The Essence of PARALLEL ALGOL

Stephen Brookes

April 1997
CMU-CS-97-124

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This is an expanded version of a paper that appeared in preliminary form in the Proceedings of the 11th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1996.

This work was partially supported by the Office of Naval Research, under grant number N00014-93-1-0750, and by the National Science Foundation, under grant number CCR 9412980.

Keywords: programming language design, concurrent programming, semantics of programming languages, procedures, communicating processes, recursion, communication

Abstract

We consider a parallel Algol-like language, combining procedures with shared-variable parallelism. Procedures permit encapsulation of common parallel programming idioms. Local variables provide a way to restrict interference between parallel commands. We provide a denotational semantics for this language, simultaneously adapting "possible worlds" [Rey81, Ole82] to the parallel setting and generalizing "transition traces" [Bro93] to the procedural setting. This semantics supports reasoning about safety and liveness properties of parallel programs, and validates a number of natural laws of program equivalence based on non-interference properties of local variables. The semantics also validates familiar laws of functional programming. We also provide a relationally parametric semantics, generalizing [Bro93] to permit reasoning about relation-preserving properties of programs, and adapting work of O'Hearn and Tennent [OT95] to the parallel setting. This semantics supports standard methods of reasoning about representational independence, adapted to shared-variable programs. The clean design of the programming language and its semantics supports the orthogonality of procedures and shared-variable parallelism.

1 Introduction

The programming language ALGOL 60 has had a major influence on the theory and practice of language design and implementation [OT97]. ALGOL shows how to combine imperative programming with an essentially functional procedure mechanism, without destroying the validity of laws of program equivalence familiar from functional programming. Moreover, procedures and local variables in ALGOL can be used to support an "object-oriented" style of programming: an abstract "object" can be represented by a collection of local variables together with procedures or "methods" used to read or write them. Although ALGOL itself is no longer widely used, an idealized form of the language has stimulated a great deal of innovative research [OT97]. Idealized Algol, as characterized by John Reynolds [Rey81], augments a simple sequential imperative language with a procedure mechanism based on the simply-typed call-by-name λ-calculus; procedure definitions, recursion, and the conditional construct are uniformly applicable to all phrase types. Reynolds identified these features as embodying the "essence" of Algol.

ALGOL 60 and Reynolds' Idealized Algol are, of course, sequential programming languages. Nevertheless the utility of procedures and local variables is certainly not limited to the sequential setting. Nowadays there is much interest in parallel programming, because of the potential for implementing efficient parallel algorithms by concurrent processes designed to cooperate in solving a common task. In this paper we focus on one of the most widely known paradigms of parallel programming, the so-called shared-variable model, in which parallel commands interact by reading and writing to shared memory. The use of procedures in such a language permits encapsulation of common parallel programming idioms. Local variable declarations provide a way to delimit the scope of interference: a local variable of one process is not shared by any other process, and is therefore unaffected by the actions of other process running concurrently.

For instance, a procedure implementing mutual exclusion [And91] with a binary semaphore could be written (in sugared form) as:

$$
\begin{array}{l}
\textbf{procedure } mutex(n_1, c_1, n_2, c_2);\\
\quad \textbf{boolean } s;\\
\quad \textbf{begin}\\
\qquad\quad s{:=}\textbf{true};\\
\qquad\quad \textbf{while true do}\\
\qquad\qquad (n_1;\ \textbf{await } s \textbf{ then } s{:=}\textbf{false};\\
\qquad\qquad\ \ c_1;\ s{:=}\textbf{true})\\
\qquad\ \ \| \ \textbf{while true do}\\
\qquad\qquad (n_2;\ \textbf{await } s \textbf{ then } s{:=}\textbf{false};\\
\qquad\qquad\ \ c_2;\ s{:=}\textbf{true})\\
\quad \textbf{end}
\end{array}
$$

Here c_1 and c_2 are parameters representing "critical" regions of code, and n_1 and n_2 represent non-critical code. The correctness of this procedure, i.e. the fact that the two critical regions are never concurrently active, relies on the inaccessibility of s to the procedure's arguments.

For another example, suppose two "worker" processes must each repeatedly execute a piece of code, can and should run concurrently, but need to stay in phase with each other, so that at each stage the two workers are executing the same iteration. If the parameters c_0 and c_1 represent the two workers' code, one way to achieve this execution pattern is represented by the following procedure:

$$\textbf{procedure } workers(c_0, c_1);\ \textbf{while true do } (c_0 \| c_1)$$

However, this program structure incurs the repeated overhead caused by creation and deletion of a pair of threads each time the loop body is executed. Although this defect has no effect on the overall correctness of the procedure, since it is obvious that the intended pattern of execution is achieved, for pragmatic reasons it might be preferable to design a program that creates two perpetually active threads, constrained to ensure that the threads stay in phase with each other. One way to achieve this, known as barrier synchronization [And91], uses a pair of local boolean variables:

$$
\begin{array}{l}
\textbf{procedure}\ barrier(c_0, c_1);\\
\quad \textbf{boolean}\ flag_0,\ flag_1;\\
\quad \textbf{procedure}\ synch(x, y);\ (x{:=}\textbf{true};\ \textbf{await}\ y;\ y{:=}\textbf{false});\\
\quad \textbf{begin}\\
\qquad flag_0{:=}\textbf{false};\ flag_1{:=}\textbf{false};\\
\qquad \textbf{while true do}\ (c_0;\\
\qquad\qquad\qquad synch(flag_0, flag_1))\\
\quad\ \ \|\ \textbf{while true do}\ (c_1;\\
\qquad\qquad\qquad synch(flag_1, flag_0))\\
\quad \textbf{end}
\end{array}
$$

The correctness of this implementation relies on locality of the flag variables. The two procedures *workers* and *barrier* are equivalent, in that for all possible arguments c_0 and c_1 the two procedure calls exhibit identical behaviors.

It is well known that parallel programs can be hard to reason about, because of the potential for undesirable interference between commands running in parallel. One might expect this problem to be exacerbated by the inclusion of procedures. Indeed, semantic accounts of shared-variable languages in the literature typically do not encompass procedures; the (usually implicit) attitude seems to be that concurrency is already difficult enough to handle by itself. Similarly, existing models for sequential Algol [Rey81, Ole82, OT95] do not handle parallelism, presumably because of the difficulty even in the sequential setting of modelling "local" state accurately [HMT83]. Nevertheless it seems intuitive that procedures and parallelism are "orthogonal" concepts, so that one ought to be able to design a programming language incorporating both seamlessly. This is the rationale behind our design of an idealized parallel Algol, blending a shared-variable parallel language with the λ-calculus while remaining faithful to Reynolds' ideals.

Even for sequential Algol the combination of procedures and local variables causes well known semantic problems for traditional, location-based store models. Such models typically fail to validate certain intuitive laws of program equivalence whose validity depends on "locality" properties of local variables [HMT83], such as the following law:

$$\textbf{new}[\textbf{int}]\ x\ \textbf{in}\ P \ = \ P,$$

when P is a free variable of type **comm** (representing a command). Intuitively, introducing a local variable x and never using it should have no effect, so that whatever the interpretation of P the two phrases should be indistinguishable; however, in a simple location-based semantics the presence of command meanings whose effect depends on the contents of specific locations will cause this equivalence to break. A more satisfactory semantics was proposed by Reynolds and Oles [Rey81, Ole82], based on a category of "possible worlds": a world W represents a set of "allowed states"; morphisms between worlds represent "expansions" corresponding to the declaration of new variables; types denote functors from the category of worlds to a category of domains and continuous functions; and well-typed phrases denote natural transformations between such functors. A command meaning at world W is a partial function from W to W. Naturality guarantees that a phrase behaves "uniformly" with respect to expansions between worlds, thereby enforcing locality constraints and validating laws such as the one discussed above.

The parallel setting requires a more sophisticated semantic structure because of the potential for interference between parallel commands. We adapt the "transition traces" semantics of [Bro93], modelling a command at world W as a set of finite and infinite traces, a subset of $(W \times W)^{\infty}$. The trace semantics given in [Bro93] covered a simple shared-variable parallel language, without procedures, with while-loops as the only means of recursion, assuming a single global set of states. This semantics was carefully designed to incorporate the assumption of *fairness* [Par79]. It is far from obvious that this kind of trace semantics can be generalized in a manner consistent with Reynolds' idealization, to include a general procedure mechanism, and a conditional construct and recursion at all types. Similarly, it is not evident that the possible worlds approach can be made to work for a parallel language. We show here that these approaches can indeed be combined. The resulting semantics brings out the stack discipline clearly yet models parallelism at an appropriate level of abstraction to permit compositional reasoning about safety and liveness properties of programs. Our categorical recasting of [Bro93] permits an improved treatment of local variables. The semantics for the λ-calculus fragment of the language is completely standard, based as usual on the cartesian closed structure of the underlying category. Thus our semantics supports the claim that procedures and parallelism are "orthogonal".

Since we are interested in proving liveness and safety properties of parallel programs it is vital to deal accurately with infinite traces. In particular, in our setting it is inappropriate to treat divergence as "catastrophic" or "undefined", and it is wrong to equate all forms of divergence, as is typically done in a conventional least-fixed-point semantics (where a single distinguished semantic value $\perp$ represents divergence). Instead, our treatment of recursion uses Tarski's theorem on greatest fixed points of monotone functions on complete lattices[Tar55]. Roughly speaking, a least-fixed-point semantics for our language would capture only the finite behaviors of programs, thus ignoring the potential for divergence; a greatest-fixed-point semantics captures both finite and infinite aspects of a program's behavior.

As we have remarked, our possible worlds semantics of Parallel Algol validates familiar laws of functional programming, as well as familiar laws of shared-variable programming, and equivalences based on locality properties. When applied to the examples listed earlier it produces the intended results; for instance, the *workers* and *barrier* procedures are indeed semantically equivalent. However, just as for the Reynolds-Oles possible worlds model of sequential Idealized Algol, certain laws of program equivalence involving the use of local variables and procedures to represent abstract data objects fail to hold, because of the presence in the model of certain insufficiently well behaved elements. These equivalences typically embody the principle of "representational independence" familiar from structured programming methodology: a program using an "object" (perhaps a member of some abstract data type) should behave the same way regardless of the object's implementation, provided its abstract properties are the same. Such equivalences are usually established by relational reasoning, typically involving some kind of invariant property that holds between the states of two programs that use alternative implementations. These problems led O'Hearn and Tennent to propose a "relationally parametric" semantics for sequential Idealized Algol [OT95], building on foundations laid in [Rey83]. In this semantics a type denotes a parametric functor from worlds to domains, and phrases denote parametric natural transformations between such functors. The parametricity constraints enforce the kind of relation-preserving properties needed to establish equivalences involving representation independence. We show how to construct a relationally parametric semantics for Parallel Algol, generalizing the O'Hearn-Tennent model to the parallel setting. We thus obtain a semantics that validates reasoning methods based on representation independence, as adapted to deal with shared-variable programs.

2 Syntax

2.1 Types and type environments

The type structure of our language is conventional [Rey81]: datatypes representing the set of integers and the set of booleans; phrase types built from expressions, variables, and commands, using product and arrow. We use τ as a meta-variable ranging over the set of datatypes, and θ to range over the set of phrase types, as specified by the following abstract grammar:

$$\begin{aligned}\theta &::= \mathbf{exp}[\tau] \mid \mathbf{var}[\tau] \mid \mathbf{comm} \mid (\theta \rightarrow \theta') \mid \theta \times \theta' \\ \tau &::= \mathbf{int} \mid \mathbf{bool}\end{aligned}$$

Let ι range over the set of identifiers. A type environment π is a partial function from identifiers to types. We write $\mathrm{dom}(\pi)$ for the domain of π, i.e. the set of identifiers for which π specifies a type. Let $(\pi \mid \iota : \theta)$ be the type environment that agrees with π except that it maps ι to θ.

2.2 Phrases and type judgements

A type judgement of form $\pi \vdash P : \theta$ is interpreted as saying that phrase P has type θ in type environment π. A judgement is valid iff it can be proven from the axioms and rules in Figure 1. We omit the rules dealing with phrases of type $\mathbf{var}[\tau]$ and $\mathbf{exp}[\tau]$, except to remark that the language contains the usual arithmetic and boolean operations. We let $\mathrm{FV}(P)$ denote the set of identifiers occurring free in P.

