The essence of Parallel Algol

Stephen Brookes
Department of Computer Science
Carnegie Mellon University
Schenley Park
Pittsburgh, PA 15213

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. The combination of local variables, procedures, and parallelism supports a form of concurrent object-oriented programming. We provide a denotational semantics for this language, simultaneously adapting "possible worlds" to the parallel setting and generalizing "transition traces" 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, to permit reasoning about relation-preserving properties of programs, adapting work of O'Hearn and Tennent 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 shows that procedures and shared-variable parallelism can be combined smoothly.

1. Introduction

The programming language Algol 60 has had a major influence on the theory and practice of language design and implementation [10]. 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. Although Algol itself is no longer widely used, an idealized form of the language has stimulated a great deal of innovative research [10]. Idealized Algol, as characterized by John Reynolds [14], 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.

Although Algol 60 and Idealized Algol are sequential programming languages 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 shared-variable model, in which parallel commands (or "threads") 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 processes running concurrently.

To illustrate the use of procedures as a means of encapsulation, a procedure for implementing mutual exclusion [2] with a binary semaphore can be written (in sugared form) as:

```
procedure $mutex(n_1, c_1, n_2, c_2)$;
  boolean $s$;
  begin
        $s$:=true;
        while true do ($n_1$; await $s$ then $s$:=false; $c_1$; $s$:=true)
      || while true do ($n_2$; await $s$ then $s$:=false; $c_2$; $s$:=true)
  end
```

Here c_1 and c_2 are parameters representing "critical" regions of code, and n_1 and n_2 represent non-critical code. The local boolean variable s represents the semaphore. 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:

```
procedure $workers(c_0, c_1)$; while true do $(c_0 \| c_1)$
```

However, this program structure incurs the repeated overhead caused by thread creation and deletion each time the loop body is executed. Although this defect does not affect the correctness of the procedure it might be preferable for pragmatic reasons 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 [2], uses a pair of local boolean variables equipped with a simple synchronization strategy:

```
procedure $barrier(c_0, c_1)$;
  boolean $flag_0$, $flag_1$;
  procedure $synch(x, y)$; ($x$:=true; await $y$; $y$:=false);
  begin
      $flag_0$:=false; $flag_1$:=false;
      while true do ($c_0$; $synch(flag_0, flag_1)$)
    || while true do ($c_1$; $synch(flag_1, flag_0)$)
  end
```

The correctness of this implementation relies on locality of the flag variables: in a call of *barrier* the code bound to c_0 and c_1 cannot access the flags. The procedures *workers* and *barrier* are equivalent, in that for all possible arguments c_0 and c_1 the two procedure calls exhibit identical behaviors.

The combination of procedures, local variables, and parallelism also supports a form of concurrent object-oriented programming. An "object" is typically described informally as having some private (or local) state and providing "methods" for accessing and updating that state. For example a one-place integer buffer can be represented as a local integer variable (holding the buffer's current contents) together with two local boolean variables (used as semaphores), with *put* and *get* methods that follow the semaphore protocol:

```
integer $data$; boolean $full$, $empty$;
procedure $put(x)$;
   begin
      await $\neg full$ then $full$:=true;
      $data := x$; $empty$:=false
   end;
procedure $get(y)$;
   begin
      await $\neg empty$ then $empty$:=true;
      $y := data$; $full$:=false
   end;
$full$:=false; $empty$:=true;
$P(put, get)$
```

Here P is a free procedure identifier representing the "rest" of the program, and the fact that its arguments include *put* and *get* but not *data*, *full* or *empty* prevents unconstrained access to the local state of the buffer. Note that P may invoke its arguments repeatedly, perhaps concurrently, and the buffer behaves in proper FIFO manner no matter what P does.

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 [14, 11, 9] do not handle parallelism, presumably because of the difficulty even in the sequential setting of modelling "local" state accurately [4]. Nevertheless it seems intuitive that Algol-style procedures and parallelism are "orthogonal" concepts, so that one ought to be able to design a programming language incorporating both seamlessly [1]. 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 [4]. Such models typically fail to validate

[1] We use the term "orthogonal" informally, to convey the idea that the semantics of procedural and parallel constructs can be given more or less in isolation of each other and combined in a modular manner. The addition of parallelism to a sequential Algol-like language does not invalidate elementary laws of equivalence from functional programming, such as the β-law, and the addition of procedures to a simple shared-variable language does not break elementary semantic equivalences involving parallel composition.

certain intuitive laws of program equivalence which express non-interference or “locality” properties of local variables, 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. For similar reasons a traditional location-based semantics cannot be used to prove correctness of the *mutex* procedure or the buffer implementation given above; for example, the *mutex* procedure can violate mutual exclusion when applied to arguments that happen to affect the location bound to s.

A more satisfactory semantics for a sequential Algol-like language was proposed by Reynolds and Oles [14, 11], 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 [3], 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 [3] covers 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* [12]. 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 models parallelism at an appropriate level of abstraction to permit compositional reasoning about safety and liveness properties of programs. Our categorical recasting of [3] permits an improved treatment of local variables, which were modelled in a rather *ad hoc* manner in the earlier paper. 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. The fact that we are able to adapt traces to the functor category setting supports the claim that procedures and parallelism are orthogonal. Like Reynolds’ semantics for sequential Algol, our semantics can be viewed as bringing out the stack discipline implicit in the procedure mechanism[2].

Since we are interested in proving liveness and safety properties of parallel programs it is vital to deal accurately with infinite traces. Recursion is the primary cause of infinite behavior, and special care is required to get the semantics of recursive programs right. In our setting it is not appropriate to regard divergence as “catastrophic”, as is done in several models of CSP [18]. It is equally wrong to equate all forms of divergence, as in a conventional least-fixed-point semantics for sequential programs,

[2] Since each parallel component in a program activates and de-activates storage in a stack-like manner independently of the other components it would be more accurate to say that our semantics brings out the “cactus stack” discipline.

which typically uses a single distinguished semantic value $\perp$ to represent non-termination. For example we must distinguish between a program that loops forever without changing the state and a program that keeps incrementing a variable repeatedly, since they satisfy different safety and liveness properties. Instead we provide a more refined treatment of recursion, making use of a fundamental "constructivity" property of programs to ensure that non-termination is modelled appropriately [3]. A least-fixed-point semantics for our language would capture only the finite behaviors of programs, and would therefore be unsuitable for liveness analysis. We use instead a greatest-fixed-point semantics that models both finite and infinite aspects of a program's behavior.

As we have remarked earlier, 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 still 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 [9], building on foundations laid in [15]. 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. This yields a powerful methodology for proving the correctness of concurrent objects.

2. Syntax

2.1. Types and type environments

The type structure of our language is conventional [14]: 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:

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

For convenience we also introduce auxiliary phrase types $\mathbf{atom}[\tau]$ ("atomic expressions" of type τ) and **atom** ("atomic commands").

[3] This is reminiscent of the role played by an analogous constructivity property in justifying the treatment of recursion in various semantics of CSP [18].

Let ι range over the set of identifiers. A type environment π is a finite partial function from identifiers to types. We write $\mathrm{dom}(\pi)$ for the domain of π, i.e. the finite 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. 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**. We omit the rules dealing with phrases of type **var**$[\tau]$ and **exp**$[\tau]$, except to remark that the language contains the usual arithmetic and boolean operations. 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.

We restrict the use of a "conditional atomic action", denoted **await** B **then** P, to cases where P is "atomic". We suppress the syntactic rules for atomic expressions and atomic commands, noting simply that atomic expressions are built from constants, identifiers, and primitive integer and boolean operations, and that an atomic command is a finite sequence of assignments involving atomic expressions, or **skip**. This syntactic constraint is common [2], guaranteeing that an atomic command always terminates, so that it is feasible to implement this construct as an indivisible action without incurring deadlock. This limitation does not significantly constrain the expressive power of our language. We use **await** B as an abbreviation for **await** B **then skip**.

