Computational adequacy of trace-semantics

Nick Benton, Martin Hofmann, and Vivek Nigam

MSR Cambridge, LMU Munich, U Paraiba

Abstract. We extend Brookes' denotational trace semantics to a higher-order language with parallelism and prove its computational adequacy with respect to a small-step operational semantics.

1 Memory model

We assume a countably infinite set $\mathbb{L}$ of physical locations $X_1, \ldots, X_n, \ldots$ and a set $\mathbb{VB}$ of "R-values" that can be stored in those references including integers, booleans, locations, and tuples of R-values, written $(v_1, \ldots, v_n)$. We assume that it is possible to tell of which form a value is and to retrieve its components in case it is a tuple. A heap h, then, is a *finite map* from $\mathbb{L}$ to $\mathbb{VB}$, written $\{(X_1, \mathsf{c}_1), (X_2, \mathsf{c}_2), \ldots, (X_n, \mathsf{c}_n)\}$, specifying that the value stored in location X_i is c_i. We write dom(h) for the domain of h. Finally, we write $\mathsf{h}[X \mapsto \mathsf{c}]$ for the heap that agrees with h except that it gives the variable X the value c. The set of heaps is denoted by $\mathbb{H}$. We also assume that *new*(h, v) yields a pair (X, h') where $X \in \mathbb{L}$ is a fresh location and $\mathsf{h}' \in \mathbb{H}$ is $\mathsf{h}[X \mapsto v]$. Note that the R-values do not contain functions and thus, we do not consider "higher-order store".

2 Syntax

In this section, we define the syntax of a metalanguage for stateful computations and higher-order functions. It is essentially a version of Plotkin's PCF with concurrency primitives, namely constructs for parallel, interleaved computation and for atomic execution. Communication between parallel computations is via a shared heap that contains updatable memory cells which hold structured values comprising in particular pointers, but we do not include the possibility of storing functions in the heap (no higher-order store). Thus, the memory model is rather akin to that of Java and similar languages. In the language itself all heap locations are understood as globally visible variables; there is no mechanism for hiding private portions of the store. On top of the simply-typed language refined type systems can be imposed which limit access to heap-allocated datastructures in various ways.

Syntax The syntax of untyped values and computations is:

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

Here, x ranges over variables, v_r ranges over R-values, and c ranges over built-in functions including arithmetic operations, test functions to tell whether a value is an integer, a function, a pair, or a reference, equality test for simple values, etc.

For each built-in function c we assume a partial function F_c on values modelling its behaviour. For example, we expect $F_+(n, n') = n + n'$ if both n, n' are integers.

The construct $\texttt{rec}\ f\ x.e$ 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 the variable f does not occur 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 $\texttt{ref}(v)$ (allocating) returns a fresh location initialized with v. The metatheory is simplified by using "let-normal form", in which the only elimination for computations is `let`, though we sometimes nest computations as shorthand for let-expanded versions in examples.

The construct $e_1 \| e_2$ is evaluated by running e_1 and e_2 in parallel until each has produced a value, say v_1 and v_2. The result of the evaluation of $e_1 \| e_2$ then is (v_1, v_2). Parallel evaluation is carried out by interleaving in an arbitrary order with the understanding that assignments to, lookups from, and allocations of locations are atomic; the evaluation of nested expressions is, however, in general not atomic. To introduce further atomicity we have the $\texttt{atomic}(e)$ construct which executes e atomically, i.e., without possibly intervention of the environment. From this primitive it is possible to define the construct $\texttt{cas}(X, v_1, v_2)$ which checks whether the content of location X is v_1 and if so replaces it atomically and without further intervention with v_2. In this case, the return value of the construct is `true`, and otherwise `false`. We can also define the more general $\texttt{await}\ e_1\ \texttt{then}\ e_2$ construct [5] which repeatedly, and atomically, evaluates e_1 until it returns the value `true` at which point e_2 is evaluated atomically and without allowing any intermediate intervention of the environment.

We define the set of free variables $FV(e)$ of a term as usual, e.g. $FV(\texttt{let}\ y = 0\ \texttt{in}\ x + y) = \{x\}$ and $FV(\texttt{rec}\ x\ f.f\ x\ y) = \{y\}$. A term e is closed if $FV(e) = \emptyset$. If v is a closed value we define the substitution $e[v/x]$ of v for x in e in the usual way. Note that $FV(e[v/x]) = FV(e) \setminus \{x\}$.

Note that all locations are global variables in the sense that they may appear in terms. Of course, such terms are not very sensible and can be formally ruled out by means of type systems or by syntactic side conditions.

Examples Some example functions can be found in the technical report [3].

3 Simple types

For technical reasons[1] we introduce a very simple type system. Its types are given by the following grammar where o represents basic values ($\mathbb{VB}$) and the other type formers stand for function and product types, respectively.

$$\tau ::= o \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$$

[1] Our adequacy result can also be established for an untyped language but then we need to define a mixed-variance predicate [9] which introduces further technicalities.

The typing rules are given in Figure 1. They use typing contexts ranged over by Γ, which are just finite maps from variables to types in the standard fashion. If e is a closed term or value we write $e : \tau$ instead of $\emptyset \vdash e : \tau$.

$$\frac{v \in \mathbb{VB}}{\Gamma \vdash v : o} \quad \frac{x \in \Gamma(x)}{x \in \mathrm{dom}(\Gamma)} \quad \frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2) : \tau_1 \times \tau_2} \quad \frac{}{\Gamma \vdash c : o \to o} \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash v : o}{\Gamma \vdash !v : o} \quad \frac{\Gamma \vdash v_1 : o \quad \Gamma \vdash v_2 : o}{\Gamma \vdash v_1 := v_2 : o} \quad \frac{\Gamma \vdash v : o}{\Gamma \vdash \mathtt{ref}(v) : o}$$

$$\frac{\Gamma \vdash v : o \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau}$$

$$\frac{\Gamma, x{:}\tau_1, f{:}\tau_1 \to \tau_2 \vdash e : \tau_2}{\Gamma \vdash \mathtt{rec}\ x\ f.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \| e_2 : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{atomic}(e) : \tau}$$

Fig. 1. Simple typing rules

4 Operational semantics

We now define inductively a relation $(\mathsf{h}, e) \longrightarrow (\mathsf{h}', e')$ (small-step operational semantics) on pairs of heaps and *closed* terms which signifies that given heap h the term e can be reduced atomically to term e' while at the same time the heap is modified to h'. The defining rules are given in Figure 2.