In addition, for convenience, we add the following rule; this allows us to elide the otherwise necessary projection for extracting the "R-value" of a variable:

$$\frac{\pi \vdash P : \mathbf{var}[\tau]}{\pi \vdash P : \mathbf{exp}[\tau]}$$

The syntax used here for phrases is essentially a simply typed λ-calculus with product types, combined with a shared-variable parallel language over ground type **comm**. Note that, in the spirit of Algol, the conditional construction **if** B **then** P_1 **else** P_2 and recursion **rec** $\iota.P$ are available at all phrase types θ. We restrict the use of a "conditional atomic action" **await** B **then** P to cases where P is "atomic", i.e. a finite sequence of assignments (or **skip**),

$$\pi \vdash \textbf{skip} : \textbf{comm}$$

$$\frac{\pi \vdash X : \textbf{var}[\tau] \quad \pi \vdash E : \textbf{exp}[\tau]}{\pi \vdash X{:=}E : \textbf{comm}}$$

$$\frac{\pi \vdash P_1 : \textbf{comm} \quad \pi \vdash P_2 : \textbf{comm}}{\pi \vdash P_1; P_2 : \textbf{comm}}$$

$$\frac{\pi \vdash P_1 : \textbf{comm} \quad \pi \vdash P_2 : \textbf{comm}}{\pi \vdash P_1 \| P_2 : \textbf{comm}}$$

$$\frac{\pi \vdash P : \textbf{exp}[\textbf{bool}] \quad \pi \vdash P_1 : \theta \quad \pi \vdash P_2 : \theta}{\pi \vdash \textbf{if } P \textbf{ then } P_1 \textbf{ else } P_2 : \theta}$$

$$\frac{\pi \vdash B : \textbf{exp}[\textbf{bool}] \quad \pi \vdash P : \textbf{comm}}{\pi \vdash \textbf{await } B \textbf{ then } P : \textbf{comm}} \quad (P \text{ atomic})$$

$$\frac{\pi \vdash B : \textbf{exp}[\textbf{bool}] \quad \pi \vdash P : \textbf{comm}}{\pi \vdash \textbf{while } B \textbf{ do } P : \textbf{comm}}$$

$$\frac{\pi, \iota : \textbf{var}[\tau] \vdash P : \textbf{comm}}{\pi \vdash \textbf{new}[\tau]\ \iota \textbf{ in } P : \textbf{comm}}$$

$$\pi \vdash \iota : \theta \quad (\text{when } \pi(\iota) = \theta)$$

$$\frac{\pi \vdash P : \theta_0 \times \theta_1}{\pi \vdash \text{fst } P : \theta_0}$$

$$\frac{\pi \vdash P_0 : \theta_0 \quad \pi \vdash P_1 : \theta_1}{\pi \vdash \langle P_0, P_1 \rangle : \theta_0 \times \theta_1}$$

$$\frac{\pi \vdash P : \theta_0 \times \theta_1}{\pi \vdash \text{snd } P : \theta_1}$$

$$\frac{\pi, \iota : \theta \vdash P : \theta}{\pi \vdash \textbf{rec } \iota.P : \theta}$$

$$\frac{\pi, \iota : \theta \vdash P : \theta'}{\pi \vdash \lambda\iota : \theta.P : (\theta \to \theta')}$$

$$\frac{\pi \vdash P : \theta \to \theta' \quad \pi \vdash Q : \theta}{\pi \vdash P(Q) : \theta'}$$

Figure 1: Type judgements

so that it is indeed feasible to implement this construct as an indivisible action. The special case **await** B **then skip** may be abbreviated by **await** B.

In displaying examples of programs it is often convenient to use a sugared form of syntax. For instance, we may write

$$\begin{array}{l}\textbf{integer}\ z;\\ \quad \textbf{begin}\ P\ \textbf{end}\end{array}$$

for **new**[**int**] z **in** P. Similarly we may write

$$\begin{array}{l}\textbf{procedure}\ f(x);\ P_0;\\ \quad \textbf{begin}\ P\ \textbf{end}\end{array}$$

instead of $(\lambda f.P)(\mathbf{rec}\ f.\lambda x.P_0)$. With this convention it is straightforward to de-sugar the examples discussed earlier into the formal syntax described here. When f does not occur free in P_0 the de-sugaring can go a little further: when the procedure is not recursive this notation corresponds to $(\lambda f.P)(\lambda x.P_0)$.

3 Possible worlds

The category $\mathbf{W}$ of possible worlds [Ole82] has as objects countable sets, called "worlds" or "store shapes", representing sets of allowed states. We let $V_{int} = \{\ldots, -1, 0, 1, \ldots\}$ and $V_{bool} = \{\mathtt{tt}, \mathtt{ff}\}$. Intuitively, the world V_τ consists of states representing a single storage cell capable of holding a value of data type τ. We will use V, W, X, and decorated versions such as W', as meta-variables ranging over $\mathbf{Ob}(\mathbf{W})$.

The morphisms from W to W' are pairs $h = (f, Q)$ where f is a function from W' to W and Q is an equivalence relation on W', such that the restriction of f to each equivalence class of Q is a bijection with W:

- $\forall x', y'.(x'Qy' \ \&\ fx' = fy' \ \Rightarrow\ x' = y')$;
- $\forall x \in W.\forall y' \in W'.\exists x'.(x'Qy' \ \&\ fx' = x)$.

Intuitively, when $(f, Q) : W \to W'$, we think of W' as a set of "large" states extending the "small" states of W with extra storage structure; f extracts the small state embedded inside a large state, and Q identifies two large states when they have the same extra structure. We will often find it convenient to blur the distinction between a relation Q on a set W' and its graph, i.e. the set $\{(x, y) \mid xQy\} \subseteq W' \times W'$.

The identity morphism on W is the pair $(\mathrm{id}_W, W \times W)$, where id_W is the identity function on the set W. For each pair of objects W and V there is an "expansion" morphism $- \times V : W \to W \times V$, given by

$$\begin{array}{l} - \times V = (\mathrm{fst} : W \times V \to W, Q), \text{ where} \\ Q = \{((w_0, v), (w_1, v)) \mid w_0, w_1 \in W \ \&\ v \in V\}. \end{array}$$

The composition of morphisms $h = (f, Q) : W \to W'$ and $h' = (g, R) : W' \to W''$, denoted $h; h' : W \to W''$, is the pair given by:

$$(f \circ g, \{(z_0, z_1) \in R \mid (gz_0, gz_1) \in Q\}).$$

As Oles has shown[Ole82], every morphism of worlds is an expansion composed with an isomorphism. Of particular relevance are structural isomorphisms reflecting the commutativity and associativity of cartesian product. For all worlds W, X, Y let

$$\begin{array}{l} swap_{W,X} : W \times X \to X \times W \\ assoc_{W,X,Y} : W \times (X \times Y) \to (W \times X) \times Y \end{array}$$

be the obvious natural isomorphisms. When equipped with the appropriate universal equivalence relation, so that there is a single equivalence class, these functions become isomorphisms in the category of worlds. For instance,

$$(swap_{W,X},\ (W \times X) \times (W \times X))$$

is an isomorphism from $X \times W$ to $W \times X$. Thus the nature of morphisms in this category captures the essence of local variable declarations in a clean and simple manner, and facilitates a "location-free" treatment of storage.

4 Semantics of types

Each type θ will be interpreted as a functor $[\![\theta]\!]$ from $\mathbf{W}$ to the category $\mathbf{D}$ of domains and continuous functions. As shown by Oles [Ole82], the category whose objects consist of such functors, with natural transformations as morphisms, is cartesian closed. We will use the categorical product and exponentiation in this ccc to interpret product types $\theta_0 \times \theta_1$ and arrow types $\theta_0 \to \theta_1$, respectively. The main differences between our parallel interpretation and the model developed by Oles and Reynolds concern the functorial treatment of the ground types **comm** and $\mathbf{exp}[\tau]$.

4.1 Commands

We interpret the type **comm** using "transition traces" [Bro93], but instead of assuming a single global state set we parameterize our definitions in terms of worlds. For each world W, $[\![\mathbf{comm}]\!]W$ will consist of sets of traces over W. A finite trace $(w_0, w'_0)(w_1, w'_1)\dots(w_n, w'_n)$ of a command represents a terminating computation from state w_0 to w'_n, during which the state was changed externally n times (by interference from another command running in parallel), the i^{th} interruption changing the state from w'_{i-1} to w_i. An infinite trace $\langle(w_n, w'_n)\rangle_{n=0}^{\infty}$ of a command represents an infinite execution, again assuming repeated interference.

When A is a set, we write A^* for the set of finite sequences over A, A^+ for the set of non-empty finite sequences over A, A^ω for the set of (countably) infinite sequences over A, and $A^\infty = A^+ \cup A^\omega$. Clearly, each of these operations extends to a functor (on **Set**), the morphism part being the appropriate "map" operation, which applies a function to each element of a sequence. Concatenation is extended to infinite traces in the usual way: $\alpha\beta = \alpha$ when α is infinite. The empty sequence, denoted ϵ, is a unit for concatenation. We extend concatenation, and finite and infinite iteration, to trace sets and to relations over traces, in the obvious componentwise manner; for instance, when $R, S \subseteq A^\infty \times A^\infty$, we let

$$R \cdot S = \{(\alpha_0\beta_0, \alpha_1\beta_1) \mid (\alpha_0, \alpha_1) \in R \ \&\ (\beta_0, \beta_1) \in S\}.$$

Using this notation, then, a command denotes a subset of $(W \times W)^\infty$. However, as in [Bro93], we let a step (w, w') in a trace represent a finite sequence of atomic actions, rather than a single atomic action. The trace set of a command is therefore closed under two natural operations: *stuttering* and *mumbling*[1]. Intuitively, stuttering involves the insertion of "idling" steps

[1]The use of closed sets of traces guarantees full abstraction for the simple shared-variable language [Bro93]. The closure conditions correspond, respectively, to reflexivity and transitivity of the $\to^*$ relation in a conventional operational semantics.

of the form (w, w) into a trace, while mumbling involves the collapsing of adjacent steps of the form $(w, w')(w', w'')$ into a single step (w, w''). We formalize this as follows.

We define relations $\mathrm{stut}_A, \mathrm{mum}_A \subseteq (A \times A)^+ \times (A \times A)^+$ by:

$$\begin{aligned}
\mathrm{stut}_A &= \{(\alpha\beta, \alpha(a,a)\beta) \mid a \in A \ \&\ \alpha\beta \in (A \times A)^+\} \\
\mathrm{mum}_A &= \{(\alpha(a,a')(a',a'')\beta, \alpha(a,a'')\beta) \mid \alpha\beta \in (A \times A)^* \ \&\ a, a', a'' \in A\}.
\end{aligned}$$

Let $\mathrm{idle}_A = \{(\alpha, \alpha) \mid \alpha \in (A \times A)^\infty\}$ denote the identity relation on $(A \times A)^\infty$. We then extend these relations to arbitrary traces, defining the relations $\mathrm{stut}_A^\infty, \mathrm{mum}_A^\infty \subseteq (A \times A)^\infty \times (A \times A)^\infty$ by [2]:

$$\begin{aligned}
\mathrm{stut}_A^\infty &= \mathrm{stut}_A^* \cdot \mathrm{idle}_A \cup \mathrm{stut}_A^\omega \\
\mathrm{mum}_A^\infty &= \mathrm{mum}_A^* \cdot \mathrm{idle}_A \cup \mathrm{mum}_A^\omega.
\end{aligned}$$

We say that a set T of traces over W is *closed* if

$$\begin{aligned}
\alpha \in T \ \&\ (\alpha, \beta) \in \mathrm{stut}_W^\infty &\Rightarrow \beta \in T; \\
\alpha \in T \ \&\ (\alpha, \beta) \in \mathrm{mum}_W^\infty &\Rightarrow \beta \in T.
\end{aligned}$$

We write $T^\dagger$ for the closure of T, that is, the smallest closed set of traces containing T as a subset.

Let $\wp^\dagger((W \times W)^\infty)$ denote the set of closed sets of traces over W, ordered by set inclusion. This forms a domain, in fact a complete lattice, with least element $\{\}$, greatest element the set of all traces, and lubs given by unions. For a morphism $h = (f, Q) : W \to W'$, $[\![\mathbf{comm}]\!]h$ should convert a set c of traces over W to the set of traces over W' that "project back" via f to a trace in c and respect the equivalence relation Q in each step. We therefore define

$$\begin{aligned}
[\![\mathbf{comm}]\!]W &= \wp^\dagger((W \times W)^\infty), \\
[\![\mathbf{comm}]\!](f, Q)c &= \{\alpha' \mid \mathrm{map}(f \times f)\alpha' \in c \ \&\ \mathrm{map}(Q)\alpha'\},
\end{aligned}$$

where $\mathrm{map}(Q)\alpha'$ indicates that each step in α' respects Q. It is straightforward to check that this is indeed a functor.

[2] Equivalently, these relations can be characterized as the greatest fixed points of the monotone functionals

$$\begin{aligned}
F(R) &= \mathrm{idle}_A \cup \mathrm{stut}_A \cdot R \\
G(R) &= \mathrm{idle}_A \cup \mathrm{mum}_A \cdot R,
\end{aligned}$$

which operate on the complete lattice of relations over traces, ordered by set inclusion.

