Concurrent Separation Logic for Pipelined Parallelization

Christian J. Bell, Andrew W. Appel, and David Walker

Princeton University, Computer Science Department,
35 Olden Drive, 08540-5233 Princeton, New Jersey
{cbell,dpw,appel}@cs.princeton.edu
http://www.cs.princeton.edu/

Abstract. Recent innovations in automatic parallelizing compilers are showing impressive speedups on multicore processors using shared memory with asynchronous channels. We have formulated an operational semantics and proved sound a concurrent separation logic to reason about multithreaded programs that communicate asynchronously through channels and share memory. Our logic supports shared channel endpoints (multiple producers and consumers) and introduces *histories* to overcome limitations with local reasoning. We demonstrate how to transform a sequential proof into a parallelized proof that targets the output of the parallelizing optimization DSWP (Decoupled Software Pipelining).

1 Introduction

We have created an operational semantics and a concurrent separation logic (CSL) to reason about the correctness of programs that share memory and use buffered channels to synchronize processes. These channels can be used directly by the programmer or automatically by a parallelizing compiler and are not restricted to point-to-point communication; multiple producers and multiple consumers may asynchronously access a channel. We have proved our CSL sound with respect to the operational semantics. Furthermore, we demonstrate how certain proofs of correctness for a sequential program can be used to generate a related proof (that maintains the original specification) for the parallelized output of an optimization.

CSL [9] is an extension of separation logic (SL), which is an extension of Hoare logic. SL facilitates local reasoning about resources used by a program, so that when analyzing a region of code, we may assume that actions made by the rest of the program cannot interfere. This is encapsulated in the "frame rule" of SL. Similarly, CSL facilitates local reasoning about resources in a concurrent program so that we can analyze just one process while assuming that other processes cannot interfere. CSL has already been used to reason about programs that use critical sections and locks [9][6].

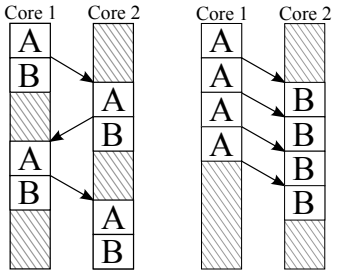
Our logic can be used in proofs about compilers. Leroy and others proved correct compilers for sequential programs; Hobor et al. outlined how such proofs can be extended to concurrent programs [8][6]. We have designed our logic to

be capable of extending these certified compilers to handle programs that use channels. Proofs in our logic can also be used in proof-carrying code frameworks.

1.1 Parallelizing transformations

A parallelizing compiler attempts to optimize a program by automatically partitioning sequential code into multiple threads. A classic example of this is the DOALL optimization, which parallelizes while-loops that have no loop-carried dependencies by distributing the iterations among multiple threads. While DOALL has had some success, particularly in scientific and numerical computing, it is common for programs to have loop-carried dependencies, thus many programs cannot benefit from DOALL.

Another optimization is DOACROSS, which can handle loop-carried dependencies. This optimization partitions iterations among several threads, but also transmits dependencies between the threads with the hope that there is a significant task within the loop that can overlap with other iterations regardless of the dependency. Figure 1a shows an example trace of a loop where each iteration is composed of tasks A and B; A and B both depend on A, but A and B do not depend on B. Because the dependencies are transmitted bidirectionally between threads, any latency in communication or an iteration stalling will cause the entire computation to stall. Therefore, DOACROSS often does not yield significant performance gains.



(a) DOACROSS (b) DSWP

Fig. 1: Trace of DOACROSS vs. DSWP. Arrows depict data flow, control/data dependencies, and communication latency.

Pipelined parallelism identifies code that can be partitioned into tasks that have acyclic dependencies. Tasks that produce dependencies can run ahead of tasks that consume them, and tasks with no dependencies between them run in parallel. Such parallelism is often leveraged at the instruction level in hardware.

Decoupled Software Pipelining (DSWP) is a compiler optimization that leverages pipelined parallelism [15][12], illustrated in Figure 1b. The dependencies are communicated between threads using asynchronous channels, which can be implemented as a shared queue in memory or in hardware [13]. Unlike DOACROSS, communication latencies and stalls will only affect consuming threads, allowing the producing threads to work ahead. DSWP is capable of a significant performance increase: in the SPEC CINT2000 benchmark suite, DSWP yielded a geometric mean speedup of 5.54 with a geometric mean of 17 threads [1].

