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.

CCS 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 $\Gamma \vdash e : \tau\&\varepsilon$ 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 be-

PPDP '16, September 05 - 07, 2016, Edinburgh, United Kingdom

ISBN 978-1-4503-4148-6/16/09…$15.00

DOI: http://dx.doi.org/10.1145/2967973.2968602

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 a 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, which may allow "cleaning-up" some of the effects ocurring during computation. Annotated function types $\tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2$ also capture the effect during a call, ε_1, the environmental interference, ε_2, and the final effect, ε_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 the equivalence class of an abstract location may change. Typing and refinement judgements may be established by generic type-based rules or semantic reasoning in the model.

We show the soundness of a number of generic equivalences, 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 $(X := !X + 1; X := !X + 1) = (X := !X + 2)$ *at* the effect type $\texttt{unit}\ \&\ \{ch_X\} \mid \varepsilon \mid \varepsilon \cup \{rd_X, wr_X\}$, *provided* that the effect, ε, of the concurrent environment does not involve X. This says that the two commands are equivalent with return type `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 ε plus reading and writing X.

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

```
readFst () = p(!X).1
readSnd () = p(!X).2
wrtFst n = (rec try _ = let m =!X in
               if cas(X, m, p⁻¹(n, p(m).2))
               then () else try () )()
wrtSnd n = (rec try _ = let m =!X in
               if cas(X, m, p⁻¹(p(m).1, n))
               then () else try () )()
```

which multiplex two abstract integer references onto a single concrete one. Note that the write functions, wrtFst and wrtSnd, use compare-and-swap, `cas`, to atomically 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 wrtFst followed by one to wrtSnd, even though they read and write the same concrete location, which looks like a race.

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

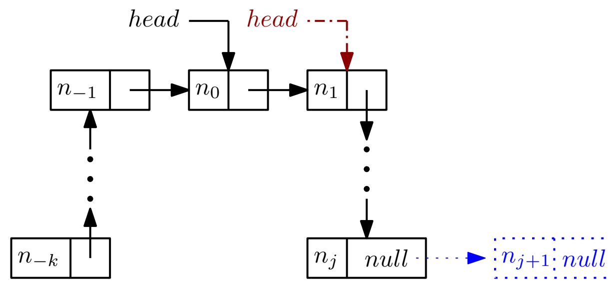

Figure 1: Illustration of a Michael-Scott Queue. The list resulting from the pointer to the element n_0 (the *head* pointer with the continuous arrow in black) contains the list of elements $[n_1, \ldots, n_j]$. The enqueueing operation is illustrated by the dotted arrow and the box with the element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

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

```
transact f = let rec try() =
let (val, ver) = atomic((!X_val, !X_ver))
 in let res = f(val) in if atomic(if !X_ver = ver then
   X_ver := ver + 1;  X_val := res; true else false)
 then () else try()
in try()
```

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

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

at type $\texttt{unit}\&\{rd_{\mathfrak{X}}, wr_{\mathfrak{X}}\} \mid \varepsilon \mid \varepsilon \cup \{rd_{\mathfrak{X}}, wr_{\mathfrak{X}}\}$ for any ε not including chaotic access, $ch_{\mathfrak{X}}$, to $\mathfrak{X}$. The environment effect ε here *may* include reading and writing $\mathfrak{X}$, 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.

