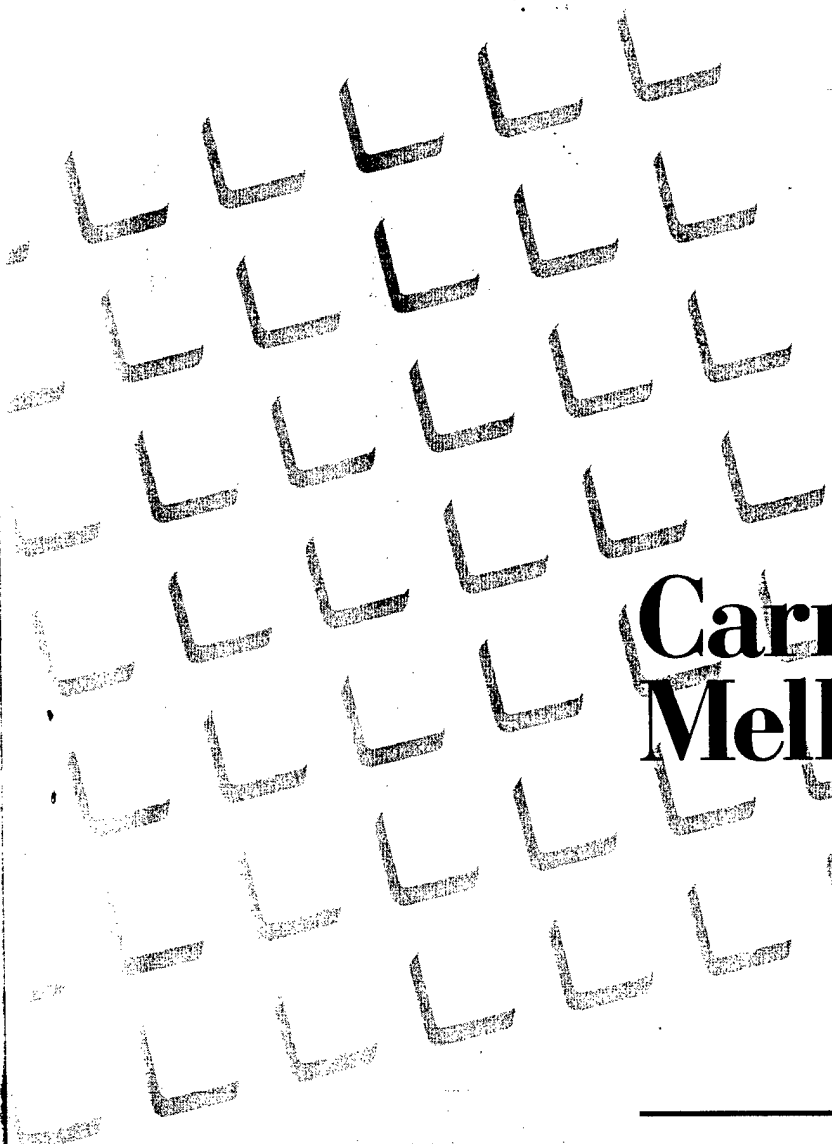
Computer Science

Idealized CSP: Combining procedures with communicating processes

Stephen Brookes

July 1997
CMU-CS-97-126



Carnegie Mellon

Idealized CSP: Combining procedures with communicating processes

Stephen Brookes

July 1997
CMU-CS-97-126

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear, Proceedings of MFPS'97, Electronic Notes in Theoretical Computer Science, Elsevier-North Holland (1997)

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

Keywords: programming language design, concurrency, semantics of programming languages, communicating processes, procedures

Abstract

Idealized CSP is a programming language combining simply typed, call-by-name procedures with asynchronous communicating processes. The language also generalizes Reynolds' Idealized Algol by adding typed channels and the ability to spawn parallel processes. Procedures permit the encapsulation of common communication protocols and parallel programming idioms. Local variables and local channel declarations provide a way to delimit the scope of interference between parallel agents. The combination of procedures and communicating parallelism raises significant semantic problems. We show—perhaps surprisingly, given the fundamental differences in underlying process model—that ideas used to model the combination of shared-variable parallelism and procedures can be adapted to the communication-based setting. This is further evidence in favor of the orthogonality of procedures and concurrency, and also shows that the shared-variable and communication-based paradigms have a lot in common, semantically. Our semantics introduces a generalization of "transition traces" and "possible worlds", incorporating an "object-oriented" treatment of channels. The semantics supports reasoning about safety and liveness properties of processes at the same time as validating natural laws of functional programming.

1 Introduction

We introduce a programming language combining a simply typed, call-by-name procedure mechanism with a generalized version of CSP[8], in which parallel processes communicate by message-passing. Our generalization of Hoare's language follows familiar lines: we allow nested parallelism, while in the original language each parallel component was required to be sequential; and we use named channels, as in *occam*, rather than process names, since this yields a more flexible communication mechanism. The inclusion of nested parallelism makes the language more uniform and causes no extra difficulties from a semantic point of view. We also allow recursive process definitions, so that processes may be created dynamically. In contrast to the original CSP, in which communication was assumed to be implemented by synchronized handshake, we assume an asynchronous model for communication: output is non-blocking, but an input request blocks until data is available. The synchronous style of communication can, nevertheless, be simulated in the asynchronous setting. To acknowledge the language's intellectual debts to Idealized Algol and CSP we call it Idealized CSP.