$$\frac{}{\mathsf{h}, (v_1, v_2).1 \longrightarrow \mathsf{h}, v_1} \quad \frac{}{\mathsf{h}, (v_1, v_2).2 \longrightarrow \mathsf{h}, v_2} \quad \frac{F_c(v) = v'}{\mathsf{h}, c\ v \longrightarrow \mathsf{h}, v'}$$

$$\frac{\mathsf{h}(X) = v}{\mathsf{h}, !X \longrightarrow \mathsf{h}, v} \quad \frac{X \in \mathrm{dom}(\mathsf{h}), v \in \mathbb{VB}}{\mathsf{h}, X := v \longrightarrow \mathsf{h}[X \mapsto v], ()} \quad \frac{\mathit{new}(\mathsf{h}, v) = (X, \mathsf{h}'), v \in \mathbb{VB}}{\mathsf{h}, \mathtt{ref}(v) \longrightarrow \mathsf{h}', X}$$

$$\frac{}{\mathsf{h}, \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \longrightarrow \mathsf{h}, e_1} \quad \frac{}{\mathsf{h}, \mathtt{if}\ \mathtt{false}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \longrightarrow \mathsf{h}, e_2}$$

$$\frac{\mathsf{h}, e_1 \to \mathsf{h}', e_1'}{\mathsf{h}, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \longrightarrow \mathsf{h}', \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2} \quad \frac{}{\mathsf{h}, \mathtt{let}\ x = v_1\ \mathtt{in}\ e_2 \longrightarrow \mathsf{h}, e_2[v_1/x]}$$

$$\frac{}{\mathsf{h}, (\mathtt{rec}\ x\ f.t)\ v \longrightarrow \mathsf{h}, t[v/x][\mathtt{rec}\ x\ f.t/f]}$$

$$\frac{\mathsf{h}, e_1 \longrightarrow \mathsf{h}', e_1'}{\mathsf{h}, e_1 \| e_2 \longrightarrow \mathsf{h}', e_1' \| e_2} \quad \frac{\mathsf{h}, t_2 \longrightarrow \mathsf{h}', t_2'}{\mathsf{h}, e_1 \| e_2 \longrightarrow \mathsf{h}', e_1 \| e_2'} \quad \frac{}{\mathsf{h}, v_1 \| v_2 \longrightarrow \mathsf{h}, (v_1, v_2)} \quad \frac{\mathsf{h}, e \to^* \mathsf{h}', v}{\mathsf{h}, \mathtt{atomic}(e) \to \mathsf{h}', v}$$

Fig. 2. Definition of small-step operational semantics

The reader may check that the example functions evaluate as predicted. If $\mathsf{h}, e \longrightarrow^* \mathsf{h}', v$ where v is a value we say that e may evaluate successfully with result v. If $\mathsf{h}, e \not\longrightarrow$ then e is stuck in h. This can only happen if there is some kind of typing error, e.g. $() + 5$ or if e tries to reference a non-existing location. An infinite sequence of steps $\mathsf{h}, e \longrightarrow h_2, e_2 \longrightarrow h_3, e_3, \ldots$ then h, e is called a divergent computation.

We are only interested in successful computations and do not distinguish between the possibility of stuck or divergent computations so long as successful computations are present ("may semantics"). This is formalised by the following definition.

Definition 1. *Let v, v' be two closed values of the same type τ and let h_{ini} be a distinguished initial heap. We say that v approximates v' operationally with respect to h_{ini}, written $v \leq^{op}_{\mathsf{h}_{ini}} v'$, if for all closed values $f : \tau \to o$ we have that whenever $\mathsf{h}_{ini}, f\ v \longrightarrow^* \mathsf{h}, b$ for some $b \in \{\mathtt{true}, \mathtt{false}\}$ then $\mathsf{h}_{ini}, f\ v \longrightarrow^* \mathsf{h}', b$ for some heap h'.*

In particular, if $v \leq^{op}_{\mathsf{h}_{ini}} v'$ and $v' \leq^{op}_{\mathsf{h}_{ini}} v$ then v and v' are indistinguishable (in the may sense [8]) by boolean valued tests.

Note that due to the presence of arbitrary locations in terms (including the tests), programs like $\lambda x.\mathtt{let}\ y = loc(1)\ \mathtt{in}\ 0$ and $\lambda x.\mathtt{let}\ y = loc(1)\ \mathtt{in}\ 0$ are not equivalent since the observing context might "know" the newly allocated location. To avoid this, one can use refined type systems to restrict the observing contexts, for example here to those that do not contain built-in locations. Of course, notions of operational equivalence based on more refined type systems can be considered which will identify those and related programs.

5 Denotational semantics

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

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

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

addition, f has a finite image then f is called a *deflation* [2]. Note that if f is a retract then $\mathrm{dom}(f) = \mathrm{img}(f)$ and if $a \in \mathrm{img}(f)$ then $a = f(a)$.

Lemma 1. *Let $f : A \rightharpoonup A$ be a deflation. If $f(a)$ is at all defined then it is a compact element.*

Proof. If $\sup_i a_i \geq f(a)$ then by idempotency, monotonicity, and continuity we have $\sup_i f(a_i) \geq f(a)$, but since f has a finite image there must exist i such that $f(a_j) = f(a_i)$ for $j \geq i$ and thus $a_i \geq f(a_i) \geq f(a)$ as required.

In the applications, it is this consequence (compactness) of image-finiteness that is needed. However, it is not stable under taking function spaces whereas image-finiteness is as shown by the next lemma.

Lemma 2. *Let $p : A \rightharpoonup A$ and $q : B \rightharpoonup B$ be deflations. Then $r(f)(a) = q(f(p(a)))$ defines a deflation on $A \rightharpoonup B$.*

Proof. It is clear that $r \circ r = r$ and $p \leq id$. As for image-finiteness observe that a function in the image of r is a partial function from the image of p to the image of q of which there are only finitely many.

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

5.1 Traces

We use traces to model terminating runs of concurrent computation. It records such a run 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 very large) set of traces which provides for all possible environment interactions. It can be likened to a graph of a function which also contains argument-value pairs for each possible argument.

Definition 2 (Traces). *A trace is a finite sequence of pairs (h, k) where $\mathsf{h}, \mathsf{k} \in \mathbb{H}$. We write Tr for the set of traces.*

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

We recall that Brookes [5] interprets while programs with parallel composition as sets of traces closed under stuttering and mumbling and shows that this semantics is adequate and fully abstract for may-equivalence when all variables are global and of integer type. The following definition extends this semantics with result values drawn from an arbitrary predomain and thus permits an extension of Brookes's model to higher-order functions and general recursion. As one might expect, the extension to higher-order is no longer fully abstract, but remains adequate.

Definition 3 (Trace Monad). *Let A be a predomain. A domain TA is defined as follows. The elements of TA are sets U of pairs (t, a) where t is a trace and $a \in A$ such that the following properties are satisfied:*

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

The elements of TA are partially ordered by inclusion.

Lemma 3. *If A is a predomain then TA is a domain.*

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

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

For example, the semantics of a program like $X :=!X+1; X :=!X+1; !X$ will contain many traces including the following, where we write $[n]$ for the heap in which X has value n:
$(([10], [12]), 12)$,
$(([10], [11])([15], [16]), 16)$,
$(([10], [11])([17], [17])([15], [16]), 16)$

Axiom [S&M] is taken from Brookes. It ensures that the semantics does not distinguish between late and early choice [11] and related phenomena which are reflected, e.g., in resumption semantics [10], but do not affect observational equivalence. Notice that a non-terminating computation is represented by the empty set and thus is invisible if it *may* happen, but does not necessarily do so ("may semantics" [8]). For example, the semantics of a program like $X := 0; \texttt{if } X{=}0 \texttt{ then } 0 \texttt{ else diverge}$ will be the same as that of $X := 0; 0$ and contain, for example $(([10], [0]), 0)$ but also (stuttering) $((([10], [0]), ([34], [34])), 0)$. Note that it is not possible to tell from a trace whether an external update of X has happened before or after the reading of X.

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

$$e_1 \equiv \texttt{if ? then } X := 0; \texttt{true else } X := 0; \texttt{false}$$
$$e_2 \equiv X := 0;\ \texttt{?}$$

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 a semantic model based on transition systems or resumptions and bisimulation these would be distinguished and special mechanisms such as history and prophecy variables [1], forward-backward simulation [7], or speculation [11] must be installed to recover from this.