Note that if c is a closed set of traces so is $[\![\mathbf{comm}]\!]hc$ as defined above. Moreover, the definition of $[\![\mathbf{comm}]\!]h$ is also applicable to a general trace set, and it is easy to see that for any set c of traces $[\![\mathbf{comm}]\!]h(c^\dagger) = ([\![\mathbf{comm}]\!]hc)^\dagger$, so that the action of $[\![\mathbf{comm}]\!]$ on morphisms interacts smoothly with closure. This observation is sometimes helpful in calculations.

The case when the morphism h is an expansion from W to $W \times V$ is worth particular attention; when c is a trace set over W, $[\![\mathbf{comm}]\!](- \times V)c$ is the trace set over $W \times V$ consisting of traces that look like a trace of c augmented with stuttering in the V-component:

$$\begin{aligned}
[\![\mathbf{comm}]\!](- \times V)c = \{&((w_0, v_0), (w'_0, v_0)) \dots ((w_n, v_n), (w'_n, v_n)) \mid \\
&\qquad (w_0, w'_0) \dots (w_n, w'_n) \in c \ \&\ \forall i \leq n.\ v_i \in V\} \\
\cup\ \{&((w_0, v_0), (w'_0, v_0)) \dots ((w_n, v_n), (w'_n, v_n)) \dots \mid \\
&\qquad (w_0, w'_0) \dots (w_n, w'_n) \dots \in c \ \&\ \forall i \geq 0.\ v_i \in V\}
\end{aligned}$$

This is as intended: here c represents the meaning of a command that uses part of the store represented by W, so when we expand the shape of the store the extra structure represented by the V-component should not be affected by the command's behavior, nor should it affect the command's behavior.

4.2 Expressions

Our treatment of expressions is similar, using a slightly simpler form of trace to reflect the assumption that expression evaluation does not cause side-effects, but with enough structure to permit a fine-grained semantics in which expression evaluation need not be atomic. We also allow for possible non-termination, and for the possibility that expression evaluation may be non-deterministic.

A finite trace of the form $(w_0 w_1 \dots w_n, v)$ represents an evaluation of an expression during which the state is changed as indicated, terminating with the result v. It suffices to allow such cases only when n is finite, since we assume fair interaction between an expression and its environment: it is impossible for the environment to interrupt infinitely often in a finite amount of time. On the other hand, if an expression evaluation fails to terminate the state may be changed arbitrarily many times and no result value is obtained; we represent such a case as an infinite trace in W^ω. Note in particular that the trace w^ω represents divergence when evaluated in state w without interference. Thus we will model the meaning of an expression of type τ, in world W, as a subset e of $W^+ \times V_\tau \cup W^\omega$; this subset will be closed under the obvious analogues of stuttering and mumbling [3]. Let $\wp^\dagger(W^+ \times V_\tau \cup W^\omega)$ denote the collection of closed sets of expression traces, ordered by inclusion. Accordingly, we define

$$\begin{aligned}
&[\![\mathbf{exp}[\tau]]\!]W = \wp^\dagger(W^+ \times V_\tau \ \cup W^\omega)\\
&[\![\mathbf{exp}[\tau]]\!](f,Q)e = \{(\rho', v) \mid (\mathrm{map} f \rho', v) \in e\} \cup \{\rho' \in W'^\omega \mid \mathrm{map} f \rho' \in e\}.
\end{aligned}$$

Again, functoriality is easy to check.

4.3 Product types

We interpret product types in the standard way, as products of the corresponding functors:

$$\begin{aligned}
[\![\theta \times \theta']\!]W &= \ [\![\theta]\!]W \times [\![\theta']\!]W\\
[\![\theta \times \theta']\!]h &= \ [\![\theta]\!]h \times [\![\theta']\!]h.
\end{aligned}$$

4.4 Arrow types

We interpret arrow types using functor exponentiation, as in [OT95]. The domain $[\![\theta \to \theta']\!]W$ consists of the families $p(-)$ of functions, indexed by morphisms from W, such that whenever $h : W \to W'$, $p(h) : [\![\theta]\!]W' \to [\![\theta']\!]W'$; and whenever $h' : W' \to W''$, $p(h) \mathbin{;} [\![\theta']\!]h' = [\![\theta]\!]h'; p(h \mathbin{;} h')$. This uniformity condition amounts to commutativity of the following diagram, for $h : W \to W'$ and $h' : W' \to W''$:

$$\begin{tikzcd}
{[\![\theta]\!]W'} \arrow[r, "p(h)"] \arrow[d, "{[\![\theta]\!]h'}"'] & {[\![\theta']\!]W'} \arrow[d, "{[\![\theta']\!]h'}"] \\
{[\![\theta]\!]W''} \arrow[r, "p(h \,;\, h')"'] & {[\![\theta']\!]W''}
\end{tikzcd}$$

The domain $[\![\theta \to \theta']\!]W$ is ordered by

$$p(-) \sqsubseteq q(-) \iff \forall W'.\forall h : W \to W'.p(h) \sqsubseteq q(h),$$

[3] For instance, for all $\rho, \sigma \in W^*$ and all $v \in V_\tau, w \in W$, $(\rho\sigma, v) \in e \Rightarrow (\rho w \sigma, v) \in e$, and $(\rho w w \sigma, v) \in e \Rightarrow (\rho w \sigma, v) \in e$. Similarly for infinite expression traces.

the obvious parametrized version of the pointwise ordering. It is easy to check that with this ordering $[\![\theta \to \theta']\!]W$ is indeed a domain, assuming that, for each W', $[\![\theta']\!]W'$ is a domain.

The morphism part of $[\![\theta \to \theta']\!]$ is defined by:

$$[\![\theta \to \theta']\!](h : W \to W')p = \lambda h' : W' \to W''.p(h\,;h').$$

This kind of λ-abstraction for denoting indexed families (here, elements of $[\![\theta \to \theta']\!]W'$) is a convenient notational abuse.

4.5 Variables

For variables we give an "object-oriented" semantics, in the style of Reynolds and Oles. A variable of type τ is a pair consisting of an "acceptor" (which accepts a value of type τ and returns a command) and an expression value. This is modelled by:

$$\begin{aligned}
[\![\mathbf{var}[\tau]]\!]W &= (V_\tau \to [\![\mathbf{comm}]\!]W) \times [\![\mathbf{exp}[\tau]]\!]W \\
[\![\mathbf{var}[\tau]]\!]h &= \lambda(a,e).(\lambda v.[\![\mathbf{comm}]\!]h(av), [\![\mathbf{exp}[\tau]]\!]he).
\end{aligned}$$

This formulation is exactly as in [Ole82], although the underlying interpretations of **comm** and $\mathbf{exp}[\tau]$ are different.

5 Semantics of phrases

A type environment π determines a functor $[\![\pi]\!]$ as an indexed product. A member u of $[\![\pi]\!]W$ is an *environment* mapping identifiers to values of the appropriate type: if $\pi(\iota) = \theta$ then $u\iota \in [\![\theta]\!]W$.

When $\pi \vdash P : \theta$ is a valid judgement, P denotes a natural transformation $[\![P]\!]$ from $[\![\pi]\!]$ to $[\![\theta]\!]$. That is, for all environments $u \in [\![\pi]\!]W$, whenever $h : W \to W'$, $[\![\theta]\!]h([\![P]\!]Wu) = [\![P]\!]W'([\![\pi]\!]hu)$. This is expressed by commutativity of the following diagram for all $h : W \to W'$:

$$\begin{tikzcd}
{[\![\pi]\!]W} \arrow[r, "{[\![P]\!]W}"] \arrow[d, "{[\![\pi]\!]h}"'] & {[\![\theta]\!]W} \arrow[d, "{[\![\theta]\!]h}"] \\
{[\![\pi]\!]W'} \arrow[r, "{[\![P]\!]W'}"'] & {[\![\theta]\!]W'}
\end{tikzcd}$$

We provide a denotational description of the semantics, beginning with the definitions for the simple shared-variable language constructs, adapting the definitions of [Bro93] to the functor category setting. In the following semantic clauses, assume that $\pi \vdash P : \theta$ and u ranges over $[\![\pi]\!]W$. In each case naturality is easy to verify, assuming naturality for the meanings of immediate subphrases.

5.1 Expressions

We omit the semantic clauses for expressions, except for two representative cases to illustrate the use of expression traces. The expression 1 always evaluates to the corresponding integer value, even if the state changes during evaluation:

$$[\![1]\!]Wu = \{(w_0 \dots w_n, 1) \mid n \geq 0 \ \&\ \forall i. w_i \in W\}$$

The following clause specifies that addition is sequential and evaluates its arguments from left to right:

$$\begin{aligned}
[\![E_1 + E_2]\!]Wu \ = \ & \\
& \{(\rho_1\rho_2, v_1 + v_2) \mid (\rho_1, v_1) \in [\![E_1]\!]Wu \ \&\ (\rho_2, v_2) \in [\![E_2]\!]Wu\}^\dagger \\
\cup\ & \{\rho_1\rho_2 \mid (\rho_1, v_1) \in [\![E_1]\!]Wu \ \&\ \rho_2 \in [\![E_2]\!]Wu \cap W^\omega\}^\dagger \\
\cup\ & \{\rho \in W^\omega \mid \rho \in [\![E_1]\!]Wu\}^\dagger
\end{aligned}$$

Other interpretations are also possible, including a parallel form of addition.

Let $\Delta_W : W \to W \times W$ denote the diagonal map: $\Delta_W(w) = (w, w)$. This may be used to coerce expression traces into command-like traces in cases (such as assignment, or conditional) where a command has a subphrase of expression type.

5.2 skip

skip has only finite traces consisting of stuttering steps:

$$\begin{aligned}
[\![\mathbf{skip}]\!]Wu \ &= \{(w, w) \mid w \in W\}^\dagger \\
&= \{(w_0, w_0)(w_1, w_1) \dots (w_n, w_n) \mid n \geq 0 \ \&\ \forall i. w_i \in W\}.
\end{aligned}$$

To show naturality of this definition, consider a morphism $(f, Q) : W \to W'$. We have

$$\begin{aligned}
[\![\mathbf{comm}]\!](f,Q)([\![\mathbf{skip}]\!]Wu) &= [\![\mathbf{comm}]\!](f,Q)\{(w_0,w_0)\ldots(w_n,w_n) \mid n \geq 0 \ \&\ \forall i.w_i \in W\} \\
&= \{(w'_0,w'_0)\ldots(w'_n,w'_n) \mid n \geq 0 \ \&\ \forall i.w'_i \in W'\} \\
&= [\![\mathbf{skip}]\!]W'([\![\pi]\!](f,Q)u)
\end{aligned}$$

because f puts each Q-class in bijection with W, so that for each w_i there is a w'_i such that $f(w'_i) = w_i$, and such a state w'_i is the unique member of its Q-class with this property.

5.3 Assignment

We specify a non-atomic interpretation for assignment, in which the source expression is evaluated first:

$$\begin{aligned}
&[\![X{:=}E]\!]Wu = \\
&\qquad \{(\mathrm{map}\Delta_W\rho)\beta \mid (\rho,v) \in [\![E]\!]Wu \ \&\ \beta \in \mathrm{fst}([\![X]\!]Wu)v\}^\dagger \\
&\quad \cup \{\mathrm{map}\Delta_W\rho \mid \rho \in [\![E]\!]Wu \cap W^\omega\}^\dagger.
\end{aligned}$$

Note the use of $\mathrm{map}\Delta_W$ to convert expression traces into command-like traces.

For instance, the assignment $x{:=}x+1$, interpreted at world $W \times V_{int}$ in an environment u in which x corresponds to the V_{int} component of state, has the following traces:

$$[\![x{:=}x{+}1]\!](W{\times}V_{int})u = \{((w_0,v_0),(w_0,v_0))((w_1,v_1),(w_1,v_0{+}1)) \mid w_0,w_1 \in W \ \&\ v_0,v_1 \in V_{int}\}^\dagger,$$

showing the potential for interruption after evaluation of the source expression $x+1$ but before the update to the target variable. Closure, in this case, implies that the command also has traces of the form $((w,v),(w,v+1))$, representing execution without interruption. In addition, closure permits the insertion of finitely many stuttering steps.

5.4 Sequential composition

Sequential composition corresponds to concatenation of traces:

$$[\![P_1;P_2]\!]Wu = \{\alpha_1\alpha_2 \mid \alpha_1 \in [\![P_1]\!]Wu \ \&\ \alpha_2 \in [\![P_2]\!]Wu\}^\dagger.$$

It is convenient to introduce a semantic sequencing construct: for arbitrary trace sets T_1 and T_2 we define $T_1;T_2 = (T_1 \cdot T_2)^\dagger$. Thus $[\![P_1;P_2]\!]Wu = [\![P_1]\!]Wu;[\![P_2]\!]Wu$.