The combination of procedures and parallelism yields a language in which CSP-like process definitions can be encapsulated and manipulated by means of procedure declarations and calls. Procedures permit encapsulation of common communication protocols, such as the alternating-bit protocol. Local variables and local channels provide a way to delimit the scope of interference between parallel agents. For example, the following procedures, written in a syntactically sugared notation, encapsulate a common way to build (integer-carrying) buffers in CSP:

procedure *buff1(in, out)* =
  newvar[**int**] x **in while true do** ($in?x$; $out!x$);

procedure *buff(in, out)* =
  newchan[**int**] *mid* **in** *buff1(in, mid)* || *buff1(mid, out);*

In any call to *buff*, locality of the channel *mid* guarantees that the actual parameters of the call are distinct from *mid.* The correct behavior of this procedure depends crucially on the inability of the two calls to *buff1* to interact except via the local channel. Intuitively, provided *in* and *out* denote distinct channels, *buff1(in,out)* behaves like a one-place buffer and *buff(in,out)*

behaves like an unbounded buffer.

Combining procedures and communicating processes raises significant semantic problems. Indeed, most existing semantic models for CSP-like languages do not incorporate procedures, and most existing semantic models for procedures seem unsuitable for a process language like CSP. This paper shows that, despite the fundamental differences in the underlying model of computation, the ideas behind our earlier work on shared-variable parallelism can be adapted to the setting of communicating processes. In [3] we used "transition traces" to build a simple fully abstract model for a shared-variable parallel language. In [4] we showed how to incorporate a procedure mechanism based on the simply typed call-by-name λ-calculus, obtaining an idealized language called Parallel Algol. Our semantics for Parallel Algol combined transition traces with "possible worlds"[13, 15, 12] in a "modular" style, bringing out the orthogonality of procedures and shared-variable concurrency. We will show that with suitable generalization and adjustment, we can obtain a semantics for Idealized CSP by similar means. In one sense, this is perhaps not so surprising: it also seems intuitive that procedures and communication-based concurrency should be orthogonal. The surprise is that transition traces, which seem to be tailored for the shared-variable paradigm, and possible worlds, which appear best suited for modelling imperative programming, can be adapted to deal with communication-based programs. This also points out the fundamental role of transition traces as a common semantic basis for the shared-variable and communication-based paradigms.

2 Syntax

The type structure of our language is standard, essentially as in [15] but with the addition of typed channels. We use τ to range over data types and θ to range over phrase types, as specified by the following abstract grammar:

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

Data types represent sets of storable and communicable values.

Let ι range over the set of identifiers. A type environment π is a partial function from identifiers to types. Let $(\pi, \iota : \theta)$ be the type environment that

agrees with π except that it maps ι to θ.

A type judgement of form $\pi \vdash P : \theta$ is interpreted as saying that phrase P has type θ in type environment π. The collection of *valid judgements* is characterized as usual by a set of syntax-directed inference rules; Figure 1 gives a representative sample. The language contains the usual arithmetic and boolean operations, together with the usual CSP-style constructs (including input-output guarded commands, parallel and sequential composition) and the simply typed λ-calculus.

3 Semantics

The combination of procedures and communicating parallelism raises significant problems: we need a semantics which clearly brings out the potential for communication and interference between parallel commands while still supporting naive methods of reasoning based on the laws of the λ-calculus (such as the β-law, or fixed-point properties of recursive procedures). Our approach generalizes transition traces[3] and possible worlds[15, 13] to deal with channels and communication. We summarize briefly some of the key background concepts.

3.1 Transition traces

A "transition trace" is a finite or infinite sequence of pairs of states,

$$\langle s_0, s_0' \rangle \langle s_1, s_1' \rangle \ldots \langle s_n, s_n' \rangle \ldots$$

representing a generalized computation of a command during which the state is changed as indicated: steps from s_i to s_i' being caused by the command, changes from s_i' to s_{i+1} being made by the command's environment. This kind of structure is very natural for modelling shared-variable parallelism, since interference is captured precisely by state changes "across step boundaries". Transition traces have been used to give denotational semantics to a simple shared-variable language, originally by Park[14], and more recently in [3] to achieve full abstraction, by imposing certain closure conditions on trace sets. In particular, a trace set T is said to be closed under *stuttering* if every trace obtained from a trace in T by inserting steps of the form $\langle s, s \rangle$ also belongs to T; and T is closed under *mumbling* if every trace obtained from a trace