Axioms [Down] and [Sup] are known from the Hoare powerdomain [10]. Recall that for a given predomain the Hoare powerdomain PA contains the subsets of A which are downclosed ([Down]) and closed under suprema of chains ([Sup]). Such subsets are also known as Scott-closed sets. Thus, TA is the restriction of $P(Tr \times A)$ to the sets closed under stuttering and mumbling. Axiom [Down] ensures that the ordering is indeed a partial order and not merely a preorder. It is also closely related to "may semantics". Additional nondeterministic outcomes that are less defined than existing ones are not recorded in the semantics.

Axiom [Sup] is needed to make the embedding of values as singletons continuous.

Definition 4. *If $U \subseteq Tr \times A$ then we denote $U^\dagger$ the least subset of TA containing U, i.e. $U^\dagger$ is the closure of U under mumbling and stuttering [S& M], [Down], [Sup].*

Lemma 4. *An element of $U \in TA$ is compact iff there exists a finite subset $V \subseteq U$ such that $U = V^\dagger$ and whenever $(t, a) \in V$ then a is a compact element of A.*

Proof. Suppose that $U = V^\dagger$ as stated and that $\sup_i U_i \supseteq U$. We must then have $V \subseteq U_i$ for some i since V is finite and the value (second) components of elements in V are compact thus cannot appear in $\sup_i U_i$ by limit closure only. But since U_i is an element of TA we have in fact $V^\dagger \subseteq U_i$ and thus $U \subseteq U_i$ as required.

Conversely, given a compact element $U \in TA$ we can find a chain U_i in TA consisting of elements of the described form and such that $\sup_i U_i = U$. By compactness of U we then have $U = U_i$ for some i and thus U is of the described form, too.

Definition 5. *Let A, B be a predomains. We define the following continuous functions*

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

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

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

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

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

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

If $c \in TA$ we define a computation $\mathit{atomic}(c) \in TA$ by

$$\begin{aligned}
&\mathit{atomic} : TA \to TA\\
&\mathit{atomic}(U) := \{((\mathsf{h}, \mathsf{k}), v) \mid ((\mathsf{h}, \mathsf{k}), v) \in U\}^\dagger
\end{aligned}$$

This function will serve as the interpretation of the "atomic" construct. Notice that due to mumbling $((\mathsf{h},\mathsf{k}),v) \in V$ iff there exists an element $((\mathsf{h}_1,\mathsf{h}_2)(\mathsf{h}_2,\mathsf{h}_3)\ldots(\mathsf{h}_{n-2},\mathsf{h}_{n-1})(\mathsf{h}_{n-1},\mathsf{h}_n),v) \in V$ where $\mathsf{h} = \mathsf{h}_1$ and $\mathsf{h}_n = \mathsf{k}$. The presence of such an element, however, models an atomic execution of the computation represented by U.

5.2 Semantic values

We define the predomain of values $\mathbb{V}$ as follows: Values are defined as either R-values, tuples of values or continuous functions from values to elements of $T\mathbb{V}$, i.e. the predomain $\mathbb{V}$ is given as the least solution of the following domain equation.

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

We tend to identify the summands of the right hand side with subsets of $\mathbb{V}$ but may use tags like $\mathit{fun}(f) \in \mathbb{V}$ when $f : \mathbb{V} \to T\mathbb{V}$ to avoid ambiguities when necessary. We will refer to the elements of $T\mathbb{V}$ as *computations*.

Fix finite subsets $\mathbb{VB}_i \subseteq \mathbb{VB}$ for $i \in \mathbb{N}$ such that $\mathbb{VB}_i \subseteq \mathbb{VB}_{i+1}$ and $\mathbb{VB} = \bigcup_i \mathbb{VB}_i$. Similarly, fix finite subsets $Tr_i \subseteq Tr$ such that $Tr_i \subseteq Tr_{i+1}$ and $\bigcup_i Tr_i = Tr$. Note that Tr is a countably infinite set since traces are essentially finite sequences of heaps of which there are countably many, too.

Definition 6. *We now define the following families of partial continuous functions $p_i : \mathbb{V} \rightharpoonup \mathbb{V}$ and $q_i : T\mathbb{V} \to T\mathbb{V}$ (note that q_i is total):*

$$\begin{aligned}
p_0(v) &= \textit{undefined}\\
p_{i+1}(v) &= v \quad \textit{if } v \in \mathbb{VB}_{i+1}\\
p_{i+1}(v) &\quad \textit{undefined} \quad \textit{if } v \in \mathbb{VB} \setminus \mathbb{VB}_{i+1}\\
p_{i+1}(v_1,\ldots,v_n) &= (p_i(v_1),\ldots,p_i(v_n))\\
p_{i+1}(g)(v) &= q_i(g(p_i(v))), \textit{ if } g : \mathbb{V} \to T\mathbb{V} \textit{ and } v \in \mathrm{dom}(p_i)\\
p_{i+1}(g)(v) &= \{\}, \textit{ if } g : \mathbb{V} \to T\mathbb{V} \textit{ and } v \notin \mathrm{dom}(p_i)\\
q_i(U) &= \{(t,v') \mid \exists v \in \mathrm{dom}(p_i).(t,v) \in U, t \in Tr_i, v' \le v\}
\end{aligned}$$

Application of partial functions is understood eagerly, i.e., if $p_i(v)$ appears in the right hand side and happens to be undefined then the entire right hand side is undefined.

Lemma 5. *The p_i and q_i each form an increasing chain of deflations such that $\sup_i p_i = id_{\mathbb{V}}$ and $\sup_i q_i = id_{T\mathbb{V}}$. Moreover, each of the deflations p_i, q_i has a finite image and the elements of the form $p_i(a)$ and $q_i(U)$ are compact.*

Proof. We first prove all the properties specific to a single p_i or q_i simultaneously by induction on i. The case $i = 0$ is trivial. Most of the other properties are obvious, too; as a specific example we show that p_{i+1} is idempotent: we have $p_{i+1}(p_{i+1}(f))(v) = q_i(q_i(f(p_i(p_i(v))))) = q_i(f(p_i(v)))$ if $v \in \mathrm{dom}(p_i)$ and undefined otherwise. We also argue image-finiteness of p_{i+1}: if $f(v) = q_i(f(p_i(v)))$ (idempotency!) then $\mathrm{dom}(f) \subseteq \mathrm{dom}(p_i) = \mathrm{img}(p_i)$ and $\mathrm{img}(f) \subseteq \mathrm{img}(q_i)$, so there number of such functions is at most $|\mathrm{img}(q_i)|^{|\mathrm{img}(p_i)|}$.

One thing that is not immediate is that indeed $q_i : T\mathbb{V} \to T\mathbb{V}$; this amounts to showing that if $U \in T\mathbb{V}$ and $v = \sup_j v_j$ and $(t,v_j) \in q_i(U)$ for each j then $(t,v) \in q_i(U)$.

By definition, all v_j are majorised an element of $\mathrm{img}(p_i)$. Thus, by image-finiteness of p_i there is a single $w \in \mathbb{V}$ such that $(t, w) \in U$ and $v_i \leq p_i(w)$ for all i. It follows that $v \leq p_i(w)$ and thus $(t, v) \in U$.

Finally, $\sup_i p_i = id_{\mathbb{V}}$ and $\sup_i q_i = id_{T\mathbb{V}}$ is direct from the way the predomain $\mathbb{V}$ is constructed as a least solution.