In Section 2 we will show how to parallelize a wide class of programs. In Section 4 we will show how to transform SL proofs of sequential programs into CSL proofs of DSWP-parallelized programs.

```
while c(p) (
    b(p);
    p:= a(p)
)
```

Fig. 3: Running example

```
while p ≠ nil (
    t:= [p.val];
    [p.val]:= t + 1;
    p:= [p.next]
)
```

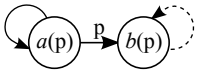
Fig. 4: Instance of running example

```
( produce d p;          ||  consume d p';
  while c(p) (          ||  while c(p') (
      p:= a(p);         ||      b(p');
      produce d p       ||      consume d p'
  )                     ||  )
)
```

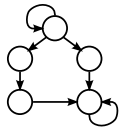
Fig. 5: Parallelized while-program

2 Parallelizing a Program

To illustrate our operational semantics and CSL, we consider the class of programs in Figure 3, where $a(\mathtt{p})$ and $b(\mathtt{p})$ represent blocks of instructions that use at least variable p. We assume operation $a(\mathtt{p})$ does not depend on $b(\mathtt{p})$, $b(\mathtt{p})$ depends only on $a(\mathtt{p})$ exactly through variable p, and $b(\mathtt{p})$ may or may not depend on itself through variables other than p; illustrated in Figure 2a. Figure 4 is an example of such a program. The dependencies in Figure 2a also generalize over more complicated dependencies such as in Figure 2b. In practice, tasks $a(\mathtt{p})$ and $b(\mathtt{p})$ are significant computations.



(a)



(b)

Fig. 2

This program is *not* a candidate for DOALL optimization because of the loop-carried dependencies from $a(\mathtt{p})$ to $a(\mathtt{p})$ and possibly from $b(\mathtt{p})$ to $b(\mathtt{p})$. However, because the dependency between $a(\mathtt{p})$ and $b(\mathtt{p})$ is in one direction, we can apply DSWP to generating the program in Figure 5.

This parallelized program, where we have two processes communicating using instructions `produce` and `consume`, computes the same values as Figure 3. To send the value of expression e through channel d (pushing the value onto the back of the channel's queue), we write `produce` d e. To receive a value (from the front of the channel's queue) into variable x, we write `consume` d x.

3 CSL with Asynchronous Channels

Our logic is an extension of SL with channel endpoints and the heap as resources. We treat channels as a way to transmit both values and resources. While a low-level view of a program may simply see integers being sent through a channel, in our logic those integers may have further meaning. For example, they can be channel identifiers or pointers. When transmitting a pointer, we can bundle the

knowledge that it is a pointer (with permission to dereference it) as a resource, and send the resource along with the pointer value.

To control how the channels are used, we assign resource invariants to each channel to act as a protocol, to which producing and consuming processes must adhere. A resource invariant is a predicate that must always hold on the state of the channel. Resource invariants were introduced to CSL by O'Hearn and have been used to control how critical sections and locks are used [9][6]. Producers use the resource invariant to determine which resources they must give up when sending a value. Consumers use the resource invariant to determine which resources they gain by receiving a value. For example, a resource invariant could state that each value in the channel points to an integer: producers must show that each value sent is such a pointer and give up the resource to dereference it, and consumers use the resource invariant to show that each received value is a valid pointer that can be dereferenced.

3.1 Channel endpoint histories

A main property of CSL is that we reason about the correctness of a process independently of all other processes. When a process sends a value to another process through a channel, neither knows what the other has or will do to the value beyond what the resource invariant states. This is enough for memory safety; access to a shared pointer can be transferred between processes to prevent race conditions. However, resource invariants cannot show that values transmitted by a producing process are correctly summed by a consuming process. We also cannot use them to ensure that every pointer sent through a channel is eventually deallocated. Local reasoning in CSL prevents us from reasoning about these behaviors from the perspective of just one process. To overcome this limitation, we delay such reasoning, by recording a *history* of the values transmitted through each endpoint, until the processes synchronize.

Histories in point-to-point communication. Consider two processes: a producer and consumer. The producer sends 1 followed by 2, so its history is $[2, 1]$. The consumer receives two values and stores them into variable x, then y. Because the consumer cannot know what the producer sent, its history is simply $[y, x]$. When the two processes join, we know that $[2, 1]$ was sent and that $[y, x]$ was received, so $[y, x] = [2, 1]$, or $y = 2$ and $x = 1$.