Naturality of this definition follows because for all trace sets T_1 and T_2 over W and all morphisms $h : W \to W'$ we have $[\![\mathbf{comm}]\!]h(T_1;T_2) = ([\![\mathbf{comm}]\!]hT_1);([\![\mathbf{comm}]\!]hT_2)$.

5.5 Parallel composition

Parallel composition of commands corresponds to fair interleaving of traces. For each set A we define the following sets:

$$\begin{array}{l} both_A = \{(\alpha,\beta,\alpha\beta),(\alpha,\beta,\beta\alpha) \mid \alpha,\beta \in A^+\} \\ one_A = \{(\alpha,\epsilon,\alpha),(\epsilon,\alpha,\alpha) \mid \alpha \in A^\infty\} \\ fairmerge_A = both_A^* \cdot one_A \cup both_A^\omega, \end{array}$$

where ϵ represents the empty sequence and we use the obvious extension of the concatenation operation on traces to sets of triples of traces:

$$t_0 \cdot t_1 = \{(\alpha_0\alpha_1, \beta_0\beta_1, \gamma_0\gamma_1) \mid (\alpha_0,\beta_0,\gamma_0) \in t_0 \ \&\ (\alpha_1,\beta_1,\gamma_1) \in t_1\}.$$

Similarly we use the obvious extensions of the Kleene iteration operators on traces. Thus, for instance, $both_A^*$ is the set of all triples obtained by concatenating together a finite sequence of triples from $both_A$.[4]

Intuitively, $fairmerge_{W\times W}$ is the set of triples (α,β,γ) of traces over W such that γ is a fair merge of α and β. Note that *fairmerge* satisfies the following "natural" property: for all functions $f : A \to B$,

$$(\alpha,\beta,\gamma) \in fairmerge_A \ \Rightarrow\ (\mathrm{map} f\alpha, \mathrm{map} f\beta, \mathrm{map} f\gamma) \in fairmerge_B.$$

We then define

$$[\![P_1 \| P_2]\!]Wu = \{\alpha \mid \exists(\alpha_1,\alpha_2,\alpha) \in fairmerge_{W\times W}.\ \alpha_1 \in [\![P_1]\!]Wu \ \&\ \alpha_2 \in [\![P_2]\!]Wu\}^\dagger.$$

Again it will be convenient to introduce a semantic parallel composition operator: for trace sets T_1 and T_2 over W let $T_1 \| T_2 = \{\alpha \mid \exists(\alpha_1,\alpha_2,\alpha) \in$

[4]Equivalently, $fairmerge_A$ can be characterized as the greatest fixed point of the monotone function $F(t) = both_A \cdot t \cup one_A$ on the complete lattice $\wp(A^\infty \times A^\infty \times A^\infty)$. The least fixed point of this functional is the subset of triples (α,β,γ) from $fairmerge_A$ in which one or both of α and β is finite.

$\mathit{fairmerge}_{W\times W}$. $\alpha_1 \in T_1$ & $\alpha_2 \in T_2\}^\dagger$. Naturality of $[\![P_1\|P_2]\!]$ follows from naturality of $[\![P_1]\!]$ and $[\![P_2]\!]$, since

$$[\![\mathbf{comm}]\!]h(T_1\|T_2) = ([\![\mathbf{comm}]\!]hT_1) \parallel ([\![\mathbf{comm}]\!]hT_2),$$

for all trace sets T_1, T_2 over W and all morphisms $h : W \to W'$.

5.6 Local variables

A trace of $\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ P$ at world W should represent an execution of P in the expanded world $W \times V_\tau$, with ι bound to a fresh variable of type τ; during this execution, P may change this variable's value but no other command has access to it. Only the changes to the W-component of the world should be reflected in the overall trace. We say that a trace $(w_n, w'_n)_{n=0}^{\infty}$ is interference-free iff for each n, $w'_n = w_{n+1}$. Thus the traces of $\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ P$ in world W and environment u should have the form $\mathrm{map}(\mathrm{fst}\times\mathrm{fst})\alpha$, where α is a trace of P in world $W \times V_\tau$ (and suitably adjusted environment) such that $\mathrm{map}(\mathrm{snd}\times\mathrm{snd})\alpha$ is interference-free:

$$\begin{aligned}[\![\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ P]\!]Wu = \{&\mathrm{map}(\mathrm{fst}\times\mathrm{fst})\alpha \mid \\ &\alpha \in [\![P]\!](W \times V_\tau)([\![\pi]\!](-\times V_\tau)u \mid \iota : (a,e))\ \&\\ &\mathrm{map}(\mathrm{snd}\times\mathrm{snd})\alpha \text{ interference-free}\}\end{aligned}$$

where the "fresh variable" $(a,e) \in [\![\mathbf{var}[\tau]]\!](W \times V_\tau)$ is defined by:

$$\begin{aligned}a &= \lambda v'{:}V_\tau.\{((w,v),(w,v')) \mid w \in W\ \&\ v \in V_\tau\}^\dagger\\ e &= \{(\rho\langle w,v\rangle\rho', v) \mid \rho\rho' \in (W\times V_\tau)^*\ \&\ w \in W\ \&\ v \in V_\tau\}^\dagger.\end{aligned}$$

5.7 Conditional

For conditional phrases we define by induction on θ, for $t \in [\![\mathbf{exp}[\mathbf{bool}]]\!]W$ and $p_1, p_2 \in [\![\theta]\!]W$, an element $\mathbf{if}\ t\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2$ of $[\![\theta]\!]W$.

- For $\theta = \mathbf{comm}$, $\mathbf{if}\ t\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2$ is

$$\begin{aligned}&\{(\mathrm{map}\Delta_W\rho)\alpha_1 \mid (\rho,\mathtt{tt}) \in t\ \&\ \alpha_1 \in p_1\}^\dagger \cup\\ &\{(\mathrm{map}\Delta_W\rho)\alpha_2 \mid (\rho,\mathtt{ff}) \in t\ \&\ \alpha_2 \in p_2\}^\dagger \cup\\ &\{\mathrm{map}\Delta_W\rho \mid \rho \in t \cap W^\omega\}.\end{aligned}$$

- For $\theta = (\theta_0 \to \theta_1)$, $(\mathbf{if}\ t\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2)(-)$ is the indexed family given by

$$\begin{aligned}&(\mathbf{if}\ t\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2)(h) = \\ &\qquad \lambda p.\mathbf{if}\ [\![\mathbf{exp}[\mathbf{bool}]]\!]ht\ \mathbf{then}\ p_1(h)p\ \mathbf{else}\ p_2(h)p.\end{aligned}$$

- For $\theta = \mathbf{var}[\tau]$ we define

$$\begin{aligned}&\mathbf{if}\ t\ \mathbf{then}\ (a_1, e_1)\ \mathbf{else}\ (a_2, e_2)\ = \\ &\quad (\lambda v{:}V_\tau.\mathbf{if}\ t\ \mathbf{then}\ a_1 v\ \mathbf{else}\ a_2 v,\ \mathbf{if}\ t\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2).\end{aligned}$$

We then define

$$\begin{aligned}&[\![\mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2]\!]Wu = \\ &\qquad\qquad \mathbf{if}\ [\![B]\!]Wu\ \mathbf{then}\ [\![P_1]\!]Wu\ \mathbf{else}\ [\![P_2]\!]Wu.\end{aligned}$$

Naturality is easy to check, by induction on the type.

5.8 Conditional atomic action

We give a "busy wait" interpretation to an await command: if the test expression B evaluates to `tt` it executes the body P without allowing interference; if the test evaluates to `ff` it waits and tries again; if evaluation of the test diverges so does the await command.

$$\begin{aligned}&[\![\mathbf{await}\ B\ \mathbf{then}\ P]\!]Wu = \\ &\qquad \{(w, w') \in [\![P]\!]Wu \mid (w, \mathtt{tt}) \in [\![B]\!]Wu\}^\dagger \\ &\qquad \cup \{(w, w) \mid (w, \mathtt{ff}) \in [\![B]\!]Wu\}^\omega \\ &\qquad \cup \{\mathrm{map}\Delta_W \rho \mid \rho \in [\![B]\!]Wu \cap W^\omega\}^\dagger.\end{aligned}$$

Recall that P is assumed to be a finite sequence of assignments or **skip**, so that $[\![P]\!]Wu$ is a set of finite traces. The singleton traces $(w, w') \in [\![P]\!]Wu$ thus represent "atomic" executions of P, during which no external state changes are permitted. If the test expression B always terminates, as is common, the third part of the clause becomes vacuously empty.

5.9 while-loops

The traces of **while** B **do** C are obtained by iteration. Define

$$\begin{aligned}[\![B]\!]_{\mathtt{tt}}Wu &= \{\mathrm{map}\Delta_W\rho \mid (\rho, \mathtt{tt}) \in [\![B]\!]Wu\} \\ &\quad \cup\{\mathrm{map}\Delta_W\rho \mid \rho \in [\![B]\!]Wu \cap W^\omega\} \\ [\![B]\!]_{\mathtt{ff}}Wu &= \{\mathrm{map}\Delta_W\rho \mid (\rho, \mathtt{ff}) \in [\![B]\!]Wu\} \\ &\quad \cup\{\mathrm{map}\Delta_W\rho \mid \rho \in [\![B]\!]Wu \cap W^\omega\}\end{aligned}$$

Then we define

$$\begin{aligned}&[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!]Wu = \\ &\quad ([\![B]\!]_{\mathtt{tt}}Wu; [\![C]\!]Wu)^*; [\![B]\!]_{\mathtt{ff}}Wu\ \cup\ ([\![B]\!]_{\mathtt{tt}}Wu; [\![C]\!]Wu)^\omega\end{aligned}$$

This definition can also be characterized as the closure of the greatest fixed point of the functional

$$F(t) = [\![B]\!]_{\mathtt{tt}}Wu \cdot [\![C]\!]Wu \cdot t\ \cup\ [\![B]\!]_{\mathtt{ff}}Wu,$$

which operates on the complete lattice of arbitrary trace sets over W, ordered by set inclusion. The reason for taking the closure only *after* constructing the fixed point, rather than taking the fixed point of the closure-preserving version of the functional (which uses ; rather than $\cdot$), is shown by the special case of the loop **while true do skip**. A similar issue will arise later in a more general context, in our treatment of recursion. We include the semantics of while-loops here explicitly, even though it will turn out to be a familiar special case of the use of recursion, because of the simplicity of the definition and the obvious connection with operational intuition. Notice also that taking the *least* fixed point of the above functional would yield only the *finite* traces of the loop, ignoring any potential for infinite iteration.

5.10 λ-calculus

The semantic clauses for identifiers, abstraction, and application are standard:

$$\begin{aligned}&[\![\iota]\!]Wu = u\iota \\ &[\![\lambda\iota : \theta.P]\!]Wuh = \lambda a : [\![\theta]\!]W'.[\![P]\!]W'([\![\pi]\!]hu \mid \iota : a) \\ &[\![P(Q)]\!]Wu = [\![P]\!]Wu(\mathrm{id}_W)([\![Q]\!]Wu),\end{aligned}$$

where, in the clause for abstraction, h ranges over morphisms from W to W'. The clauses for pairing and projections are also standard, using the cartesian structure of the functor category:

$$\begin{array}{l}
[\![\langle P_0, P_1\rangle]\!]Wu = ([\![P_0]\!]Wu, [\![P_1]\!]Wu) \\
[\![\text{fst } P]\!]Wu = \text{fst}([\![P]\!]Wu) \\
[\![\text{snd } P]\!]Wu = \text{snd}([\![P]\!]Wu).
\end{array}$$

5.11 Recursion

It is possible to give a least-fixed-point interpretation for recursion, as noted above for while-loops, but this would only account for finite traces and would therefore preclude reasoning about safety and liveness properties of programs. Instead we make use of greatest fixed points to obtain a model containing both finite and infinite traces.