The lemma shows in particular that $\mathbb{V}$ and $T\mathbb{V}$ are *bifinite* (equivalently SFP) (pre-)domains [2] and as such also Scott (pre-)domains.

Definition 7. *Let P be a subset of a predomain A. We define $Adm(P)$ as the least admissible superset of P. Concretely, $a \in Adm(P)$ iff there exists a chain $(a_i)_i$ such that $a_i \in P$ for all i and $a = \sup_i a_i$.*

The following lemma is rather obvious; we include it mainly for illustration.

Lemma 6. *Let $f : A \to B$ and $P \subset A$, $Q \subset B$ with Q admissible. If $f : P \to Q$ then $f : Adm(P) \to Q$.*

Proof. Suppose that $a \in Adm(P)$ so that $a = \sup_i a_i$ where $a_i \in P$. By assumption, $f(a_i) \in Q$ so, by admissibility of Q, we get $\sup_i(f(a_i)) \in Q$, but $f(a) = \sup_i(f(a_i))$ by continuity of f.

Lemma 7. *Let A, B be predomains and $P \subset A$, $Q \subset B$. We have $Adm(P) \times Adm(Q) = Adm(P \times Q)$.*

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

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

The following Lemma gives conditions under which the admissible closure ("*Adm*") commutes with function spaces. It is taken from [6].

Lemma 8. *Let A, B be predomains and let $(p_i)_i$ be a chain of retracts on B such that $p_i(b)$ is compact for each i and $\sup_i p_i = id$ and $b \in Q$ implies $p_i(b) \in Q$ for all i.*

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

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

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

Remark 1. Suppose that we have a monad M on the category of predomains. This means that we have functions $\eta : A \to MA$ and $ap : (A{\to}MB) \to MA{\to}MB$ such that the usual [] equations are satisfied. We do not need those equations here.

Furthermore, suppose that for each A and subset $P \subseteq A$ we have a subset $MP \subseteq MA$ such that $\eta : P \to MP$ and $ap : (Q \to MP) \to MQ \to MP$ whenever $P \subseteq A$ and $Q \subseteq B$.

We can then form $M'P = Adm(MP)$ and show that M' has a similar property provided that the preconditions of Lemma 8 are satisfied in certain cases. Indeed, $\eta : P \to M'P$ always holds simply because Adm is increasing. But now suppose that $P \subseteq A$ and $Q \subseteq B$ and that there is an increasing family of deflations $q_i : MB \to MB$ with supremum id_{MB}. We then also have

$$ap : (P \to M'Q) \to M'P \to M'Q$$

To see this, suppose that $f : A \to B$ and $f : P \to M'Q$ and $x \in M'P$. By Lemma 8, we have $f \in Adm(P \to MQ)$ and so, by Lemma 7, $(f, x) \in Adm(P \times MQ)$. But then $ap(f)(x) \in Adm(MQ) = M'Q$ as required.

The reason for wanting to compose a monad M with admissible closure is that it then becomes compatible with the fixpoint combinator.

Namely suppose that MA is always a domain (has a least element).

We then have

$$(-)^\dagger : ((A \to MB) \to (A \to MB)) \to (A \to MB)$$

Now suppose that $P \subseteq A$ and $Q \subseteq B$ and $\bot \in MQ$. We then have

$$(-)^\dagger : ((P \to M'Q) \to (P \to M'Q)) \to (P \to M'Q)$$

Indeed, if $F : (P \to M'Q) \to (P \to M'Q)$ then $F^n(\lambda x.\bot) \in P \to M'Q$ for all n, so, since $M'Q$ is admissible, we also get $F^\dagger = \sup_n F^n(\lambda x.\bot) \in M'Q$.

With M' replaced by M there is no reason why the analogous property should hold.

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.

We use environments, ranged over by ρ to map variables to values ($\mathbb{V}$). We represent such environments as tuples of values using some implicit enumeration of the variables. This allows us to treat envorinments as elements of $\mathbb{V}$ themselves. We use the standard notations $\rho(x)$ stands for the i-th projection from $\rho \in \mathbb{V}$ if x is the i-th variable and $\rho[x \mapsto v]$ to (functionally) update the i-th slot in ρ.

6 Program equivalences

The denotational semantics can directly validate some expected program equivalences that are more difficult to obtain directly from the operational semantics. An example is fixpoint unrolling which is at the basis of various loop optimisations: For any term t we have

$$[\![\texttt{rec}\ x\ f.e]\!]\rho = [\![\lambda x.t[(\texttt{rec}\ x\ f.e)/f]]\!]\rho$$

The proof of this is direct from the interpretation of recursive definition as least fixpoints and the following substitution lemma:

$$
\begin{aligned}
\llbracket x \rrbracket\rho &= \rho(x) \\
\llbracket v_r \rrbracket\rho &= v_r \\
\llbracket (v_1, v_2) \rrbracket\rho &= (\llbracket v_1 \rrbracket\rho, \llbracket v_2 \rrbracket\rho) \\
\llbracket v.i \rrbracket\rho &= d_i \text{ if } i = 1, 2,\ \llbracket v \rrbracket\rho = (d_1, d_2) \\
\llbracket c \rrbracket\rho &= \mathit{fun}(f) \\
&\text{where } f(v) = \eta(F_c(v)) \\
&\text{if } F_c(v) \text{ is defined and } f(v) = \emptyset, \text{ otherwise.} \\
\llbracket \texttt{rec}\ f\ x.t \rrbracket\rho &= \mathit{fun}(g^{\ddagger}\,\rho) \\
&\text{where } g(\rho, u) = \lambda d.\llbracket t \rrbracket\rho[f \mapsto u, x \mapsto d]
\end{aligned}
$$

$$
\begin{aligned}
\llbracket v \rrbracket\rho &= \eta(\llbracket v \rrbracket\rho) \\
\llbracket \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \rrbracket\rho &= \mathit{ap}(\lambda d.\llbracket t_2 \rrbracket\rho[x \mapsto d], \llbracket t_1 \rrbracket\rho) \\
&= \{(t_1 t_2, v) \mid (t_1, u) \in \llbracket e_1 \rrbracket\rho, (t_2, v) \in \llbracket e_2 \rrbracket\rho[x \mapsto u]\}^{\dagger} \\
\llbracket v_1\ v_2 \rrbracket\rho &= \llbracket v_1 \rrbracket\rho(\llbracket v_2 \rrbracket\rho) \\
\llbracket \texttt{if}\ v\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \rrbracket\rho &= \llbracket t_1 \rrbracket\rho, \text{ if } \llbracket v \rrbracket\rho = \texttt{true} \\
\llbracket \texttt{if}\ v\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \rrbracket\rho &= \llbracket t_2 \rrbracket\rho, \text{ if } \llbracket v \rrbracket\rho = \texttt{false} \\
\llbracket !v \rrbracket\rho &= \mathit{fromstate}(\lambda \mathsf{h}.(\mathsf{h}, \mathsf{h}(X))), \text{ when } \llbracket v \rrbracket\rho = X \\
\llbracket v_1 := v_2 \rrbracket\rho &= \mathit{fromstate}(\lambda \mathsf{h}.(\mathsf{h}[X \mapsto \llbracket v_2 \rrbracket\rho], \mathit{int}(0))), \text{ if } \llbracket v_1 \rrbracket\rho = X \\
\llbracket \texttt{ref}(v) \rrbracket\rho &= \mathit{fromstate}(\lambda \mathsf{h}.\mathit{new}(\mathsf{h}, \llbracket v \rrbracket\rho)) \\
\llbracket \texttt{atomic}(t) \rrbracket\rho &= \mathit{atomic}(\llbracket t \rrbracket) \\
\llbracket t_1 \parallel t_2 \rrbracket\rho &= \llbracket t_1 \rrbracket\rho \parallel \llbracket t_2 \rrbracket\rho \\
\llbracket v \rrbracket\rho &= 0, \text{ otherwise} \\
\llbracket t \rrbracket\rho &= \emptyset, \text{ otherwise}
\end{aligned}
$$