Histories with multiple producers and consumers. We allow a channel endpoint to be split among multiple producers or consumers. Each piece of an endpoint records its own *local* history, and when all the pieces of an endpoint are joined, the history is *global*. Recording histories through a piece of an endpoint is trickier than recording the history in the full endpoint because the order in which the processes send/receive from the channel is nondeterministic. For this reason, we record *sets* of possible histories through each endpoint.

$$
\begin{array}{lr}
A ::= \ d_\pi H! & \text{Share } \pi \text{ of the produce endpoint of channel } d, \text{ with histories } H \\
\mid d_\pi H? & \text{Share } \pi \text{ of the consume endpoint of channel } d, \text{ with histories } H \\
\mid A[\mathsf{p}\colon d \mathrel{+}= v] & A \text{ holds after appending } v \text{ to } d\text{'s local produce histories} \\
\mid A[\mathsf{c}\colon d \mathrel{+}= v] & A \text{ holds after appending } v \text{ to } d\text{'s local consume histories} \\
\mid \mathrm{PHist}\ d\ H & \text{The local produce histories of channel } d \text{ are } H \\
\mid \mathrm{CHist}\ d\ H & \text{The local consume histories of channel } d \text{ are } H \\
\mid e \Downarrow v & \text{The expression evaluates to value } v \\
\mid e_1 \mapsto e_2 \ \mid \ A_1 * A_2 \ \mid \ A_1 \mathrel{-\!\!*} A_2 \ \mid \ A_1 \implies A_2 \ \mid \ e_1 = e_2 \ \mid \ \mathrm{emp} \ \mid \ B &
\end{array}
$$

Fig. 6: Predicates

Consider the case of multiple processes producing through the same channel. Each producer records the sequence of the values it sends, but not what other producers send. When two of the processes join, we do not know the order of one history with respect to the other. Thus the combined endpoint records all possible orderings by interleaving the two histories; the actual order in which the values were sent must be within this set. When a new value is subsequently produced through the endpoint, it is appended to each history in the set.

When two sets of histories join, we compute the new set of histories by interleaving every pair of histories from the two sets. We call this operation a *merging* of the histories. Every history in a set of histories is a permutation of all other histories in the set; merging preserves this behavior. For multiple consumers, we record histories in the same way.

Example 1 (Multiple consumers). Assume we have one producing process and two consuming processes, each with an initial set of histories equal to $\{\mathrm{nil}\}$ (none have sent or received any values). The producer sends 1, 2, 3, then 4. Consumer 1 receives two values, w then x; consumer 2 receives one value, y; then they join and one more value, z, is consumed. The resulting sets of histories are:

Producer	Consumer 1	Consumer 2	After C1 & C2 join, then consume z
$\{[4,3,2,1]\}$	$\{[x,w]\}$	$\{[y]\}$	$z::(\{[x,w]\} \between \{[y]\}) =$ $z::\{[x,w,y],[x,y,w],[y,x,w]\} =$ $\{[z,x,w,y],[z,x,y,w],[z,y,x,w]\}$

The merged ($\between$) consume histories cannot exactly reconstruct the order in which the two consumers received values with respect to each other. They can show, however, that $z = 4$ and $z > x > w$.

3.2 Predicate logic

Figure 6 lists some predicates used in our logic. Metavariable A is a predicate that ranges over formulae, e is an expression, H is a set of histories (each a list of values), and d is a channel name. π is a fractional share of a resource that ranges over $[0, 1]$, where $\pi > 0$ grants permission for [shared] use of the resource

and $\pi = 0$ does not. A value is either an integer or list of values. The predicates in the last line are conventional: separating conjunction, separating implication, implication, expression equality, no share of any resources, and a pure logical formula. The logic has universal and existential quantification ranging over values. We use the forcing relation $r; s \models A$ to state that predicate A holds under environment s and exactly resources r (resources are defined in Section 5.1). Predicate A_1 entails A_2 if $\forall r, s.\ r; s \models A_1 \implies r; s \models A_2$, written $A_1 \vdash A_2$.

Predicates PHist/CHist hold for any resource if the produce/consume history is exactly H. They are used as side conditions for some of the Hoare rules.