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

```
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 ()) ()

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

enqueue(x) =   (rec try (p) =
                 if !p.next = null then
                   if atomic(if !p.next = null then
                   !p.next := ref(x, null); true else false)
                   then () else try (!p.next)
                 else try (!p.next)) !head

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 $\xrightarrow[\varepsilon_2]{\emptyset \mid \varepsilon_2, rd_{MSQ}}$ bool provided the environment does not access the MSQ chaotically, *i.e.*, $ch_{MSQ} \notin \varepsilon_2$. This typing denotes that mem has the effect of reading the MSQ, both during execution and as overall effect. With more assumptions on the environment effects ε_2, namely, that it does not enqueue nor dequeue MSQ, mem may participate in many of the equations we prove sound, *e.g.*, commuting, deadcode.

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

2. SYNTAX

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 $X_1, \ldots, X_n, \ldots$ and a set $\mathbb{VB}$ 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 $\mathsf{h} \in \mathbb{H}$, then, is a *finite map* from $\mathbb{L}$ to $\mathbb{VB}$, written $\{(X_1, \mathsf{c}_1), (X_2, \mathsf{c}_2), \ldots, (X_n, \mathsf{c}_n)\}$, specifying that the value stored in location X_i is c_i. We write dom(h) for the domain of h and write $\mathsf{h}[X \mapsto \mathsf{c}]$ for the heap that agrees with h except that it maps X to c. We also assume that $new(\mathsf{h}, v)$ yields a pair (X, h') where $X \in \mathbb{L}$ is a fresh location and $\mathsf{h}' \in \mathbb{H}$ is $\mathsf{h}[X \mapsto v]$.

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

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

Here, x ranges over variables, v_r over R-values, and c over built-in functions, 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 $F_+(n, n') = n + n'$ for integers n, n'.

The construct $\mathtt{rec}\ f\ x = e$ defines a recursive function with body e and recursive calls made via f; we use $\lambda x.e$ as syntactic sugar in the case when f is not free in e. Next, $!v$ (reading) returns the contents of location v, $v_1 := v_2$ (writing) updates location v_1 with value v_2, and $\mathtt{ref}(v)$ (allocating) returns a fresh location initialized with v. The metatheory is simplified by using "let-normal form", where the only elimination for computations is let, though we nest computations as a shorthand 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. The command $\mathtt{atomic}(e)$ evaluates e in one step, without any environmental interference. One can then define a (more realistic) compare-and-swap operation $\mathtt{cas}(X, v_1, v_2)$ as

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

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* $(\mathsf{h}_1,\mathsf{k}_1)(\mathsf{h}_2,\mathsf{k}_2)\cdots(\mathsf{h}_n,\mathsf{k}_n)$ *where for* $1\le j\le i\le n$*, we have* $\mathsf{h}_i,\mathsf{k}_i\in\mathbb{H}$ *and* $\mathrm{dom}(\mathsf{h}_j)\subseteq\mathrm{dom}(\mathsf{h}_i), \mathrm{dom}(\mathsf{h}_j)\subseteq\mathrm{dom}(\mathsf{k}_i), \mathrm{dom}(\mathsf{k}_j)\subseteq\mathrm{dom}(\mathsf{h}_i), \mathrm{dom}(\mathsf{k}_j)\subseteq\mathrm{dom}(\mathsf{k}_i)$. *We write* Tr *for the set of traces.*

A trace of the form $u\,(\mathsf{h},\mathsf{h})\,v$ where $t=uv$ is said to arise from t by stuttering. A trace of the form $u(\mathsf{h},\mathsf{k})v$ where $t=u(\mathsf{h},\mathsf{q})(\mathsf{q},\mathsf{k})v$ is said to arise from t by mumbling. If $t=(\mathsf{h}_1,\mathsf{k}_1)(\mathsf{h}_2,\mathsf{k}_2)(\mathsf{h}_3,\mathsf{k}_3)$, say, then $(\mathsf{h}_1,\mathsf{k}_1)(\mathsf{h},\mathsf{h})(\mathsf{h}_2,\mathsf{k}_2)(\mathsf{h}_3,\mathsf{k}_3)$ arises from t by stuttering. If $\mathsf{k}_1=\mathsf{h}_2$, then the trace $(\mathsf{h}_1,\mathsf{k}_2)(\mathsf{h}_3,\mathsf{k}_3)$ arises from t by mumbling. A set of traces U is closed under stuttering and mumbling if whenever t' arises from $t\in U$ 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 (ω-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:*

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

The elements of TA *are partially ordered by inclusion.*

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

For example, the semantics of $X :=!X+1; X :=!X+1; !X$ 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 [S&M] is taken from Brookes. It ensures that the semantics does not distinguish between late and early choice [31] and related phenomena which are reflected, e.g., in resumption semantics [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.

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

Both e_1 and e_2 admit the same traces, namely $(([x],[0]),\texttt{true})$ and $(([x],[0]),\texttt{false})$ and stuttering variants thereof. In 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 nondeterministic outcomes that are less defined than existing ones are not recorded in the semantics.

Definition 3.3. *If* $U\subseteq Tr\times A$ *then* $U^\dagger$ *is the least subset of* TA *containing* U*, i.e.* $U^\dagger$ *is the closure of* U *under [S&M], [Down], [Sup].*

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

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

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

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

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

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

where $\mathsf{h}=\mathsf{h}_1$ and $\mathsf{h}_n=\mathsf{k}$. Such an element models an atomic execution of the computation represented by U.

3.1 Semantic values

The predomain $\mathbb{V}$ of values is the least solution of

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

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

There are (canonical) families of deflations $p_i:\mathbb{V}\multimap\mathbb{V}$ and $q_i:T\mathbb{V}\to T\mathbb{V}$, such that that $(p_i)_i$ and $(q_i)_i$ are ascending chains converging to the identity. A consequence is that $\mathbb{V}$ and $T\mathbb{V}$ 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 $[\![v]\!] \in \mathbb{V} \to \mathbb{V}$ and terms $[\![t]\!] \in \mathbb{V} \to T\mathbb{V}$ are given by the recursive clauses in Figure 3. Environments, ρ, are properly tuples of values; we abuse notation slightly by treating them as maps from variables, x, to values, v, (and write $\rho[x \mapsto v]$ for functional update) to avoid mentioning an explicit context in which untyped terms are well-formed.

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 a regarded as a single "location" that can be written to and read from. Essentially, an abstract location is given by a partial equivalence relation on heaps modelling well-formedness and equality, together with a transitive relation modelling allowed modifications of the abstract location. Abstract locations then allow certain commands that modify the physical heap to be treated as read-only or even pure if they respect the contracts. Abstract locations are related to *islands* [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 $\mathfrak{l}$ *comprises:*

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

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

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