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:

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

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

integer z;
  begin P **end**

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

procedure $f(x);\ P_0$;
  begin P **end**

instead of $(\lambda f.P)($**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 **W** of possible worlds [11] 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

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

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

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

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

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

$$\frac{\pi \vdash B : \textbf{exp}[\textbf{bool}] \quad \pi \vdash P : \textbf{atom}}{\pi \vdash \textbf{await } B \textbf{ then } P : \textbf{comm}}$$

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

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

$$\frac{}{\pi \vdash \iota : \pi(\iota)} \quad \text{when } \iota \in \text{dom}(\pi)$$

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

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

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

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

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

Figure 1. Type judgements

of data type τ. We will use V, W, X, and decorated versions such as W', as meta-variables ranging over **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:

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

We will use the notation $[w']_Q$ for the equivalence class of w', and we will write $f_{w'} : [w']_Q \rightarrow W$ for the corresponding restriction of f, and $f_{w'}^{-1} : W \rightarrow W'$ for its inverse.

Intuitively, when $(f, Q) : W \rightarrow W'$, 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 $\{(x', y') \mid x'Qy'\}$.

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 $- \times V : W \rightarrow W \times V$, given by

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

Intuitively an expansion of form $- \times V_\tau$ models the effect (on the shape of the store) of a single local variable declaration.

The composition of morphisms $h = (f, Q) : W \rightarrow W'$ and $h' = (g, R) : W' \rightarrow W''$, which we write as $h; h' : W \rightarrow W''$, is the pair given by:

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

It is easy to check that this pair satisfies the requirements listed above, so that this does indeed define a valid morphism.

As Oles has shown[11], 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 the functions

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

be the obvious natural isomorphisms. 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 $W \times X$. The composition of an expansion from W to $W \times V_1$ with an expansion from $W \times V_1$ to $(W \times V_1) \times V_2$ yields the same result as an expansion from W to $W \times (V_1 \times V_2)$, up to associativity. 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 **W** to the category **D** of domains and continuous functions. As shown by Oles [11], 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 **exp**$[\tau]$.

4.1. Atomic commands

Atomic commands are given a conventional interpretation, along lines familiar from the sequential setting, slightly simplified because atomic phrases always terminate. At world W an atomic command denotes a total function from W to W. The corresponding functor is:

$$\begin{aligned}&[\![\textbf{atom}]\!]W = W \to W\\ &[\![\textbf{atom}]\!](f, Q) = \lambda\gamma.\, \lambda w'.\, f_{w'}^{-1}(\gamma(f w')).\end{aligned}$$

For example, $[\![\textbf{atom}]\!](- \times V_\tau)\gamma(w, v) = (\gamma w, v)$ for all $\gamma \in W \to W$ and all $w \in W, v \in V_\tau$.

4.2. Commands

We interpret the type **comm** using "transition traces" [3], but instead of assuming a single global state set we parameterize our definitions in terms of worlds. For each world W, $[\![\textbf{comm}]\!]W$ will consist of sets of traces over W. A finite trace $(w_0, w'_0)(w_1, w'_1)\ldots(w_n, w'_n)$ 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 $\langle(w_n, w'_n)\rangle_{n=0}^{\infty}$ 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 [3], 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*[4]. Intuitively, stuttering involves the insertion of "idling" steps 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.

[4] The use of closed sets of traces guarantees full abstraction for the simple shared-variable language [3]. The closure conditions correspond, respectively, to reflexivity and transitivity of the $\to^*$ relation in a conventional operational semantics.

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 [5]:

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

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 $(W \times W)^\infty$ 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

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

It is straightforward to check that this is indeed a functor.

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:

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

This is as intended: c represents the meaning of a command that affects the part of the store represented by W, and when we expand the shape of the store to $W \times V$ the extra structure represented by the V component should not be affected by the command's behavior.

Note that if c is a closed set of traces so is $[\![\mathbf{comm}]\!]hc$. 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. In addition $[\![\mathbf{comm}]\!]h$ interacts simply with concatenation and iteration: $[\![\mathbf{comm}]\!]h(T_1 \cdot T_2) = [\![\mathbf{comm}]\!]hT_1 \cdot [\![\mathbf{comm}]\!]hT_2$, and hence $[\![\mathbf{comm}]\!]h(T^+) = ([\![\mathbf{comm}]\!]hT)^+$, and similarly for infinite iteration. These observations are sometimes helpful in calculations.

[5] 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.

4.3. Atomic expressions

For atomic expressions again the interpretation is simple. At world W an atomic expression of type τ denotes a total function from W to V_τ:

$$\begin{aligned} [\![\textbf{atom}[\tau]]\!]W &= W \rightarrow V_\tau \\ [\![\textbf{atom}[\tau]]\!](f, Q) &= \lambda e.\, e \circ f. \end{aligned}$$

4.4. Expressions

For expression types **exp**$[\tau]$ we use traces, since expressions can be used in non-atomic contexts. However, since we assume that expression evaluation does not cause side-effects, we can employ a slightly simpler form of trace than was used for commands. We also allow for possible non-termination, and for the possibility that expression evaluation may be non-deterministic.

A finite trace of the form $(w_0 w_1 \ldots w_n, v)$ 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 during evaluation, and no result value is obtained; we represent such a case as an infinite trace $\langle w_n \rangle_{n=0}^{\infty}$ 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 at world W as a subset e of $W^+ \times V_\tau \cup W^\omega$, closed under the obvious analogues of stuttering and mumbling [6]. 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

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

Again, functoriality is easy to check.

4.5. Product types

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

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

4.6. Arrow types

We interpret arrow types using functor exponentiation, as in [9]. The domain $[\![\theta \rightarrow \theta']\!]W$ consists of the families $p(-)$ of functions, indexed by morphisms from W, such that whenever $h : W \rightarrow W'$, $p(h) : [\![\theta]\!]W' \rightarrow [\![\theta']\!]W'$; and whenever $h' : W' \rightarrow W''$, $p(h)\ ;\ [\![\theta']\!]h' = [\![\theta]\!]h'; p(h\ ;\ h')$. This uniformity

[6] 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.

condition amounts to commutativity of the following diagram, for all W', W'', $h : W \to W'$ 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

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

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 $[\![\theta']\!]$ is a functor from worlds to domains.

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 is a convenient notational abuse.

4.7. 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:

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

This formulation is exactly as in [11], although the underlying interpretations of **comm** and **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 W' and 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 [3] 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:
$$[\![1]\!]Wu = \{(w, 1) \mid w \in W\}^{\dagger} = \{(\rho, 1) \mid \rho \in W^{+}\}.$$
- The following clause specifies that addition is sequential and evaluates its arguments from left to right:
$$\begin{aligned}[\![E_1 + E_2]\!]Wu \ =& \\ &\{(\rho_1\rho_2, v_1 + v_2) \mid (\rho_1, v_1) \in [\![E_1]\!]Wu \ \& \ (\rho_2, v_2) \in [\![E_2]\!]Wu\}^{\dagger} \\ \cup& \{\rho_1\rho_2 \mid \exists v_1.\, (\rho_1, v_1) \in [\![E_1]\!]Wu \ \& \ \rho_2 \in [\![E_2]\!]Wu \cap W^{\omega}\}^{\dagger} \\ \cup& \{\rho \in W^{\omega} \mid \rho \in [\![E_1]\!]Wu\}^{\dagger}\end{aligned}$$

  Note that this interpretation invalidates algebraic laws such as $E_1 + E_2 = E_2 + E_1$, which hold in sequential Algol but fail in the parallel setting with this non-atomic sequential form of addition. Other interpretations are also possible, such as a parallel non-atomic form of addition for which the commutative law does hold.

Let $\Delta_W : W \to W \times W$ denote the diagonal function: $\Delta_W(w) = (w, w)$. 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. Atomic commands and expressions

The semantics of atomic phrases is standard, essentially as in the Reynolds-Oles semantics of expressions and commands in sequential Algol. The main difference is that atomic phrases always terminate, so that we work with total functions rather than partial. When convenient we will identify the function denoted by an atomic phrase with its graph, and we will also regard this graph as a set of "singleton" traces, viewing for example a pair (w, w') as a command trace of length 1.