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

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

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

$$\frac{\pi \vdash h : \mathbf{chan}[\tau] \quad \pi \vdash E : \mathbf{exp}[\tau]}{\pi \vdash h!E : \mathbf{comm}}$$

$$\frac{\pi \vdash h : \mathbf{chan}[\tau] \quad \pi \vdash X : \mathbf{var}[\tau]}{\pi \vdash h?X : \mathbf{comm}}$$

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

$$\frac{\pi, \iota : \mathbf{chan}[\tau] \vdash P : \mathbf{comm}}{\pi \vdash \mathbf{newchan}[\tau]\ \iota\ \mathbf{in}\ P : \mathbf{comm}}$$

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

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

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

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

Figure 1: Some rules for type judgements

in T by replacing adjacent steps of the form $\langle s, s'\rangle\langle s', s''\rangle$ by $\langle s, s''\rangle$ is also in T. The closure conditions are designed to ensure that each step $\langle s, s'\rangle$ in a trace represents a finite (possibly empty) sequence of atomic actions. When T is a trace set we write $T^\dagger$ for its closure, the smallest closed set containing T as a subset.

3.2 Possible worlds

The main idea behind the extension to include procedures[4] is the realization that possible-worlds semantics[15, 13] and the more refined relationally parametric model of [12], used earlier to interpret a sequential Algol-like language, can be adapted to the shared-variable setting.

Instead of assuming a global set S of states, we work with a category whose objects, or possible worlds, represent states with a given store shape, and whose morphisms correspond to the introduction of local variables, or more abstractly to "expansions" of store shape. The trace semantics may then be parameterized in terms of the underlying world: each type denotes a functor from worlds into domains, and each well-typed phrase denotes a natural transformation from an environment functor to the corresponding result type functor. Because of the nature of morphisms in the worlds category, naturality imposes certain "locality" constraints, intuitively of the form that a "global" entity cannot interfere with a "local" entity. The functor category, whose objects are functors from worlds to domains, with natural transformations as morphisms, is cartesian closed; this provides a canonical way to interpret the λ-calculus.

3.3 Incorporating communications into state

In traditional accounts of the semantics of communicating processes channel names (or process names, or some similar kind of communication label) play a prominent role. For instance [5] uses "communication traces" of the form (s, ρ, s'), where s and s' are states and ρ is a sequence of labels of the form $h?n$, $h!n$ or ϵ (indicating an "internal" action such as assignment). Yet from an abstract point of view the reliance on channel names seems awkward. By analogy, in traditional storage-based models of sequential imperative programming languages *locations* play a prominent role; yet the precise nature of locations is irrelevant, and details concerning storage management may

invalidate certain natural laws of program equivalence. We therefore seek a semantics in which channel names are handled more implicitly than in traditional trace semantics.

The location problem is avoided in the possible worlds setting, because locations are only dealt with implicitly: in essence, a location corresponds to a component in the structure of a world. For instance, the world $V_{int} \times V_{bool}$ consists of states with two "locations", one capable of holding an integer and one of holding a truth value. The world $V_{bool} \times V_{int}$ is isomorphic to the above world, and therefore essentially indistinguishable. In seeking a name-free account of channels we will adopt a similar approach.

Each channel potentially carries a sequence of data values. Over the course of an entire computation an individual channel may participate in an infinite sequence of communications, but at each stage only finitely many actions have occurred. It follows that we can regard the state as a collection of components representing (ordinary) variables, together with a collection of components representing channels, each holding a finite sequence of data values. For instance, the world $V_{int} \times V_{bool} \times V_{int}^*$ represents states with one integer variable, one boolean variable, and one integer channel. In a particular state the sequence of values in the channel component represents the values that have been output to the channel and not yet taken off by an inputting process. By embedding channel histories into states in this way, we pave the way for an adaptation of the transition traces approach to the setting of communicating processes. In outline, the meaning of a process at world W is a set of finite or infinite sequences of pairs of states:

$$[\![\mathbf{comm}]\!]W = \wp^{\dagger}((W \times W)^{\infty}),$$

exactly as in our earlier semantics of shared-variable programs. Again as in the earlier model, we work with trace sets closed under stuttering and mumbling, so that each step in a trace represents a finite sequence of atomic actions. Now, however, since channels form part of the state, we can account properly for message-passing between processes. A trace of the form indicated above now represents a possible computation of a process assuming certain patterns of communication with its environment (modelled as "state changes between steps"). Assuming a fair scheduler, the behavior of a system of processes can be built by fairly interleaving traces of the individual processes, using a suitable form of interleaving that permits synchronization of matching input and output.

3.4 An object-oriented interpretation of channels

A key ingredient in the possible worlds semantics of sequential Algol is an "object-oriented" view of variables, originally proposed by Reynolds [15]: a variable of data type τ can be represented as a pair consisting of an *acceptor*, which when supplied with a value of type τ yields a *command value* whose effect describes how to "update" the variable, and an *expression value* giving the "current" value of the variable. A key innovation in our new set-up is the realization that *channels* may also be given an "object-oriented" semantics.