These data must satisfy the following conditions where $\mathsf{h} : \mathfrak{l}$ *stands for* $\mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}$.

1. *If* $\mathsf{h} : \mathfrak{l}$ *then* $\mathsf{h} \overset{\mathfrak{l}}{=} \mathsf{h}$;

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

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

We now 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:

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

Two heaps are semantically equivalent w.r.t. $\mathfrak{int}(X)$ if the values stored in X are equal integers; the step relation requires all other concrete locations to be unchanged. We may write rd_X, wr_X, ch_X for $rd_{\mathfrak{int}(X)}, wr_{\mathfrak{int}(X)}, ch_{\mathfrak{int}(X)}$.

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

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

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

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

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

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

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

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

$$
\begin{array}{rcl}
(\!|x|\!)\rho & = & \rho(x) \\
(\!|v_r|\!)\rho & = & v_r \\
(\!|(v_1, v_2)|\!)\rho & = & ((\!|v_1|\!)\rho, (\!|v_2|\!)\rho) \\
(\!|v.i|\!)\rho & = & d_i \text{ if } (\!|v|\!)\rho = (d_1, d_2) \\
(\!|c|\!)\rho & = & \mathit{fun}(f) \\
\multicolumn{3}{l}{\quad \text{where } f(v) = \mathit{rtn}(F_c(v)) \text{ if } F_c(v) \text{ is defined}} \\
\multicolumn{3}{l}{\quad \text{and } f(v) = \emptyset, \text{ otherwise.}} \\
(\!|\mathtt{rec}\ f\ x = e|\!)\rho & = & \mathit{fun}(g^{\ddagger}(\rho)) \\
\multicolumn{3}{l}{\quad \text{where } g(\rho, u) = \lambda d.(\!|e|\!)\rho[f \mapsto u, x \mapsto d]} \\
(\!|v|\!)\rho & = & 0, \text{ otherwise}
\end{array}
$$

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

Figure 3: Denotational semantics

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

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

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

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

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

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

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

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

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

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

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

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 $\mathsf{h} \models \mathsf{w}$. 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 $\mathit{rd}_{\mathfrak{l}}$ (reading from $\mathfrak{l}$), $\mathit{wr}_{\mathfrak{l}}$ (writing to $\mathfrak{l}$), and $\mathit{ch}_{\mathfrak{l}}$ (chaotic access), for each abstract location $\mathfrak{l}$. An *effect*, ranged over by ε, 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 wr_X effect, but e_1 and e_2 are distinguishable by contexts that assume the wr_X-effect. Thus, e_1 and e_2 are not equal "at type" wr_X. At type ch_X they are, however, equal, because a context that copes with this effect may not assume that both produce equal results.

So $\mathit{ch}_{\mathfrak{l}}$ 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 $X := !X + 1; X := !X + 1$ is equivalent to $X := !X + 2$ "at type" $\mathtt{unit}\ \&\ \mathit{ch}_X \mid \varepsilon \mid \varepsilon \cup \{\mathit{rd}_X, \mathit{wr}_X\}$, where ε is any effect such that $X \notin \mathrm{locs}(\varepsilon)$. This means that the two computations are indistinguishable by environments that do not read, let alone modify X during the computation and assume regular read-write access once it is completed. The 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 ch_X. The ch effects are akin to the private regions from [11], but seem more permissive.

We use the notation $\mathrm{rds}(\varepsilon)$, $\mathrm{wrs}(\varepsilon)$, $\mathrm{chs}(\varepsilon)$ to refer to the abstract locations $\mathfrak{l}$ for which ε contains $\mathit{rd}_{\mathfrak{l}}$, $\mathit{wr}_{\mathfrak{l}}$, and $\mathit{ch}_{\mathfrak{l}}$, respectively. We write $\mathrm{locs}(\varepsilon) := \mathrm{rds}(\varepsilon) \cup \mathrm{wrs}(\varepsilon) \cup \mathrm{chs}(\varepsilon)$.

Our semantics of effects follows the relational style [8, 11]. Intuitively, two computations are related at 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 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 ch_X, then arbitrary modifications of X are allowed.

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

A specification $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ approximates the behavior of a computation e as follows: ε_1 summarizes side effects that may occur during the execution of e (corresponding to a guarantee condition in the rely-guarantee formalism [16]); ε_2 summarizes effects of the interacting environment that e can tolerate while still functioning as expected (a rely condition). Finally, ε_3 summarizes the side effects that may occur between start and completion of e. All the effects that the environment might introduce must be recorded in ε_3 because they are not under "our" control and might happen at any time, even as the very last thing before the final result is returned. The effects flagged in ε_1, on the other hand, do not necessarily show up in ε_3, for a computation might be able to clean up those effects prior to returning a final result. The requirement that $\text{rds}(\varepsilon) \cap \text{chs}(\varepsilon) = \emptyset$ is owed to the fact that all effects should preserve their own precondition; the precondition of $rd_{\mathfrak{l}}$ is agreement on $\mathfrak{l}$, which is not preserved by $ch_{\mathfrak{l}}$. The requirement $\text{chs}(\varepsilon) \subseteq \text{wrs}(\varepsilon)$ reflects that $ch_{\mathfrak{l}}$ includes $wr_{\mathfrak{l}}$ as a special case.

Consider computations $e_1 = X := !X + 1; X := !X + 1$ and $e_2 = X := !X + 2$. Let ε_X stand for $\{rd_X, wr_X\}$. Each of the two computations can be assigned the effect $(\varepsilon_X, \emptyset, \varepsilon_X)$, but they are distinguishable at that effect typing. Let e be **if** $X = 1$ **then** *diverge*, which has effect specification $(\emptyset, \varepsilon_X, \varepsilon_X)$. Assuming that $e_1 = e_2$ at type $(\varepsilon_X, \emptyset, \varepsilon_X)$, then from our parallel congruence rule (in Figure 5) we could derive that $e_1 \| e = e_2 \| e$ at effect type $(\varepsilon_X, \varepsilon_X, \varepsilon_X)$, which is clearly not true. Under the looser specification $(\{ch_X\}, \emptyset, \varepsilon_X)$, 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 $\texttt{int}\ \&\ rd_X \mid \varepsilon \mid \varepsilon, rd_X$. However, it can also be assigned the effect $\texttt{int}\ \&\ \emptyset \mid \varepsilon \mid \varepsilon, rd_X$. Some effect specifications seem not be needed in practice. The important ones are those $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ that do not have read effects in $\varepsilon_1 \cup \varepsilon_2$.

We write ε^C for ε with all read effects removed and each $wr_{\mathfrak{l}}$ in ε replaced by $ch_{\mathfrak{l}}$. We sometimes write rd_X, wr_X, ch_X for $rd_{\texttt{int}(X)}, wr_{\texttt{int}(X)}, ch_{\texttt{int}(X)}$. Note that if $\varepsilon^C \cup \varepsilon_1$ is a well-formed effect, then $\text{rds}(\varepsilon_1) \cap (\text{wrs}(\varepsilon) \cup \text{chs}(\varepsilon)) = \emptyset$. We use this observation to simplify some side conditions, abbreviating $\{ch_{\mathfrak{l}}, wr_{\mathfrak{l}}\}$ by just $ch_{\mathfrak{l}}$ in examples, so the chaotic effect silently implies the write effect.