- Whenever $\pi \vdash P :$ **atom** we have $[\![P]\!]Wu \in W \to W$. For example, when P_1 and P_2 are atomic commands we define
$$[\![P_1; P_2]\!]Wu = [\![P_2]\!]Wu \circ [\![P_1]\!]Wu.$$
- Whenever $\pi \vdash E :$ **atom**$[\tau]$ and $u \in [\![\pi]\!]W$ we have $[\![E]\!]Wu \in W \to V_{\tau}$. For example, when E_1 and E_2 are atomic expressions
$$[\![E_1 + E_2]\!]Wu = \{(w, v_1 + v_2) \mid (w, v_1) \in [\![E_1]\!]Wu \ \& \ (w, v_2) \in [\![E_2]\!]Wu\}.$$

  Obviously atomic addition is commutative.

Note that atomic commands are also commands: when $\pi \vdash P : \textbf{atom}$ is valid, so is $\pi \vdash P : \textbf{comm}$. The atomic semantics of P is related to its trace semantics in the expected way: the atomic semantics of P is determined by the traces of length 1. Thus

$$[\![P : \textbf{atom}]\!]Wu = \{(w, w') \mid (w, w') \in [\![P : \textbf{comm}]\!]Wu\}.$$

A similar relationship holds for atomic expressions:

$$[\![E : \textbf{atom}[\tau]]\!]Wu \quad = \quad \{(w, v) \mid (w, v) \in [\![E : \textbf{exp}[\tau]]\!]Wu\}.$$

5.3. skip

skip has only finite traces consisting of stuttering steps:

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

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

$$\begin{aligned}[\![\textbf{comm}]\!](f, Q)([\![\textbf{skip}]\!]Wu) \quad &= [\![\textbf{comm}]\!](f, Q)\{(w, w) \mid w \in W\}^{+} \\ &= ([\![\textbf{comm}]\!](f, Q)\{(w, w) \mid w \in W\})^{+} \\ &= \{(w', w') \mid w' \in W'\}^{+} \\ &= [\![\textbf{skip}]\!]W'([\![\pi]\!](f, Q)u)\end{aligned}$$

because f puts each Q-class in bijection with W and stuttering steps obviously project back to stuttering steps.

5.4. Assignment

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

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

Note the use of $\text{map}\Delta_W$ 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 under mumbling 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.

5.5. 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 $[\![\mathbf{comm}]\!]h(T_1;T_2) = ([\![\mathbf{comm}]\!]hT_1);([\![\mathbf{comm}]\!]hT_2)$.

5.6. Parallel composition

Parallel composition of commands corresponds to fair interleaving of traces. For each set A we define the following subsets of $A^\infty \times A^\infty \times A^\infty$:

$$\begin{aligned}
&\mathit{both}_A = \{(\alpha,\beta,\alpha\beta),(\alpha,\beta,\beta\alpha) \mid \alpha,\beta \in A^+\}\\
&\mathit{one}_A = \{(\alpha,\epsilon,\alpha),(\epsilon,\alpha,\alpha) \mid \alpha \in A^\infty\}\\
&\mathit{fairmerge}_A = \mathit{both}_A^* \cdot \mathit{one}_A \cup \mathit{both}_A^\omega,
\end{aligned}$$

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.[7]

Intuitively, $\mathit{fairmerge}_{W\times W}$ 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 $f : A \to B$,

$$(\alpha,\beta,\gamma) \in \mathit{fairmerge}_A \;\Rightarrow\; (\mathrm{map} f\alpha, \mathrm{map} f\beta, \mathrm{map} f\gamma) \in \mathit{fairmerge}_B.$$

We then define

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

Again it will be convenient to introduce a semantic parallel composition operator: for trace sets T_1 and T_2 over W let $T_1 \| T_2 = \{\alpha \mid \exists(\alpha_1,\alpha_2,\alpha) \in \mathit{fairmerge}_{W\times W}.\; \alpha_1 \in T_1 \;\&\; \alpha_2 \in T_2\}^\dagger$. 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 $h : W \to W'$.

[7] Equivalently $\mathit{fairmerge}_A$ can be characterized as the greatest fixed point of the monotone function $F(t) = \mathit{both}_A \cdot t \cup \mathit{one}_A$ 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 $\mathit{fairmerge}_A$ in which one or both of α and β is finite. The greatest fixed point also includes the cases where α and β are both infinite.

5.7. Local variables

A trace of **new**$[\tau]$ ι **in** P at world W should be constructed from an execution of P in the expanded world $W \times V_\tau$, with ι bound to a fresh variable of type τ, during which 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 is interference-free iff for each pair of consecutive steps (w_n, w'_n) and (w_{n+1}, w'_{n+1}) in the trace we have $w'_n = w_{n+1}$. Thus the traces of **new**$[\tau]$ ι **in** P in world W and environment u should have the form map(fst × fst)α, where α is a trace of P in world $W \times V_\tau$ (and suitably adjusted environment) such that map(snd × snd)α 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:

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

5.8. 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 **if** t **then** p_1 **else** p_2 of $[\![\theta]\!]W$.

- For $\theta = \mathbf{exp}[\tau]$, **if** t **then** p_1 **else** p_2 is

$$\begin{aligned}&\{\rho\rho_1 \mid (\rho, \mathtt{tt}) \in t\ \&\ \rho_1 \in p_1\}^\dagger \cup\\ &\{\rho\rho_2 \mid (\rho, \mathtt{ff}) \in t\ \&\ \rho_2 \in p_2\}^\dagger \cup\\ &\{\rho \mid \rho \in t \cap W^\omega\}\end{aligned}$$

- For $\theta = \mathbf{comm}$, **if** t **then** p_1 **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 $\theta = (\theta_0 \to \theta_1)$, (**if** t **then** p_1 **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

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

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

5.9. Conditional atomic action

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

$$\begin{aligned}&[\![\textbf{await } B \textbf{ then } P]\!]Wu = \\ &\quad \{(w, w') \in [\![P]\!]Wu \mid (w, \mathtt{tt}) \in [\![B]\!]Wu\}^{\dagger} \\ &\quad \cup \{(w, w) \mid (w, \mathtt{ff}) \in [\![B]\!]Wu\}^{\omega} \\ &\quad \cup \{(w, w)^{\omega} \mid w^{\omega} \in [\![B]\!]Wu\}^{\dagger}.\end{aligned}$$

Recall that P is assumed to be an atomic command, so that $[\![P]\!]Wu$ is a total function from W to W whose graph determines a set of singleton traces that represent interference-free executions of P. In particular $[\![\textbf{await true then } P]\!]Wu = ([\![P : \mathbf{atom}]\!]Wu)^{\dagger}$.

If the test expression B always terminates, as is common, for example when B is atomic, the third part of the clause becomes vacuously empty.

5.10. 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

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

This trace set 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. Note that this functional is "constructive", in the intuitive sense that for each $n \geq 0$, the first $n + 1$ steps of traces in $F(t)$ are uniquely determined by the first n steps of traces in t, because of the "stuttering" caused by evaluating B.

The need to take the closure only *after* constructing the fixed point is shown by the special case of the loop **while true do skip**. This command does nothing but stutter forever, so that we would expect

$$[\![\textbf{while true do skip}]\!]Wu = \{(w, w) \mid w \in W\}^{\omega}.$$

Both the iterative formula given above and the greatest fixed point of F agree with this. However, the closure-preserving functional

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

interpreted on closed trace sets, coincides with the identity function when B is **true** and C is **skip**. The greatest fixed point of G is therefore the set of *all* traces over W, which does not agree with the operational characterization.

Notice also that taking the *least* fixed point of F would yield only the *finite* traces of the loop, ignoring any potential for infinite iteration.

5.11. Recursion

The above discussion of while-loops showed the need to take the greatest fixed point of a functional on arbitrary trace sets, and pointed out the role of stuttering in ensuring that divergence is modelled accurately. Similar needs arise in interpreting more general recursive programs.