Channel endpoints interact with the separating conjunction as follows:

$$\begin{aligned} d_{\pi_1} H_1! * d_{\pi_2} H_2! &\iff d_{\pi_1+\pi_2}(H_1 \between H_2)! \\ d_{\pi_1} H_1? * d_{\pi_2} H_2? &\iff d_{\pi_1+\pi_2}(H_1 \between H_2)? \end{aligned}$$

Example 2. Assume that Example 1 uses channel d and the two consumers have permissions π_1 and π_2. Just before the consumers join, they have resources $d_{\pi_1}\{[x, w]\}?$ and $d_{\pi_2}\{[y]\}?$ respectively. After joining, their resources are:

$$\begin{aligned} d_{\pi_1}\{[x, w]\}? * d_{\pi_2}\{[y]\}? &\vdash d_{\pi_1+\pi_2}(\{[x, w]\} \between \{[y]\})? \\ &\vdash d_{\pi_1+\pi_2}\{[x, w, y], [x, y, w], [y, x, w]\}? \end{aligned}$$

And after consuming into variable z, their resources are:

$$\begin{aligned} & d_{\pi_1+\pi_2}(z{::}\{[x, w, y], [x, y, w], [y, x, w]\})? \\ \vdash\ & d_{\pi_1+\pi_2}\{[z, x, w, y], [z, x, y, w], [z, y, x, w]\}? \end{aligned}$$

3.3 Resource invariants

We use resource invariants to verify that values and resources are transmitted through each channel according to protocol. The state of a channel is composed of the resources r stored in the channel, the list of values l_q queued in the channel, and the list of previously consumed values l_c (not a *set* of histories). A resource invariant R holds on the state (r, l_q, l_c) of a channel only if $r; . \models R(l_q, l_c)$. Concatenating the queue l_q and consume values l_c of a channel together, written $l_q @ l_c$, yields the list of produced values for the channel. Channel resource invariants must satisfy the predicate $\vdash R$ `R-okay`, specified as follows:

$$\frac{\begin{array}{ll} \text{emp} & \vdash R(\text{nil}, \text{nil}) \\ \forall l_q, l_c.\ R(l_q, l_c) & \text{is } \textit{closed} \text{ and } \textit{precise} \\ \forall l_c.\ R(\text{nil}, l_c) & \vdash \text{emp} \end{array}}{\vdash R \texttt{ R-okay}} \text{ Resource-Okay}$$

The first premise states that the resource invariant must be satisfied and own no resources for a channel that has not yet been used. The second ensures that resources can be transferred between process environments and that the resource invariant is sufficient to determine exactly what resources are transferred to and

$$\iota ::= \iota_1; \iota_2 \;\mid\; x\texttt{:=}\, e \;\mid\; x\texttt{:=}\, \texttt{[}e\texttt{]} \;\mid\; \texttt{[}x\texttt{]:=}\, e \;\mid\; \texttt{while}\; e\; \iota \;\mid\; \texttt{assert}\; A$$
$$\mid\; [\Gamma_1; A_1]\; \iota_1 \parallel [\Gamma_2; A_2]\; \iota_2 \;\mid\; \texttt{produce}\; d\; e \;\mid\; \texttt{consume}\; d\; x \;\mid\; \texttt{skip}$$

Instruction sequencing, local variable assignment, fetch from heap, store into heap, repeat ι until e is false, assert that predicate A holds, run two blocks of instructions in parallel, produce a value, consume a value into a local variable, no-op.

Fig. 7: Instructions

from the channel. Finally, the third premise attaches resources only to values in the queue and not the consume history to ensure that all resources can eventually be extracted from the channel by consuming values.

Example 3. A resource invariant that specifies the first value produced/consumed is 0 and that the values transmitted are strictly increasing:

$$\begin{aligned} R \triangleq \lambda l_q.\lambda l_c.\ &\text{match } l_q, l_c \text{ with} \\ &\mid \text{nil}, \text{nil} \Rightarrow \text{emp} \\ &\mid \text{nil}, v{::}\text{nil} \Rightarrow v = 0 \wedge \text{emp} \\ &\mid \text{nil}, v_1{::}v_2{::}l_c \Rightarrow v_1 > v_2 \wedge R(\text{nil}, v_2{::}l_c) \\ &\mid l_q@v{::}\text{nil}, l_c \Rightarrow R(l_q, v{::}l_c) \end{aligned}$$

Example 4. A resource invariant for the parallelization of Figure 4 in Figure 5. To pass permission to dereference `p.val` through the channel:

$$\begin{aligned} R \triangleq \lambda l_q.\lambda l_c.\ &\text{match } l_q, l_c \text{ with} \\ &\mid \text{nil}, \_ \Rightarrow \text{emp} \\ &\mid v{::}l'_q, \_ \Rightarrow R(l'_q, l_c) * v.\texttt{val} \mapsto - \end{aligned}$$

3.4 Instructions

In Figure 7, x is a program variable, ι is an instruction, A is a predicate, Γ is a set of free variables, and d is the name of a channel. Instruction $[\Gamma_1; A_1]\; \iota_1 \parallel [\Gamma_2; A_2]\; \iota_2$ uses Γ and A to specify how variables and resources are split between processes ι_1 and ι_2. We use `assert` to prove the partial correctness of programs.