Notations: For well-formed effects $\varepsilon, \varepsilon'$ we write $\varepsilon \perp \varepsilon'$ to mean $\text{rds}(\varepsilon) \cap \text{wrs}(\varepsilon') = \text{rds}(\varepsilon') \cap \text{wrs}(\varepsilon) = \text{wrs}(\varepsilon) \cap \text{wrs}(\varepsilon') = \emptyset$. Note that this implies $\text{chs}(\varepsilon) \cap \text{rds}(\varepsilon') = \emptyset$, etc. We write $\mathsf{h} \overset{\text{rds}(\varepsilon)}{\sim} \mathsf{h}'$ to mean $\mathsf{h} \overset{\mathfrak{l}}{\sim} \mathsf{h}'$ for each $\mathfrak{l} \in \text{rds}(\varepsilon)$. We write $\overset{\varepsilon}{\to}$ for the transitive closure of $(\bigcup_{\mathfrak{l} \in \text{wrs}(\varepsilon)} \overset{\mathfrak{l}}{\to}) \cup \bigcup_{\mathfrak{l} \in \mathsf{w}} (\overset{\mathfrak{l}}{\to} \cap \overset{\mathfrak{l}}{=})$. Thus, $\overset{\varepsilon}{\to}$ allows steps by locations recorded as writing in ε and silent steps by all locations in the current world. We define $\varepsilon_1 \sqcup \varepsilon_2$, appearing in the parallel congruence rule, by $\varepsilon_1 \sqcup \varepsilon_2 = (\varepsilon_1 \cup \varepsilon_2) \setminus \{wr_\ell \mid wr_\ell \notin \varepsilon_1 \cap \varepsilon_2\} \setminus \{ch_\ell \mid ch_\ell \notin \varepsilon_1 \cap \varepsilon_2\}$.

6. TYPING AND CONGRUENCE RULES

Types are given by the grammar

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

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

We use two judgments:

- $\Gamma \vdash v \le v' : \tau$ specifying that values v and v' have type τ and that v approximates v',
- $\Gamma \vdash e \le e' : \tau\ \&\ \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$ specifying that the programs e and e' under the context Γ have type τ, with the effect specification $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ specifying, respectively, the effects during execution, the effects of the interacting environment and the start and completion effects. Moreover, e approximates e' at this specification.

We assume an ambient set of *axioms* of the form (v, v', τ) where v, v' are values and τ is a type, meaning that v and v' are claimed to be of type τ 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 $\Gamma \vdash e : \tau\ \&\ \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$ simply to be abbreviations for the 'diagonal' part of the inequational judgements, i.e. they hold when $\Gamma \vdash v \le v : \tau$ and $\Gamma \vdash e \le e : \tau\ \&\ \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$ 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 appoximation 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 parallization rule further down.

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

The rules labelled (Sem) make available 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 [11]. The other ones are concurrent version of analogous rules for sequential computation 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 congruence rule in that it requires the participating computations to mutually tolerate each other. This time, however, since the two computations being compared will do rather different things temporarily they must be oblivious against chaotic access, hence the $(-)^C$ strengthenings in the premise.

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

The following definitions provide the semantics of effects.

Definition 6.1 (Tiling). *Assume* $\mathsf{w} \vdash \varepsilon$. *Then we write* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$ *to mean that (i)* $\mathsf{h} \models \mathsf{w} \Rightarrow \mathsf{h} \xrightarrow{\varepsilon} \mathsf{h}_1$ *and (ii)* $\mathsf{h}' \models \mathsf{w} \Rightarrow \mathsf{h}' \xrightarrow{\varepsilon} \mathsf{h}_1'$ *and (iii)* $\mathsf{h} \overset{\mathrm{rds}(\varepsilon)}{\sim} \mathsf{h}'$ *and* $\mathfrak{l} \in \mathrm{wrs}(\varepsilon) \setminus \mathrm{chs}(\varepsilon)$ *imply* $(\mathsf{h} \overset{\mathfrak{l}}{=} \mathsf{h}_1 \wedge \mathsf{h}' \overset{\mathfrak{l}}{=} \mathsf{h}_1') \vee \mathsf{h}_1 \overset{\mathfrak{l}}{\sim} \mathsf{h}_1'$.

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

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 operational, version is inspired by that of Birkedal et al [11].

Lemma 6.2. *Suppose that* $\mathsf{w} \vdash \varepsilon$, $\mathsf{w} \vdash \varepsilon_1$, $\mathsf{w} \vdash \varepsilon_2$. *The following hold whenever well-formed.*

1. $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$ *and* $[\varepsilon](\mathsf{h}_1, \mathsf{h}_1', \mathsf{h}_2, \mathsf{h}_2')$ *imply* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_2, \mathsf{h}_2')$
2. $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}, \mathsf{h}')$
3. *If* $\varepsilon_1 \subseteq \varepsilon_2$ *then* $[\varepsilon_1](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1') \Rightarrow [\varepsilon_2](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$
4. $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1') \Rightarrow [\varepsilon^C](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$
5. *If* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{k}, \mathsf{k}')$ *and* $\mathsf{h} \overset{\mathrm{rds}(\varepsilon)}{\sim} \mathsf{h}'$ *then* $\mathsf{k} \overset{\mathrm{rds}(\varepsilon)}{\sim} \mathsf{k}'$. *(this relies on* $\mathrm{rds}(\varepsilon) \cap \mathrm{chs}(\varepsilon) = \emptyset$.*)*
6. *Suppose* $[\varepsilon](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$. *If* $\mathsf{h} \models \mathsf{w}$ *then* $\mathsf{h}_1 \models \mathsf{w}$*; if* $\mathsf{h}' \models \mathsf{w}$ *then* $\mathsf{h}_1' \models \mathsf{w}$.

7. LOGICAL RELATION

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

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

Similarly, a computation specification is a relation $Q \subseteq T\mathbb{V} \times T\mathbb{V}$ *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. Admissibility is needed for the soundness of recursion and, finally, closure under the canonical deflations makes admissible closure interact well with arrows [6].

Definition 7.2. *If* $E \subseteq \mathbb{V} \times \mathbb{V}$ *and* $Q \subseteq T\mathbb{V} \times T\mathbb{V}$ *then the relation* $E{\to}Q \subseteq \mathbb{V} \times \mathbb{V}$ *is defined by*

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

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

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

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

Definition 7.4. *Let* $E \subseteq \mathbb{V} \times \mathbb{V}$ *be a value specification and* $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ *an effect specification. We define the relations* $T_0(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *and* $T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *between sets of trace-value pairs, i.e. on* $\mathcal{P}(Tr \times Values)$*:*
$(U, U') \in T_0(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *if and only if*

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

