).
5. If \([e](h, h', k, k') \) and \( h \text{dfs}(e) h' \) then \( k \text{dfs}(e) k' \).
6. If \( h \vdash w \Rightarrow h_1 \vdash w \Rightarrow h_1' \vdash w \).

**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 \( x \text{E} y \) then \( x_1 \text{E} y_1 \);
- if \( (x_i), (y_i) \) are chains such that \( x_i \text{E} y_i \) then \( \text{sup}_i x_i \text{E} \sup_i y_i \), i.e., \( E \) is admissible qua relation;
- if \( x \text{E} y \) then \( p_i(x) \text{E} p_i(y) \) 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 \( Q \subseteq Q \) and \( Q \) is admissible qua relation and \( Q \) is closed under the canonical deflations \( q \).

The requirement \( Q \subseteq E \subseteq Q \) ensures smooth interaction with the down-closure built into our trace monad. Admissibility is needed for the soundness of recursion and, finally, closure under the canonical deflections, is needed so that Lemma A.2 can be applied.

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

\[ f \text{E} \rightarrow Q' \iff \forall x, y. (x \text{E} f(y)) \Rightarrow (f(x) \text{Q'} f'(y')) \]

In particular, for \( f \text{E} \rightarrow Q' \) 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 very intuitive proofs.

**Definition 7.4.** Let \( E \subseteq \mathbb{V} \times \mathbb{V} \) 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

\[ \forall (h_1, k_1, \ldots, h_n, k_n), (a, 0) \in U \Rightarrow w \vdash h_1, k_1' \vdash w \Rightarrow h_1 \text{dfs}(e) h_1' \Rightarrow \ldots \Rightarrow \exists k'_1, (e_1) (h_1, h'_1, k_1, k'_1) \wedge \exists k'_2, (e_2) (h_2, h'_2, k_2, k'_2) \wedge \exists k'_3, (e_3) (h_1, h'_1, k_1, k'_1) \wedge \exists k'_4, (a') (\in (e_4) (h'_1', k'_1', k'_1') \wedge \exists a' \in U' \wedge (a', a') \in E \wedge ((h'_1', k'_1', k'_1'), (a', a')) \in U' \]

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. We explain in [7] that due to the fact the underlying predominates are SFP, these admissible closures can safely be ignored in proofs.

The game-theoretic view of \( T_0(E, e_1, e_2, e_3) \) may be understood as follows. Given \( U, U' \in TV \) we can consider a game between a proponent (who believes \( (U, U') \) is \( TV \)) and an opponent who believes otherwise. The game begins by the opponent selecting an element \( (h_1, k_1, \ldots, h_n, k_n), (a, 0) \in U \) and \( h_1 \vdash w \), the pilot trace, and a start heap \( h_1' \vdash w \) such that \( h_1 \text{dfs}(e) h_1' \) to begin a trace.

7  2016/5/22
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.

**Example 7.6.** Consider again the programs $e_1 = (X := !X + 1; X := !X + 1)$ and $e_2 = (X := !X + 2)$. Let $\Sigma = \text{Int}(X)$ be the abstract location for a single integer stored at $X$ (see Section 4). Let $E = \text{unit}\{X\} = \{(0, 0)\}$ be the value specification for the unit type.

We show that $\langle \{X = 1\}, \{X = 2\} \rangle \in \Sigma(E, [e_1]_\Sigma, [e_2]_\Sigma)$ under the assumption that $[c_{\text{rd}}]_\Sigma = x$, 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 that reads or writes $X$.

Let us now prove the claim when $[c_{\text{rd}}]_\Sigma$. The opponent picks a pilot trace in the semantics of $e_1$, for example, $(\langle (h_1, k_1)(h_2, k_2), () \rangle)$ where $h_1(X) = n$ and $k_1(X) = n + 1$ and $h_2(X) = n'$ and $k_2(X) = n' + 1$. The other possible traces are stuttering or mumbling variants of this one and do not present additional difficulties. The opponent

in $U'$. Then, the proponent answers with a matching heap $k'_1$ so
that $[e_1][h_1, h'_1, k_2, k'_2]$. If $h'_1$ does not hold, proponent
does not need to ensure that writes are in sync. The opponent then
plays a heap $h'_2$ so that $[e_2][k_1, k'_1, h'_2, h'_2]$. At this point, it is in
the proponents interest to make sure that $k'_1$ for otherwise
opponent may make "funny" moves.