Consider for example the command **rec** $\iota.\iota$, which 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, exactly as for the divergent loop considered above:

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

This trace set is not the greatest fixed point of the *identity function* on $[\![\textbf{comm}]\!]W$, 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 functional

$$F = \lambda c.\{(w,w)\alpha \mid w \in W \;\&\; \alpha \in c\},$$

operating on the complete lattice $[\textbf{comm}]W = \wp((W \times W)^{\infty})$ of *arbitrary* trace sets; intuitively, the extra initial stutter mimics an operational step in which the recursion is unwound. Obviously any fixed point of F contains only infinite traces; moreover the initial stutter inserted by F permits a proof by induction that for all $n \geq 0$ and all trace sets c the first n steps of each trace in $F^n(c)$ are stutters. Thus the greatest fixed point of F is $\bigcap_{n=0}^{\infty} F^n((W \times W)^{\infty}) = \{(w,w) \mid w \in W\}^{\omega}$ as claimed. This trace set is already closed under stuttering and mumbling, so it belongs to $[\![\textbf{comm}]\!]W = \wp^{\dagger}((W \times W)^{\infty})$. We can therefore define $[\![\textbf{rec }\iota.\iota]\!]Wu = \nu F$. To show naturality of this definition let $h : W \to W'$ and let F' be the functional on $[\textbf{comm}]W'$ given by

$$F' = \lambda c'.\{(w',w')\alpha' \mid w' \in W' \;\&\; \alpha' \in c'\},$$

so that $[\![\textbf{rec }\iota.\iota]\!]W'([\![\pi]\!]hu) = \nu F' = \{(w',w') \mid w' \in W'\}^{\omega}$. We have

$$\begin{array}{rcl}
[\![\textbf{comm}]\!]h(\nu F) & = & [\![\textbf{comm}]\!]h(\{(w,w) \mid w \in W\}^{\omega}) \\
& = & ([\![\textbf{comm}]\!]h\{(w,w) \mid w \in W\})^{\omega} \\
& = & \{(w',w') \mid w' \in W'\}^{\omega} \\
& = & \nu F',
\end{array}$$

as required for naturality. Note, however, that the successive pairs of approximations to these fixed points are *not* naturally related. For instance, when $h = (f,Q) : W \to W'$ is a non-trivial expansion morphism, so that Q has more than one equivalence class,

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

Nevertheless, the stutters induced by F and F' support a proof by induction that for all $n \geq 0$ the first n steps of $F^n((W \times W)^\infty)$ and $F'^n((W' \times W')^\infty)$ are naturally related, and in the limit we get full naturality.

The discussion above relies crucially on the fact that $[\textbf{comm}]W$ is a complete lattice, so that the existence of the relevant fixed point is guaranteed by Tarski's Theorem [19]. However, the generalization to all types is not so straightforward, since the domain $[\theta \rightarrow \theta']W$ does not possess a top element. We can see this as follows, by considering the special case of $[\textbf{comm} \rightarrow \textbf{comm}]W$. The obvious order-theoretic candidate for top of this domain is the family $\mathrm{top}(-)$ such that for all $h : W \rightarrow W'$,

$$\mathrm{top}(h) = \lambda c' : [\textbf{comm}]W'.\,(W' \times W')^\infty.$$

However, as was shown above, $[\textbf{comm}]h$ does not preserve top; hence this family lacks the naturality property required for membership in $[\textbf{comm} \rightarrow \textbf{comm}]W$. Furthermore, the obvious natural candidate for tophood, i.e. the family *top*$(-)$ given by

$$\textit{top}(h) = \lambda c' : [\textbf{comm}]W'.\,[\textbf{comm}]h((W \times W)^\infty),$$

is not even the order-theoretic top among the natural elements, since it does not dominate the identity family $\mathrm{id}(h) = \lambda c' : [\textbf{comm}]W'.\,c'$.

Nevertheless $[\textbf{comm} \rightarrow \textbf{comm}]W$ is clearly a sub-domain of the complete lattice $\langle\textbf{comm} \rightarrow \textbf{comm}\rangle W$ consisting of the *arbitrary* families $p(-)$ such that for all $h : W \rightarrow W'$, $p(h) : [\textbf{comm}]W' \rightarrow [\textbf{comm}]W'$, i.e. the lattice obtained by relaxing the naturality requirement. The top element of this lattice is clearly the family $\mathrm{top}(-)$ introduced above. A recursive phrase of this type determines a continuous functional F on this lattice. For example, consider the divergent phrase $\textbf{rec}\ \iota.\iota : \textbf{comm} \rightarrow \textbf{comm}$. Intuitively this should denote, at world W, the procedure meaning which causes infinite stuttering whenever it is called:

$$[\textbf{rec}\ \iota.\iota : \textbf{comm} \rightarrow \textbf{comm}]Wu = \lambda h : W \rightarrow W'.\,\lambda c'.\{(w', w') \mid w' \in W'\}^\omega.$$

This can be characterized as (the closure of) the greatest fixed point of the functional

$$F = \lambda p.\lambda h.\lambda c'.\{(w', w')\alpha' \mid w' \in W' \ \&\ \alpha' \in phc'\},$$

operating on the lattice $\langle\textbf{comm} \rightarrow \textbf{comm}\rangle W$. Note that the successive approximants $F^n(\mathrm{top})$ to the fixed point are not natural and thus do not qualify for membership in $[\textbf{comm} \rightarrow \textbf{comm}]W$. Nevertheless for each $n \geq 0$ it can be seen intuitively that $F^n(\mathrm{top})$ is natural "for n steps", and in the limit we achieve full naturality. Thus $\nu F \in [\textbf{comm} \rightarrow \textbf{comm}]W$, as required for this construction to make sense. Moreover, this definition is natural, since whenever $h : W \rightarrow W'$ we have

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

We can generalize the above discussion to more general recursive phrases as follows.

Each type θ denotes a functor $[\theta]$ from worlds to domains, defined as for $[\![\theta]\!]$ except that we omit the use of closure. For each $n \geq 0$, and each morphism $h : W \rightarrow W'$, we define a chain of approximations $[\theta]_n h : [\theta]W \rightarrow [\theta]W'$ whose limit is $[\theta]h$. For example,

$$[\textbf{comm}]_n(f, Q)c = \{\alpha' \mid \mathrm{map}(f \times f)\alpha' \in c \ \&\ \alpha' \text{ respects } Q \text{ for } n \text{ steps}\}.$$

The semantic definitions given for $[\![-]\!]$ can be systematically adjusted, by dropping the use of the closure operator $(-)^\dagger$, yielding a semantics $[-]$ based on arbitrary trace sets. For example, the semantic clause for sequential composition becomes $[P_1; P_2]Wu = [P_1]Wu \cdot [P_2]Wu$, with u interpreted as an environment based on arbitrary trace sets. When $\pi \vdash P : \theta$ is valid, $[P]$ is a natural transformation from $[\pi]$ to $[\theta]$. Moreover, $[P]$ is *non-destructive* [8], in the sense that, whenever $\pi \vdash P : \theta$ is valid, for all $n \geq 0$ and all $h : W \to W'$ we have

$$[P]W' \circ [\pi]_n h \sqsubseteq [\theta]_n h \circ [P]W.$$

We generalize the idea of inserting an extra initial stutter to all types, inductively, obtaining for each type θ a natural transformation stut_θ from $[\theta]$ to $[\theta]$. At ground types this is straightforward, as described above for commands; at arrow types we transform a procedure meaning so as to cause an extra stutter to occur each time the procedure is called. For example,

$$\begin{array}{l} \mathrm{stut}_{\mathbf{comm}} Wc = \{(w, w)\alpha \mid w \in W \ \&\ \alpha \in c\} \\ \mathrm{stut}_{\theta \to \theta'} Wp = \lambda h : W \to W'.\, \mathrm{stut}_{\theta'} W' \circ (ph). \end{array}$$

We then have, for all n, all $h : W \to W'$, and all θ,

$$\mathrm{stut}_\theta W' \circ [\theta]_n h = [\theta]_{n+1} h \circ \mathrm{stut}_\theta W.$$

Hence when $\pi \vdash P : \theta$ is valid the natural transformation $\mathrm{stut}_\theta \circ [P]$ is *constructive*, in that for all n, and all $h : W \to W'$,

$$(\mathrm{stut}_\theta W' \circ [P]W') \circ [\pi]_n h \sqsubseteq [\theta]_{n+1} h \circ (\mathrm{stut}_\theta W \circ [P]W),$$

making precise the informal notion of constructivity alluded to earlier.