Fig. 3. Denotational semantics

Lemma 9. *Let t be a term, v a value, and ρ an environment. We always have*

$$\llbracket e \rrbracket\rho[x \mapsto \llbracket v \rrbracket\rho] = \llbracket e[v/x] \rrbracket\rho$$

In a similar way, we can prove various call-by-value reduction laws, e.g.

$$\llbracket (\lambda x.e)v \rrbracket\rho = \llbracket e[x \mapsto v] \rrbracket\rho$$

and a similar one for `let`-expressions.

Combined with logical relations the denotational semantics can also be used to validate effect-dependent program equivalences [4], e.g. swapping two consecutive `let`-definitions provided their side-effects commute or memoising repeated `let`-definitions under similar conditions. This will be detailed in a forthcoming companion paper.

For all these equivalences to be meaningful for physical reality it is, however, crucial that the predictions made by the denotational semantics agree with those made by the operational one. The latter then, must of course still be shown to agree with what is actually going on in a computer.

We will therefore now show computational adequacy of the denotational semantics with respect to the denotational one which is the established technical formalisation of this agreement. In particular, computational adequacy entails that all semantic equivalences like the ones above hold in the sense of observational equivalence.

7 Computational adequacy

The goal of this section is to prove the following theorem.

Theorem 1. *Suppose that $\vdash e : o$.*

1. *If $\mathsf{h}_{ini}, e \longrightarrow^* \mathsf{h}', v$ then $((\mathsf{h}_{ini}, \mathsf{h}'), v) \in \llbracket e \rrbracket$ (*correctness*).*
2. *If $((\mathsf{h}_{ini}, \mathsf{h}'), v) \in \llbracket e \rrbracket$ then $\mathsf{h}_{ini}, e \longrightarrow^* \mathsf{h}', v$. (*adequacy*).*

Corollary 2 (Soundness w.r.t. observational approximation). *Suppose that $\vdash v : \tau$ and $\vdash v' : \tau$ and $\llbracket v \rrbracket \leq \llbracket v' \rrbracket$. Then $v \leq^{op}_{\mathsf{h}_{ini}} v'$.*

Proof. Let $f : \tau \to o$ be a (syntactic) value and $\mathsf{h}_{ini}, f\ v \longrightarrow^* \mathsf{h}, b$. By correctness, we have $((\mathsf{h}_{ini}, \mathsf{h}'), v) \in [\![f\ v]\!]$. But now, $[\![f\ v]\!] = \lceil\!\lceil f \rceil\!\rceil(\lceil\!\lceil v \rceil\!\rceil) \subseteq \lceil\!\lceil f \rceil\!\rceil(\lceil\!\lceil v' \rceil\!\rceil) = [\![f\ v']\!]$. So, adequacy yields $\mathsf{h}_{ini}, f\ v' \longrightarrow^* \mathsf{h}, b$ as required.

In fact, even the weaker assumption $[\![f\ v]\!] \subseteq [\![f\ v']\!]$ for all closed values $f : \tau \to o$, a semantic version of observational approximation, would be enough to conclude $v \leq^{op}_{\mathsf{h}_{ini}} v'$, see Theorem 3, below.

Let us now come to the proof of the adequacy theorem. The following lemma essentially establishes correctness:

Lemma 10. *Suppose that* $\mathsf{h}, e \longrightarrow \mathsf{h}', e'$ *and that* $(t, v) \in [\![e']\!]$. *Then* $((\mathsf{h}, \mathsf{h}')t, v) \in [\![e]\!]$.

Proof. This is by induction on the rules defining the operational semantics.

First, we consider that the closure under stuttering and mumbling commutes with prefixing a trace so that we can assume without loss of generality that the trace (t, v) got into $[\![e']\!]$ by the semantic clause "directly" and not by closure under stuttering and mumbling. E.g. suppose that $t = (\mathsf{h}_0, \mathsf{h}_0)t'$ where $(t', v) \in [\![e']\!]$. Then, if inductively $((\mathsf{h}, \mathsf{h}')t', v) \in [\![e]\!]$ we also have $((\mathsf{h}, \mathsf{h}')(\mathsf{h}_0, \mathsf{h}_0)t', v) \in [\![e]\!]$ by closure of $[\![e]\!]$ under stuttering.

In most of the rules from Fig. 2 we have $[\![e]\!] = [\![e']\!]$ and $\mathsf{h} = \mathsf{h}'$. E.g. the first four rules, the rule for recursion, one of the let rules are of that form. In these cases the claim is direct by closure of $[\![e]\!]$ under stuttering.

Consider now that $e = X := v$ and $e' = ()$ and $(t, ()) \in [\![()]\!]$. Note that in this case t is just an accumulation of stuttering steps. Clearly, by the definition of $[\![X := v]\!]$ we then have $((\mathsf{h}, \mathsf{h}[X \mapsto v])t, ()) \in [\![e]\!]$ as required.

Next, suppose that $e = \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ and $t' = \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2$ and $\mathsf{h}, e_1 \longrightarrow \mathsf{h}', e_1'$. If $(t, v) \in [\![e']\!]$ then (recalling that closure under stuttering and mumbling can be safely ignored here) $t = t_1 t_2$ where $(t_1, v_1) \in [\![e_1']\!]$. Inductively, we obtain $((\mathsf{h}, \mathsf{h}')t_1, v_1) \in [\![e_1]\!]$ and hence $((\mathsf{h}, \mathsf{h}')t, v) \in [\![e]\!]$.

Now, consider that $e = e_1 \| e_2$ and that $e' = e_1' \| e_2$ and $\mathsf{h}, e_1 \longrightarrow \mathsf{h}', e_1'$. If $(t, v) \in [\![e_1' \| e_2']\!]$ then $inter(t_1, t_2, t)$ and $v = (v_1, v_2)$ where $(t_1, v_1) \in [\![e_1']\!]$ and $(t_2, v_2) \in [\![e_2]\!]$. Inductively, we obtain $((\mathsf{h}, \mathsf{h}')t_1, v_1) \in [\![e_1]\!]$ and hence the claim.

The other cases are similar.

In order to get the converse ("adequacy") we need to work a little harder.

Denote the set of closed (syntactic) values by V and the set of closed terms by C considered as flat predomains.

Definition 8 (syntactic traces). *If $e \in C$ then we define its set of syntactic traces $Tr(e)$ as the set of pairs (t, v) where $t = (\mathsf{h}_1, \mathsf{k}_1) \ldots (\mathsf{h}_n, \mathsf{k}_n)$ is a trace, $v \in V$ is a (syntactic) value and there exist terms $e_1 = e, e_2, \ldots, e_n$ such that $\mathsf{h}_i, e_i \longrightarrow^* \mathsf{k}_i, e_{i+1}$ for $i < n$ and $\mathsf{h}_n, e_n \longrightarrow^* \mathsf{k}_n, v$.*

The syntactic traces are somewhat related to the traces in the denotational semantics as shown by the following lemma whose proof is direct.