A channel (of data type τ) has two capabilities: one can output a value of type τ to it, which will have the effect of enqueueing this value; and one can input from the channel, which (when the channel sequence is non-empty) yields a value of the appropriate type obtained by dequeueing. Both the dequeueing and enqueueing operations can be modelled abstractly as *command values*, since — with this incorporation of channels into the state — they cause a state change. This suggests the following semantics for channel types[1]:

$$[\![\mathbf{chan}[\tau]]\!]W = (V_\tau \to [\![\mathbf{comm}]\!]W) \times ([\![\mathbf{exp}[\tau]]\!]W \times [\![\mathbf{comm}]\!]W)$$

In this paper we give a treatment of channels in which output is regarded as asynchronous (non-blocking) but an attempt to input from an empty channel will block. Alternatives are discussed briefly at the end of the paper.

3.5 Examples

Because of space limitations, we omit the semantic details in favor of some simple examples which illustrate the main ideas.

Suppose we have declared an integer variable x and an integer channel h, corresponding to a world of form $W = V_{int} \times V_{int}^*$. Let u be an environment consistent with this set-up, so that x is bound to a variable corresponding to the first component of the state, and h is bound to a channel value whose input and output capabilities involve the second component of the state. We will write $n.\rho$ and $\rho.n$ to indicate concatenation at either end of a finite list, and we write ϵ for the empty sequence.

[1] An alternative formulation, ostensibly simpler but conveying the same semantic information, is possible if we allow expressions with side-effects. In this case we could take $[\![\mathbf{chan}[\tau]]\!]W = (V_\tau \to [\![\mathbf{comm}]\!]W) \times [\![\mathbf{exp}[\tau]]\!]W$.

The command $h?x$ at world W and in environment u has traces of the form

$$\langle (v, n.\rho), (n, \rho) \rangle$$

for $v, n \in V_{int}, \rho \in V_{int}^*$. The command $h!(x+1)$ has traces of the form

$$\langle (n, \rho), (n, \rho.(n+1)) \rangle,$$

and the command $h?x \| h!(x+1)$ has traces including the following forms:

- $\langle (v_0, n_0.\rho_0), (n_0, \rho_0) \rangle \langle (v_1, \rho_1), (v_1, \rho_1.(n_1+1)) \rangle$, representing input followed by output, possibly after a state change from the environment;
- $\langle (v_0, \rho_0), (v_0, \rho_0.(v_0+1)) \rangle \langle (v_1, n_1.\rho_1), (n_1, \rho_1) \rangle$, representing output followed by input, again possibly after action by the environment;
- $\langle (v, \epsilon), (v+1, \epsilon) \rangle$, representing a synchronized input-output, which behaves like a "distributed assignment". This trace actually arises by mumbling together an output $\langle (v, \epsilon), (v, v+1) \rangle$ and an input $\langle (v, v+1), (v+1, \epsilon) \rangle$.

Traces in which a state change occurs across a step boundary, such as the first two cases above, reflect the potential for interaction with another process executing concurrently.

The above discussion did not cover the case when a process wants to perform input but the intended channel is empty. It seems reasonable to model this situation as a form of *busy waiting*, since such a process will keep waiting for an output to the channel by another process; while waiting, the process never changes the state, and the waiting continues provided the channel stays empty. In trace-theoretic terms this amounts to a form of infinite stuttering. Thus, revisiting the above example, the traces of $h?x$ also include infinite stuttering traces of the form

$$\langle (v_0, \epsilon), (v_0, \epsilon) \rangle \langle (v_1, \epsilon), (v_1, \epsilon) \rangle \ldots \langle (v_n, \epsilon), (v_n, \epsilon) \rangle \ldots$$

This is, of course, consistent with our view of output as non-blocking, input as blocking when the intended channel is empty.

3.6 Semantic issues

The discussion above indicates how to model input, output, and parallel composition. As usual, sequential composition is modelled by concatenation of traces. Assignment, conditional and while-loops may be handled in the standard way too, as in our earlier treatment of shared-variable parallelism. Recursion and while-loops are interpreted via *greatest fixed points*[16] in order to deal appropriately with both finite and infinite traces.

As in CSP our language includes a form of "external" choice; however, because of our assumption that output is non-blocking, it seems most natural to permit the use of external choice only when the guards involve input. It is then straightforward to model a combination such as

$$(a?x \to P_1) \Box (b?x \to P_2),$$

which can either input on a and behave like P_1, or input on b and behave like P_2, or busy-wait while a and b are *both* empty. In contrast an "internal" choice

$$(a?x \to P_1) \sqcap (b?x \to P_2)$$

can busy-wait if *either* a or b is empty.