When $\pi \vdash \mathbf{rec}\ \iota.P : \theta$ is valid, so that $\pi, \iota : \theta \vdash P : \theta$ is also valid, and $u \in [\pi]W$, the function

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

is a continuous map on the complete lattice $\langle\theta\rangle W \supseteq [\theta]W$, and restricts to a function from $[\theta]W$ to $[\theta]W$. Its greatest fixed point νF belongs to $[\theta]W$. We therefore take

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

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

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

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

so that $[\mathbf{rec}\ \iota.P]W'([\pi]hu) = \nu F'$. We must show that $[\theta]h(\nu F) = \nu F'$. We argue as follows.

- By definition of F', naturality of P, naturality of stut_θ, and the fixed point property, we have:

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

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

[8] Again this terminology is reminiscent of a related notion used in models of CSP[18].

- For the converse inequality let top and top$'$ be the greatest elements of $\langle\theta\rangle W$ and $\langle\theta\rangle W'$, respectively. We show first by induction that for all $k \geq 0$ we have
$$F'^k(\mathrm{top}') \sqsubseteq [\theta]_0 h(F^k(\mathrm{top})),$$
from which it follows that $\nu F' \sqsubseteq [\theta]_0 h(\nu F)$. Then we show, using the fixed point property and constructivity of $\mathrm{stut}_\theta \circ [P]$, that whenever $\nu F' \sqsubseteq [\theta]_n h(\nu F)$ we have
$$\begin{array}{rcl} \nu F' = F'(\nu F') & \sqsubseteq & F'([\theta]_n h(\nu F)) \\ & \sqsubseteq & [\theta]_{n+1} h(F(\nu F)) \\ & = & [\theta]_{n+1} h(\nu F). \end{array}$$
Thus by induction we have for all $n \geq 0$, $\nu F' \sqsubseteq [\theta]_n h(\nu F)$, and hence $\nu F' \sqsubseteq [\theta] h(\nu F)$ as required.

We can generalize the closure operator $(-)^\dagger$ to all types inductively, obtaining for each type θ a natural transformation $\theta^\dagger : [\theta] \to [\![\theta]\!]$. For example, $\mathbf{comm}^\dagger W$ is just the closure operator on trace sets over W, exactly as before; and closure at an arrow type is defined by

$$(\theta \to \theta')^\dagger W p = \lambda h : W \to W'.\, \lambda x : [\![\theta]\!]W'.\, \theta'^\dagger W'(phx).$$

Whenever $\pi \vdash P : \theta$ is valid, $[P]$ respects closure, in that for all $u_0, u_1 \in [\pi]W$,

$$\pi^\dagger W(u_0) = \pi^\dagger W(u_1) \;\Rightarrow\; \theta^\dagger([P]Wu_0) = \theta^\dagger W([P]Wu_1).$$

In other words, the closure of $[P]Wu$ depends only on the closure of u. Thus it makes sense to define

$$[\![\mathbf{rec}\ \iota.P]\!]Wu^\dagger = \theta^\dagger W([\mathbf{rec}\ \iota.P]Wu),$$

where u is any environment in $[\pi]W$ with closure $u^\dagger$.

Indeed with this as the interpretation of recursion the closed trace sets semantic function $[\![-]\!]$ can be obtained as the quotient of $[-]$: whenever $\pi \vdash P : \theta$ is valid, we have $[\![P]\!]W(\pi^\dagger u) = \theta^\dagger W([P]Wu)$. Since closure "absorbs" initial stuttering, i.e. for all types θ we have $\theta^\dagger \circ \mathrm{stut}_\theta = \theta^\dagger$, the validity of the usual unrolling rule for recursive phrases follows:

$$\begin{array}{rcl} [\![\mathbf{rec}\ \iota.P]\!]Wu^\dagger & = & \theta^\dagger W(\nu p.\, \mathrm{stut}_\theta W([P]W(u \mid \iota : p))) \\ & = & \theta^\dagger W(\mathrm{stut}_\theta W([P]W(u \mid \iota : [\mathbf{rec}\ \iota.P]Wu))) \\ & = & \theta^\dagger W([P](u \mid \iota : [\mathbf{rec}\ \iota.P]Wu)) \\ & = & [\![P]\!]W(u^\dagger \mid \iota : [\![\mathbf{rec}\ \iota.P]\!]Wu). \end{array}$$

The Appendix contains further details.

It is easy to check that this semantics for recursion does indeed prescribe the operationally expected meanings for the divergent phrase $\mathbf{rec}\ \iota.\iota$, at type **comm** and at type **comm** $\to$ **comm**. Similarly the meaning ascribed to the divergent integer expression $\mathbf{rec}\ n.n+1$ at world W is W^ω, again consistent with operational intuition: no matter what state changes may occur as the result of parallel activity the expression evaluation never stops.

It is also easy to verify that the meaning given to

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

coincides with the semantics given earlier for the loop **while** B **do** C, when ι does not occur free in C.

5.12. λ-calculus

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

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

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:

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

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'):

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

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

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

These laws amount to naturality (of the meaning of P) 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 being a composition of suitably chosen *swap* and *assoc* isomorphisms as discussed earlier.

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

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

where $P[P'/\iota]$ is the phrase obtained by replacing every free occurrence of ι in P by P', with renaming when necessary to avoid capture. In fact these equivalences follow easily from the semantic definitions when combined with the following Substitution Theorem: whenever $\pi \vdash P : \theta$ is valid, $\pi(\iota) = \theta'$ and $\pi \vdash P' : \theta'$ is valid, and $u \in [\![\pi]\!]W$,

$$[\![P[P'/\iota]]\!]Wu = [\![P]\!]W(u \mid \iota : [\![P']\!]Wu).$$

As usual the Substitution Theorem may be proved by structural induction on the derivation of $\pi \vdash P : \theta$.

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

$$\begin{array}{l}
(\textbf{if}\ B\ \textbf{then}\ P_1\ \textbf{else}\ P_2)(P)\ =\ \textbf{if}\ B\ \textbf{then}\ P_1(P)\ \textbf{else}\ P_2(P) \\
\lambda\iota : \theta.\textbf{if}\ B\ \textbf{then}\ P_1\ \textbf{else}\ P_2 = \textbf{if}\ B\ \textbf{then}\ \lambda\iota : \theta.P_1\ \textbf{else}\ \lambda\iota : \theta.P_2 \quad \text{if } \iota \text{ not free in } B,
\end{array}$$

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

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

Our semantics also equates **while true do skip** and **await false then skip**, because of the 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", generalizing the barrier synchronization example mentioned earlier:

$$\begin{array}{l}\textbf{boolean } \mathit{flag}_0, \ \mathit{flag}_1; \\ \textbf{procedure } \mathit{synch}(x, y); \ (x{:=}\textbf{true}; \ \textbf{await } y; \ y{:=}\textbf{false}); \\ \quad \mathit{flag}_0{:=}\textbf{false}; \ \mathit{flag}_1{:=}\textbf{false}; \\ \quad P(\mathit{synch}(\mathit{flag}_0, \mathit{flag}_1), \ \mathit{synch}(\mathit{flag}_1, \mathit{flag}_0))\end{array}$$

Here P is a free identifier of type $(\textbf{comm} \times \textbf{comm} \rightarrow \textbf{comm})$. 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

$$\begin{array}{l}\textbf{boolean } \mathit{flag}_0, \ \mathit{flag}_1; \\ \textbf{procedure } \mathit{synch}(x, y); \ (x{:=}\textbf{false}; \ \textbf{await } \neg y; \ y{:=}\textbf{true}); \\ \quad \mathit{flag}_0{:=}\textbf{true}; \ \mathit{flag}_1{:=}\textbf{true}; \\ \quad P(\mathit{synch}(\mathit{flag}_0, \mathit{flag}_1), \ \mathit{synch}(\mathit{flag}_1, \mathit{flag}_0))\end{array}$$

This is an example of the principle of representation independence. Our semantics for Parallel Algol validates this equivalence, by virtue of the existence of an isomorphism of worlds that relates the two implementations. To be specific, for all worlds W there is an isomorphism

$$\begin{array}{l}\mathit{dual} : W \times V_{\mathit{bool}} \rightarrow W \times V_{\mathit{bool}} \\ \mathit{dual} = (\lambda(w, b).(w, \neg b), \ (W \times V_{\mathit{bool}})^2)\end{array}$$

Naturality of the meaning of P with respect to this isomorphism 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

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

we recover the barrier synchronization example discussed earlier.