3.5 Hoare logic

A Hoare triple describes the precondition and postcondition of executing a command. If the precondition is met and the command terminates, then the postcondition establishes the new state of the program. We give the Hoare triple rules for our logic in Figure 8. $FV(X)$ denotes the set of free variables in X, where X ranges over instructions and predicates. The Hoare triple $\bar{R}; \Gamma \vdash_{\text{i}} \{A\}\; \iota\; \{B\}$ is composed of an instruction ι, precondition A, postcondition B, environment

$$l \Subset_\pi H \triangleq \begin{cases} l \in H & \text{if } \pi = 1 \\ \exists H'.\ l \in H \between H' & \text{if } \pi \neq 1 \end{cases}$$

$$\frac{\begin{array}{l} B \vdash \text{PHist } d\ \{\text{nil}\} \qquad A \vdash \text{PHist } d\ e{::}H \\ \forall l_q, l_c.\ \ l_q@l_c \Subset_\pi H \implies B * \bar{R}[d](l_q, l_c) \vdash \bar{R}[d](e{::}l_q, l_c) \end{array}}{\bar{R}; \Gamma \vdash_{\text{i}} \{d_\pi H! * (d_\pi e{::}H! \mathrel{-\!\!*} B * A)\}\ \texttt{produce}\ d\ e\ \{A\}}\ \text{H-Produce}$$

$$\frac{\begin{array}{l} x \in \Gamma \quad \forall v.\ B(v) \vdash \text{CHist } d\ \{\text{nil}\} \quad A \vdash \text{CHist } d\ x{::}H \\ \forall l_q, v, l_c.\ \ l_c \Subset_\pi H \implies \\ \qquad \bar{R}[d](l_q@v{::}\text{nil}, l_c) \vdash B(v) * \bar{R}[d](l_q, v{::}l_c) \end{array}}{\begin{array}{l} \bar{R}; \Gamma \vdash_{\text{i}} \{d_\pi H? * (\forall v.\ (d_\pi H? * B(v))[\texttt{c: } d \mathrel{+}= v] \mathrel{-\!\!*} A[v/x])\} \\ \qquad \texttt{consume}\ d\ x \\ \qquad \{A\} \end{array}}\ \text{H-Consume}$$

$$\frac{\begin{array}{l} \bar{R}; \Gamma_1 \vdash_{\text{i}} \{A_1\}\ \iota_1\ \{B_1\} \quad FV(A_1) \subseteq \Gamma_1 \qquad \Gamma_1 \# \Gamma_2 \\ \bar{R}; \Gamma_2 \vdash_{\text{i}} \{A_2\}\ \iota_2\ \{B_2\} \quad FV(A_2) \subseteq \Gamma_2 \quad \Gamma_1 \cup \Gamma_2 \subseteq \Gamma \end{array}}{\bar{R}; \Gamma \vdash_{\text{i}} \{A_1 * A_2\}\ [\Gamma_1; A_1]\ \iota_1 \parallel [\Gamma_2; A_2]\ \iota_2\ \{B_1 * B_2\}}\ \text{H-Parallel}$$

$$\frac{x \in \Gamma}{\bar{R}; \Gamma \vdash_{\text{i}} \{\exists v.\ e \mapsto v * (e \mapsto v \mathrel{-\!\!*} A[v/x])\}\ x\texttt{:= [}e\texttt{]}\ \{A\}}\ \text{H-Fetch}$$

$$\frac{}{\bar{R}; \Gamma \vdash_{\text{i}} \{e_1 \mapsto - * (e_1 \mapsto e_2 \mathrel{-\!\!*} A)\}\ \texttt{[}e_1\texttt{]:= }e_2\ \{A\}}\ \text{H-Store}$$

$$\frac{}{\bar{R}; \Gamma \vdash_{\text{i}} \{A\}\ \texttt{assert}\ A\ \{A\}}\ \text{H-Assert} \qquad \frac{x \in \Gamma}{\bar{R}; \Gamma \vdash_{\text{i}} \{A[e/x]\}\ x\texttt{:= }e\ \{A\}}\ \text{H-Assign}$$