Then, again, proponent plays a heap $k'_2$ such that $[e_1][h_2, h'_2, k_2, k'_2]$ and so on until, proponent has played $k'_2$ so that $[e_1][h_2, h'_2, k_2, k'_2]$. After that final heap has been played, it is checked that $[e_2][h, h'_2, k, k'_2]$. If not, proponent loses. If yes, then proponent must also play a value $a'$ and it is then checked whether or not $(h'_1, k'_1), \ldots (h'_n, k'_n), a' \in U'$ 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 $(U, U') \in T_\Sigma(E, e_1, e_2, e_3)$ iff the proponent has a winning strategy for that game.
also chooses a heap $h'_t$ such that $h \vdash h'_t$, i.e., $h'(X) = n$. Now the proponent will choose to stutter for the time being and thus selects $k'_t := h'_t$. Indeed, $\langle ch \rangle(h_t, h'_t, k_t, k'_t)$ holds, so this is legal. The opponent now presents $h'_t$ such that $\langle e \rangle(k_t, k'_t, h_t, h'_t)$. By the assumption on $e$ we know that $n' = h_t(X) = k_t(X) + n + 1$ and also $h_t(X) = k'_t(X)$ = $n$. The proponent now answers with $k'_t := h'_t[X \mapsto 0 + 2]$. It follows that $\langle ch \rangle(k_t, h_t, k'_t, k'_t)$ and also $[r_d, w_r][h_t, h'_t, k_t, k'_t]$. Finally, by stuttering $(h'_t, h'_t)[h'_t, h'(X \mapsto 0 + 2)]\in [e_2]$ so that the proponent wins the game.

**Example 7.7.** Consider the following programs $e_1$ and $e_2$:

$$X := \{X + 1 \mid Y := Y + 1\} \quad \text{and} \quad (X := \{X + 1 \mid Y := Y + 1\})$$

We show $([e_1], [e_2]) \in \mathcal{T}(E, \{ch, ch_1\}, e, e \cup \{r_d, w_d, r_w, w_r\})$. Provided $e$ 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 $\langle [k_1], [k_1, k_2 + 1], [k_1, k_2 + 1], [k_1, k_2 + 1] \rangle$. Notice that in this trace, $Y$ is incremented before $X$ and since $e$ 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 $h'_t$ that agrees with the initial heap $[k_1, k_2]$ on the reads of $e$ and $[r_d, w_d, r_w, w_r]$. Thus, $h'_t$ should be of the form $[k_1, k_2]$.

We now play the move $([k_1], [k_1, k_2])$. This is a valid move in the game as $\langle ch, ch_1 \rangle([k_1], [k_1, k_2], [k_1, k_2], [k_1, k_2 + 1])$. The environment moves returning $[k_1 + 1, k_2 + 1]$ as it does not read nor modify $X$ and $Y$. We can now match the trace above by playing $([k_1, k_2], [k_1, k_2 + 1])$ and returning $(\), (\))$, winning the game.

The following is one of the main technical results of our paper 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 attached Appendix.

**Theorem 7.8.** The following hold whenever well-formed.

1. If $(U, U') \in T(E, e_1, e_2, e_3)$ then $q(U, q(U')) \in T(E, e_1, e_2, e_3)$.
2. If $(e, e_1, e_2, e_3)$ is a computation specification.
3. If $(U, U') \in T(E, e_1, e_2, e_3)$ then $(U', U'') \in T(E, e_1, e_2, e_3)$.
4. If $(a, a') \in E$ then $(r_m(a), r_m(a')) \in T(E, e_1, e_2, e_3)$.
5. Suppose that $(e_1, e_2, E)$ is an effect specification where $e_1 \subseteq e_2 \subseteq e_3$.

**Example 7.9.** Consider the following heap $h'_t$ such that $h'$ $\vdash\vdash (\langle e \rangle(h_t, h'_t, k_t, k'_t))$. Since $rds(e_1, e_2, e_3) = rds(e_1, e_2), rds(e_2, e_3)$, we can apply strategy $S_1$ to guide us through the first part of the game, obtaining:

$$(h'_t, k'_t) \cdot (h'_t, k'_t)$$

Moreover, we have an environment move which forms the tile $\langle e \rangle(h_t, h'_t, k_t, k'_t, k_t, k_t)$. Thus, we have the tile $\langle e \rangle(h_t, h'_t, k_t, k'_t, k_t, k_t)$. This 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:

$$(h'_t, k'_t) \cdot (h'_t, k'_t)$$

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 $(U \cup U')$.

Let $(a, a')$ be the final values reached at the end. It is clear that $\langle e \cup e' \cup e_1, k_t, h'_t, k'_t \rangle$ and also $aEa'\delta$.

It remains to assert the stronger statement $\langle e \cup e' \cup e_1, k_t, h'_t, k'_t \rangle$. To see this suppose that $w_r \in e_1 \cup e_2 \cup e' \cup e \cup e' \cup e_1$. Since the entire game can be viewed as an instance of the game $U_1 \cup U'_1$ with interventions by $U_2 \cup U'_2$ regarded as environment interactions we have $\langle e \cup e' \cup e_1, h'_t, k'_t, h'_t \rangle$ so that in fact $h \vdash k'_t$ and $h' \vdash h'_t$. The case of $ch$ and $e_1, e_2$ interchanged is analogous.

We assign a value specification $\tau$ to each refined type by:

- $\tau = \{ (v, v') \mid v \in \mathbb{N} \}$
- $\tau \cdot \tau \cdot \tau = \tau \cdot \tau \cdot \tau$
- $\tau \cdot \tau \cdot \tau = \tau \cdot \tau \cdot \tau$
- $\tau \cdot \tau \cdot \tau = \tau \cdot \tau \cdot \tau$
- $\tau \cdot \tau \cdot \tau = \tau \cdot \tau \cdot \tau$

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

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

- The initial heap satisfies the current world: $h_{\text{init}} \models w$.
- Each axiom is type sound: whenever $(v, v', \tau)$ is an axiom then $(v, v) \in \tau$ and $(v', v) \in \tau$.
- Each axiom is inequationally sound: whenever $(v, v', \tau)$ is an axiom then $(v, v') \in \tau$.