Lemma 11. *1. Let e be a term. The set $Tr(t)$ is closed under mumbling and stuttering.*

2. *Let e_1 and e_2 be closed terms. We have*

$$Tr(e_1 \| e_2) = \{(t, v) \mid inter(t_1, t_2, t), (t_1, v_1) \in Tr(t_1), (t_2, v_2) \in Tr(t_2)\}$$

3. *Let e_1 be a closed term and $FV(e_2) \subseteq \{x\}$. We have*

$$Tr(\texttt{let } x{=}e_1 \texttt{ in } e_2) = \{(e_1 t_2, v) \mid (t_1, v_1) \in Tr(e_1), (t_2, v) \in Tr(e_2[v_1/x])\}$$

4. *Let $FV(e) \subseteq \{x, f\}$ and v a closed value. We have*

$$Tr((\texttt{rec } f\ x.e)\ v) = Tr(e[(\texttt{rec } f\ x.e)/f, v/x])$$

Definition 9. *For each type τ we now define admissible relations $[\![\tau]\!] \subseteq V \times \mathbb{V}$ and $[\![T\tau]\!], [\![T_0\tau]\!] \subseteq C \times T\mathbb{V}$ by induction on types as follows:*

$$\begin{array}{ll}
[\![o]\!] & = \{(v, v) \mid v \in \mathbb{VB}\} \\
[\![\tau_1 \times \tau_2]\!] & = \{((v_1, v_2), (w_1, w_2)) \mid (v_1, w_1) \in [\![\tau_1]\!], (v_2, w_2) \in [\![\tau_2]\!]\} \\
[\![\tau_1 \to \tau_2]\!] & = \{(v, f) \mid \forall (v', w) \in [\![\tau_1]\!].(v\ v', f(w)) \in [\![T\tau_2]\!]\} \\
[\![T\tau]\!] & = Adm([\![T_0\tau]\!]) \\
[\![T_0\tau]\!] & = \{(e, U) \mid \forall (t, w) \in U \exists v \in V.(t, v) \in Tr(e), (v, w) \in [\![\tau]\!]
\end{array}$$

Lemma 12. *If $(v, w) \in [\![\tau]\!]$ then $(v, \eta(w)) \in [\![T\tau]\!]$.*

Proof. If $(t, w') \in \eta(w)$ then $w' = w$ and $t = (\mathsf{h}_1, \mathsf{h}_1) \ldots (\mathsf{h}_n, \mathsf{h}_n)$, i.e., arises from the empty trace by mumbling and stuttering. But since $\mathsf{h}, v \longrightarrow^* \mathsf{h}, v$ holds for any h we then have $(t, v) \in Tr(v)$ and hence the claim.

Lemma 13. *If $(v, w) \in [\![\tau]\!]$ and $w' \leq w$ then $(v, w') \in [\![\tau]\!]$.*
If $(e, U) \in [\![T\tau]\!]$ and $U' \leq U$ then $(e, U') \in [\![T\tau]\!]$.
If $(e, U) \in [\![T\tau]\!]$ then $(e, q_i(U)) \in [\![T_0\tau]\!]$.

Proof. By induction on τ. Most cases are obvious (note that $\mathbb{VB}$ is flat). The only interesting case is the third assertion. So, suppose that $(e, U) \in [\![T\tau]\!]$. We then have $U = \sup_j U_j$ where $(e, U_j) \in [\![T_0\tau]\!]$. But since $q_i(U)$ is compact, we must have $q_i(U) = q_i(U_j)$ for some j. Now, suppose that $(t, w) \in q_i(U) = q_i(U_j)$. Then $(t, w_0) \in U_j$ for some w_j with $q_i(w_j) = w$. By assumption, we find v such that $(t, v) \in Tr(t)$ and $(v, w_0) \in [\![\tau]\!]$. The IH yields $(v, w) \in [\![\tau]\!]$ and hence $(e, q_i(U)) \in [\![T_0\tau]\!]$.

The following lemma will allow us to remove $(-)^\dagger$ closures in many cases.

Lemma 14. *Let e be a closed term of type τ. Suppose that $U = U_0^\dagger$ and that for all $(t, w) \in U_0$ there is $v \in V$ such that $(t, v) \in Tr(e)$ and $(v, w) \in [\![\tau]\!]$. Then $(e, U) \in [\![T\tau]\!]$.*

Proof. Since $U = \sup_i q_i(U)$, it suffices to show that $(e, q_i(U)) \in [\![T\tau]\!]$ for each i.
We claim that, in fact, $(e, q_i(U)) \in [\![T_0\tau]\!]$.
Define

$$V = \{(t, w) \mid \forall w' \leq q_i(w).\exists v' \in V.(v, w') \in [\![\tau]\!], (t, v) \in Tr(e)$$

Now we have $V = V^\dagger$. Indeed, $w = \sup_j w_j$ is a chain then $q_i(w) = q_i(w_j)$ for some j which establishes closure of V under [Sup]. Closure under [M& S] follows from Lemma 11 and [Down] follows from Lemma 13.

We also have $U_0 \subseteq V$: suppose $w' \leq q_i(w) \leq w \in U_0$ and suppose using the assumption on U_0 that $(v, w) \in [\![\tau]\!]$ and $(t, v) \in Tr(e)$. Then $(v, w') \in [\![\tau]\!]$ by Lemma 13.

It follows from this that $U \subseteq V$ and hence the result.

Lemma 15. *If* $(e_1, U_1) \in [\![T\tau_1]\!]$ *and* $(e_2, U_2) \in [\![T\tau_2]\!]$ *then* $(e_1 \| e_2, U_1 \parallel U_2) \in [\![T(\tau_1 \times \tau_2)]\!]$.

Proof. Consider the continuous function $\Phi : (C \times T\mathbb{V})^2 \to C \times T\mathbb{V}$ given by $\Phi((t_1, U_1), (t_2, U_2)) = (t_1 \| t_2, U_1 \parallel U_2)$.

We claim $\Phi : [\![T_0\tau_1]\!] \times [\![T_0\tau_1]\!] \to [\![T(\tau_1 \times \tau_2)]\!]$. Indeed, suppose that $(e_1, U_1) \in [\![T_0\tau_1]\!]$ and $(e_2, U_2) \in [\![T_0\tau_2]\!]$. We have $U_1 \parallel U_2 = U_0^\dagger$, where $U_0 = \{(t, (w_1, w_2)) \mid inter(t_1, t_2, t), (t_1, w_1) \in U_1, (t_2, w_2) \in U_2\}$. We need to show $(e_1 \| e_2, U_0^\dagger) \in [\![T(\tau_1 \times \tau_2)]\!]$ and use Lemma 14 for this. So let $(t, (w_1, w_2)) \in U_0$ where $inter(t_1, t_2, t), (t_1, w_1) \in U_1, (t_2, w_2) \in U_2$. By assumption, we have $v_1, v_2 \in V$ so that $(v_i, w_i) \in [\![\tau_i]\!]$ and $(t_i, v_i) \in Tr(e_i)$ for $i = 1, 2$. It follows that $(t, (v_1, v_2)) \in Tr(e_1 \| e_2)$ by Lemma 11 and $((v_1, v_2), (w_1, w_2))) \in [\![\tau_1 \times \tau_2]\!]$. This proves the claim.