This synchronizer object works well in the above context, but less satisfactorily in cases where several threads can compete. For example, consider what can happen if we use for P the procedure

$$\begin{array}{l}\lambda(\mathit{left},\ \mathit{right}).\\ \quad (\mathit{left};\ c_0) \parallel (\mathit{left};\ c_1) \parallel (\mathit{right};\ c_2;\ \mathit{right};\ c_3)\end{array}$$

with the intention that the resulting program be equivalent to

$$((c_0 \| c_2); (c_1 \| c_3))\ \textbf{or}\ ((c_1 \| c_2); (c_0 \| c_3)),$$

where **or** is interpreted as non-deterministic choice [9]. Intuitively this equivalence may fail because it is possible for two threads concurrently to execute $synch(\mathit{flag}_0, \mathit{flag}_1)$ to completion, leading to the simultaneous parallel activity of c_0, c_1 and c_2.

A more robust synchronizer can be defined as follows, using a conditional atomic action to guarantee mutual exclusion between such competitor threads:

```
boolean $flag_0,\ flag_1$;
procedure $synch(x, y)$; (await $\neg x$ then $x{:=}$true; await $y$ then $y{:=}$false);
   $flag_0{:=}$false; $flag_1{:=}$false;
   $P(synch(flag_0, flag_1),\ synch(flag_1, flag_0))$
```

When P is instantiated as above the resulting program does behave as intended. This more sophisticated synchronizer object also has an equivalent dualized version (in which **false** and **true** are interchanged systematically).

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:

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

where P is a free identifier of type **comm** → **comm**. No matter how P is instantiated this should have the same effect as $P(\textbf{skip})$. 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 [9]. 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:

```
boolean $z$;
procedure $flick$; ($z{:=}$true);
procedure $on$; return $z$;
   $z{:=}$false;
   $P(flick,\ on)$
```

[9] It is straightforward to add this construct to the programming language. The corresponding semantic clause is simply $[\![P_1\ \textbf{or}\ P_2]\!]Wu = [\![P_1]\!]Wu \cup [\![P_2]\!]Wu$.

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

```
**integer** $z$;
**procedure** *flick*; $(z{:=}z + 1)$;
**procedure** *on*; **return** $(z > 0)$;
   $z{:=}0$;
   $P($*flick*, *on*$)$
```

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.

For an example exploiting parallelism, we remark that there is also a non-isomorphic implementation of our synchronizer object, in which flags take on successive integer values and the parity of a flag is used to indicate availability:

```
**integer** *flag*$_0$, *flag*$_1$;
**procedure** *synch*$(x, y)$; (**await** *even*$(x)$ **then** $x{:=}x + 1$; **await** *odd*$(y)$ **then** $y{:=}y + 1)$;
   *flag*$_0{:=}0$; *flag*$_1{:=}0$;
   $P($*synch*$($*flag*$_0$, *flag*$_1)$,
$mbox$*synch*$($*flag*$_1$, *flag*$_0))$
```

The equivalence of this and the above robust synchronizer cannot be proven in the model given so far.

7. Relational parametricity

In response to this inadequacy O’Hearn and Tennent [9] formulated a more refined semantics for Idealized Algol embodying “relational parametricity”, in which values of procedure type are constrained by certain relation-preservation 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 [9].

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 **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 **W** to **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" [5, 1]. The last condition is a parametricity constraint.

When $\pi \vdash P : \theta$ is valid $[\![P]\!]$ is a *parametric natural transformation* from $[\![\pi]\!]$ to $[\![\theta]\!]$, i.e. a natural transformation obeying the following parametricity constraints: whenever $R : W_0 \leftrightarrow W_1$, $(u_0, u_1) \in [\![\pi]\!]R \Rightarrow ([\![P]\!]W_0 u_0, [\![P]\!]W_1 u_1) \in [\![\theta]\!]R$. This property may be expressed in diagram form as follows:

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

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 [9].

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 [9]. 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. We also suppress the details of atomic types, since their treatment is standard.

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

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

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 (non-deterministic) 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 $\mathrm{map\,fst}\,\alpha_0 = \mathrm{map\,fst}\,\alpha_1$ and $\mathrm{map\,snd}\,\alpha_0 = \mathrm{map\,snd}\,\alpha_1$, 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 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 $(c_0, c_1) \in [\![\textbf{comm}]\!]R$.

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

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

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

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

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

We also have

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

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

7.4. Expressions

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

$$\begin{aligned} &(e_0, e_1) \in [\![\textbf{exp}[\tau]]\!]R \iff \\ &\qquad (\forall \rho_0 \in e_0 \cap W^\omega.\ \forall \rho_1.\ (\rho_0, \rho_1) \in \mathsf{map}(R) \Rightarrow \rho_1 \in e_1 \\ &\qquad \ \&\ \forall (\rho_0, v) \in e_0.\ \forall \rho_1.\ (\rho_0, \rho_1) \in \mathsf{map}(R) \Rightarrow (\rho_1, v) \in e_1) \\ &\quad \&\ \ (\forall \rho_1 \in e_1 \cap W^\omega.\ \forall \rho_0.\ (\rho_0, \rho_1) \in \mathsf{map}(R) \Rightarrow \rho_0 \in e_0 \\ &\qquad \ \&\ \forall (\rho_1, v) \in e_1.\ \forall \rho_0.\ (\rho_0, \rho_1) \in \mathsf{map}(R) \Rightarrow (\rho_0, v) \in e_0) \end{aligned}$$

Intuitively, two expression meanings are related if when evaluated in related ways they both terminate 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 as before, so that

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

and assuming that u is a suitable environment, we have

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

7.5. Semantic definitions

The possible worlds semantics given above can be adapted 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 $[\![\textbf{skip}]\!]$ amounts to the fact that

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

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 $[\![\textbf{comm}]\!]R$.

Recursion again requires a careful treatment. We define $[\![\textbf{rec}\ \iota.P]\!]$ as the closure of $[\textbf{rec}\ \iota.P]$, making use of a parametric version of the semantics $[-]$ based on arbitrary trace sets, defined as before but with suitable modifications to fit the relational setting. Also as before, we recover the closed trace set semantics $[\![-]\!]$ as the quotient of $[-]$ with respect to the equivalence induced by taking closure. We again define $[\textbf{rec}\ \iota.P]Wu = \nu p.\text{stut}_{\theta}W([P]W(u \mid \iota : p))$, where the fixed point is taken over the complete lattice $\langle\theta\rangle W$ extending $[\theta]W$. The proof that this fixed point belongs to the subset $[\theta]W$, and that this semantic definition is natural, depends as before on constructivity and on naturality of $[P]$. We also need to show that this is a parametric definition, i.e. for all $R : W_0 \leftrightarrow W_1$, whenever $(u_0, u_1) \in [\pi]R$,

$$([\textbf{rec}\ \iota.P]W_0u_0,\ [\textbf{rec}\ \iota.P]W_1u_1) \in [\theta]R.$$

Let F_0 and F_1 be given by:

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

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 continuous function on a complete lattice, and maps the subset $[\theta]R$ into itself. Let top_0 and top_1 be the top elements of $\langle\theta\rangle W_0$ and $\langle\theta\rangle W_1$ respectively. One can then show, by induction on n, that for all $n \geq 0$ we have $(F_0^n(\text{top}_0), F_1^n(\text{top}_1)) \in [\theta]R$. From this it follows easily that $(\nu F_0, \nu F_1) \in [\theta]R$, as required, by an obvious completeness property of $[\theta]R$.

7.6. Examples of parametric reasoning

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

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

where P is a free identifier of type $\textbf{comm} \to \textbf{comm}$. This can be shown with the help of the relation $R : W \leftrightarrow W \times V_{int}$ 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 ι to the "fresh variable" corresponding to 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 an environment in which x denotes the variable corresponding to the V_{int} component in states of shape $W \times V_{int}$, and R is the relation

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

we have

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

It follows by parametricity 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 $(\mathbf{exp}[\mathbf{int}] \times \mathbf{comm} \to \mathbf{comm})$. This example shows the equivalence in the parallel setting of two implementations of an abstract "counter". An analogous result was shown for the sequential setting by O'Hearn and Tennent[9], but the validation of such equivalences in parallel contexts requires our more detailed semantic model.