**Corollary 7.9.** Suppose that $\tau \vdash v : \tau$ and $\tau \vdash e : \tau \land e_1 \land e_2 \land e_3$. Then $(\langle v \rangle) \in \tau$ (interpreting a context as a cartesian product) implies $(\langle v \rangle) \in \tau$ and $(\langle v \rangle) \in \tau$.

Proof. By induction on derivations. Most cases are already subsumed by Theorem 7.8. 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 $v : \tau$ and $v' : \tau$. We say that $v$ observationally approximates $v'$ at type $\tau$ if for all $f$ such that $\tau : f : \tau \rightarrow \tau$, we write $\vdash v \approx v'$ if $v \approx v' \forall \tau \in \text{type}(\tau)$.

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 we will respect the contracts set up by the type r. 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 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']) ∈ [τ]. Then v ⊑obs v' → τ.

**Proof.** If v → τ := ε→ε | int then by Thm 7.9 we have ([f], [f]) ∈ [τ := ε→ε | int], so ([f], [f']) ∈ T([int], e1, e2, ε3)∗.

Let (hobs, k), v ∈ [f]. We have hobs := w and thus in particular hobs = w. This means there exist a matching heap k' and a value v' such that ([hobs, k'], v') ∈ [f v'] and v = v' ∈ Z.

This means that the examples from earlier on give rise to valid transformations in the sense of observational approximation. For instance, for e1 and e2 form Example 7.6 we find that λ e1 e2 at type unit [l:(x:τ)|r:(x:τ)] ε whenever X does not appear in ε.

### 9. Effect-dependent transformations

We will now establish the semantic theory of the effect-dependent program transformations given in Figure 5. It includes concurrent versions of the effect-dependent equations from [9, 30], but the side conditions on the environmental interaction are by no means obvious. We also note that some equations now hold only 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 := λ x = ? in (x, x). Then, even though ? does not read nor write any location we only have e ⊑ ? ≤ ε, but not (∈, ?) ≤ ε for (ε, ?) admits the result (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 e1 and e2 in the premises are replaced by ε1 and ε2 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 e1 and e2 can be modified in any way (see Example 7.7). However, in the global effect, we can still guarantee the effects ε1 and ε2 because of the ⊑-conditions. This intuition appears directly in the soundness proofs.

The following thus constitutes the second main technical result of our paper. We sketch the soundness proof for parallelization. The detailed proofs appear in the attached Appendix.

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

- **Commuting** If (U1, U′1) ∈ T(E1, e1, e′1, e2 ∪ e3) and (U2, U′2) ∈ T(E2, e2, e′2, e2 ∪ e3) and e1 ⊑ e and e2 ⊑ e and e′1 ⊑ e′2 then
  
  \[ ((t1, t2), (v1, v2)) \mid (t1, v1) \in U1, (t2, v2) \in U2 \],
  
  \[ ((t′2, ′1, v′2, v′1)) \mid (t′1, v′1) \in U′1, (t′2, v′2) \in U′2 \] \in T(E1 × E2, e1 ∪ e2 ∪ e3, e, e′1 ∪ e′2)

- **Duplicated** If (U, U′) ∈ T(E, e1, e′1, e′2) and rds(e′) ∩ wrs(e) = ø and e2 ⊑ e3, then
  
  \[ ((t, v)) \mid (t, v) \in U \],
  
  \[ ((t′1, v′1)) \mid (t′1, v′1) \in U′ \] \in T(E, e1, e2, e3 ∪ e′1).

- **Pure** Let (U, U′) ∈ T(E, e1, e′1, e′2), such that e1 ⊑ e2. If ((q1, k1) . . . (qk, k0), v) ∈ U for some arbitrary t = (q1, k1) . . . (qk, k0) (with q1 = w) and value v, then (τ v(v)) ∈ T(E, e′1, e2, e3);

- **Dead** Suppose that (U, U′) ∈ T(unit, e1, e2, e′1, e′2). Then w reminds(e1) = ø and e2 ⊑ e3. Then (U, τ r m()) ∈ T(unit, e1, e2, e′1, e′2).

- **Parallelization** If (U1, U′1) ∈ T(E1, e1, e′1, e′2, e2 ∪ e3) and (U2, U′2) ∈ T(E2, e2, e′2, e′2 ∪ e′3) and e1 ⊑ e2 and e1 ⊑ e3 and e2 ⊑ e3, then
  
  \[ (U1, U′2, (t′1, v′1, v′2)) \mid (t′1, v′1) \in U′1, (t′2, v′2) \in U′2 \] \in T(E1 × E2, e′1 ∪ e′2, e, e′1 ∪ e′2)

**Proof.** (Sketch) Parallelization.

Assume w.l.o.g. that the pilot trace takes the form (t1, (v1, v2)) where inter((t1, t2, t3) and (t1, v3) ∈ U1. Just as in the commuting case we set up two side games U1 vs. U′1 on t1, v1. Unlike, in 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 τ being played on. 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 ε ∪ e′1 ∪ e′2 location by location using the definition of tiling.