Local channel declarations can be handled rather simply using an extension of the idea used in our shared-variable model to deal with local variables. The traces of **newchan**$[\tau]$ h **in** P at world W and environment u are obtained from traces of P in world $W \times V_\tau^*$ and suitably adjusted environment u', by projecting onto the W-components in each step, assuming that initially the V_τ^*-component is ϵ and that this component never changes across step boundaries. In other words, only the *locally interference-free* traces of P contribute in this construction, where we say that a trace is locally interference-free if it has the form

$$\langle (w_0, \epsilon), (w_0', \rho_1) \rangle \langle (w_1, \rho_1), (w_1', \rho_2) \rangle \ldots \langle (w_n, \rho_n), (w_n', \rho_{n+1}) \rangle \ldots$$

(either finite or infinite). When applied to the example discussed above, this shows that

$$\textbf{newchan}[\tau]\ h\ \textbf{in}\ (h!e \| h?x)$$

has the same traces (modulo stuttering and mumbling) as the assignment $x{:=}e$, as expected. Moreover, in this interpretation we also have

$$\textbf{newchan}[\tau]\ h\ \textbf{in}\ (h!0; P)\ =\ P$$

if h does not occur free in P. Note also that

$$\mathbf{newchan}[\tau]\ h\ \mathbf{in}\ (h?x; P)$$

has only infinite stuttering traces, because of the unrequited request for input. Again, notice how these laws reflect our assumptions that an attempt to input from an empty channel is blocked but output is asynchronous.

3.7 Category-theoretic issues

In Oles' category a world is a countable set W, typically a product of the form $V_1 \times \cdots \times V_k$, whose elements (states) specify values for the finitely many variables currently in scope. A morphism from world W to world X is a pair (f, Q) in which $f : X \to W$ is a function, Q is an equivalence relation on X, and f puts each equivalence class in bijection with W. Intuitively, X is a set of "large" states each extending a "small" state in W; f maps each large state to the corresponding small state; and Q identifies all pairs of large states with identical "extra" structure. For each pair of objects W and V there is a canonical "expansion" morphism from W to $W \times V$ of the form $(\mathrm{fst} : W \times V \to V, Q)$, where $(w, v)Q(w', v') \iff v = v'$. Every set-theoretic isomorphism, such as $swap : W \times V \to V \times W$, yields an isomorphism of worlds when equipped with the obvious universal equivalence relation (so that there is a single equivalence class). Oles showed that every morphism of worlds is expressible as the composition of an isomorphism with an expansion.

In adapting the possible worlds approach to deal with channels we need to generalize the nature of worlds and of morphisms. A typical world in the above discussion had a structure, up to isomorphism, like

$$(V_1 \times \cdots \times V_k) \times (H_1^* \times \cdots \times H_n^*)$$

where each V_i and H_j is the set of values for some data type τ. Expansion morphisms still account for local variable declarations adequately, and it is tempting to try to model a local channel declaration by means of an expansion introducing an extra V^* component to the state. However, we need to refine the notion of morphism so that we only permit isomorphisms that respect the queue structure of channel components. To show why this degree of care is needed, consider the list-reversal function from V_τ^* to V_τ^*. Even

though this is a set-theoretic isomorphism, it does not make sense computationally as a morphism of worlds: it is not sensible to expect a command's meaning to be essentially unchanged if the contents of all channel queues are reversed. (For example, consider an input command $h?x$.) The presence of such a morphism of worlds would thus prevent us from interpreting input and output commands as natural transformations, precluding the development of a functor category semantics. Instead we must constrain our choice of morphisms so that the only allowed isomorphisms between channels are those that respect the prefix ordering on channel contents.

To generalize the Oles category in a satisfactory manner, building in enough extra structure to permit an abstract treatment of channels, we therefore work with worlds equipped with a partial order (based on the use of the prefix ordering on sequences). Without loss of generality we may restrict attention to countable distributive posets W in which all elements are finite and dominate only finitely many other elements. Posets formed as products of flat domains and prefix-ordered sets of finite sequences have these properties, so that this class of posets is general enough to include the worlds arising in our semantic definitions. We take as morphisms from $(W, \leq_W)$ to $(X, \leq_X)$ all pairs (f, Q) in which $f : X \to W$ is monotone and induces an *order-isomorphism* with W on each Q-class. This clearly generalizes the Oles category in a natural way and does not permit morphisms that violate the queue discipline. Every morphism can be expressed as the composition of an expansion with an order-isomorphism.

Following the line of development in [13] we then show that, using our category $\mathbf{W}$ of worlds and the category $\mathbf{D}$ of domains and continuous functions, the functor category $\mathbf{D}^{\mathbf{W}}$ is cartesian closed. This then permits us to use the ccc structure to interpret the λ-calculus fragment of our language. As usual, each phrase type denotes a functor from worlds to domains, and each well-typed phrase denotes a natural transformation from an environment functor to a result functor. Naturality enforces certain locality properties, and as a consequence the model validates laws of equivalence such as

$$\begin{aligned}&\textbf{newchan}[\tau_1]\ h_1\ \textbf{in}\ \textbf{newchan}[\tau_2]\ h_2\ \textbf{in}\ P\\ &\quad = \textbf{newchan}[\tau_2]\ h_2\ \textbf{in}\ \textbf{newchan}[\tau_1]\ h_1\ \textbf{in}\ P,\end{aligned}$$

expressing the property that the order of declaration of channels is irrelevant, and

$$\textbf{newchan}[\tau]\ h\ \textbf{in}\ (P_1 \| P_2) \ = \ (\textbf{newchan}[\tau]\ h\ \textbf{in}\ P_1) \| P_2,$$

provided h does not occur free in P_2.