We know from Tarski's theorem [Tar55] that every monotone function on a complete lattice has a greatest fixed point. This might suggest that we begin by establishing that each domain $[\![\theta]\!]W$ is a complete lattice. Unfortunately this is not generally true. Although $[\![\mathbf{comm}]\!]W$ is a complete lattice for each world W, with top element the set of all traces over W, the functions $[\![\mathbf{comm}]\!]h$ do not generally preserve top. For instance, when $h = (f, Q) : W \to W'$ is a non-trivial expansion morphism, so that Q has more than one equivalence class,

$$\begin{array}{rcl}
[\![\mathbf{comm}]\!]h(\text{top}_{[\![\mathbf{comm}]\!]W}) & = & [\![\mathbf{comm}]\!]h(\wp((W \times W)^\infty)) \\
& = & \{\alpha' \in [\![\mathbf{comm}]\!]W' \mid \text{map}(Q)\alpha'\} \\
& \neq & \text{top}_{[\![\mathbf{comm}]\!]W'}.
\end{array}$$

As a consequence, $[\![\mathbf{comm} \to \mathbf{comm}]\!]W$ is *not* a complete lattice, because it does not possess a top. We can see this as follows. The obvious order-theoretic candidate for top of $[\![\mathbf{comm} \to \mathbf{comm}]\!]W$, i.e. the family $\text{top}(-)$ such that for all $h : W \to W'$,

$$\text{top}(h) = \lambda d' : [\![\mathbf{comm}]\!]W'.\text{top}_{[\![\mathbf{comm}]\!]W'},$$

lacks the naturality property required for membership in $[\![\mathbf{comm} \to \mathbf{comm}]\!]W$, as was just shown above. Furthermore, the obvious natural candidate for tophood, i.e. the family $\mathit{top}(-)$ given by

$$\mathit{top}(h) = \lambda d' : [\![\mathbf{comm}]\!]W'.[\![\mathbf{comm}]\!]h(\text{top}_{[\![\mathbf{comm}]\!]W}),$$

is not the order-theoretic top, since it does not dominate the identity family $\mathrm{id}(h) = \lambda d' : [\![\mathbf{comm}]\!]W'.d'$.

The resolution of this dilemma is suggested by the operational behavior of the command $\mathbf{rec}\ \iota.\iota$; this command simply diverges, without ever changing the state, no matter how its environment tries to interfere. Its trace set should therefore consist of the infinite stuttering sequences. This trace set is not the greatest fixed point of the *identity function* on $[\![\mathbf{comm}]\!]W$, as might be suggested by the syntactic form of the command. Instead it can be characterized as (the closure of) the greatest fixed point of the monotone functional $\lambda c.\{(w, w)\alpha \mid w \in W \ \&\ \alpha \in c\}$, operating on the complete lattice $\wp((W \times W)^\infty)$ of *arbitrary* trace sets; intuitively, the extra initial stutter mimics an operational step in which the recursion is unwound. It is easy to prove that the greatest fixed point of this functional does indeed consist of the infinite stuttering sequences. Clearly this trace set is also closed under stuttering and mumbling, so belongs to the sublattice $\wp^\dagger((W \times W)^\infty)$. Moreover, when $h : W \to W'$ we have

$$\begin{aligned}
&[\![\mathbf{comm}]\!]h([\![\mathbf{rec}\ \iota.\iota]\!]Wu) \\
&\quad = [\![\mathbf{comm}]\!]h(\{(w, w) \mid w \in W\}^\omega) \\
&\quad = \{(w', w') \mid w' \in W'\}^\omega \\
&\quad = [\![\mathbf{rec}\ \iota.\iota]\!]W'([\![\pi]\!]hu),
\end{aligned}$$

so that $[\![\mathbf{rec}\ \iota.\iota]\!]$ is indeed a natural transformation.

A similar argument can be given for a recursive phrase $\mathbf{rec}\ \iota.P$ at general type θ. The key to a general definition of $[\![\mathbf{rec}\ \iota.P]\!]W$ is to embed each $[\![\theta]\!]W$ in a suitable lattice $[\theta]W$, and generalize the insertion of an initial stutter, and the notion of closure, to all phrase types. For each type θ we define a functor $[\theta]$ from the category of worlds to the category of complete lattices and monotone functions; in essence, $[\theta]$ is like $[\![\theta]\!]$ as defined before, except that we relax the naturality requirements at arrow types and the closure requirements at ground types. For each type θ we define a natural transformation stut_θ from $[\theta]$ to $[\theta]$; at ground types this inserts a stuttering step at the beginning of all traces in a trace set, and at arrow types it produces a procedure meaning that induces an extra initial stuttering step at result type, whenever the proecedure is called. We then define a natural transformation clos_θ from $[\theta]$ to $[\![\theta]\!]$ that restores closure. The semantic definitions given earlier, modified to omit the use of closure, serve to define a semantic function $[P]$ such that, when $\pi \vdash P : \theta$ and $u \in [\pi]W$, we have $[P]Wu \in [\theta]W$.

In particular, when $\pi, \iota : \theta \vdash P : \theta$ and $u \in [\![\pi]\!]W$, the function $F(p) = \mathrm{stut}_\theta W([P]W(u \mid \iota : p)$ is a monotone map on the complete lattice $[\theta]W$. Its greatest fixed point, which we denote by $\nu p.F(p)$, is in $[\theta]W$, and the closure of this fixed point is in $[\![\theta]\!]W$. We therefore take

$$[\![\mathbf{rec}\ \iota.P]\!]Wu = \mathrm{clos}_\theta W(\nu p.\mathrm{stut}_\theta W([P]W(u \mid \iota : p)).$$

This definition is natural, in that $[\![\theta]\!]h([\![\mathbf{rec}\ \iota.P]\!]Wu) = [\![\mathbf{rec}\ \iota.P]\!]W'([\![\pi]\!]hu)$. Stuttering plays a crucial role in the proof of this result. Indeed, in the absence of $\mathrm{stut}_\theta W$ naturality would fail, as seen when P is ι. The Appendix contains further details.

Note that this semantic definition does indeed provide the command **rec** $\iota.\iota$ with the desired denotation, i.e.

$$[\![\mathbf{rec}\ \iota.\iota : \mathbf{comm}]\!]W = \{(w, w) \mid w \in W\}^\omega.$$

Moreover, the analogous recursion at procedure type $\mathbf{comm} \to \mathbf{comm}$ denotes the family

$$\lambda h : W \to W'.\lambda a : [\![\mathbf{comm}]\!]W'.\{(w', w') \mid w' \in W'\}^\omega,$$

corresponding to a procedure that causes divergence whenever called. Again this conforms with our operational expectations.

It is also easy to verify that the meaning given to

$$\mathbf{rec}\ c.\mathbf{if}\ B\ \mathbf{then}\ C; c\ \mathbf{else\ skip}$$

coincides with the semantics given earlier for the loop **while** B **do** C, provided $c \notin \mathrm{FV}(C)$.

6 Reasoning about program behavior

The semantics validates a number of natural laws of program equivalence, including (when ι does not occur free in P'):

$$\begin{array}{l}
\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ P' = P' \\
\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ (P \| P') = (\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ P) \| P' \\
\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ (P; P') = (\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ P); P'.
\end{array}$$

Similarly the semantics validates laws such as the following, which show that the order in which local variables are declared is irrelevant:

$$\begin{array}{l}\mathbf{new}[\tau_1]\ \iota_1\ \mathbf{in}\ \mathbf{new}[\tau_2]\ \iota_2\ \mathbf{in}\ P = \mathbf{new}[\tau_2]\ \iota_2\ \mathbf{in}\ \mathbf{new}[\tau_1]\ \iota_1\ \mathbf{in}\ P \\ \mathbf{new}[\tau_1]\ \iota_1\ \mathbf{in}\ \mathbf{new}[\tau_2]\ \iota_2\ \mathbf{in}\ P(\iota_1, \iota_2) = \mathbf{new}[\tau_1]\ \iota_1\ \mathbf{in}\ \mathbf{new}[\tau_2]\ \iota_2\ \mathbf{in}\ P(\iota_2, \iota_1)\end{array}$$

These laws amount to naturality (for a phrase P of the appropriate type) with respect to the natural isomorphism of worlds $(W \times V_{\tau_1}) \times V_{\tau_2}$ and $(W \times V_{\tau_2}) \times V_{\tau_1}$, this isomorphism being a composition of suitably chosen *swap* and *assoc* morphisms as discussed earlier.

The semantics also validates familiar laws of functional programming, such as β-equivalence and the usual recursion law:

$$\begin{array}{l}(\lambda\iota : \theta.P)(Q) = P[Q/\iota] \\ \mathbf{rec}\ \iota.P = P[\mathbf{rec}\ \iota.P/\iota],\end{array}$$

where $P[Q/\iota]$ is the phrase obtained by replacing every free occurrence of ι in P by Q, with renaming when necessary to avoid capture.

Similarly the model validates laws relating the conditional construct with functional abstraction and application:

$$\begin{array}{l}(\mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2)(Q)\ =\ \mathbf{if}\ B\ \mathbf{then}\ P_1(Q)\ \mathbf{else}\ P_2(Q) \\ \lambda\iota : \theta.\mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 = \mathbf{if}\ B\ \mathbf{then}\ \lambda\iota : \theta.P_1\ \mathbf{else}\ \lambda\iota : \theta.P_2 \quad \text{if } \iota \notin \mathrm{FV}(B),\end{array}$$

and the semantics validates laws familiar from imperative programming, such as

$$\begin{array}{l}(\mathbf{if}\ B\ \mathbf{then}\ X_1\ \mathbf{else}\ X_2){:=}E\ =\ \mathbf{if}\ B\ \mathbf{then}\ X_1{:=}E\ \mathbf{else}\ X_2{:=}E \\ \mathbf{while}\ B\ \mathbf{do}\ C\ =\ \mathbf{if}\ B\ \mathbf{then}\ C; \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{else}\ \mathbf{skip} \\ \mathbf{skip}\|C\ =\ C\|\mathbf{skip}\ =\ C \\ \mathbf{skip}; C\ =\ C; \mathbf{skip}\ =\ C\end{array}$$

Our semantics also equates **while true do skip** and **await false then skip**, because of our busy-wait interpretation of conditional atomic actions.

The semantics supports compositional reasoning about safety and liveness properties. For instance, it is possible to show the correctness of the mutual exclusion procedure discussed earlier, and to show the equivalence of the *workers* and *barrier* procedures.

For a more complex example involving parallelism, consider the following implementation of a synchronization "object", exploiting two local boolean variables and a pair of procedures which can be invoked to set up synchronization:

$$
\begin{array}{l}
\textbf{boolean}\ flag_0,\ flag_1; \\
\textbf{procedure}\ synch(x, y);\ (x{:=}\textbf{true};\ \textbf{await}\ y;\ y{:=}\textbf{false}); \\
\quad flag_0{:=}\textbf{false};\ flag_1{:=}\textbf{false}; \\
\quad P(synch(flag_0, flag_1),\ synch(flag_1, flag_0))
\end{array}
$$

Here P is a free identifier of type $(\mathbf{comm} \times \mathbf{comm} \to \mathbf{comm})$. Since P is a non-local identifier, the only way for this phrase to access the flag variables is by one of the two pre-packaged ways to call *synch*. Intuitively, the behavior of this phrase should remain identical if we use a "dualized" implementation of the flags, interchanging the roles of the two truth values. Thus, this phrase should be equivalent to

$$
\begin{array}{l}
\textbf{boolean}\ flag_0,\ flag_1; \\
\textbf{procedure}\ synch(x, y);\ (x{:=}\textbf{false};\ \textbf{await}\ \neg y;\ y{:=}\textbf{true}); \\
\quad flag_0{:=}\textbf{true};\ flag_1{:=}\textbf{true}; \\
\quad P(synch(flag_0, flag_1),\ synch(flag_1, flag_0))
\end{array}
$$

This is an example of the principle of representation independence. Our semantics for Parallel Algol validates this equivalence, by virtue of the existence of a suitable isomorphism of worlds that relates the two implementations. To be specific, for all worlds W there is an isomorphism $dual : W \times V_{bool} \to W \times V_{bool}$ involving the function $\lambda(w, b).(w, \neg b)$,equipped with the universal equivalence relation on $W \times V_{bool}$. Naturality of the meaning of P with respect to this morphism is enough to establish the desired equivalence. Note that this is an equivalence between two terms containing a free identifier. In essence, no matter how the "rest" of the program is filled in, provided it is only allowed access to the two flags by calling one of the supplied procedures, the two implementations are indistinguishable. For example, if we substitute for P the procedure

$$\lambda(left,\ right).\ (\textbf{while true do}\ (c_0;\ left) \parallel \textbf{while true do}\ (c_1;\ right))$$

we recover the barrier synchronization example discussed earlier.

Although the above semantics validates many laws of program equivalence related to locality in parallel programming, there remain equivalences for which we can give convincing informal justification, yet which are not valid in this model. Consider for example the following phrase:

$$\mathbf{new}[\mathbf{int}]\ x\ \mathbf{in}\ (x{:=}0;\ P(x{:=}x+1)),$$

where P is a free identifier of type $\mathbf{comm} \to \mathbf{comm}$. No matter how P is instantiated this should have the same effect as $P(\mathbf{skip})$. As observed by O'Hearn and Tennent, this equivalence holds for the sequential language yet is not validated by the sequential possible worlds semantics. Indeed, the equivalence should still hold in the parallel setting, because the two phrases obviously treat the non-local part of the state the same way. This argument may be formalized by establishing an invariant relationship between the states arising during executions of the two phrases; however, the preservation of this invariant does not follow immediately from naturality of $[\![P]\!]$.