**Theorem 9.2.** Suppose that Γ ⊢ v ≤ v′ : τ and Γ ⊢ e ≤ e′ : τ & e1 | e2 | e3 and assume that for each axiom (x, y) it holds that (x, y) ∈ Ω[v]. Then, (η, η′) ∈ Γ ⊢ Ω[v] interpreting a context as a cartesian product implies ([v], η) (∃x, η′) ∈ Γ[v] and ((∃x, η)[v], hη'[v]) ∈ T(Γ[v], e1, e2, e3)∗.

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

Define

\[ Γ ⊢ τ [f, f] = ((f, f′) \triangleright η, η′) ∈ Γ \] (f, f′) \triangleright η, η′ ∈ Γ \]

\[ Γ ⊢ τ ∪ r(r1, e1, e2, e3) = ((f, f′) \triangleright η, η′) \triangleright η ∈ Γ \]

\[ ((f, f′) \triangleright η, η′) ∈ T(Γ, [r(r1, e1, e2, e3)]) \]

\[ Γ ⊢ τ ∪ τ(r1, e1, e2, e3) = ((f, f′) \triangleright η, η′) ∈ Γ \]

\[ ((f, f′) \triangleright η, η′) ∈ T(Γ, r(r1, e1, e2, e3)) \]

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 = let x = e1 in x : e′ and e′ = let x = e′ in x. Inductively, we know ([e1], [e′1]) ∈ Γ ⊢ τ ⊃ (x : e1, e2, e3) ∗[[v]] and ([x], [e′1]) ∈ Γ, x : τ ⊃ τ ∪ r(r1, e1, e2, e3) ∗[[v]] for some n1, n2 > 0. By Theorem 7.9, we also have ([e1], [e′1]) ∈ Γ ⊢ τ ⊃ (x : e1, e2, e3) ∗[[v]] and analogous statements for e′1, e2, e′2. We
can, therefore, assume, w.l.o.g. that \( n_1 = n_2 \) and then use Theorem 7.8 (6) repeatedly \( (n_1 \text{ times}) \) so as to conclude \( (\langle e_1, \langle f \rangle \rangle) \in \Gamma \Rightarrow (e_1 \& (f_1, e_2, f_2))^\mathbf{ff} \).

The rules for dead code and pure lambda hoist rely on the cases “Dead” and “Pure” of Theorem 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 \( (\langle v \rangle, \langle v' \rangle) \in \Gamma \) where \((-)^v\) denotes transitive closure. Then \( \vdash v \leqsub v' : \tau \).

**Proof.** If \( \vdash f : \tau \rightarrow t \rightarrow \int \) then by Thm. 7.9 we have \( (\langle f \rangle, \langle f \rangle) \in \Gamma \Rightarrow (f_1 \& (e_1, e_2, f_2))^\mathbf{ff} \) and \( e_1 \& (f_1, e_2, f_2) \).

Let \( (h_{\text{init}}, k, v) \in \Gamma \). We have \( h_{\text{init}} \Rightarrow w \) and thus in particular \( h_{\text{init}} \Rightarrow (h_{\text{init}}) \). There must therefore exist a matching heap \( k' \) and a value \( v' \) such that \( (\langle h_{\text{init}}, k', v' \rangle) \in \Gamma \) 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 \( \epsilon \) does not read nor write \( X \):

\[
\text{readFst: unit} \rightarrow \int \text{ and writeFst: int} \rightarrow \text{unit}
\]

The obvious and 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 \( \text{writeSnd} \). We show how the game is played against itself using the typing shown above. We start with a "pilot trace", say: \((123, [23]), ([12][7], [17][1]), ()\) where \([x_1y_1]z\) 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 \( \text{writeFst}(17) \):

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

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 \( \text{writeFst}(17) \). 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 \( \text{readFst}, \text{writeFst} \), and another program, \( e_2 \), that only calls \( \text{readSnd}, \text{writeSnd} \). Since the former functions have disjoint effects to the latter ones, \( e_1 \) and \( e_2 \) will have effect specifications, respectively, of the form \( (e_1, e_2) \Rightarrow (e_2, e_1) \) and \( (e_1, e_2) \Rightarrow (e_2, e_1) \), where \( e_1 \cap e_2 = \emptyset \) and \( e_1 \cap e_2 = \emptyset \). 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 \( \text{enqueue} \) and \( \text{dequeue} \) functions described in Section 1 for the Michael-Scott Queue have the same behavior as their atomic versions. We only show the case for \( \text{dequeue} \), as the case for \( \text{enqueue} \) is similar. More precisely, we now justify the axiom

\[
(\text{dequeue}, \text{atomic}(\text{dequeue}), \text{unit}) \Rightarrow (\text{MSQ}, \text{MSQ}_1, \text{MSQ}_2, \text{int})
\]

where \( \text{MSQ} = (\text{rd}_\text{max} + \text{wr}_\text{max}) \). 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 mumbing. After consuming a dummy argument \( () \) the resulting pilot trace be \( (h_1, k_1) \ldots (h_n, k_n) \ldots (h_n, k_n, a) \) and \( h_1 \) and \( h_1' \) be the start heap to match. We can now assume that the messages from \( k_1 \) to \( h_n \) follow the protocol, i.e. \( k_1 \rightarrow_\text{msq} h_n \). (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 \( (h_n, k_n) \) the element \( a \) is dequeued and \( h_1 = k_1 \) holds for \( j \neq i \). We can thus match this trace by a trace in the semantics of \( \text{atomic}(\text{dequeue}) () \) by stuttering until \( i : (h'_i, h'_i) \ldots (h'_i, \ldots \).