4 Examples

4.1 Buffers

The semantics correctly handles the examples discussed earlier. Thus the one-place buffer procedure *buff1*(*left*, *right*), when called at world $V_{int}^* \times V_{int}^*$ in a suitable environment, has the trace

$$\begin{array}{l} \langle(0,\epsilon),(\epsilon,\epsilon)\rangle \\ \quad \langle(1,\epsilon),(1,0)\rangle \\ \quad\quad \langle(1,0),(\epsilon,0)\rangle \\ \quad\quad\quad \langle(2,0),(2,01)\rangle \\ \quad\quad\quad\quad \ldots \end{array}$$

representing input of 0, output of 0, input of 1, and so on. It also has a busy-wait trace $\langle(\epsilon,\epsilon),(\epsilon,\epsilon)\rangle^\omega$, representing its behavior when the input channel is persistently empty. The unbounded buffer process *buff*(*left*, *right*) also has these traces, but in addition has traces such as

$$\begin{array}{l} \langle(0,\epsilon),(\epsilon,\epsilon)\rangle \\ \quad \langle(1,\epsilon),(\epsilon,\epsilon)\rangle \\ \quad\quad \langle(2,\epsilon),(2,0)\rangle \\ \quad\quad\quad \ldots \end{array}$$

showing the ability to input two items before outputting one. Note that the unbounded buffer process cannot accept input forever without eventually yielding an output; this liveness property is captured by the use of fairmerge.

4.2 Prime numbers

Another traditional example is the Sieve of Eratosthenes, involving a procedure *sieve* of type **expint** $\times$ **chan**[**int**] $\rightarrow$ **comm** and a procedure *filter* of type **exp**[**int**] $\times$ **chan**[**int**] $\times$ **chan**[**int**] $\rightarrow$ **comm**:

procedure *filter*$(p,\ in,\ out) =$
  new[**int**] x **in while true do** $(in?x;$ **if** x **mod** $p \neq 0$ **then** $out!x)$;
procedure *sieve*$(p, c) =$
  $(c!p;$ **newchan**[**int**] h **in** *filter*$(p, h, c) \parallel$ *sieve*$(p+1, h))$;

If c is an integer-carrying channel variable the call $sieve(2, c)$ results in the outputting of the prime numbers in ascending order on this channel. Note that each recursive call to *sieve* introduces new parallel processes sharing a local channel, and each call to *filter* makes use of a local variable to hold the integer currently being tested for divisibility. Actually this implementation of the sieve method is rather inefficient, creating a "filter" even for non-prime numbers. Here is an alternative version in which this inefficiency is avoided:

```
procedure sift(in, out) =
  newchan[int] h in
  new[int] p in
    begin
      in?p; out!p;
      filter(p, in, h) || sift(h, out)
    end
procedure nats(k, c) = (c!k; nats(k + 1, c));
procedure primes(c) =
  newchan[int] h in nats(2, h) || sift(h, c)
```

The phrase $sieve(2, c)$, evaluated in world V^*_{int} with c bound to the obvious channel, has the trace

$$\langle \epsilon,\ 2\rangle\langle 2,\ 2.3\rangle\langle 2.3,\ 2.3.5\rangle \ldots$$

as expected, corresponding to the ability to output the prime numbers in ascending order. The alternative version $primes(c)$ also has this trace.

4.3 Concurrent queues

The following examples illustrate two isomorphic implementations of a "concurrent object" that represents an integer queue equipped with methods for enqueueing and dequeueing. Each implementation involves the use of a local channel. The second implementation uses integer negation to "code" and

"decode" all transmitted items.

```
newchan[int] h in
  procedure put(y) = h!y;
  procedure get(z) = new[int] x in (h?x; z:=x);
  begin
    P(put, get)
  end
```

```
newchan[int] h in
  procedure put(y) = h!(−y);
  procedure get(z) = new[int] x in (h?x; z:=(−x));
  begin
    P(put, get)
  end
```

These two phrases, in which P is a free identifier of type

$$(\mathbf{exp[int]} \to \mathbf{comm}) \times (\mathbf{var[int]} \to \mathbf{comm}) \to \mathbf{comm},$$

are semantically equivalent. This can be established by appealing to naturality of $[\![P]\!]$ with respect to the obvious isomorphism built from $\mathrm{id}_W \times \mathrm{map}(\lambda n.(-n))$ on $W \times V_{int}^*$.