Similarly, and exactly as in the Reynolds–Oles semantics of Idealized Algol, our semantics typically fails to support proofs of representation independence involving *non-isomorphic* representations. This is illustrated by the following example, adapted from [OT95]. Consider an abstract "switch" object, initially "off", with two capabilities which can be thought of as a method for turning the switch "on" and a test to see if the switch has been turned on. One implementation uses a boolean variable:

$$\begin{array}{l}
\mathbf{boolean}\ z; \\
\mathbf{procedure}\ \mathit{flick};\ (z{:=}\mathbf{true}); \\
\mathbf{procedure}\ \mathit{on};\ \mathbf{return}\ z; \\
\quad z{:=}\mathbf{false}; \\
\quad P(\mathit{flick},\ \mathit{on})
\end{array}$$

Another implementation uses an integer variable, and treats all positive integers as "on", zero as "off":

$$\begin{array}{l}
\mathbf{integer}\ z; \\
\mathbf{procedure}\ \mathit{flick};\ (z{:=}z+1); \\
\mathbf{procedure}\ \mathit{on};\ \mathbf{return}\ (z>0); \\
\quad z{:=}0; \\
\quad P(\mathit{flick},\ \mathit{on})
\end{array}$$

Intuitively, even if P is allowed to use parallelism, and even though assignment is not assumed to be atomic, these two phrases will always be equivalent. Yet the possible worlds semantics fails to validate this equivalence.

Informally an argument supporting the equivalence can be given, by establishing an invariant relation between the states produced during execution of the two phrases. The problem is that naturality is not a sufficiently stringent requirement on phrase denotations, since it does not imply the kind of relation-preserving properties necessary to justify equivalences such as this.

7 Relational parametricity

In response to this inadequacy O'Hearn and Tennent [OT95] formulated a more refined semantics for Idealized Algol embodying "relational parametricity", in which values of procedure type are constrained by certain parametricity properties that guarantee good behavior. This parametric model of Idealized Algol then supports relational reasoning of the kind needed to establish program equivalences based on representation independence. We will show how to generalize their approach to the shared-variable setting. We first summarize some background material from [OT95].

7.1 Relations between worlds

We introduce a category whose objects are relations R between worlds; we write $R : W \leftrightarrow W'$ or $R \subseteq W \times W'$. For each world W we let $\Delta_W : W \leftrightarrow W$ denote the identity relation on W, i.e. $\Delta_W = \{(w, w) \mid w \in W\}$.

A morphism from $R : W_0 \leftrightarrow W_1$ to $S : X_0 \leftrightarrow X_1$ is a pair $(h_0 : W_0 \to X_0, h_1 : W_1 \to X_1)$ of morphisms in $\mathbf{W}$, such that, letting $h_0 = (f_0, Q_0)$ and $h_1 = (f_1, Q_1)$,

- for all $(x_0, x_1) \in S$, $(f_0 x_0, f_1 x_1) \in R$;
- for all $(x_0, x_1) \in S$, $x'_0 \in X_0$ and $x'_1 \in X_1$, if $(x'_0, x_0) \in Q_0$ & $(x'_1, x_1) \in Q_1$ then $(x'_0, x'_1) \in S$.

Loosely, we refer to these properties as saying that h_0 and h_1 respect R and

S. We represent such a morphism in the following diagrammatic form:

$$\begin{tikzcd} W_0 \arrow[r, "h_0"] \arrow[d, leftrightarrow, "R"'] & X_0 \arrow[d, leftrightarrow, "S"] \\ W_1 \arrow[r, "h_1"'] & X_1 \end{tikzcd}$$

The identity morphism from R to R corresponds to the diagram

$$\begin{tikzcd} W_0 \arrow[r, "\mathrm{id}_{W_0}"] \arrow[d, leftrightarrow, "R"'] & W_0 \arrow[d, leftrightarrow, "R"] \\ W_1 \arrow[r, "\mathrm{id}_{W_1}"'] & W_1 \end{tikzcd}$$

Composition in this category of relations is defined in the obvious way, building on composition in the category of worlds: when $(h_0, h_1) : R \leftrightarrow R'$ and $(h_0', h_1') : R' \leftrightarrow R''$ the composite morphism is $(h_0, h_1); (h_0', h_1') = (h_0; h_0', h_1; h_1')$.

7.2 Parametric functors and natural transformations

For each type θ we define a *parametric functor* $[\![\theta]\!]$ from worlds to domains, i.e. a functor $[\![\theta]\!]$ from $\mathbf{W}$ to $\mathbf{D}$ equipped with an action on relations, such that:

- whenever $R : W_0 \leftrightarrow W_1$, $[\![\theta]\!]R : [\![\theta]\!]W_0 \leftrightarrow [\![\theta]\!]W_1$;
- for all W, $[\![\theta]\!]\Delta_W = \Delta_{[\![\theta]\!]W}$;
- whenever

$$\begin{tikzcd} W_0 \arrow[r, "h_0"] \arrow[d, leftrightarrow, "R"'] & X_0 \arrow[d, leftrightarrow, "S"] \\ W_1 \arrow[r, "h_1"'] & X_1 \end{tikzcd}$$

holds then so does

$$\begin{tikzcd}
[\![\theta]\!]W_0 \arrow[r, "[\![\theta]\!]h_0"] \arrow[d, leftrightarrow, "[\![\theta]\!]R"'] & [\![\theta]\!]X_0 \arrow[d, leftrightarrow, "[\![\theta]\!]S"] \\
[\![\theta]\!]W_1 \arrow[r, "[\![\theta]\!]h_1"'] & [\![\theta]\!]X_1,
\end{tikzcd}$$

by which we mean that

$$(d_0, d_1) \in [\![\theta]\!]R \;\Rightarrow\; ([\![\theta]\!]h_0 d_0, [\![\theta]\!]h_1 d_1) \in [\![\theta]\!]S.$$

The first two conditions above say that $[\![\theta]\!]$ constitutes a "relator" [MS93, AJ91]. The last condition is a parametricity constraint.

Each well-typed phrase denotes a *parametric natural transformation* $[\![P]\!]$ between the parametric functors $[\![\pi]\!]$ and $[\![\theta]\!]$, i.e. a natural transformation obeying the following parametricity constraints: whenever $R : W_0 \leftrightarrow W_1$, $(u_0, u_1) \in [\![\pi]\!]R \;\Rightarrow\; ([\![P]\!]W_0 u_0, [\![P]\!]W_1 u_1) \in [\![\theta]\!]R$, as expressed by the following diagram:

$$\begin{tikzcd}
[\![\pi]\!]W_0 \arrow[r, "[\![P]\!]W_0"] \arrow[d, leftrightarrow, "[\![\pi]\!]R"'] & [\![\theta]\!]W_0 \arrow[d, leftrightarrow, "[\![\theta]\!]R"] \\
[\![\pi]\!]W_1 \arrow[r, "[\![P]\!]W_1"'] & [\![\theta]\!]W_1
\end{tikzcd}$$

Parametric natural transformations compose in the usual pointwise manner. The category having all parametric functors from **W** to **D** as objects, and all parametric natural transformations as morphisms, is cartesian closed [OT95].

Hence we may use the cartesian closed structure of this category in a perfectly standard way to interpret the λ-calculus fragment of our language, exactly along the lines developed in [OT95]. To adapt these ideas to the parallel setting, we must give trace-theoretic interpretations to types **comm**, **var**$[\tau]$, and **exp**$[\tau]$. We give details only **comm** and **exp**$[\tau]$, the definitions for **var**$[\tau]$ then being derivable.

7.3 Commands

We define $[\![\mathbf{comm}]\!]W$ and $[\![\mathbf{comm}]\!]h$ as before. To define $[\![\mathbf{comm}]\!]R : [\![\mathbf{comm}]\!]W_0 \leftrightarrow [\![\mathbf{comm}]\!]W_1$, when $R : W_0 \leftrightarrow W_1$, let $\mathrm{map}(R)$ be the obvious extension of R to traces of the same length, so that $\mathrm{map}(R) \subseteq W_0^\infty \times W_1^\infty$. We then define

$$\begin{aligned}
&(c_0, c_1) \in [\![\mathbf{comm}]\!]R \iff \\
&\qquad (\forall \alpha_0 \in c_0.\ \forall \rho_1.\ (\mathrm{map}\,\mathrm{fst}\,\alpha_0,\ \rho_1) \in \mathrm{map}(R) \Rightarrow \\
&\qquad \exists \alpha_1 \in c_1.\ \mathrm{map}\,\mathrm{fst}\,\alpha_1 = \rho_1 \ \&\ (\mathrm{map}\,\mathrm{snd}\,\alpha_0,\ \mathrm{map}\,\mathrm{snd}\,\alpha_1) \in \mathrm{map}(R)) \\
&\quad \&\ \ (\forall \alpha_1 \in c_1.\ \forall \rho_0.\ (\rho_0,\ \mathrm{map}\,\mathrm{fst}\,\alpha_1) \in \mathrm{map}(R) \Rightarrow \\
&\qquad \exists \alpha_0 \in c_0.\ \mathrm{map}\,\mathrm{fst}\,\alpha_0 = \rho_0 \ \&\ (\mathrm{map}\,\mathrm{snd}\,\alpha_0,\ \mathrm{map}\,\mathrm{snd}\,\alpha_1) \in \mathrm{map}(R)).
\end{aligned}$$

This is intended to capture the following intuition: $[\![\mathbf{comm}]\!]R$ relates two command meanings iff, whenever started in states related by R and interrupted in related ways, the commands respond in related ways. This, informally, expresses the idea that a trace set represents a (nondeterministic) state-transformation "extended in time".

It is straightforward to verify that $[\![\mathbf{comm}]\!]$ is indeed a parametric functor. In particular, since $\mathrm{map}\Delta_W$ is the identity relation on W^∞, and two traces α_0 and α_1 over $W \times W$ are equal iff $\mathrm{map}\,\mathrm{fst}\,\alpha_0 = \mathrm{map}\,\mathrm{fst}\,\alpha_1$ and $\mathrm{map}\,\mathrm{snd}\,\alpha_0 = \mathrm{map}\,\mathrm{snd}\,\alpha_1$, it is easy to see that

$$(c_0, c_1) \in [\![\mathbf{comm}]\!]\Delta_W \iff c_0 = c_1,$$

as required. Now suppose $(h_0, h_1) : R \to S$ and $(c_0, c_1) \in [\![\mathbf{comm}]\!]R$. We must show that

$$([\![\mathbf{comm}]\!]h_0 c_0, [\![\mathbf{comm}]\!]h_1 c_1) \in [\![\mathbf{comm}]\!]S.$$

This follows by a routine calculation, using the fact that the morphisms h_0 and h_1 respect the relations R and S.

As an example to illustrate this definition, suppose x is a variable of data type **int** corresponding to the V_{int}-component in states of shape $W \times V_{int}$. Let u be a corresponding environment. Let c_0 and c_1 be the trace sets corresponding to $x{:=}x+1$ and $x{:=}x-1$, respectively, i.e.

$$\begin{aligned}
c_0 &= \{((w_0, v_0), (w_0, v_0))((w_1, v_1), (w_1, v_0 + 1)) \mid w_0, w_1 \in W \ \&\ v_0, v_1 \in V_{int}\}^\dagger \\
c_1 &= \{((w_0, v_0), (w_0, v_0))((w_1, v_1), (w_1, v_0 - 1)) \mid w_0, w_1 \in W \ \&\ v_0, v_1 \in V_{int}\}^\dagger
\end{aligned}$$

Let R be the relation on $W \times V_{int}$ given by

$$(w, v)R(w', v') \iff w = w' \ \& \ v = -v'.$$

Then $(c_0, c_1) \in [\![\mathbf{comm}]\!]R$.

As a further example, let $c \in [\![\mathbf{comm}]\!]W$ and define the relation $R : W \leftrightarrow W \times V$ by

$$wR(w', v) \iff w = w'.$$

Then $(c,\ [\![\mathbf{comm}]\!](- \times V)c) \in [\![\mathbf{comm}]\!]R$.