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 \( \text{op}_1 || \cdots \| \text{op}_n \) and \( \text{atomic}(\text{op}_1) || \cdots || \text{atomic}(\text{op}_n) \) for \( \text{op}_i \) 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 [9, 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 [17], 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 over-
lap in concrete storage whilst appearing independent to clients also appears in work on ‘fictional’ separation [18, 22].

Most previous work aims at proving particular, concrete equivalences and refinements. Sophisticated logics like CaReSL [31] can verify more complex fine-grained algorithms than our system. However, such logics do not directly capture the simpler, more general patterns of behaviour expressed by effect-refined types, or the soundness of the associated generic transformation rules.

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, the invariants are based on simply-typed concrete locations and thus do not capture effects at the level of whole datastructures, as abstract locations do. The examples in [12] are consequently more elementary 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 a denotational model gives a rather simpler and more extensional definition of the logical relation by comparison with [12]. While some of the complexity is certainly attributable to dynamic allocation and higher-order store, 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 for higher-order concurrent programming.

Brookes’s trace model is also used in, for example, Turon and Wand’s work on refinement [32], and we certainly found it a usefully simpler base than transition systems or resumptions. Brookes [16] extends his original semantics to model a parallel Algol-like language. Explicit powerdomains are not required for that language, but the semantics incorporates both a possible-worlds treatment of local variables and potentially infinite traces for modelling liveness as well as safety.

There are various directions for further work. 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, we believe, allow us to model and reason about lock-based protocols in an elegant way. It would also be natural to integrate this work with effects that track non-determinism [10]. Other possible extensions include higher-order store and weak concurrency models. It might be possible to factor the semantics of an effect system into an abstract layer treating single locations, like [12], with a separate refinement, like [31], to concrete implementations 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.

References

A. Proof of Theorem 7.8

We write Adm(P) for the admissible closure of the predicate P.

Lemma A.1. Let A, B be predomains and P ⊆ A and Q ⊆ B. We have Adm(P × Q) = Adm(P) × Adm(Q).

Proof. The direction ⊆ is obvious. For the other one, fix b ∈ Adm(Q) and define S = {a | (a, b) ∈ Adm(P × Q)}. Since S is admissible and contains P, we have Adm(P) ⊆ S and the claim follows.

Lemma A.2. Let A, B be predomains and let (p_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 ∈ Q implies p_i(b) ∈ Q for all i. Then P → Adm(Q) = Adm(P → Q).

Proof. The second equality and the ≤-direction of the first equation are obvious. So, for “⊇” assume f : P → Adm(Q). Thus, for each a ∈ P, we can choose a chain (b_i(a)), so that sup_i b_i(a) = f(a) and b_i(a) ∈ Q for each j. Now, for each i we have r_i(b_i(a)) = sup_i r_i(b_i(a)) and, since r_i(f(a)) is compact, we even have r_i(f(a)) = r_i(b_i(a)) for some j. Note that r_i(b_i(a)) ≥ r_i(b_i(a)) always holds by monotonicity. Thus, since b_j(a) ∈ Q and r_i : Q → Q we obtain r_i(f(a)) ∈ Q for all i. As a result, for each i we have r_i f : P → Q and thus, since f = sup_i r_i f, we finally get f ∈ Adm(P → Q) as required.

Proof of Theorem 7.8 In each case, using Lemmas A.1 and A.2 (for case 6), we can in fact assume w.l.o.g. that the assumed pairs are in T_1(...), rather than T(...).

Ad 1. Let (t, a) ∈ q_i(U_i). i.e. a = p_i(a_0) where (t, a_0) ∈ U. By down-closure (↓Down) we also have (t, a) ∈ U. We can now play the strategy guaranteed by the assumption (U, U′) ∈ T(Ei, εi, E3) which will yield (depending on the opponent’s moves) a trace t′ and a value a′ such that (t′, a′) ∈ U′ and (p_i(a′), a′) ∈ E. Now, since E is a specification we get (p_i(a), p_i(a′) ∈ 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′) ∈ T_j(Ei, ε_i, E3, ε3). Since (T(E_i, E3, ε3) is closed under suprema it suffices to show that (q_j(U′), q_j(U″)) ∈ T(E_i, E3, ε3) for each j. Fix such j and pick (t, p_i(a)) ∈ q_j(U″), thus (t, a) ∈ U″. By induction on the closure process we can assume w.l.o.g. that (t, a) arises from (t, a) ∈ U by a single mumbling or stuttering step or that (t, a) ∈ U for some a ≥ a or else that (t, a) ∈ U where sup a = 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) ∈ U″ hence (t, p_i(a)) ∈ q_j(U″) as required.