We define the relation $T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3) \subseteq T\mathbb{V} \times T\mathbb{V}$ *as the 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 predomains being SFP allows these admissible closures to be safely 'ignored' in proofs.*

The game-theoretic view of $T_0(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ may be understood as follows. Given $U, U' \in T\mathbb{V}$ we can consider a game between a proponent (who believes $(U, U') \in T\mathbb{V}$) and an opponent who believes otherwise. The game begins by the opponent selecting an element $((\mathsf{h}_1, \mathsf{k}_1)\dots(\mathsf{h}_n, \mathsf{k}_n), a) \in U$ and $\mathsf{h}_1 \models \mathsf{w}$, the *pilot trace*, and a start heap $\mathsf{h}_1' \models \mathsf{w}$ such that $\mathsf{h}_1 \overset{\mathrm{rds}(\varepsilon_3)}{\sim} \mathsf{h}_1'$ to begin a trace in U'. Then, the proponent answers with a matching heap k_1' so that $[\varepsilon_1](\mathsf{h}_1, \mathsf{h}_1', \mathsf{k}_1, \mathsf{k}_1')$. If $\mathsf{h}_1 \overset{\mathrm{rds}(\varepsilon_1)}{\sim} \mathsf{h}_1'$ does not hold, proponent does not need to ensure that writes are in sync. The opponent then plays a heap h_2' so that $[\varepsilon_2](\mathsf{k}_1, \mathsf{k}_1', \mathsf{h}_2, \mathsf{h}_2')$. At this point, it is in the proponents interest to make sure that $\mathsf{k}_1 \overset{\mathrm{rds}(\varepsilon_2)}{\sim} \mathsf{k}_1'$ for otherwise opponent may make "funny" moves.

Then proponent plays heap k_2' such that $[\varepsilon_1](\mathsf{h}_2, \mathsf{h}_2', \mathsf{k}_2, \mathsf{k}_2')$, etc. until proponent has played k_n' so that $[\varepsilon_1](\mathsf{h}_n, \mathsf{h}_n', \mathsf{k}_n, \mathsf{k}_n')$. After that final heap has been played, it is checked that $[\varepsilon_3](\mathsf{h}, \mathsf{h}', \mathsf{k}_n, \mathsf{k}_n')$ holds. If not, proponent loses. If yes, then proponent must also play a value a' and it is then checked whether or not $((\mathsf{h}_1', \mathsf{k}_1')\dots(\mathsf{h}_n', \mathsf{k}_n'), a') \in U'$ and $(a\,E\,a')$. If this is the case or if at any one point in the game the opponent was unable to move because there exists no appropriate heap then the proponent has won the game. Otherwise the opponent wins and we have $(U, U') \in T_0(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ iff the proponent has a winning strategy for that game.

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.

$$\frac{}{\Gamma \vdash \texttt{true} \le \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} \le \texttt{false} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash n \le n : \texttt{int}} \qquad \frac{}{\Gamma, x : \tau \vdash x \le x : \tau} \qquad \frac{\Gamma \vdash v \le v' : \tau_1 \times \tau_2}{\Gamma \vdash v.i \le v'.i : \tau_i}$$

$$\frac{\Gamma \vdash e_1 \le e_2 : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad \Gamma \vdash e_1 \le e_2 : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}{\Gamma \vdash e_1 \le e_3 : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3} \qquad \frac{\Gamma \vdash v \le v' : \tau}{\Gamma \vdash v \le v' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3} \qquad \frac{\Gamma \vdash v_i \le v_i' : \tau_1 \ i = 1,2}{\Gamma \vdash (v_1, v_2) \le (v_1', v_2') : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash v_1 \le v_1' : \tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2 \quad \Gamma \vdash v_2 \le v_2' : \tau_1}{\Gamma \vdash v_1\ v_2 \le v_1'\ v_2' : \tau_2 \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3} \qquad \frac{\begin{array}{c}\Gamma \vdash v \le v' : \texttt{bool} \\ \Gamma \vdash e_1 \le e_1' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad \Gamma \vdash e_2 \le e_2' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3\end{array}}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \le\ \texttt{if}\ v'\ \texttt{then}\ e_1'\ \texttt{else}\ e_2' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \le e_1' : \tau_1 \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \\ \Gamma, x{:}\tau_1 \vdash e_2 \le e_2' : \tau_2 \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3\end{array}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \le \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' : \tau_2 \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3} \qquad \frac{\Gamma, f{:}\tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2, x{:}\tau_1 \vdash e \le e' : \tau_2 \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}{\Gamma \vdash \texttt{rec}\ f\ x = e \le \texttt{rec}\ f\ x = e' : \tau_1 \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \tau_2}$$

$$\frac{\Gamma \vdash e_1 \le e_1' : \tau_1 \mathbin{\&} \varepsilon_1 \mid \varepsilon \cup \varepsilon_2 \mid \varepsilon \cup \varepsilon_2 \cup \varepsilon' \quad \Gamma \vdash e_2 \le e_2' : \tau_2 \mathbin{\&} \varepsilon_2 \mid \varepsilon \cup \varepsilon_1 \mid \varepsilon \cup \varepsilon_1 \cup \varepsilon'}{\Gamma \vdash e_1 \| e_2 \le e_1' \| e_2' : \tau_1 \times \tau_2 \mathbin{\&} \varepsilon_1 \cup \varepsilon_2 \mid \varepsilon \mid \varepsilon \cup \varepsilon' \cup (\varepsilon_1 \sqcup \varepsilon_2)} \qquad \frac{(v, v', \tau)\ \text{an axiom}}{\Gamma \vdash v \le v : \tau}\ \mathrm{Ax}_1$$

$$\frac{\Gamma \vdash e \le e : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad [\![e]\!] = [\![e']\!]}{\Gamma \vdash e' \le e' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}\ \mathrm{Sem}_1 \qquad \frac{\Gamma \vdash e \le e : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad [\![e]\!] = [\![e']\!]}{\Gamma \vdash e \le e' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3}\ \mathrm{Sem}_2 \qquad \frac{(v, v', \tau)\ \text{an axiom}}{\Gamma \vdash v' \le v' : \tau}\ \mathrm{Ax}_2$$

$$\frac{\Gamma \vdash e \le e' : \tau \mathbin{\&} \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3 \quad \varepsilon_1 \subseteq \varepsilon_1' \quad \varepsilon_2' \subseteq \varepsilon_2 \quad \varepsilon_3 \subseteq \varepsilon_3'}{\Gamma \vdash e \le e' : \tau \mathbin{\&} \varepsilon_1' \mid \varepsilon_2' \mid \varepsilon_3'} \qquad \frac{\Gamma \vdash e \le e' : \tau \mathbin{\&} \varepsilon_1 \mid \emptyset \mid \varepsilon_3}{\Gamma \vdash \texttt{atomic}(e) \le \texttt{atomic}(e') : \tau \mathbin{\&} \varepsilon_3 \mid \varepsilon_2 \mid \varepsilon_2 \cup \varepsilon_3}\ \mathrm{Atom}$$

Figure 4: Typing and congruence rules

EXAMPLE 7.6. Consider again the programs $e_1 = (X := {!X} + 1; X := {!X} + 1)$ and $e_2 = (X := {!X} + 2)$. Let $\mathfrak{l} = \mathfrak{int}(X)$ be the abstract location for a single integer stored at X (see Section 4). Let $E = [\![\texttt{unit}]\!] = \{((), ())\}$ be the value specification for the unit type.

We show that $([\![e_1]\!], [\![e_2]\!]) \in T(E, \{ch_\mathfrak{l}\}, \varepsilon, \varepsilon \cup \{rd_\mathfrak{l}, wr_\mathfrak{l}\}\}$ under the assumption that $\{ch_\mathfrak{l}\} \perp \varepsilon$, that is, when the environment does not read nor write X. This condition is clearly necessary, for e_1 and e_2 can be distinguished by an environment that reads or writes X.