$$\frac{\begin{array}{l} \bar{R}; \Gamma \vdash_{\text{i}} \{A\}\ \iota\ \{B\} \qquad FV(\iota) \cap FV(C) = \emptyset \\ \forall d \in FV(\iota).\ C \vdash \text{PHist } d\ \{\text{nil}\} \wedge \text{CHist } d\ \{\text{nil}\} \end{array}}{\bar{R}; \Gamma \vdash_{\text{i}} \{A * C\}\ \iota\ \{B * C\}}\ \text{H-Frame}$$

$$\frac{\bar{R}; \Gamma \vdash_{\text{i}} \{A \wedge e\}\ i\ \{A\}}{\bar{R}; \Gamma \vdash_{\text{i}} \{A\}\ \texttt{while}\ e\ i\ \{A \wedge \neg e\}}\ \text{H-While}$$

$$\frac{\begin{array}{l} A \vdash A' \qquad B' \vdash B \\ \bar{R}; \Gamma \vdash_{\text{i}} \{A'\}\ \iota\ \{B'\} \end{array}}{\bar{R}; \Gamma \vdash_{\text{i}} \{A\}\ \iota\ \{B\}}\ \text{H-Consequence} \qquad \frac{\begin{array}{l} \bar{R}; \Gamma \vdash_{\text{i}} \{A\}\ \iota_1\ \{B\} \\ \bar{R}; \Gamma \vdash_{\text{i}} \{B\}\ \iota_2\ \{C\} \end{array}}{\bar{R}; \Gamma \vdash_{\text{i}} \{A\}\ \iota_1\texttt{;}\iota_2\ \{B\}}\ \text{H-Seq}$$

Fig. 8: Inference rules of the Hoare logic

domain Γ, and a channel-indexed list of resource invariants $\bar{R}$. We write $\Gamma_1 \# \Gamma_2$ if the two domains are disjoint. For any element X in our formal system, we write $\bar{X}$ to denote a list of such elements and $\bar{X}[i]$ to access the i^{th} element.

H-Produce. Consider a program that produces pointers to nodes in a linked list with the intention that the consumer only accesses the val member of each node, such as in Figure 5. To execute $\texttt{produce}\ d\ \texttt{p}$ and express that $\texttt{p.}\mathsf{val} \mapsto -$ is transferred to the channel (to eventually be transferred to a consumer), that the produce histories are appended with $\texttt{p}$, and that the resource $\texttt{p.}\mathsf{next} \mapsto -$ is retained, we could use the Hoare triple and resource invariant:

$$\bar{R}; \Gamma \vdash_{\text{i}} \{d_1 H! * \texttt{p.}\mathsf{val} \mapsto - * \texttt{p.}\mathsf{next} \mapsto -\}\ \texttt{produce}\ d\ \texttt{p}\ \{d_1 \texttt{p}{::}H! * \texttt{p.}\mathsf{next} \mapsto -\}$$

$$\begin{aligned} \bar{R}[d] \triangleq \lambda l_q.\lambda l_c.\ &\text{match } l_q, l_c \text{ with} \\ &\mid \text{nil}, \_ \Rightarrow \text{emp} \\ &\mid v{::}l'_q, \_ \Rightarrow R(l'_q, l_c) * v.\mathtt{val} \mapsto - \end{aligned}$$

To prove this triple, we apply rules H-Produce and H-Consequence and prove these side conditions:

$$\forall l_q, l_c.\ l_q @ l_c \Subset_1 H \implies \big(\texttt{p.}\mathsf{val} \mapsto - * R(l_q, l_c) \vdash R(\texttt{p}{::}l_q, l_c)\big) \tag{1}$$

$$\texttt{p.}\mathsf{val} \mapsto - \vdash \text{PHist}\ d\ \{\text{nil}\} \tag{2}$$

$$d_1 \texttt{p}{::}H! * \texttt{p.}\mathsf{next} \mapsto - \vdash \text{PHist}\ d\ \texttt{p}{::}H \tag{3}$$

$$\begin{pmatrix} d_1 H! * \texttt{p.}\mathsf{val} \mapsto - \\ * \texttt{p.}\mathsf{next} \mapsto - \end{pmatrix} \vdash \begin{pmatrix} d_1 H! * \big(d_1 \texttt{p}{::}H! \mathrel{-\!\!*} (\texttt{p.}\mathsf{val} \mapsto - \\ * d_1 \texttt{p}{::}H! * \texttt{p.}\mathsf{next} \mapsto -)\big) \end{pmatrix} \tag{4}$$

These obligations are easy to prove for this example. (The antecedent in obligation 1 can be ignored because it is not needed to prove the consequent).