If (t, a) arises by stuttering, so that u = ω(h, h′, h′′) 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 [ε_i][h, h′, h′′] is always true (Lemma 6.2) so this is a legal move. We continue as just as in the original strategy. In the special case where ν is empty we must also show that [ε_i][h, h′, h′′] knowing [ε_i][h, h′, k, K] where u = (h_1, h_2) ... (h_n, k) and u′ = (h′_1, k′) ... (h′_n, k′). The matching trace. We have [ε_i][h, h′, k, K] for opponent’s playing h′ would have been illegal. Since, by assumption ε_2 ⊆ ε_3, we can conclude [ε_i][h, h′, h′′] and then [ε_i][h, h′, h′′] by Lemma 6.2(3&1).

If (t, a) arises by mumbling then we must have v = u(h_1, h_2)v and t_1 = (h_1, h_2)(h_3, h_4)v. We play until the strategy has produced a match h′_3 for h_3. So far, the player has produced a trace u′ matching u, and a state h′_3 so that [ε_i][h_1, h′_2, h_2, h_3]. Now, we can ask what the original strategy would produce if we gave it (temporarily assuming opponent’s role) the state h_3′ as a match for h_2. Note that this is legal because [ε_2][h_2, h′_3, h_3, h_4]. The strategy will then produce h′_3′ such that [ε_i][h_2, h′_2, h_3, h′_3′] and our answer in the play on the new trace against the challenge h′_3 will be this very h′_3′. Indeed, by composing tiles (Lemma 6.2) we have [ε_i][h, h′_3, h_3, h_4] as required. Thereafter, the play continues according to the original strategy.

For down-closure, we play the strategy against (t, a) yielding a match (t′, a′) ∈ U′ where a ∈ Adm. That same strategy also wins against (t, a) because aEa′ since E is a value specification.

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

Ad 4. Suppose aEa′. By 3 which we have just proved we only need to match elements of the form ((h, h′), h). The opponent plays h′ where h ~ 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 ((h, h_i), a) where c(h) = (h_i, a). In this case, suppose that the opponent plays h′ where h ~ h′. The assumption gives (h′_i, a′) such that c′(h′) = (h′_i, a′) and [ε_i][h, h′, h′_i, h′_i] and aEa′. We thus play h′_i and a′ and indeed [ε_i][h, h′, h′_i, h′_i] and aEa′ hold so this is a winning move.

Ad 6. Suppose (f, f′) ∈ E → T(E_2, E_2, E_3) and (U, U′) ∈ (E_1, E_2, E_3). Suppose that (u, b) ∈ apf(U) where (u, a) ∈ U and (v, b) in f(a) (note that we can ignore the ↑-closure). We need to produce a trace (u′, b′) ∈ apf′(U′) such that (u′, b′) ∈ U′ and (v′, b′) in f′(a′) and E_2b′. Assume that:

u = (h_1, k_1) ... (h_n, k_n) and v = (h_{n+1}, k_{n+1}) ... (h_{m+1}, k_{m+1})

We are given a heap h′, such that h ~ h′ and h′ ∈ C. We can use the strategy S_1 from (U, U′) ∈ T(E_1, E_2, E_3) for (u, a). We play according to S_1 to work off the α-part. This results in a matching trace u′ ∈ U′:

u′ = (h′_1, k′_1) ... (h′_m, k′_m)

where [ε_i][h_1, h′_1, k_1, K] and (a, a′) ∈ E_2. We get (f(a), f′(a′)) ∈ T(E_2, E_2, E_3). Now, we are given a heap h′_1′ which is an environment move forming the tile ([ε_i][h_1, k_1, h′_1, h′_1]). From the fact that E_2 ⊆ E_3 and Lemma 6.2(5) we can conclude h_{n+1} ~ h_{n+1}.

Thus we can continue our play by using the strategy S_2 from (f(a), f′(a′)) ∈ T(E_2, E_2, E_3) which yields a continuation v′ of our trace and a final answer b′. It is then clear that (u′, b′) ∈ bm(d′(f′, U′)) so this combination of strategies does indeed win.

Ad 7. Suppose that (U_1, U′_1) ∈ T(E_1, E_1, E_1, E_1, E_1, U_1) and (U_2, U′_2) ∈ T(E_2, E_2, E_2, E_1, E_1, U_2) and let (t, a, b) ∈ U_1 ∪ U_2, thus inter(t_1, t_2, t) (ignoring ↑ by item 3) where (t_1, a) ∈ U_1 and (t_2, b) ∈ 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:

t = (h_1, k_1) ... (h_{n+1}, k_{n+1}) (h_{n+2}, k_{n+2}) ... (h_{m+1}, k_{m+1})

composed of pieces of the traces t_1 and t_2. Assume w.l.o.g. that the first piece (h_1, k_1) ... (h_{n+1}, k_{n+1}) is a part of t_1. We are given an initial heap h such that h ~ h′ and h′ ∈ C. Since rh_{ni} ⊆ rh_{ni}, we can apply strategy S_1 to guide us through the first part of the game, obtaining:

(h′_1, k′_1) ... (h′_m, k′_m)
Moreover, we have an environment move which forms the tile $[e][h,v,k,u_1,u_2]$. Thus, we have the tile $[e][h,v,k,u_1,u_2]$ which can be seen as an environment move for $t$. Therefore, we can use strategy $S$ for the $U'$ and continue the game, obtaining the trace piece:

$$(h'_1,k'_1)\cdots(h'_n,k'_n)$$

Now, we can return to the $S_1$ game as the trace above is seen as an environment move for $U$. Alternatively, these strategies, we get a trace $t$ which is in $(U \cup U')$. Let $(a',b')$ be the final values reached at the end. It is clear that $[e \cup e' \cup e_1 \cup e_2](h',h',h',h')$ and also $a \equiv a' \equiv b \equiv b'$.