Note also that the above definition of $[\![\mathbf{comm}]\!]R$ makes sense even when applied to arbitrary trace sets, i.e. closure is not crucial for the definition. Clearly we have

$$(c_0, c_1) \in [\![\mathbf{comm}]\!]R \Rightarrow (c_0^\dagger, c_1^\dagger) \in [\![\mathbf{comm}]\!]R.$$

We also have

$$\begin{array}{l}(p_0, q_0) \in [\![\mathbf{comm}]\!]R \ \& \ (p_1, q_1) \in [\![\mathbf{comm}]\!]R \ \Rightarrow \ (p_0; p_1,\ q_0; q_1) \in [\![\mathbf{comm}]\!]R \\ (p_0, q_0) \in [\![\mathbf{comm}]\!]R \ \& \ (p_1, q_1) \in [\![\mathbf{comm}]\!]R \ \Rightarrow \ (p_0 \| p_1,\ q_0 \| q_1) \in [\![\mathbf{comm}]\!]R\end{array}$$

so that sequential and parallel composition (and hence also iteration) interact smoothly with the action of $[\![\mathbf{comm}]\!]$ on relations.

7.4 Expressions

For expressions, we define $[\![\mathbf{exp}[\tau]]\!]W$ and $[\![\mathbf{exp}[\tau]]\!]h$ as before. When $R : W_0 \leftrightarrow W_1$ we define

$$\begin{array}{ll}(e_0, e_1) \in [\![\mathbf{exp}[\tau]]\!]R \iff & \\ & (\forall \rho_0 \in e_0 \cap W^\omega.\ \forall \rho_1.\ (\rho_0, \rho_1) \in \mathrm{map}(R) \ \Rightarrow \ \rho_1 \in e_1 \\ & \ \& \ \forall (\rho_0, v) \in e_0.\ \ \forall \rho_1.\ (\rho_0, \rho_1) \in \mathrm{map}(R) \ \Rightarrow \ (\rho_1, v) \in e_1) \\ \& & (\forall \rho_1 \in e_1 \cap W^\omega.\ \forall \rho_0.\ (\rho_0, \rho_1) \in \mathrm{map}(R) \ \Rightarrow \ \rho_0 \in e_0 \\ & \ \& \ \forall (\rho_1, v) \in e_1.\ \ \forall \rho_0.\ (\rho_0, \rho_1) \in \mathrm{map}(R) \ \Rightarrow \ (\rho_0, v) \in e_0)\end{array}$$

Intuitively, two expression meanings are related if when evaluated in related ways they either terminate together with the same answer, or both fail to terminate.

As an example, suppose again that x is a variable of type **int** corresponding to the V_{int} component in states of shape $W \times V_{int}$. Using the same relation R as above, so that

$$(w, v) R (w', v') \iff w = w' \ \&\ v = -v',$$

and assuming that u is a suitable environment, we have

$$([\![x]\!](W \times V_{int})u,\ [\![-x]\!](W \times V_{int})u) \in [\![\mathbf{exp}[\mathbf{int}]]\!]R.$$

7.5 Semantic definitions

The possible worlds semantics given above can be adapted immediately to the parametric setting, provided we show that each phrase denotes a parametric natural transformation. This is straightforward, using structural induction. For instance, it is easy to see that when $R : W \leftrightarrow W'$, parametricity of $[\![\mathbf{skip}]\!]$ amounts to the fact that

$$(\{(w, w) \mid w \in W\}^\dagger,\ \{(w', w') \mid w' \in W'\}^\dagger) \in [\![\mathbf{comm}]\!]R,$$

which holds obviously. Similarly, for the parallel construct the parametricity of $[\![P_1 \| P_2]\!]$ follows from parametricity of $[\![P_1]\!]$ and $[\![P_2]\!]$, since interleaving of trace sets respects $[\![\mathbf{comm}]\!]R$. Recursion requires a careful argument based on co-inductive properties of greatest fixed points.

To show the parametricity of recursion, let $\pi, \iota{:}\theta \vdash P : \theta$ and assume that P denotes a parametric natural transformation. We need to show that for all $R : W_0 \leftrightarrow W_1$, whenever $(u_0, u_1) \in [\![\pi]\!]R$,

$$([\![\mathbf{rec}\ \iota.P]\!]W_0 u_0,\ [\![\mathbf{rec}\ \iota.P]\!]W_1 u_1) \in [\![\theta]\!]R.$$

This may be achieved by means of a temporary detour using the parametric analogues of the functors $[\theta]$ used earlier. Let F_0 and F_1 be given by:

$$\begin{aligned} F_0(p_0) &= \mathrm{stut}_\theta W_0([\![P]\!]W_0(u_0 \mid \iota : p_0)), \\ F_1(p_1) &= \mathrm{stut}_\theta W_1([\![P]\!]W_1(u_1 \mid \iota : p_1)). \end{aligned}$$

By assumption on P, whenever $(p_0, p_1) \in [\![\theta]\!]R$ it follows that $(F_0(p_0), F_1(p_1)) \in [\![\theta]\!]R$. Consequently the functional $F : [\theta]W_0 \times [\theta]W_1 \to [\theta]W_0 \times [\theta]W_1$ given by

$$F(p_0, p_1) = (F_0(p_0), F_1(p_1))$$

is a monotone function on a complete lattice, and maps $[\![\theta]\!]R$ into itself. One can then show that the closure of its greatest fixed point is in $[\![\theta]\!]R$, and coincides with the pair $([\![\mathbf{rec}\ \iota.P]\!]W_0u_0, [\![\mathbf{rec}\ \iota.P]\!]W_1u_1)$.

7.6 Examples of reasoning

In addition to the laws and examples listed earlier, the relationally parametric semantics also validates the problematic equivalence discussed above:

$$\mathbf{new}[\mathbf{int}]\ \iota\ \mathbf{in}\ (\iota{:=}0;\ P(\iota{:=}\iota+1)) \ = \ P(\mathbf{skip}),$$

where P is a free identifier of type $\mathbf{comm} \to \mathbf{comm}$. To prove this, one can use a relation of form $R : W \leftrightarrow W \times V_{int}$, given by $wR(w', v) \iff w = w' \in W \ \&\ v \in V_{int}$. It is easy to show that, when u is a suitable environment in $[\![\pi]\!]W$ and u' binds x to the "fresh variable" represented by the V_{int} component of state, we get

$$([\![\mathbf{skip}]\!]Wu,\ [\![\iota{:=}\iota+1]\!](W \times V_{int})u') \in [\![\mathbf{comm}]\!]R.$$

The desired result follows by parametricity of $[\![P]\!]$.

Similarly, the parametric semantics validates the following equivalence,

$$\mathbf{new}[\mathbf{int}]\ \iota\ \mathbf{in}\ (\iota{:=}1; P(\iota)) \ = \ P(1),$$

when P is a free identifier of type $\mathbf{exp}[\mathbf{int}] \to \mathbf{comm}$.

Recall that we showed earlier that, when u is a suitable environment in which x denotes the V_{int} component of states of shape $W \times V_{int}$, and R is the relation

$$(w, v)R(w', v') \iff w = w' \ \&\ v = -v',$$

we have

$$\begin{array}{l} ([\![x{:=}x+1]\!](W \times V_{int})u,\ [\![x{:=}x-1]\!](W \times V_{int})u) \in [\![\mathbf{comm}]\!]R \\ ([\![x]\!](W \times V_{int})u,\ [\![-x]\!](W \times V_{int})u) \in [\![\mathbf{exp}[\mathbf{int}]]\!]R \end{array}$$

It follows by parametricity of $[\![P]\!]$ that

$$\mathbf{new}[\mathbf{int}]\ x\ \mathbf{in}\ (x{:=}0;\ P(x{:=}x+1)) = \mathbf{new}[\mathbf{int}]\ x\ \mathbf{in}\ (x{:=}0;\ P(x{:=}x-1)),$$

whenever P is a free identifier of type $\mathbf{comm} \to \mathbf{comm}$. Similarly,

$$\mathbf{new}[\mathbf{int}]\ x\ \mathbf{in}\ (x{:=}0;\ P(x,\ x{:=}x{+}1)) = \mathbf{new}[\mathbf{int}]\ x\ \mathbf{in}\ (x{:=}0;\ P(x,\ x{:=}x{-}1))$$

when P is a free identifier of type $(\mathbf{exp[int]} \times \mathbf{comm} \to \mathbf{comm})$. This example shows the equivalence of two implementations of an abstract "counter". This was shown for the sequential language by O'Hearn and Tennent[OT95].

To illustrate the subtle differences between sequential and parallel settings, consider the following phrase

$$\mathbf{new[int]}\ x\ \mathbf{in}\ (x{:=}0;\ P(x/2,\ x{:=}x+2)),$$

which amounts to yet another representation for an abstract counter, and is equivalent to both versions discussed above. In sequential Algol it is also equivalent to

$$\mathbf{new[int]}\ x\ \mathbf{in}\ (x{:=}0;\ P(x/2,\ x{:=}x+1; x{:=}x+1)),$$

but this equivalence fails in the parallel model. The reason lies in the inequivalence of $x{:=}x+1; x{:=}x+1$ and $x{:=}x+2$, and the ability, by looking at the value of x in the intermediate state, to detect the difference.

Despite this example, the phrases

$$\mathbf{new[int]}\ x\ \mathbf{in}\ (x{:=}0;\ P(x{:=}x+1; x{:=}x+1))$$

and

$$\mathbf{new[int]}\ x\ \mathbf{in}\ (x{:=}0;\ P(x{:=}x+2))$$

are equivalent in sequential Algol *and* in parallel Algol, even though $x{:=}x+1; x{:=}x+1$ and $x{:=}x+2$ are not semantically equivalent in the parallel model; no matter how P uses its argument, the only differences involve the local variable, whose value is ignored. To establish the equivalence, one can use the relation $R : W \leftrightarrow W \times V_{\mathrm{int}}$ given by $(w, (w', z)) \in R \iff w = w'$.

In contrast the phrases

$$\begin{array}{l}\mathbf{new[int]}\ x\ \mathbf{in}\\ \quad (x{:=}0;\ P(x{:=}x+1; x{:=}x+1);\\ \quad\ \mathbf{if}\ even(x)\ \mathbf{then\ diverge\ else\ skip})\end{array}$$

and

$$\begin{array}{l}\mathbf{new[int]}\ x\ \mathbf{in}\\ \quad (x{:=}0;\ P(x{:=}x+2);\\ \quad \mathbf{if}\ even(x)\ \mathbf{then\ diverge\ else\ skip}),\end{array}$$

where **diverge** is a divergent command, are equivalent in sequential but not in parallel Algol. For example, if P is $\lambda c.c\|c$, then the first phrase has an execution in which each argument thread reads x as 0, then each sets x to 1, and then the two final increments occur sequentially, leaving x with the value 3, causing termination; the other phrase, however, must diverge. The relation $(w, (w', z)) \in R \iff w = w' \;\&\; even(z)$ works for the sequential model but not for the parallel.

Indeed, in sequential Algol, the phrase

$$\begin{array}{l} \mathbf{new}[\mathbf{int}]\ x\ \mathbf{in} \\ \quad (x{:=}0;\ P(x{:=}x+2); \\ \quad\ \mathbf{if}\ even(x)\ \mathbf{then\ diverge\ else\ skip}) \end{array}$$

discussed above is equivalent to **diverge**. This is because the semantics of a command is taken to be a state transformation, and matter how many times P calls its argument the value of the local variable x stays even, causing the phrase to diverge. This equivalence fails for parallel Algol, because our semantics “observes” intermediate states during execution. Instead the phrase is equivalent to $P(\mathbf{skip}); \mathbf{diverge}$.

In the O'Hearn-Tennent model $\mathbf{if}\ x = 0\ \mathbf{then}\ f(x)\ \mathbf{else}\ 1$ and $\mathbf{if}\ x = 0\ \mathbf{then}\ f(0)\ \mathbf{else}\ 1$ fail to be semantically equivalent, because the model includes procedure meanings that violate the irreversibility of state change [OT95], yet the phrases behave identically in all sequential contexts. In contrast the equivalence should (and does) fail in our parallel model, because expression evaluation may not be atomic. For example, if f is $\lambda y.y$ and the phrase is evaluated in parallel with a command that may change the value of x from 0 to 2, the first case might yield the result 2.

The two isomorphic implementations of synchronizers discussed earlier:

$$\begin{array}{l} \mathbf{boolean}\ flag_1 = \mathbf{false},\ flag_2 = \mathbf{false}; \\ \mathbf{procedure}\ synch(x, y) = (x{:=}\mathbf{true};\ \mathbf{await}\ y;\ y{:=}\mathbf{false}) \\ P(synch(flag_1, flag_2), synch(flag_2, flag_1)) \end{array}$$

and the dualized version, in which the roles of the two truth values are reversed, can also be proved equivalent by an easy argument involving parametricity. Let $X = (W \times V_{bool}) \times V_{bool}$, and define the relation $R : X \leftrightarrow X$ by

$$((w, b_1), b_2) R ((w', b'_1), b'_2) \iff w = w' \;\&\; b_1 = \neg b'_1 \;\&\; b_2 = \neg b'_2.$$

The crucial step is to show that, when u is an environment binding $flag_1$ and $flag_2$ to variables corresponding to the intended components of state,

$$([\![synch(flag_1, flag_2)]\!]Xu,\ [\![synch(flag_2, flag_1)]\!]Xu) \in [\![\mathbf{comm}]\!]R.$$

The desired equivalence then follows straightforwardly.