But now, Lemmas 6 and 7 give $\Phi : [\![T\tau_1]\!] \times [\![T\tau_1]\!] \to [\![T(\tau_1 \times \tau_2)]\!]$ which gives the result.

Lemma 16. *If* $(e, U) \in [\![T\tau]\!]$ *then* $(\mathtt{atomic}(e), atomic(U)) \in [\![T\tau]\!]$.

Proof. Consider the continuous function $\Phi : (C \times T\mathbb{V}) \to C \times T\mathbb{V}$ given by $\Phi(t, U) = (\mathtt{atomic}(t), atomic(U))$.

We claim $\Phi : [\![T\tau]\!] \to [\![T\tau]\!]$. Indeed, suppose that $(e, U) \in [\![T_0\tau]\!]$. We have $atomic(U) = U_0^\dagger$, where $U_0 = \{((\mathsf{h}, \mathsf{k}), v) \mid ((\mathsf{h}, \mathsf{k}), v) \in U\}$. We need to show $(\mathtt{atomic}(e), U_0^\dagger) \in [\![T\tau]\!]$ and use Lemma 14 for this. So let $((\mathsf{h}, \mathsf{k}), w) \in U_0$ where $((\mathsf{h}, \mathsf{k}), w) \in U$. By assumption, we have $v \in V$ so that $(v, w) \in [\![\tau]\!]$ and $((\mathsf{h}, \mathsf{k}), v) \in Tr(e)$. It follows that $((\mathsf{h}, \mathsf{k}), v) \in Tr(\mathtt{atomic}(e))$. This proves the claim.

Now, Lemma 6 furnishes $\Phi : [\![T\tau]\!] \to [\![T\tau]\!]$ which gives the result.

Lemma 17. *Let* $f : \mathbb{V} \to T\mathbb{V}$ *be a continuous function and suppose that* $(e_1, U_1) \in [\![T\tau_1]\!]$ *and* $\lambda(v, w).(e_2[v/x], f(w)) \in [\![\tau_1]\!] \to [\![T\tau_2]\!]$ *then* $(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2, ap(f, U_1)) \in [\![T\tau_2]\!]$.

Proof. Denote $C(x)$ the set of terms e with $FV(e) \subseteq \{x\}$.

Consider the continuous function

$$\Phi : (C \times T\mathbb{V}) \times (C(x) \times (\mathbb{V} \to T\mathbb{V}) \to C \times T\mathbb{V}$$

given by $\Phi((e_1, U_1), (e_2, f)) = (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2, ap(f, U_1))$.

We claim

$$\Phi : [\![T_0\tau_1]\!] \times ([\![\tau_1]\!] \to [\![T_0\tau_2]\!] \to [\![T\tau_2]\!]$$

Indeed, suppose that $(e_1, U_1) \in [\![T_0\tau_1]\!]$ and $(\lambda(v, w).e_2[v/x], f(w)) \in [\![\tau_1]\!] \to [\![T_0\tau_2]\!]$. We have $ap(f, U_1) = U_0^\dagger$, where $U_0 = \{(t_1 t_2, w_2) \mid (t_1, w_1) \in U_1, (t_2, w_2) \in f(w_1)\}$. We need to show $(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2, U_0^\dagger) \in [\![T\tau_2]\!]$ and use Lemma 14 for this. So let

$(t_1 t_2, w_2) \in U_0$ where $(t_1, w_1) \in U_1, (t_2, w_2) \in f(w_1)$. The assumption $(e_1, U_1) \in [\![T_0 \tau_1]\!]$ yields $v_1 \in V$ so that $(v_1, w_1) \in [\![\tau_1]\!]$ and $(t_1, v_1) \in Tr(e_1)$. The other assumption about e_2 then provides $v_2 \in V$ such that $(v_2, f(w_1)) \in [\![\tau_2]\!]$ and $(t_2, v_2) \in Tr(e_2[v_1/x])$. It follows that $(t_1 t_2, v_2) \in Tr(\texttt{let}\ x = e_1\ \texttt{in}\ e_2)$ and thus the claim.

But now, Lemmas 7 and 8 furnish the stronger typing

$$\Phi : [\![T\tau_1]\!] \times ([\![\tau_1]\!] \to [\![T\tau_2]\!] \to [\![T\tau_2]\!]$$

from which the result follows.

Lemma 18. *Let $F : (\mathbb{V} \to T\mathbb{V}) \to (\mathbb{V} \to T\mathbb{V}$ be a continuous function and suppose that whenever $(v_x, w) \in [\![\tau_1]\!]$ and $(v_f, \varphi) \in [\![\tau_1 \to \tau_2]\!]$ then $(e[v_f/f, v_x/x], F(\varphi)(w)) \in [\![T\tau_2]\!]$. Then $(\texttt{rec}\ f\ x.e, F^\ddagger) \in [\![\tau_1 \to \tau_2]\!]$.*

Proof. Define $F_n : \mathbb{V} \to T\mathbb{V}$ by $F_0(w) = \emptyset$ and $F_{n+1} = F(F_n)$. Note that $F^\ddagger = \sup_n F_n$. We claim that $(\texttt{rec}\ f\ x.e, F_n) \in [\![\tau_1 \to \tau_2]\!]$ holds for all n. The result then follows since $[\![\tau_1 \to T\tau_2]\!]$ is admissible.

We prove the claim by induction on n. If $n = 0$ then this is trivially true since then $F_n(w) = \emptyset$.

For the inductive step assume $(\texttt{rec}\ f\ x.e, F_n) \in [\![\tau_1 \to \tau_2]\!]$ for some n and pick $(v_x, w) \in [\![\tau_1]\!]$. By assumption, we then have $(e[(\texttt{rec}\, f\, x.e)/f, v_x/x], F(F_n)(w)) \in [\![T\tau_2]\!]$. But now, by Lemma 11 and the fact that the definition of "$(e, U) \in [\![T\tau]\!]$" only depends on $Tr(e)$ and not e itself, it follows that $(\texttt{rec}\ f\ x.e, F_{n+1}) \in [\![\tau_1 \to \tau_2]\!]$ as required.

Theorem 2. *Suppose that $\Gamma \vdash e : \tau$, that E maps $\mathrm{dom}(\Gamma) =: \{x_1, \ldots, x_n\}$ to V and that ρ maps $\mathrm{dom}(\Gamma)$ to $\mathbb{V}$. If $(E(x), \rho(x)) \in [\![\tau]\!]$ for all $x \in \mathrm{dom}(\Gamma)$ then $(e[E(x_1)/x_1, \ldots, E(x_n)/x_n], [\![e]\!]\rho) \in [\![T\tau]\!]$. If $e = v$ is a value then $(v[E(x_1)/x_1, \ldots, E(x_n)/x_n], \lceil v \rceil \rho) \in [\![\tau]\!]$.*

Proof. We abbreviate $e[E(x_1)/x_1, \ldots, E(x_n)/x_n]$ by $e[E/\boldsymbol{x}]$ and use analogous notation for values and other terms.

If $e[E/\boldsymbol{x}]$ is a value then $[\![e[E/\boldsymbol{x}]]\!]\rho = \eta(\lceil e[E/\boldsymbol{x}] \rceil \rho)$ and so, in this case, it suffices to establish $(e[E/\boldsymbol{x}], \lceil e[E/\boldsymbol{x}] \rceil \rho) \in [\![\tau]\!]$ for the other part $(e[E/\boldsymbol{x}], [\![e[E/\boldsymbol{x}]]\!]\rho) \in [\![T\tau]\!]$ then follows by Lemma 12.