It remains to assert the stronger statement $[e \cup e' \cup (e_1 \cup e_2)](h',h',h',h')$. To see this suppose that $w_1 \in e_1 \setminus e_2 \setminus e' \setminus e_2$. Since the entire game can be viewed as an instance of the game $U_1 \cup U'$ with interventions by $U_2$ vs. $U'_2$ regarded as environment interactions we have $[e \cup e_2 \cup e'](h',h',h',h')$ so in fact $h \equiv h_1$ and $h_1 \equiv h_2'$. The case of $c_1$ and $e_1 = e_2$ interchanged is analogous.

Ad 8. This is direct from the definition of atomic and appealing on the fact that $(U,U') \in T(E,e_1,0,e_2)$.

### B. Proof of Theorem 9.1

**Proof.** Commuting. By Theorem 7.8(3) we can assume our pilot trace $t$ to be of the form:

$$(h_1,k_1)(h_2,k_2)\cdots(h_n,k_n) = (h_{u_1},k_{u_1})(h_{u_2},k_{u_2})\cdots(h_{u_m},k_{u_m})$$

where

- $t_1 = (h_1,k_1)(h_2,k_2)\cdots(h_n,k_n) = v_1 \in U_1$
- $t_2 = (h_{u_1},k_{u_1})(h_{u_2},k_{u_2})\cdots(h_{u_m},k_{u_m}) = v_2 \in U_2$

We make similar use of Theorem 7.8(3) in the subsequent cases without explicit mention.

We are also given a heap $h'_1$ such that $h'_1 \equiv h_2$. Because $e'_1 \equiv e'_2$, $h_1$ and $h_{u_1}$ agree on the reads of $e'_1$. 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

$$(h'_1,k'_1)(h'_2,k'_2)\cdots(h'_m,k'_m) = v_2$$

where $t_2 = (h'_1,k'_1)(h'_2,k'_2)\cdots(h'_m,k'_m)$, and $v_2$. 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 $[e_1 \cup e_2 \cup e'_1 \cup e'_2](h_1,k_1,k_2,k_3,k_4)$. This follows from the fact that $e'_1 \equiv e_1 \equiv e_2$ and $(1)$ and $(2)$. In particular, from $(1)$ and $e_1 \equiv e_2$, we get that $k_{u_1}$ and $k_{u_2}$ agree on the locations in $e'_1$, while from $(2)$ we get that $k_{u_m}$ and $k_{u_m}$ agree on the locations in $e'_2$. This finishes the proof.

**Duplicated.** Assume given a trace $t$ in $U$:

$$t = (h_1,k_1)\cdots(h_n,k_n) \nu$$

and a heap $h'_1$ such that $h_1 \equiv h'_1 \equiv h'_2$. Recall that $\text{dstr}(e') \cap \text{w 其s}(e') = \emptyset$ and moreover, since $e'_1 \cup e'_2 \equiv e_1 \cup e_2$ is well formed, we also have $\text{dstr}(e') \cap (\text{w 其s}(e') \cup \text{ch 其s}(e'_2)) = \emptyset$. Thus $h_1$ and $k_1$ agree on the reads of $e'_1 \cup e'_2$, i.e., the reads of $e'$.

We start by simply stuttering:

$$t' = (h'_1,h'_1)(h'_2,h'_2)\cdots(h'_n,?,?)$$

leaving the final heap $? \equiv ?$ 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 $[e'_1](h_1,k_1,k_1,k_2)$ and a chaotic effect on a location allows any changes to that location. Moreover, we may assume $[e_1](h_{u_1},k_{u_1},k_{u_2})$ for otherwise we would have won immediately. As a result, since $e_1 \equiv e_2$, we inductively get $h \equiv h_1 \equiv h_2$ and, of course, $h_1 \equiv h_2$.

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 $h_1$, where the environment moves are simply stutter moves. Recall that $h_1 \equiv h_2$.

$h_1$ has already been asserted above. Thus we obtain the following trace $t_1 \in U'$:

$$t_1 = (h'_1,q_1)(q_1,q_2)\cdots(q_{n-1},q_n)$$

where $v_1$ and $[e'_1 \cup e'_2](h_1,k_1,k_1,k_2)$. Notice that using stuttering environment moves is valid as $[e'_1 \cup e'_2](h_1,k_1,q_1)$ for $1 \leq i \leq n - 1$.