4.4 The Sleeping Barber

Next we give an example of a solution to one of the classic synchronization problems from the literature, the so-called Sleeping Barber Problem [7, 1]. Imagine a barber's shop in which a solitary barber repeatedly cuts hair, one customer at a time, and sleeps when not busy. If a customer enters the premises and finds the barber sleeping he wakes the barber and takes his place at the barber's chair for a haircut, after which the barber gets paid and the customer leaves; if another customer is waiting the barber continues operating, otherwise he goes back to sleep. A customer arriving while the barber is busy will wait. We can model this set-up in Idealized CSP as follows. Since the local channels used here are intended purely for synchronization, we use a slightly abbreviated form of syntax for input and output (writing *enter?*

and *hello!*, for example). The procedure *visit* represents the protocol followed by a customer; the argument c represents what the customer intends to do while having his hair cut. Similarly the procedure *cut* models the protocol used by the barber, and its argument represents the task to be performed by the barber during haircut:

newchan *enter*, *leave*, *hello*, *bye*
procedure *visit*(c) = (*enter*?; *hello*!; c; *leave*?; *bye*!);
procedure *cut*(c) = (*enter*!; *hello*?; c; *leave*!; *bye*?)
in P(*visit*, *cut*)

For instance, if P is instantiated as

λ(*visit*, *cut*).
(**while true do** *cut*(b)
|| **while true do** *visit*(c_0)
|| **while true do** *visit*(c_1))

then the above phrase becomes semantically equivalent to

while true do $(b \parallel (c_0$ **or** $c_1))$,

where c_0 **or** c_1 is a command that chooses, non-deterministically, to behave either like c_0 or like c_1.

Note that this program structure does not prevent individual starvation – there are fair executions of this program in which c_1 never occurs, so that the second customer never gets his hair cut. To avoid starvation one can modify this example to make use of "tickets", so that on entering the barber shop a customer must take a numbered ticket, at the same time incrementing a local counter to prevent multiple uses of the same ticket number. We then modify the customer definition to include a wait until the assigned ticket is "next". One way to achieve this employs a "counter" maintained by the barber. We leave the details to the reader.

4.5 A sorting network

Here is a well known way to construct a network of merge processes for sorting a sequence of integers [1]. We suppose that the sequence to be sorted is presented along a channel *in*, terminated by the special value `EOS`. When

$n \geq 0$ and *in* and *out* are distinct integer channels, $sort(n, in, out)$ reads in 2^n integers from channel *in* and outputs the corresponding sorted sequence on *out*, followed by the EOS signal. First we specify the merge procedure, of type

$$\textbf{chan[int]} \times \textbf{chan[int]} \times \textbf{chan[int]} \rightarrow \textbf{comm},$$

as follows:

$$\begin{array}{l}
\textbf{procedure } merge(a, b, c) = \\
\quad \textbf{new[int]} \ v_1, v_2 \ \textbf{in} \\
\quad\quad (a?v_1;\ b?v_2; \\
\quad\quad \textbf{while } v_1 \neq \texttt{EOS} \ \& \ v_2 \neq \texttt{EOS} \ \textbf{do} \\
\quad\quad\quad \textbf{if } v_1 \leq v_2 \ \textbf{then } (c!v_1;\ a?v_1) \ \textbf{else } (c!v_2;\ b?v_2); \\
\quad\quad \textbf{while } v_1 \neq \texttt{EOS} \ \textbf{do } (c!v_1;\ a?v_1); \\
\quad\quad \textbf{while } v_2 \neq \texttt{EOS} \ \textbf{do } (c!v_2;\ b?v_2); \\
\quad\quad c!\texttt{EOS})
\end{array}$$

The sorting procedure, of type $\textbf{exp[int]} \times \textbf{chan[int]} \times \textbf{chan[int]} \rightarrow \textbf{comm}$, is then:

$$\begin{array}{l}
\textbf{procedure } sort(n,\ in,\ out) = \\
\quad \textbf{if } n = 0 \ \textbf{then new[int]} \ v \ \textbf{in } (in?v;\ out!v;\ out!\texttt{EOS}) \ \textbf{else} \\
\quad\quad \textbf{newchan[int]} \ in_1,\ in_2 \ \textbf{in} \\
\quad\quad\quad sort(n-1,\ in,\ in_1) \parallel sort(n-1,\ in,\ in_2) \parallel merge(in_1,\ in_2,\ out)
\end{array}$$

Correctness of this procedure can be shown by induction on n.

5 Related and Future Work

The semantics outlined above is, when restricted to the CSP-like subset of the programming language, close in spirit to the "communication traces" semantics described in [5]. This earlier semantics was fully abstract with respect to a simple notion of program behavior. It seems likely that, at least at ground types, full abstraction can be achieved for Idealized CSP by imposing certain reasonable closure conditions on trace sets, akin to the use of "stuttering" and "mumbling"[3], and ensuring that the language includes a suitable form of conditional atomic action.

Our semantics treats channels in a more implicit manner than earlier trace-based approaches such as [5]. Indeed, these earlier semantics assume that a network of processes behaves as if it really executes in a step-by-step manner according to some interleaving of atomic actions, since a trace typically represents an interleaving of the histories of all channels. However, when modelling a network of processes it seems natural to want to reason about the histories of separate channels as separate entities, rather than reasoning about a single combined history. A transition trace does not blend the histories on separate channels into a single sequence, and the internal sequencing of a trace can accurately reflect information about the relative order of activity on different channels during computation. Thus our semantics is closer in spirit to "true concurrency" than traditional trace models.