Let us now prove the claim when $\{ch_\mathfrak{l}\} \perp \varepsilon$. The opponent picks a pilot trace in the semantics of e_1, for example, $((\mathsf{h}_1, \mathsf{k}_1)(\mathsf{h}_2, \mathsf{k}_2), ())$ where $\mathsf{h}_1(X) = n$ and $\mathsf{k}_1(X) = n + 1$ and $\mathsf{h}_2(X) = n'$ and $\mathsf{k}_2(X) = n' + 1$. The other possible traces are stuttering or mumbling variants of this one and do not present additional difficulties. The opponent also chooses a heap h_1' such that $\mathsf{h}_1 \overset{\mathfrak{l}}{\sim} \mathsf{h}_1'$, i.e., $\mathsf{h}_1'(X) = n$. Now the proponent will choose to stutter for the time being and thus selects $\mathsf{k}_1' := \mathsf{h}_1'$. Indeed, $[ch_\mathfrak{l}](\mathsf{h}_1, \mathsf{h}_1', \mathsf{k}_1, \mathsf{k}_1')$ holds, so this is legal. The opponent now presents h_2' such that $[\varepsilon](\mathsf{k}_1, \mathsf{k}_1', \mathsf{h}_2, \mathsf{h}_2')$. By the assumption on ε we know that $n' = \mathsf{h}_2(X) = \mathsf{k}_1(X) = n + 1$ and also $\mathsf{h}_2'(X) = \mathsf{k}_1'(X) = n$. The proponent now answers with $\mathsf{k}_2' := \mathsf{h}_2'[X \mapsto n+2]$. It follows that $[ch_\mathfrak{l}](\mathsf{h}_2, \mathsf{h}_2', \mathsf{k}_2, \mathsf{k}_2')$ and also $[rd_\mathfrak{l}, wr_\mathfrak{l}](\mathsf{h}_1, \mathsf{h}_1', \mathsf{k}_2, \mathsf{k}_2')$. Finally, by stuttering $(\mathsf{h}_1', \mathsf{h}_1')(\mathsf{h}_2', \mathsf{h}_2'[X \mapsto n + 2]) \in [\![e_2]\!]$ so that proponent wins the game.

EXAMPLE 7.7. Consider $e_1 = (X := {!X} + 1 \| Y := {!Y} + 1)$ and $e_2 = (X := {!X} + 1; Y := {!Y} + 1)$. We show $([\![e_1]\!], [\![e_2]\!]) \in T(E, \{ch_X, ch_Y\}, \varepsilon, \varepsilon \cup \{rd_X, rd_Y, wr_X, wr_Y\})$, provided ε does not read nor modify X and Y. This equivalence could be deduced syntactically using our parallelization equation shown in Figure 5. For illustrative purpose, however, we describe its semantic proof using a game.

The opponent picks a pilot trace in $[\![e_1]\!]$, for example, the trace $([n_1|n_2], [n_1|n_2+1])([n_1|n_2+1], [n_1+1|n_2+1])((), ())$, where $[n_X|n_Y]$ denotes a heap where X and Y store n_X and n_Y, respectively. Notice that in this trace, Y is incremented before X and since ε does not read nor modify X and Y, the environment move does not change the values in X nor Y. We are also given an initial heap h_1' that agrees with the initial heap $[n_1|n_2]$ on the reads of $\varepsilon \cup \{rd_X, rd_Y, wr_X, wr_Y\}$. Thus, h_1' should be of the form $[n_1|n_2]$.

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

The following is one of our main technical results, and shows that the computation specifications $T(\dots)$ 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* $(U, U') \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *then*
$$(q_i(U), q_i(U')) \in T(E, \varepsilon_1, \varepsilon_2).$$
2. $T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *is a computation specification.*
3. *If* $(U, U') \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ *then*
$$(U^\dagger, U'^\dagger) \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3).$$
4. *If* $(a, a') \in E$ *then* $(rtn(a), rtn(a'))$ *is in* $T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$.

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

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

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

$$\frac{(v, v', \tau) \text{ an axiom}}{\Gamma \vdash v \leq v' : \tau} \text{ Ax} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \,\&\, \varepsilon_1 \mid \varepsilon^C \mid \varepsilon^C \qquad \Gamma, x : \tau_3, y : \tau_1 \vdash e_2 : \tau_2 \,\&\, \varepsilon_2 \mid \varepsilon \mid \varepsilon \cup \varepsilon_2 \qquad \varepsilon \perp \varepsilon_1}{\Gamma \vdash \texttt{let}\, y{=}e_1 \,\texttt{in}\, \lambda x.e_2 \leq \lambda x.\texttt{let}\, y{=}e_1 \,\texttt{in}\, e_2 : \tau_3 \xrightarrow[\varepsilon]{\varepsilon_1^C \cup \varepsilon_2 \mid \varepsilon \cup \varepsilon_3} \tau_2 \,\&\, \varepsilon_1^C \mid \varepsilon \mid \varepsilon} \text{ Lambda Hoist}$$

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

Figure 5: Effect-dependent transformations.

5. *Suppose that $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ is an effect specification where $\varepsilon_1 \cup \varepsilon_2 \subseteq \varepsilon_3$. Suppose that whenever $\mathsf{h} \overset{\mathrm{rds}(\varepsilon_1)}{\sim} \mathsf{h}'$ and $c(\mathsf{h}) = (\mathsf{h}_1, a)$ then there exist (h_1', a') such that $c'(\mathsf{h}') = (\mathsf{h}_1', a')$ and $[\varepsilon_1](\mathsf{h}, \mathsf{h}', \mathsf{h}_1, \mathsf{h}_1')$ and aEa'. Then for any ε_2, $(\mathit{fromstate}(c), \mathit{fromstate}(c')) \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_3)$.*
6. *If $(f, f') \in E_1 {\to} T(E_2, \varepsilon_1, \varepsilon_2, \varepsilon_3)$ and*
$$(U, U') \in T(E_1, \varepsilon_1, \varepsilon_2, \varepsilon_3)$$
then $(\mathit{bnd}(f, U), \mathit{bnd}(f', U')) \in T(E_2, \varepsilon_1, \varepsilon_2, \varepsilon_3)$.
7. *If $(U_1, U_1') \in T(E_1, \varepsilon_1, \varepsilon {\cup} \varepsilon_2, \varepsilon {\cup} \varepsilon_2 {\cup} \varepsilon')$ and $(U_2, U_2') \in T(E_2, \varepsilon_2, \varepsilon \cup \varepsilon_1, \varepsilon \cup \varepsilon_1 \cup \varepsilon')$ then $(U_1 \mid U_1', U_2 \mid U_2') \in T(E_1 \times E_2, \varepsilon_1 \cup \varepsilon_2, \varepsilon, \varepsilon \cup \varepsilon' \cup (\varepsilon_1 \sqcup \varepsilon_2))$.*
8. *If $(U, U') \in T(E, \varepsilon_1, \emptyset, \varepsilon_3)$, we have $(\mathit{at}(U), \mathit{at}(U')) \in T(\varepsilon_3, \varepsilon_2, \varepsilon_2 \cup \varepsilon_3)$.*

Proof. Ad 7. Suppose that $(U_1, U_1') \in T(E_1, \varepsilon_1, \varepsilon {\cup} \varepsilon_2, \varepsilon \cup \varepsilon_2 \cup \varepsilon')$ and $(U_2, U_2') \in T(E_2, \varepsilon_2, \varepsilon \cup \varepsilon_1, \varepsilon \cup \varepsilon_1 \cup \varepsilon')$ and let $(t, (a, b)) \in U_1 \mid U_2$, thus $\mathit{inter}(t_1, t_2, t)$ (ignoring † by item 3) where $(t_1, a) \in U_1$ and $(t_2, b) \in U_2$. Let S_1, S_2 be corresponding winning strategies. The idea is to use S_1 when we are in t_1 and to use S_2 when we are in t_2. Supposing that t starts with a t_1 fragment we begin by playing according to S_1. Let t be of the form:

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

composed of pieces of the traces t_1 and t_2. Assume w.l.o.g. that the first piece $(\mathsf{h}_1, \mathsf{k}_1) \cdots (\mathsf{h}_n, \mathsf{k}_n)$ is a part of t_1. We are given a initial heap h_1' such that $\mathsf{h} \overset{\mathrm{rds}(\varepsilon \cup \varepsilon' \cup (\varepsilon_1 \sqcup \varepsilon_2))}{\sim} \mathsf{h}'$. Since $\mathrm{rds}(\varepsilon_1 \sqcup \varepsilon_2) = \mathrm{rds}(\varepsilon_1) \cup \mathrm{rds}(\varepsilon_2)$, we can apply strategy S_1 to guide us through the first part of the game, obtaining:

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

Moreover, we have an environment move which forms the tile $[\varepsilon](\mathsf{k}_n, \mathsf{k}_n', \mathsf{h}_{n+1}, \mathsf{h}_{n'+1})$. So the tile $[\varepsilon \cup \varepsilon_1](\mathsf{h}_1, \mathsf{h}_1', \mathsf{h}_{n+1}, \mathsf{h}_{n+1}')$ 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:

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

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

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

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

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

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

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

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

Corollary 7.9. *Suppose that $\Gamma \vdash v : \tau$ and $\Gamma \vdash e : \tau \,\&\, \varepsilon_1 \mid \varepsilon_2 \mid \varepsilon_3$. Then $(\eta, \eta') \in [\![\Gamma]\!]$ (interpreting a context as a cartesian product) implies $([\![v]\!]\eta, [\![v]\!]\eta') \in [\![\tau]\!]$ and $([\![e]\!]\eta, [\![e]\!]\eta') \in T([\![\tau]\!], \varepsilon_1, \varepsilon_2, \varepsilon_3)$.*