Since $h_1$ and $k_1$ agree on the reads of $e'$ and $q_1$ and $k_1$ agree on $rds(e')$ from $[e'_1 \cup e'_2](h_1,k_1,k_1,k_2)$, we can run the game $U$ vs. $U'$ again on $q_1$ and $t$ with stutter environment moves:

$$(q_1,q_1)(q_1,q_2)\cdots(q_{n-1},q_n)$$

where $v_2$ and $[e'_1 \cup e'_2](h_1,k_1,k_1,k_1)$. Thus, $(v,v)(E \times E)(v'_1,v'_2)$. This trace is again valid for the same reasons above, namely $e'_2$ allows any internal moves, and since $e_1 \equiv e_2$, the environment moves are also legal.

We now put $?? \equiv q_{n+1}$ which leads to a valid trace due to repeated mumbled. Finally, we shall show that $[e'_1 \cup e'_2](h_1,k_1,k_1,k_{n+1})$ that is $k_1$ and $q_{n+1}$ agree on the reads of $e_2$ and of $e'$.

- They agree on the reads of $e'$ because $[e'_2 \cup e'](h_1,q_1,k_1,k_{n+1})$ obtained from the game above;
- They agree on the reads of $e_1$ because $e_1 \equiv e_2$. The internal moves did not affect the locations read by $e_2$.

**Duplicated for result value unit:** We can show that equality holds and not just $\leq$ when the result type is $\text{unit}$. 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 $\text{unit}$, 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 $\text{rtr}(t)$, for example $(h_1,h_1)\nu$ and an arbitrary heap $h_1$. We now consider the game involving $U$ vs. $U'$ on $t$, $v$ and $h_1$.

$$t' = (q_1,k_1)(k_1,k_2)\cdots(q_{n-1},k_n)\nu$$

and $h_1$ such that $h_1 \equiv h'_1$. We now initiate a side game $U$ vs. $U'$ on this trace and respond in the main game by stuttering. Thus, we obtain traces $(h_1,h_1)\cdots(h_1,h_1)$ in the main game and $(h'_1,h'_1)\cdots(h'_1,h'_1)$ in the side game.

The main trace is in $\text{rtr}(t)$. The side game tells us that $v = ()$ and that $h_1 \equiv h_2$ and therefore $[e'_1 \cup e'_2](h_1,h_1,h_1)$. It remains to show that $[e_1 \cup e'_2 \cup e'_2](h_1,h_1,k_1,k_2)$. This follows from the fact that $e_1$ has only reads as $h_1$ and $k_1$ agree on all locations.
**Parallelization.** We start with a trace in $U_1||U_2$. Assume that the trace is of the following form:

$$t_1, t_2, t_3, t_4, \ldots, t_n$$

where each $t_i$ is a possibly empty sequence of moves of the form $(h_i^{1,j}, k_i^{1,j}) \cdots (h_i^{m,j}, k_i^{m,j})$ 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 $h_i^t$ such that $h_i^{1,1} \sim h_i^t$. We also have $h_i^{1,1} \sim \epsilon h_i^t$. We run a side game $U_1$ vs. $U_2$ using $h_i^t$ and $t_1$, yielding:

$$t_1', t_2'$$

Assume that $(h_i^t, k_i^t)$ and $(h_i^t, k_i^t)$ are, respectively, the first and last moves of this trace. We have $v_1 E_1 v_1'$ and (1) $[e^f \cup e^2 \cup e_i^t](h_i^{1,1}, k_i^{1,1}, h_i^{1,1}, k_i^{1,1})$. Notice that these are legal moves in the global game as we have $[e^f \cup e^2 \cup e_i^t]$ tiles for the player moves and $[e]$ times for the environment moves.

Now, assume there is an environment move $(k_h, h_{io+1})$. Since $e_1 \perp e_2$ and $e \perp e_i$, the heaps $h_i^{1,1}$ and $h_i^{1,1}$ agree on the ends of $e_i^2$ and $h_i^t$ and $h_i^{1,1}$ also agree on the reads of $e_i^2$. (Notice as well that $wrs(e_1), wrs(e_1) = 0$ as $e^f \cup e^2 \cup e_i^t$ is a valid effect.) Therefore, we can invoke an $U_2$ game using $h_i^{1,1}$ and $t_2$, obtaining the trace:

$$t_2', t_3'$$

Assume that $(h_i^{1,1}, k_i^{1,1})$ and $(h_i^{1,1}, k_i^{1,1})$ are, respectively, the first and last moves of this trace. We have $v_2 E_2 v_2'$ and (2) $[e^f \cup e^2 \cup e_i^t](h_i^{1,1}, h_i^{1,1}, k_i^{1,1}, k_i^{1,1})$. 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 $[e^f \cup e^2 \cup e_i^t](h_i^{1,1}, h_i^t, k_i^{1,1}, k_i^{1,1}, k_i^{1,1}, k_i^{1,1})$. From (1) and $e_1 \perp e_2$ and $e \perp e_i$, we have that $k_i^{1,1}$ and $k_i^{1,1}$ agree on the locations of $e_1$. Similarly, $k_i^{1,1}$ and $k_i^{1,1}$ agree on the locations of $e_2$. Since there are only $e$ tiles and $e \perp e_1$ and $e \perp e_2$, $k_i^{1,1}$ and $k_i^{1,1}$ agree on the locations of $e$. This finishes the proof.

$\square$