The proof now proceeds by induction on typing derivations.

Case $e = v \in \mathbb{VB}$. Then $e[E/\boldsymbol{x}] = e = v$ and $\lceil e \rceil \rho = v$. Clearly $(v, v) \in [\![o]\!]$.

Case $e = x_i$. Then $e[E/\boldsymbol{x}] = v_i$ and $\lceil e \rceil \rho = \rho(x)$ and the claim follows directly from the assumption.

Case $e = (v_1, v_2)$ and $\tau = \tau_1 \times \tau_2$. The induction hypothesis then yields $(v_i[E/\boldsymbol{x}], \lceil v_i \rceil \rho) \in [\![\tau_i]\!]$ for $i = 1, 2$ so $(e[E/\boldsymbol{x}], \lceil e \rceil \rho) \in [\![\tau]\!]$.

Case $e = c$. Towards showing that $(c, \lceil c \rceil) \in [\![o \to o]\!]$ assume $(v, w) \in [\![o]\!]$. We have $v = w$ and should prove $(c\ v, \lceil c \rceil(v)) \in [\![To]\!]$. If $F_c(v)$ is undefined then $\lceil c \rceil(v) = \emptyset$ so this is trivially true. If $F_c(v) = v'$ then $\lceil c \rceil(v) = \eta(v')$ and the result follows from Lemma 12.

Case $e = \texttt{rec}\ f\ x.e'$ and $\tau = \tau_1 \to \tau_2$. Again, e is a value and we only need to show that $(e[E/\boldsymbol{x}], \lceil e \rceil \rho) \in \lceil \tau \rceil$.

We have $e[E/\boldsymbol{x}] = \texttt{rec} f x.e'[E/\boldsymbol{x}]$ and will use Lemma 18 with $F(\varphi)(w) = [\![e]\!]\rho[f \mapsto \varphi, x \mapsto \varphi]$. Note that $\lceil e \rceil \rho = F^\dagger$. Assume $(v_x, w) \in [\![\tau_1]\!]$ and $(v_f, \varphi) \in [\![\tau_1 \to \tau_2]\!]$. Using the lemma

we only need to show $(e'[E/\boldsymbol{x}, v_f/f, v_x/x], F(\varphi)(w)) \in [\![T\tau_2]\!]$. But this is direct from the induction hypothesis applied to e'.

Case $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$. Here $e[E/\boldsymbol{x}] = \texttt{let}\ x = e_1[E/\boldsymbol{x}]\ \texttt{in}\ e_2[E/\boldsymbol{x}]$ so the result follows directly from Lemma 17 with $f(w) = [\![e_2]\!]\rho[x \mapsto w]$.

Case $e = v_1 := v_2$. Here the induction hypothesis gives $(v_1[E/\boldsymbol{x}], \lceil\!\lceil v_1 \rceil\!\rceil\rho) \in [\![o]\!]$ and $(v_2[E/\boldsymbol{x}], \lceil\!\lceil v_2 \rceil\!\rceil\rho) \in [\![o]\!]$ thus $v_1[E/\boldsymbol{x}] = \lceil\!\lceil v_1 \rceil\!\rceil\rho$ and $v_2[E/\boldsymbol{x}] = \lceil\!\lceil v_2 \rceil\!\rceil\rho$. If $(t, w) \in [\![e]\!]$ then $w = ()$ and $\lceil\!\lceil v_1 \rceil\!\rceil = X \in \mathbb{L}$ and t arises by mumbling and stuttering from $(\mathsf{h}, \mathsf{h}[X \mapsto [\![v_1]\!]\rho])$ for some h. Now $\mathsf{h}, e[E/\boldsymbol{x}] \longrightarrow \mathsf{h}[X \mapsto v_1[E/\boldsymbol{x}]], ()$, so $(t, w) \in Tr(e[E/\boldsymbol{x}])$ and $(w, w) \in [\![o]\!]$ as required.

Case $e = \texttt{atomic}(e')$. This is direct from Lemma 16.

The remaining cases are analogous.

We now come to the proof of the adequacy theorem.

Proof (of Theorem 1). Let $\vdash e : o$.

1. Suppose that $\mathsf{h}_{ini}, e \longrightarrow^* \mathsf{h}', v$. It is clear that $v \in \mathbb{VB}$, so We have $((\mathsf{h}', \mathsf{h}'), v) \in [\![v]\!] = \eta(\lceil\!\lceil v \rceil\!\rceil)$. By Lemma 10 and mumbling we have $((\mathsf{h}_{textitinit}, \mathsf{h}'), v) \in [\![e]\!]$.
2. Suppose that $((\mathsf{h}_{ini}, \mathsf{h}'), v) \in [\![e]\!]$. By Theorem 2 we have $(e, [\![e]\!]) \in [\![o]\!]$. Therefore, $((\mathsf{h}_{ini}, \mathsf{h}'), v) \in Tr(e)$ thus $\mathsf{h}_{ini}, e \longrightarrow^* \mathsf{h}', v$ as required.

To conclude this section we give another application of adequacy which is at the heart of the semantic justification of observational equivalence under refined typing assumption.

Suppose that we have a subset of "well typed" observations. This could be those that obey some refined typing discipline, for example, restricting or delineating the use of side-effects. We can then ask whether two closed terms of type τ are indistinguishable as far as observations from O are concerned. Adequacy entails that it does not matter whether we use operational or denotational semantics for this:

Theorem 3. *Let $O \subseteq \{v \mid \vdash v : \tau \to\}$ and $\vdash v_i : \tau$ for $i = 1, 2$. Suppose that for all $o \in O$ and $(t, w) \in [\![o\ v_1]\!]$ where t starts with h_{ini} we have $(t, w) \in [\![o\ v_2]\!]$.*

If $\mathsf{h}_{ini}, o\ v_1 \longrightarrow^* \mathsf{h}, b$ for some $o \in O$ and $b \in \{\texttt{true}, \texttt{false}\}$ then $\mathsf{h}_{ini}, o\ v_2 \longrightarrow^* \mathsf{h}, b$, too.

Proof. If $\mathsf{h}_{ini}, o\ v_1 \longrightarrow^* \mathsf{h}, b$ then by Theorem 1 (correctness) we have $((\mathsf{h}_{ini}, \mathsf{h}), b) \in [\![o\ v_1]\!]$ so, by assumption, $((\mathsf{h}_{ini}, \mathsf{h}), b) \in [\![o\ v_2]\!]$. Again by Theorem 1 (adequacy), we then get $\mathsf{h}_{ini}, o\ v_2 \longrightarrow^* \mathsf{h}, b$ as required.

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

4. N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations. In N. Kobayashi, editor, *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*, volume 4279 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2006.
5. S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.
6. M. Hofmann. Logical relations and nondeterminism. In *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*, pages 62–74, 2015.
7. N. A. Lynch and F. W. Vaandrager. Forward and backward simulations, ii: Timing-based systems. *Inf. Comput.*, pages 1–25, 1996.
8. R. D. Nicola and M. Hennessy. Testing equivalence for processes. In *ICALP*, pages 548–560, 1983.
9. A. M. Pitts. Relational properties of domains. *Inf. Comput.*, 127(2):66–90, 1996.
10. G. D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5(3):452–487, 1976.
11. A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 343–356. ACM, 2013.