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

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

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

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

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

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

and

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

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

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

and

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

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

$$(w,(w',z)) \in R \iff w = w' \ \&\ \textit{even}(z)$$

works for the sequential model but not for the parallel.

Indeed, in sequential Algol, the phrase

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

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 $P(\textbf{skip}); \textbf{diverge}$.

In the O'Hearn-Tennent model **if** $x = 0$ **then** $f(x)$ **else** 1 and **if** $x = 0$ **then** $f(0)$ **else** 1 fail to be semantically equivalent, because the model includes procedure meanings that violate the irreversibility of state change [9], yet the phrases behave identically in all sequential contexts. In contrast the equivalence should (and does) fail in our parallel model, because expression evaluation need 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 dual implementations of synchronizers discussed earlier can be proven 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 $flag_0$ and $flag_1$ to variables corresponding to the intended components of state,

$$([\![\textit{synch}(\textit{flag}_0,\textit{flag}_1)]\!]Xu,\ [\![\textit{synch}(\textit{flag}_1,\textit{flag}_0)]\!]Xu) \in [\![\textbf{comm}]\!]R.$$

The desired equivalence then follows immediately.

The equivalence of boolean-based synchronizer and the parity-based version can be shown by means of the relation $R : W \times V_{bool} \leftrightarrow W \times V_{int}$ given by

$$(w, b)R(w', n) \iff w = w' \;\&\; (b = \textit{even}(n)).$$

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, supports the intuition that 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".

The trace semantics $[\![-]\!]$ was designed carefully to incorporate *closure* as a basic property of the trace set of a command; each step in a trace represents the effect of a finite (possibly empty) sequence of atomic actions, and an entire trace records a fair interaction between a command and its environment. Given a conventional operational semantics, in which the transition relation $\rightarrow$ describes the effect of a single atomic action, the closed trace set semantics is based on $\rightarrow^*$, the reflexive, transitive closure of the transition relation. We also introduced an auxiliary semantics $[-]$ based on arbitrary (not necessarily closed) trace sets, in which each step represents the effect of a single atomic action, so that this semantics is based directly on the one-step transition relation. Clearly $[-]$ is a more concrete semantics than $[\![-]\!]$, distinguishing for example between **skip** and **skip**; **skip**. We therefore prefer $[\![-]\!]$, which identifies these two commands and validates many laws of program equivalence that fail in the step-by-step semantics. Nevertheless the step-by-step semantics is a key ingredient in understanding recursion. Indeed, note that the single-step transition relation $\rightarrow$ can be used to define both $\rightarrow^*$ and $\rightarrow^\omega$ (the divergence predicate), whereas $\rightarrow^*$ by itself does not determine $\rightarrow^\omega$. It is not surprising, therefore, that we needed to make a detour. The relationship between the two semantic frameworks is simple: the closed trace set semantics can be obtained by taking the quotient of the step-by-step semantics under closure equivalence.

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 [3]: 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 [8], preventing full abstraction at higher types. Recent work of Reddy [13], and of O'Hearn and Reynolds [8], 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.

Shared-variable programs are typically designed to include parallel components intended to *cooperate*, but semantically there is little distinction between cooperation and interference: both amount to patterns of interactive state change, and the only pragmatic distinction concerns whether the state changes are beneficial or detrimental to the achievement of some common goal, such as the satisfaction of some safety or liveness property. As we have shown, local variables can be used to limit the scope of interference between parallel components of a program, thus providing a form of "syntactic control of interference", somewhat in the spirit of [17, 16]. It would be interesting to see if this earlier work on syntactic control of interference in the sequential setting, together with related developments [20, 6, 7], can be adapted to the shared-variable parallel setting.

9. Acknowledgements

The work from which this paper grew began during a 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 appeared in the Proceedings of the 11th Annual IEEE Conference on Logic in Computer Science (IEEE Computer Society Press, 1996) and was later incorporated as a chapter in 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.

This work was sponsored in part by the Office of Naval Research, under Grant No. N00014-95-1-0567, and in part by the National Science Foundation, under Grant No. CCR-9412980.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] P. W. O'Hearn. A model for syntactic control of interference. *Mathematical Structures in Computer Science*, 3(4):435–465, 1993.

[7] P. W. O'Hearn, A. Power, M. Takeyama, and R. Tennent. Syntactic control of interference revisited. In *Proceedings of* 11^{th} *Conference on Mathematical Foundations of Programming Semantics*. Elsevier Science, 1995.

[8] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *J.ACM*, 47(1):167–223, 2000.

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

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

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

[12] 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.

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

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

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

[16] J. C. Reynolds. Syntactic control of interference, part 2. In *Proceedings of the* 16^{th} *International Colloquium on Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722. Springer-Verlag, Berlin, 1989.

[17] J. C. Reynolds. Syntactic control of interference. In *Conference Record of* 5^{th} *Annual ACM Symposium on Principles of Programming Languages*, pages 39–46. ACM, New York, January 1978.

[18] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

[19] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[20] R. D. Tennent. Semantics of interference control. *Theoretical Computer Science*, 27:297–310, 1983.

10. Appendix

Here we provide some of the details behind the step-by-step semantics $[-]$, and summarize some of the relevant properties, each of which can be proved by structural induction.

- For each type θ, the functor $[\theta]$ from worlds to domains is given by:

$$
\begin{array}{l}
[\mathbf{comm}]W = \wp((W \times W)^{\infty}) \\
[\mathbf{comm}](f, Q) = \lambda c.\{\alpha' \mid \text{map}(f \times f)\alpha' \in c \;\&\; \alpha' \text{ respects } Q\} \\
[\mathbf{comm}]_n(f, Q) = \lambda c.\{\alpha' \mid \text{map}(f \times f)\alpha' \in c \;\&\; \alpha' \text{ respects } Q \text{ for } n \text{ steps}\} \\
\\
[\mathbf{exp}[\tau]]W = \wp((W^{+} \times V_{\tau}) \cup W^{\omega}) \\
[\mathbf{exp}[\tau]]h = \lambda e.\{(\rho', v) \mid (\text{map} f\rho, v) \in e\} \cup \{\rho' \mid \text{map} f\rho' \in e \cap W^{\omega}\} \\
[\mathbf{exp}[\tau]]_n h = [\mathbf{exp}[\tau]]h \\
\\
[\mathbf{var}[\tau]]W = (V_{\tau} \to [\mathbf{comm}]W) \times [\mathbf{exp}[\tau]]W \\
[\mathbf{var}[\tau]]h = \lambda(a, e).(\lambda v.[\mathbf{comm}]h(av),\ [\mathbf{exp}[\tau]]he) \\
[\mathbf{var}[\tau]]_n h = \lambda(a, e).(\lambda v.[\mathbf{comm}]_n h(av),\ [\mathbf{exp}[\tau]]_n he) \\
\\
[\theta \times \theta'] = [\theta] \times [\theta'] \\
[\theta \times \theta']h = [\theta]h \times [\theta']h \\
[\theta \times \theta']_n h = [\theta]_n h \times [\theta']_n h \\
\\
[\theta \to \theta']W = \{p(-) \mid \forall h : W \to W'.\ p(h) : [\theta]W' \to [\theta']W' \;\& \\
\qquad\qquad\qquad\qquad \forall h' : W' \to W''.\ [\theta']h' \circ ph = p(h; h') \circ [\theta]h' \;\& \\
\qquad\qquad\qquad\qquad\qquad \forall n \geq 0.\ [\theta']_n h' \circ ph \sqsupseteq p(h; h') \circ [\theta]_n h'\} \\
[\theta \to \theta']hp = \lambda h' : W' \to W''.p(h; h') \\
[\theta \to \theta']_n h = [\theta \to \theta']h
\end{array}
$$

  The ordering on $[\mathbf{comm}]W$ and on $[\mathbf{exp}[\tau]]W$ is set inclusion, and this is extended componentwise and pointwise to other types as appropriate.