Proof. By induction on derivations. Most cases are already subsumed by Theorem 7.8. The typing rules regarding functions and recursion follow from the definitions and from the fact that all specifications are admissible. □

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 τ if for all f such that $\vdash f : \tau \xrightarrow[\varepsilon]{\varepsilon_1 \mid \varepsilon_3} \texttt{int}$ ("observations") it is the case that if $((\mathsf{h}_{init}, \mathsf{k}), n) \in [\![f\ v]\!]$ for $v \in \mathbb{Z}$ and starting from h_{init} then $((\mathsf{h}_{init}, \mathsf{k}'), n) \in [\![f\ v']\!]$ for some k'. We write $\vdash v \leq_{obs} v'$ in this case. We say that v and v' are observationally equivalent at type τ, written $\vdash v =_{obs} v'$ if both $\vdash v \leq_{obs} v' : \tau$ and $\vdash v' \leq_{obs} v : \tau$.*

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

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 $\vdash v \leq_{obs} v' : \tau$.*

PROOF. If $\vdash f : \tau \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \texttt{int}$ then by Thm 7.9 we have $([\![f]\!], [\![f]\!]) \in [\![\tau \xrightarrow[\varepsilon_2]{\varepsilon_1 \mid \varepsilon_3} \texttt{int}]\!]$, so

$$([\![f\ v]\!], [\![f\ v']\!]) \in T([\![\texttt{int}]\!], \varepsilon_1, \varepsilon_2, \varepsilon_3)^+.$$

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

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 $\lambda\_.e_1 =_{obs} \lambda\_.e_2$ at type $\texttt{unit} \xrightarrow[\varepsilon]{\{ch_{\mathfrak{l}}\} \mid \varepsilon \cup \{rd_{\mathfrak{l}}, wr_{\mathfrak{l}}\}} \texttt{unit}$ whenever X does not appear in ε.

9. EFFECT-DEPENDENT TRANSFORMATIONS

We will now establish the semantic soundness of the inequational theory of effect-dependent program transformations given in Figure 5. It includes concurrent versions of the effect-dependent equations from [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 $e := \texttt{let}\ x = \texttt{?}\ \texttt{in}\ (x, x)$. Then, even though `?` does not read nor write any location we only have $e \leq (\texttt{?}, \texttt{?})$, but not $(\texttt{?}, \texttt{?}) \leq e$ for $(\texttt{?}, \texttt{?})$ admits the result $(\texttt{true}, \texttt{false})$ but e does not. Furthermore, due to presence of nontermination the equations for dead code elimination and pure lambda hoist also hold in one direction only. It might be possible to restore both directions of said equations by introducing special effects for nondeterminism and nontermination; we have not explored this avenue. We concentrate the individual effect-dependent transformations before summarising the foregoing results in the general soundness Theorem 9.2.

In many of the equations, co-effects play an important role. For example, in the commuting and parallelization equations, the internal effects ε_1 and ε_2 in the premises are replaced by ε_1^C and ε_2^C in the internal effects of the conclusion. This makes sense intuitively because the computations are run in a different order, so for the internal moves, the locations in ε_1 and ε_2 can be modified in any way (see Example 7.7). However, in the global effect, we can still guarantee the effects ε_1' and ε_2' because of the $\perp$-conditions. This intuition appears directly in the soundness proofs.

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

- ***Commuting*** *If $(U_1, U_1') \in T(E_1, \varepsilon_1, \varepsilon^C, \varepsilon^C \cup \varepsilon_1')$ and $(U_2, U_2') \in T(E_2, \varepsilon_2, \varepsilon^C, \varepsilon^C \cup \varepsilon_2')$ and $\varepsilon_1 \perp \varepsilon$ and $\varepsilon_2 \perp \varepsilon$ and $\varepsilon_1' \perp \varepsilon_2'$ then*

$$\begin{gathered}(\{(t_1 t_2, (v_1, v_2)) \mid (t_1, v_1) \in U_1, (t_2, v_2) \in U_2\}^\dagger, \\ \{(t_2' t_1', (v_1', v_2')) \mid (t_1', v_1') \in U_1', (t_2', v_2') \in U_2'\}^\dagger) \\ \in T(E_1 \times E_2, (\varepsilon_1 \cup \varepsilon_2)^C, \varepsilon, \varepsilon \cup \varepsilon_1' \cup \varepsilon_2')\end{gathered}$$

- ***Duplicated*** *Given $(U, U') \in T(E, \varepsilon_1, \varepsilon_2^C, \varepsilon_2^C \cup \varepsilon')$ with $\mathrm{rds}(\varepsilon') \cap \mathrm{wrs}(\varepsilon') = \emptyset$ and $\varepsilon_2 \perp \varepsilon_1$, we have*

$$\begin{gathered}(\{(t, (v, v)) \mid (t, v) \in U\}^\dagger, \{(t_1' t_2', (v_1', v_2')) \mid (t_1', v_1') \in U', \\ (t_2', v_2') \in U'\}^\dagger) \in T(E, \varepsilon_1, \varepsilon_2, \varepsilon_2 \cup \varepsilon')\end{gathered}$$

- ***Pure*** *Let $(U, U') \in T(E, \varepsilon_1, \varepsilon_2^C, \varepsilon_2^C)$, such that $\varepsilon_1 \perp \varepsilon_2$. If $((q_1, k_1) \ldots (q_n, k_n), v) \in U$ for some* arbitrary *trace $t = (q_1, k_1) \ldots (q_n, k_n)$ (with $q_1 \models \mathsf{w}$) and value v, then $(rtn(v), U') \in T(E, \varepsilon_1^C, \varepsilon_2, \varepsilon_2)$;*

- ***Dead*** *Suppose that $(U, U') \in T(\texttt{unit}, \varepsilon_1, \varepsilon_2, \varepsilon_2 \cup \varepsilon_1')$, where $\mathrm{wrs}(\varepsilon_1') = \emptyset$ and $\varepsilon_1 \perp \varepsilon_2$. Then $(U, rtn(())) \in T(\texttt{unit}, \varepsilon_1^C, \varepsilon_2, \varepsilon_2 \cup \varepsilon_1')$.*

- ***Parallelization*** *If $(U_1, U_1') \in T(E_1, \varepsilon_1, \varepsilon^C \cup \varepsilon_2^C, \varepsilon^C \cup \varepsilon_2^C \cup \varepsilon_1')$ and $(U_2, U_2') \in T(E_2, \varepsilon_2, \varepsilon^C \cup \varepsilon_1^C, \varepsilon^C \cup \varepsilon_1^C \cup \varepsilon_2')$ and $\varepsilon_1 \perp \varepsilon_2$ and $\varepsilon_1 \perp \varepsilon$ and $\varepsilon_2 \perp \varepsilon$, then*

$$\begin{gathered}(U_1 \| U_2, \{(t_1' t_2' (v_1', v_2')) \mid (t_1', v_1') \in U_1', (t_2', v_2') \in U_2'\}^\dagger) \in \\ T(E_1 \times E_2, \varepsilon_1^C \cup \varepsilon_2^C, \varepsilon, \varepsilon \cup \varepsilon_1' \cup \varepsilon_2')\end{gathered}$$

PROOF. 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 $inter(t_1, t_2, t)$ and $(t_i, v_i) \in U_i$. Just as in the commuting case we set up two side games U_i vs. U_i' on t_i, v_i. 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 $\varepsilon \cup \varepsilon_1' \cup \varepsilon_2'$ location by location using the definition of tiling. □

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

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

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

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

The various cases now follow from earlier results in a straightforward manner. 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 $e \equiv \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ and $e' \equiv \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2'$. Inductively, we know $([\![e_1]\!], [\![e_1']\!]) \in [\![\Gamma \vdash \tau_1\&(\varepsilon_1, \varepsilon_2, \varepsilon_3)]\!]^{n_1}$ and $([\![e_1]\!], [\![e_1']\!]) \in [\![\Gamma, x{:}\tau_1 \vdash \tau\&(\varepsilon_1, \varepsilon_2, \varepsilon_3)]\!]^{n_2}$ for some $n_1, n_2 > 0$. By Theorem 7.9, we also have $([\![e_1]\!], [\![e_1]\!]) \in [\![\Gamma \vdash \tau_1\&(\varepsilon_1, \varepsilon_2, \varepsilon_3)]\!]$ and analogous statements for e_1', e_2, e_2'. We can, therefore, assume, w.l.o.g. that $n_1 = n_2$ and then use Theorem 7.8 (6) repeatedly (n_1 times) so as to conclude