Our treatment of deadlock was rather straightforward, but might be criticized on the grounds that we equate deadlock with busy-waiting. Nevertheless such a busy-wait interpretation is consistent with an operational notion of behavior in which we are allowed to observe the state periodically during execution, as well as to observe (successful) termination. A process waiting for input would then be indistinguishable from one executing a busy loop. It would be interesting to see if the "fair failures" semantics[6, 10] of CSP can also be adapted to the procedural setting.

Our framework appears to be robust enough to handle a variety of paradigms of communicating process, ranging from synchronized communication (as in CSP) to asynchronous communication using unbounded buffers (as shown here); with suitable adjustments, it should even be possible to model asynchronous communication with bounded buffers. In this paper we have allowed processes to share state. Our framework can also handle the case where processes are required to have disjoint states, as in the original CSP. The main difference is that when states of parallel processes can be assumed disjoint it makes sense to combine states in a tensor-like manner when forming a parallel composition, rather than assuming that the component processes share the same state. It is also worth noting that our rather abstract object-oriented view of channels is general enough to model a wide variety of communication mechanisms, such as "lossy" channels which may lose or duplicate data. This suggests again that the ideas presented here may prove to be more widely applicable.

The trace semantics given here for Idealized CSP may also be recast into a parametric setting, taking account of relations between worlds[12]. This

permits an elegant generalization of the principle of representation independence, familiar from the use of abstract datatypes and modules in sequential programming, to the CSP-like setting. Recent work of Lazic and Roscoe[11] has shown that relational parametricity can be used to tame the combinatorial explosion inherent in applying model-checking techniques to networks of communicating processes. Their use of parametricity seems fundamentally different from our use of possible worlds and relations between worlds; we plan to investigate the connections between their approach and ours. A related issue is polymorphism; it is obvious that some of our examples, such as buffer procedures, may be given a sensible polymorphic type, in this case $\forall\tau.(\mathbf{chan}[\tau] \times \mathbf{chan}[\tau] \to \mathbf{comm})$. We plan to examine the consequences of allowing polymorphic types, at least permitting quantification over datatypes as shown here. The use of this form of type indicates a "data independence" property of the phrase in question: a buffer behaves the same way regardless of the type of its contents.

Our examples involving concurrent queues illustrate a methodology for using local variables and channels together with procedures that operate on them to achieve an "object-oriented" style, in which the rest of the program is only permitted to interact with the local data by calling one of the supplied procedures. In the concurrent setting, this technique can be used to mimic the use of "monitors" [9, 2], originally proposed as a concept for structuring the design of operating systems. Thus our semantics can be used to reason about the correctness of classic algorithms from the operating systems literature.

References

[1] Andrews, Gregory R., **Concurrent Programming**, The Benjamin/Cummings Publishing Company, Inc. (1991).

[2] Brinch Hansen, P., *Concurrent programming concepts*, ACM Computing Surveys, Vol. 5, No. 4, 223-245 (December 1973).

[3] Brookes, Stephen., *Full abstraction for a shared-variable parallel language*, Information and Computation, Vol. 127, No. 2, 145–163 (June 1996).

[4] Brookes, Stephen, *The essence of Parallel Algol*, Proc. 11th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1996) 164–173.

[5] Brookes, Stephen, *Fair communicating processes.* In A. W. Roscoe, editor, **A Classical Mind: Essays in Honour of C. A. R. Hoare**, Prentice-Hall International (1994), 59–74.

[6] Brookes, Stephen and Older, Susan, *Full abstraction for strongly fair communicating processes*, Proc. 11th MFPS (1995), ENTCS vol. 1, Elsevier Science B. V.

[7] Dijkstra, E. W., *Cooperating sequential processes.* In F. Genuys, editor, **Programming Languages**, Academic Press, New York, 43–112 (1968).

[8] Hoare, C. A. R., *Communicating Sequential Processes*, Comm. ACM, 21(8):666–677 (1978).

[9] Hoare, C. A. R., *Monitors: an operating system structuring concept*, Comm. ACM, 17(10):549–557 (1974).

[10] Older, Susan, *A Denotational Framework for Fair Communicating Processes*, Ph.D. thesis, Carnegie Mellon University, (January 1997).

[11] Lazic, Ranko and Roscoe, Bill, *Using Logical Relations for Automated Verification of Data-independent CSP*, Proc. Oxford Workshop on Automated Formal Methods (1996), to appear, ENTCS, Elsevier Science B. V.

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

[13] Oles, Frank. J., *A Category-Theoretic Approach to the Semantics of Programming Languages*, Ph.D. thesis, Syracuse University, 1982.

[14] Park, D., *On the semantics of fair parallelism.* In D. Bjørner, editor, **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86 (1979), 504–526.

[15] Reynolds, J. C., *The essence of Algol.* In van Vliet and de Bakker, editors, **Algorithmic Languages**, North-Holland, Amsterdam (1981), 345-372.

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