Rule H-Produce requires some permission ($d_\pi H!$) to access the produce endpoint and implies that the post state will have histories H appended with the value sent. The first judgement in H-Produce, $B \vdash \text{PHist}\ d\ \{\text{nil}\}$, prevents the resources B that are transferred to the channel from specifying histories for the same channel. Judgement $A \vdash \text{PHist}\ d\ e :: H$ prevents the postcondition from specifying histories in addition to $e :: H$. These side conditions involving PHist and CHist are necessary to prove soundness using our current model of histories. They do not prevent channels from sending shares of themselves (with histories $\{\text{nil}\}$) or shares and histories of other channels.

The third judgement requires that, for any state of the channel (the queue l_q and consumed values l_c) such that the resource invariant holds, the invariant remains satisfied after pushing e onto the back of the queue and adding resources B to the channel. Its antecedent, $l_q @ l_c \Subset_\pi H$, restricts the set of channel states to those that support the local histories we have observed.

When $\pi = 1$, $l_q @ l_c \Subset_\pi H$ simplifies to $l_q @ l_c \in H$ (H contains all possible orderings of produced values), and when $\pi < 1$, it implies that every value in H is present in the list of produced values $l_q @ l_c$. This constraint, although weak, still has uses. For example, it allows producing a 0 to a channel with the resource

invariant "all values must be 1 until a 0 is produced, at which point only 0's can be produced" if all we know is that a 0 appears in the local produce histories.

H-Consume. Consider the consumer process from the previous example program, which consumes values that are pointers to nodes in a linked list, only to dereference the val member of each node. To execute `consume` d `p'`, express that `p'`.val$\mapsto -$ is received from the channel, and that the consume histories are appended with `p'`, we could use the Hoare triple:

$$\bar{R}; \Gamma, x \vdash_{\mathrm{i}} \{d_1 H?\}\ \texttt{consume}\ d\ \texttt{p'}\ \{d_1\texttt{p'}::H? * \texttt{p'}.\mathsf{val}\mapsto -\}$$

To prove this triple, we apply rules H-Consume and H-Consequence and prove:

$$\forall l_q, v, l_c.\ l_c \Subset_1 H \implies \big(R(l_q @ v::\mathrm{nil}, l_c) \vdash v.\mathsf{val}\mapsto - * R(l_q, v::l_c)\big) \tag{5}$$

$$v.\mathsf{val}\mapsto - \vdash \mathrm{CHist}\ d\ \{\mathrm{nil}\} \tag{6}$$

$$d_1\texttt{p'}::H? * \texttt{p'}.\mathsf{val}\mapsto - \vdash \mathrm{CHist}\ d\ \texttt{p'}::H \tag{7}$$

$$d_1 H? \vdash \begin{pmatrix} d_1 H? * \big(\forall v.\ (d_1 H? * v.\mathsf{val}\mapsto -)[\mathsf{c}\colon d \mathrel{+}= v] \\ \qquad -\!\!* (d_1\texttt{p'}::H? * \texttt{p'}.\mathsf{val}\mapsto -)[v/\texttt{p'}]\big) \end{pmatrix} \tag{8}$$

These particular obligations are also easy to prove. Obligation 8, although convoluted, can be deduced easily:

$$d_1 H? \vdash \begin{pmatrix} d_1 H? * \big(\forall v.\ (d_1 H? * v.\mathsf{val}\mapsto -)[\mathsf{c}\colon d \mathrel{+}= v] \\ \qquad -\!\!* (d_1\texttt{p'}::H? * \texttt{p'}.\mathsf{val}\mapsto -)[v/\texttt{p'}]\big) \end{pmatrix}$$

$$emp \vdash \begin{pmatrix} \forall v.\ (d_1 H? * v.\mathsf{val}\mapsto -)[\mathsf{c}\colon d \mathrel{+}= v] \\ -\!\!* (d_1\texttt{p'}::H? * \texttt{p'}.\mathsf{val}\mapsto -)[v/\texttt{p'}]) \end{pmatrix}$$

$$(d_1 H? * v.\mathsf{val}\mapsto -)[\mathsf{c}\colon d \mathrel{+}= v] \vdash (d_1\texttt{p'}::H? * \texttt{p'}.\mathsf{val}\mapsto -)[v/\texttt{p'}]$$

$$d_1 v::H? * v.\mathsf{val}\mapsto - \vdash d_1 v::H? * v.\mathsf{val}\mapsto -$$

Rule H-Consume requires permission ($d_\pi H?$) to access the consume endpoint. The fourth judgement in H-Consume requires that, for all states of the channel (the queue $l_q @ v :: \mathrm{nil}$ and consumed values l_c) such that its resource invariant holds, the invariant remains satisfied after popping v from the front of the queue, recording it in the list of consumed values, and removing the resources $B(v)$ tied to the value. (B is a function from values to predicates). The antecedant, $l_c \Subset_\pi H$, restricts the consumed values to those that are supported by the local consume histories. Resources $B(v)$ are transferred to the consuming process. Finally, the local consume histories are appended with the consumed value.