$$([\![e]\!], [\![e]\!]) \in [\![\Gamma \vdash \tau\&(\varepsilon_1, \varepsilon_2, \varepsilon_3)]\!]^{n_1}.$$

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

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 ε does not read nor write X:

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

The analogous typings for readSnd and 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: $([2|3], [2|3]), ([17|3], [17|3]), (())$
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).
$([13|12], [13|12]), ([13|12], [17|12]), (())$
If, however, the environment plays $[18|21]$ (a modification of both components of X has occurred), then we answer $[17|21]$. Again,
$([13|12], [13|12]), ([18|21], [17|21]), (())$
is a possible trace for writeFst(17). It is easy to check that there is a strategy that justifies the typing given above.
Now, consider a program, e_1, that only calls readFst, writeFst, and another program, e_2, that only calls readSnd, writeSnd. Since the former functions have disjoint effects to the latter ones, e_1 and e_2 will have effect specifications, respectively, of the form $(\varepsilon_1, \varepsilon^C \cup \varepsilon_2^C, \varepsilon^C \cup \varepsilon_2^C \cup \varepsilon_1)$ and $(\varepsilon_2, \varepsilon^C \cup \varepsilon_1^C, \varepsilon^C \cup \varepsilon_1^C \cup \varepsilon_2)$, where $\varepsilon_1 \cap \varepsilon_2 = \varepsilon_1 \cap \varepsilon = \varepsilon_2 \cap \varepsilon = \emptyset$. Thus we can use the parallelization rule shown in Figure 5 to conclude that the behavior of $e_1 \| e_2$ is the same as executing these programs sequentially, although they read and write to the same concrete location.

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

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

where $MSQ = \{rd_{\mathfrak{msq}(X)}, wr_{\mathfrak{msq}(X)}\}$. That is, they approximate each other at a type where the environment is allowed to operate on the queue as well. We also note that the converse of the axiom is obvious by stuttering and mumbling. After consuming a dummy argument () let the resulting pilot trace be $(\mathsf{h}_1, \mathsf{k}_1) \ldots (\mathsf{h}_i, \mathsf{k}_i) \ldots (\mathsf{h}_n, \mathsf{k}_n)a$ and h_1' be the start heap to match. We can now assume that the passages from k_i to h_{i+1} follow the protocol, i.e. $\mathsf{k}_i \xrightarrow{\mathfrak{msq}(X)} \mathsf{h}_{i+1}$. (Should this not be the case we are free to make arbitrary moves and still win the game by default of the environment player.) Therefore, there must exist i such that in the move $(\mathsf{h}_i, \mathsf{k}_i)$ the element a is dequeued and $\mathsf{h}_j = \mathsf{k}_j$ holds for $j \neq i$. We can thus match this trace by a trace in the semantics of $\mathtt{atomic}(\mathsf{dequeue}\ ())$ by stuttering until i:
$(\mathsf{h}_1', \mathsf{h}_1') \ldots (\mathsf{h}_i', \ldots$
where h_j and h_j' have the same content, but not necessarily the exact same layout. Given the environment's allowed effects it is then clear that also h_i and h_i' have the same content, but not necessarily the same as h_1 and h_1' because in the meantime other operations on the queue might have succeeded. We then dequeue the corresponding element from h_i' leading to k_i' and continue by stuttering.
$\ldots, \mathsf{k}_i')(\mathsf{h}_{i+1}', \mathsf{h}_{i+1}') \ldots (\mathsf{h}_n', \mathsf{h}_n')a'$
It is now clear that this is a matching trace and that $a = a'$ so we are done.

Notice that the congruence rules now allow us to deduce the equivalence of $op_1 \parallel \cdots \parallel op_n$ and $\mathtt{atomic}(op_1) \parallel \cdots \parallel \mathtt{atomic}(op_n)$ for 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 [8, 30]. Notably, in the presence of concurrency the rules for code duplication, motion, and deletion, which in the sequential realm are fairly intuititive, 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, 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 [11] 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 [11] 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 [11]. 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 [15] 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 [5]. 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 [9]. 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 [11], 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.

11. REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] S. Abramsky and A. Jung. Domain theory, 1994. Online Lecture Notes, avaliable from CiteSeerX.

[3] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 340–353, 2009.

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

[5] N. Benton, M. Hofmann, and V. Nigam. Abstract effects and proof-relevant logical relations. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 619–632, 2014.

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

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

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

[9] N. Benton, A. Kennedy, M. Hofmann, and V. Nigam. Counting successes: Effects and transformations for non-deterministic programs. In *A List of Successes*

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

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

[11] L. Birkedal, F. Sieczkowski, and J. Thamsborg. A concurrent logical relation. In 26th International Workshop/21st Annual Conference of the EACSL, CSL, volume 16 of *LIPIcs*, pages 107–121, 2012.

[12] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1996.

[13] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *15th European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*. Springer, 2006.

[14] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

[15] S. D. Brookes. The essence of parallel algol. *Inf. Comput.*, 179(1):118–149, 2002.

[16] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput.*, 17(4):807–841, 2007.

[17] R. De Nicola and M. Hennessy. Testing equivalence for processes. In *Automata, Languages and Programming, 10th Colloquium*, pages 548–560, 1983.

[18] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 287–300, 2013.

[19] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

[20] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI*, 2003.

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

[22] J. B. Jensen and L. Birkedal. Fictional separation logic. In *Proceedings of the 21st European Conference on Programming Languages and Systems* ESOP, pages 377–396, 2012.

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

[24] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL* , pages 455–468. ACM, 2012.

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

[26] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, May 1998.

[27] N.Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *3rd ACM Workshop on Types in Language Design and Implementation TLDI*, 2007.

[28] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of the 26 ACM Symposium on Principles of Programming Languages POPL*, 1999.

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

[30] J. Thamsborg and L. Birkedal. A Kripke logical relation for effect-based program transformations. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP*, pages 445–456. ACM, 2011.

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

[32] A. J. Turon and M. Wand. A separation logic for refining concurrent objects. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 247–258. ACM, 2011.