The two non-isomorphic implementations of a "switch", discussed earlier, can be proved equivalent using the relation $R : W \times V_{bool} \leftrightarrow W \times V_{int}$ given by

$$(w, b)R(w', v) \iff w = w' \ \&\ b = (v > 0).$$

8 Conclusions

We have shown how to give semantic models for a parallel Algol-like language. The semantic models combine ideas from the theory of sequential Algol (possible worlds, relational parametricity) with ideas from the theory of shared-variable parallelism (transition traces) in a rather appealing manner which, we believe, brings out the sense in which shared-variable parallelism and call-by-name procedures are orthogonal. We have shown that certain laws of program equivalence familiar from shared-variable programming remain valid when the language is expanded to include procedures; and certain laws of equivalence familiar from functional programming remain valid when parallelism is added. Although we do not claim a full conservative extension property, these results suggest that our language Parallel Algol combines functional and shared-variable programming styles in a disciplined and well-behaved manner. We have discussed a variety of examples intended to show the utility of the language and the ability of our semantics to support rigorous arguments about the correctness properties of programs. Our parametric model offers a formal and general way to reason about "concurrent objects".

Our semantics inherit both the advantages and limitations of the corresponding sequential models and of the trace model for the simple shared-variable language. At ground type **comm** we retain the analogue of the full abstraction properties of [Bro93]: two commands have the same meaning if and only if they may be interchanged in all contexts without affecting the behavior of the overall program. The extra discriminatory power provided by the λ-calculus facilities does not affect this. However, like their sequential forebears, our models still include procedure values that violate the irreversibility of state change [OR95], preventing full abstraction at higher types. Recent work of Reddy [Red96], and of O'Hearn and Reynolds [OR95], incorporating ideas from linear logic, appears to handle irreversibility for sequential Algol; we conjecture that similar ideas may also work for the parallel language, with suitable generalization; this will be the topic of further research.

9 Acknowledgements

The work from which this paper grew began during a three-month visit (July-September 1995) to the Isaac Newton Institute for the Mathematical Sciences (Cambridge, England), as part of a research programme on Semantics of Computation. An early version of this paper appeared in the Proceedings of the 11th Annual IEEE Conference on Logic in Computer Science, published by IEEE Computer Society Press, and was later incorporated as a chapter of volume 2 of **Algol-like Languages**, edited by Peter O'Hearn and Bob Tennent. Thanks to Peter O'Hearn, John Reynolds, Edmund Robinson, Pino Rosolini, Philip Scott, Bob Tennent, and Glynn Winskel for helpful discussions and comments.

References

[AJ91] S. Abramsky and T. P. Jensen. A relational approach to strictness analysis for higher-order polymorphic functions. In *Conf. Record 18th ACM Symposium on Principles of Programming Languages*, pages 49–54. ACM Press, 1991.

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.

[Bro93] S. Brookes. Full abstraction for a shared variable parallel language. In *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*, pages 98–109. IEEE Computer Society Press, June 1993.

[HMT83] J. Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or What makes the free list free? In *ACM Symposium on Principles of Programming Languages*, pages 245–257, 1983.

[MS93] J. C. Mitchell and A. Scedrov. Notes on sconing and relators. In E. Boerger, editor, *Computer Science Logic '92, Selected Papers*, volume 702 of *Lecture Notes in Computer Science*, pages 352–378. Springer-Verlag, 1993.

[Ole82] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, 1982.

[OR95] P. W. O'Hearn and J.C. Reynolds. From Algol to Polymorphic Linear Lambda-calculus. Talk given at Isaac Newton Institute for Mathematical Sciences, Cambridge, 1995.

[OT95] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, May 1995.

[OT97] P. W. O'Hearn and R. D. Tennent. *Algol-like Languages*. Birkhäuser, 1997.

[Par79] D. Park. On the semantics of fair parallelism. In D. Bjørner, editor, *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 504–526. Springer-Verlag, 1979.

[Red96] U. S. Reddy. Global state considered unnecessary: object-based semantics of interference-free imperative programming. *Lisp and Symbolic Computation*, 9(1):7–76, February 1996.

[Rey81] J. C. Reynolds. The essence of Algol. In *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.

[Rey83] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing 83*, pages 513–523. North-Holland, Amsterdam, 1983.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5, 1955.

10 Appendix: Naturality of recursion

Throughout this Appendix suppose $\pi, \iota : \theta \vdash P : \theta$.

With each phrase type θ we associate a functor $[\theta]$ from the category of worlds to the category of complete lattices and monotone functions, defined by induction on θ:

$$
\begin{aligned}
&[\mathbf{comm}]W = \wp((W \times W)^\infty)\\
&[\mathbf{comm}]h = \lambda c.\{\alpha' \mid \mathrm{map}(f \times f)\alpha' \in c \;\&\; \mathrm{map}(Q)\alpha'\}\\
&[\mathbf{exp}[\tau]]W = \wp((W^+ \times V_\tau) \cup W^\omega)\\
&[\mathbf{exp}[\tau]]h = \lambda e.\{(\rho', v) \mid (\mathrm{map} f\rho, v) \in e\} \cup \{\rho' \mid \mathrm{map} f\rho' \in e \cap W^\omega\}\\
&[\theta \times \theta'] = [\theta] \times [\theta']\\
&[\theta \to \theta']W = \{p(-) \mid \forall h : W \to W'.p(h) : [\![\theta]\!]W' \to [\theta']W'\}\\
&[\theta \to \theta']hp = \lambda h' : W' \to W''.p(h; h')
\end{aligned}
$$

Intuitively, $[\theta]W$ is like $[\![\theta]\!]W$ without the closure requirements at ground types and naturality requirements at arrow types. Again we use the pointwise ordering on $[\theta \to \theta']W$. For each type θ and morphism h, $[\theta]h$ is continuous, and $[\![\theta]\!]W \subseteq [\theta]W$.

We define, for each phrase type θ, a natural transformation stut_θ from $[\theta]$ to $[\theta]$, embodying what it means to insert an extra stuttering step at that type. Again the definition is by structural induction on θ:

$$
\begin{aligned}
&\mathrm{stut}_{\mathbf{comm}} Wc = \{(w, w)\alpha \mid w \in W \;\&\; \alpha \in c\}\\
&\mathrm{stut}_{\mathbf{exp}[\tau]} We = \{(w\rho, v) \mid w \in W \;\&\; (\rho, v) \in e\} \cup \{w\rho \mid w \in W \;\&\; \rho \in e \cap W^\omega\}\\
&\mathrm{stut}_{\theta \times \theta'} = \mathrm{stut}_\theta \times \mathrm{stut}_{\theta'}\\
&\mathrm{stut}_{\theta \to \theta'} Wp = \lambda h : W \to W'.\mathrm{stut}_{\theta'} W' \circ (ph)
\end{aligned}
$$

We define induction on θ, a natural transformation clos_θ from $[\theta]$ to $[\![\theta]\!]$:

$$
\begin{aligned}
&\mathrm{clos}_{\mathbf{comm}} Wc = c^\dagger\\
&\mathrm{clos}_{\mathbf{exp}[\tau]} We = e^\dagger\\
&\mathrm{clos}_{\theta \times \theta'} = \mathrm{clos}_\theta \times \mathrm{clos}_{\theta'}\\
&\mathrm{clos}_{\theta \to \theta'} Wp = \lambda h : W \to W'.\mathrm{clos}_{\theta'} W' \circ (ph)
\end{aligned}
$$

The semantic definitions given earlier (minus the use of closure) then yield natural transformations $[P]$ from $[\pi]$ to $[\theta]$, such that $[\![P]\!]Wu = \mathrm{clos}_\theta W([P]Wu)$ for all $u \in [\![\pi]\!]W$. This may be shown by induction on the proof of the judgement $\pi \vdash P : \theta$.

For example, for sequential composition we put $[P_1; P_2]Wu = ([P_1]Wu) \cdot ([P_2]Wu)$ for all $u \in [\pi]W$. When $u \in [\![\pi]\!]W$ we then get

$$\begin{aligned}
[\![P_1; P_2]\!]Wu &= [\![P_1]\!]Wu; [\![P_2]\!]Wu \\
&= ([P_1]Wu)^\dagger; ([P_2]Wu)^\dagger \\
&= ([P_1]Wu \cdot [P_2]Wu)^\dagger \\
&= [P_1; P_2]Wu)^\dagger
\end{aligned}$$

When $\pi, \iota : \theta \vdash P : \theta$, and $u \in [\![\pi]\!]W$, the function

$$F = \lambda x : [\theta]W.\mathrm{stut}_\theta W([P]W(u \mid \iota : x))$$

is a monotone map on the complete lattice $[\theta]W$. Its greatest fixed point, which we denote by $\nu x.F(x)$, is in $[\theta]W$, and the closure of this fixed point is in $[\![\theta]\!]W$.

We therefore take

$$[\![\mathbf{rec}\ \iota.P]\!]Wu = \mathrm{clos}_\theta W(\nu x.\mathrm{stut}_\theta W([P]W(u \mid \iota : x)).$$

This definition is natural, in that $[\![\theta]\!]h([\![\mathbf{rec}\ \iota.P]\!]Wu) = [\![\mathbf{rec}\ \iota.P]\!]W'([\![\pi]\!]hu)$.

To show naturality, let $h : W \to W'$ and let $F' : [\theta]W' \to [\theta]W'$ be given by:

$$F' = \lambda x' : [\theta]W'.\mathrm{stut}_\theta W'([P]W([\![\pi]\!]hu \mid \iota : x')).$$

We must show that $[\theta]h(\nu F) = \nu F'$. We argue as follows.

- By definition of F', naturality of stut_θ, naturality of P (assumed as induction hypothesis), and the fixed point property of F, we have:

$$\begin{aligned}
F'([\theta]h(\nu F)) = &\ \mathrm{stut}_\theta W'([P]W'([\pi]hu \mid \iota : [\theta]h(\nu F))) \\
= &\ \mathrm{stut}_\theta W'([\theta]h([P]W'([\pi, \iota : \theta]h(u \mid \iota : \nu F)))) \\
= &\ [\theta]h(\mathrm{stut}_\theta W([P]W(u \mid \iota : \nu F))) \\
= &\ [\theta]h(\nu F),
\end{aligned}$$

  so that $[\theta]h(\nu F)$ is a fixed point of F'. Hence $[\theta]h(\nu F) \sqsubseteq \nu F'$.

- For the converse inequality, i.e. $\nu F' \sqsubseteq [\theta]h(\nu F)$, we show that $\nu F' \sqsubseteq [\theta]h(\mathrm{top}_{[\theta]W})$, from which the result follows by continuity of $[\theta]h$, and the fact that νF is equal to $F^\beta(\mathrm{top})$ for some ordinal β. We sketch the proof, focussing on the most difficult case (when θ is **comm**). In this case we need to show that every trace of $\nu F'$ respects (the equivalence relation of) h. To prove this we first need some definitions. Say that a value $p' \in [\mathbf{comm}]W'$ respects h for n steps if each trace in p' respects the equivalence relation of h for its first n steps. We will also say that a value p' in $[\theta_0 \to \theta_1]W'$ respects h for n steps if, for all $h' : W' \to W''$, and all $a \in [\theta_0]W''$, if a respects $h; h'$ for n steps then so does $p'h'a$. We say that an environment $u' \in [\pi]W'$ respects h for n steps if for all $\iota \in \mathrm{dom}(\pi)$, $u'(\iota)$ respects h for n steps. We then prove, by induction on P, that whenever u' respects h for n steps so does $[P]W'u'$. It follows easily that when u' and x' respect h for n steps, then $\mathrm{stut}_{\mathbf{comm}}W'([P]W'(u' \mid \iota : x'))$ respects h for $n+1$ steps. Clearly, when $u \in [\![\pi]\!]W$, $[\![\pi]\!]hu$ respects h for *all* steps. By the fixed point property of $\nu F'$ and the definition of $\mathrm{stut}_{\mathbf{comm}}$, the first step of every trace in $\nu F'$ is a stutter, which obviously preserves any equivalence relation. Hence, $\nu F'$ respects h for 1 step. Using the fixed point property of $\nu F'$ again one can then show by an easy induction on n that $\nu F'$ respects h for all steps, as required.

This proof extends to cover phrases of product type, arrow type, $\mathbf{exp}[\tau]$ and $\mathbf{var}[\tau]$, in a straightforward manner.