  For each type θ and morphism h, $[\theta]h$ is the limit of the $[\theta]_n h$, in that for all $n \geq 0$, $[\theta]_{n+1}h \sqsubseteq [\theta]_n h$, and $[\theta]h = \sqcap_{n=0}^{\infty}[\theta]_n h$.

- For each type θ the functor $\langle\theta\rangle$ from worlds to the category of complete lattices and continuous functions is given by:

$$
\begin{array}{l}
\langle\mathbf{comm}\rangle = [\mathbf{comm}] \\
\langle\mathbf{exp}[\tau]\rangle = [\mathbf{exp}[\tau]] \\
\langle\mathbf{var}[\tau]\rangle W = [\mathbf{var}[\tau]] \\
\langle\theta \times \theta'\rangle = \langle\theta\rangle \times \langle\theta'\rangle \\
\langle\theta \to \theta'\rangle W = \{p(-) \mid \forall h : W \to W'.\ p(h) : \langle\theta\rangle W' \to \langle\theta'\rangle W'\}
\end{array}
$$

  In each case the action of $\langle\theta\rangle$ on morphisms is defined exactly as for $[\theta]$.

  For each world W and each type θ the domain $[\theta]W$ is a subset of $\langle\theta\rangle W$.

- For each type θ the natural transformation stut_θ from $[\theta]$ to $[\theta]$ is defined by:

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

These definitions also make sense as natural transformations from $\langle\theta\rangle$ to $\langle\theta\rangle$.

- Whenever $\pi \vdash P{:}\theta$ is valid, $[P] : [\pi] \dot{\to} [\theta]$ is defined as follows, by structural induction:

$$\begin{array}{l}
[1]Wu = \{(w,1) \mid w \in W\} \\
[E_1 + E_2]Wu \ = \\
\qquad \{(\rho_1\rho_2, v_1 + v_2) \mid (\rho_1, v_1) \in [E_1]Wu \ \&\ (\rho_2, v_2) \in [E_2]Wu\} \\
\quad \cup \{\rho_1\rho_2 \mid \exists v_1.\, (\rho_1, v_1) \in [E_1]Wu \ \&\ \rho_2 \in [E_2]Wu \cap W^\omega\} \\
\quad \cup \{\rho \in W^\omega \mid \rho \in [E_1]Wu\} \\
[\mathbf{skip}]Wu = \{(w,w) \mid w \in W\} \\
[X{:=}E]Wu = \\
\qquad\qquad \{(\mathrm{map}\Delta_W \rho)\beta \mid (\rho, v) \in [E]Wu \ \&\ \beta \in \mathrm{fst}([X]Wu)v\} \\
\qquad\quad \cup \{\mathrm{map}\Delta_W \rho \mid \rho \in [E]Wu \cap W^\omega\} \\
[\mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2]Wu = \mathbf{if}\ [B]Wu\ \mathbf{then}\ [P_1]Wu\ \mathbf{else}\ [P_2]Wu \\
[\mathbf{while}\ B\ \mathbf{do}\ P]Wu = ([B]_{\mathtt{tt}}Wu \cdot [P]Wu)^* \cdot [B]_{\mathtt{ff}}Wu \ \cup\ ([B]_{\mathtt{tt}}Wu \cdot [P]Wu)^\omega \\
[P_1; P_2]Wu = [P_1]Wu \cdot [P_2]Wu \\
[P_1 \| P_2]Wu = \{\alpha \mid \exists \alpha_1 \in [P_1]Wu, \alpha_2 \in [P_2]Wu.\, (\alpha, \alpha_1, \alpha_2) \in \mathit{fairmerge}_W\} \\
[\mathbf{new}[\tau]\ \iota\ \mathbf{in}\ P]Wu = \{\mathrm{map}(\mathrm{fst} \times \mathrm{fst})\alpha \mid \\
\qquad\qquad\qquad\qquad\qquad \alpha \in [P](W \times V_\tau)([\pi](- \times V_\tau)u \mid \iota : (a,e))\ \& \\
\qquad\qquad\qquad\qquad\qquad \mathrm{map}(\mathrm{snd} \times \mathrm{snd})\alpha \text{ interference-free}\} \\
[\mathbf{rec}\ \iota.P]Wu = \nu p : \langle\theta\rangle W.\, \mathrm{stut}_\theta W([P]W(u \mid \iota{:}p))
\end{array}$$

In the clause for local variable declarations the “fresh variable” $(a,e) \in [\mathbf{var}[\tau]](W \times V_\tau)$ is defined by:

$$\begin{array}{l}
a = \lambda v'{:}V_\tau.\{((w,v),(w,v')) \mid w \in W \ \&\ v \in V_\tau\} \\
e = \{((w,v),v) \mid w \in W \ \&\ v \in V_\tau\}.
\end{array}$$

Non-destructivity of $[P]$, and the corresponding constructivity of $\mathrm{stut}_\theta \circ [P]$, is needed to show that the fixed point used to interpret $[\mathbf{rec}\ \iota.P]Wu$ belongs to $[\theta]W$, and to show naturality.

- For each type θ we define a natural equivalence relation clos_θ on $[\theta]$:

$$\begin{array}{rcl}
\mathrm{clos}_{\mathbf{comm}} W &=& \{(c_0, c_1) \mid c_0^\dagger = c_1^\dagger\} \\
\mathrm{clos}_{\mathbf{exp}[\tau]} W &=& \{(e_0, e_1) \mid e_0^\dagger = e_1^\dagger\} \\
\mathrm{clos}_{\mathbf{var}[\tau]} W &=& \{((a_0, e_0), (a_1, e_1)) \mid \forall v.\, (a_0 v, a_1 v) \in \mathrm{clos}_{\mathbf{comm}} W \ \&\ (e_0, e_1) \in \mathrm{clos}_{\mathbf{exp}[\tau]} W\} \\
\mathrm{clos}_{\theta\times\theta'} W &=& \{((x_0, y_0), (x_1, y_1)) \mid (x_0, x_1) \in \mathrm{clos}_\theta W \ \&\ (x_1, y_1) \in \mathrm{clos}_{\theta'} W\} \\
\mathrm{clos}_{\theta\to\theta'} W &=& \{(p_0, p_1) \mid \forall h : W \to W'.\, \forall (x_0, x_1) \in \mathrm{clos}_\theta W'.\, (p_0 h x_0, p_1 h x_1) \in \mathrm{clos}_{\theta'} W'\}
\end{array}$$

For each θ and W, $\mathrm{clos}_\theta W$ is an equivalence relation on $[\theta]W$, and

$$\forall h : W \to W'.\, \forall (x_0, x_1) \in \mathrm{clos}_\theta W.\, ([\theta]hx_0, [\theta]hx_1) \in \mathrm{clos}_\theta W'.$$

- Whenever $\pi \vdash P{:}\theta$ is valid,

$$(u_0, u_1) \in \text{clos}_\pi W \;\Rightarrow\; ([P]Wu_0, [P]Wu_1) \in \text{clos}_\theta W.$$

- For all types θ, the natural transformation $\theta^\dagger : [\theta] \dot{\rightarrow} [\![\theta]\!]$ is given by:

$$\begin{array}{l}
\textbf{comm}^\dagger W c = c^\dagger \\
\textbf{exp}[\tau]^\dagger W e = e^\dagger \\
\textbf{var}[\tau]^\dagger(a, e) = (\lambda v.\, \textbf{comm}^\dagger W(av),\; \textbf{exp}[\tau]^\dagger We) \\
(\theta \times \theta')^\dagger W(p_0, p_1) = (\theta^\dagger W p_0, \theta'^\dagger W p_1) \\
(\theta \rightarrow \theta')^\dagger W p = \lambda h : W \rightarrow W'.\, \lambda x : [\![\theta]\!]W'.\, \theta'^\dagger W'(phx).
\end{array}$$

The connection between $\theta^\dagger$ and clos_θ is expressed by:

$$\text{clos}_\theta W = \{(p_0, p_1) \mid \theta^\dagger p_0 = \theta^\dagger p_1\}.$$

Moreover, for all types θ, $\theta^\dagger \circ \text{stut}_\theta = \theta^\dagger$.

- When $\pi \vdash P : \theta$ is valid, $[\![P]\!]W(\pi^\dagger u) = \theta^\dagger W([P]Wu)$.