4 Parallelized Program with Proof

Consider proofs of correctness for programs such as Figure 3 (and particularly Figure 4) that follow the schema in Figure 9. We give an example instance of such

$\bar{R}; \Gamma \vdash_\iota$
$\{C(\mathrm{nil}, \mathtt{p}) * F(\mathtt{p}) * D(\mathrm{nil})\}$
$\{\exists h.\ C(h, \mathtt{p}) * F(\mathtt{p}) * D(h)\}$
`while` c(p) (
$\{C(h, \mathtt{p}) * F(\mathtt{p}) * D(h) \wedge c(\mathtt{p})\}$
b(p) ;
$\{C(h, \mathtt{p}) * D(\mathtt{p}{::}h) \wedge c(\mathtt{p}) \wedge \mathtt{p} \Downarrow v\}$
p:= a(p)
$\{C(v{::}h, \mathtt{p}) * F(\mathtt{p}) * D(v{::}h)\}$
$\{\exists h.\ C(h, \mathtt{p}) * F(\mathtt{p}) * D(h)\}$
)
$\{\exists h.\ C(h, \mathtt{p}) * F(\mathtt{p}) * D(h) \wedge \neg c(\mathtt{p})\}$

Fig. 9: While-program with proof

$$C(l, v) \triangleq \mathtt{plist}\ l^{\mathrm{R}}\ v * \exists l'.\ j = l^{\mathrm{R}} @ l'$$
$$\wedge\ (v \neq \mathrm{nil} \iff v = \mathtt{hd}\ l')$$
$$\wedge\ (F(v) \mathrel{-\!\!*} \mathtt{list}\ l'\ \mathrm{nil})$$
$$F(v) \triangleq \text{if } v \neq \mathrm{nil} \text{ then } v.\mathsf{val} \mapsto - \text{ else emp}$$
$$D(l) \triangleq \mathtt{vlist}\ l$$

$$\mathtt{plist}\ l\ v \triangleq \text{match } l \text{ with } \mid \mathrm{nil} \Rightarrow \mathrm{emp}$$
$$\mid x{::}\mathrm{nil} \Rightarrow x.\mathsf{next} \mapsto v \wedge x \neq \mathrm{nil}$$
$$\mid w{::}x{::}l \Rightarrow w.\mathsf{next} \mapsto x * \mathtt{plist}\ x{::}l\ v \wedge w \neq \mathrm{nil}$$
$$\mathtt{vlist}\ l \triangleq \text{match } l \text{ with } \mid \mathrm{nil} \Rightarrow \mathrm{emp}$$
$$\mid x{::}l' \Rightarrow x.\mathsf{val} \mapsto - * \mathtt{vlist}\ l' \wedge x \neq \mathrm{nil}$$
$$\mathtt{list}\ l\ t \triangleq \mathtt{vlist}\ l * \mathtt{plist}\ l\ t$$

Fig. 10

a proof in Figure 10 for the program in Figure 4. The example proof ensures that the shape and order of the linked list is preserved, and holds for the properties:

$$\bar{R}; \Gamma \vdash_{\mathrm{i}} \{C(h, \mathtt{p}) * D(h) \wedge c(\mathtt{p}) \wedge \mathtt{p} \Downarrow v\}\ \mathtt{p:=}\ a(\mathtt{p})\ \{F(\mathtt{p}) * D(h) * C(v{::}h, \mathtt{p}) \wedge c(v)\} \tag{9}$$

$$\bar{R}; \Gamma \vdash_{\mathrm{i}} \{C(h, \mathtt{p}) * F(\mathtt{p}) * D(h) \wedge c(\mathtt{p})\}\ b(\mathtt{p})\ \{C(h, \mathtt{p}) * D(\mathtt{p}{::}h) \wedge c(\mathtt{p})\} \tag{10}$$

Furthermore, if the following properties hold, then we can construct a proof for programs characterized by Figure 3 and optimized by DSWP:

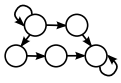
$$\bar{R}; \Gamma \vdash_{\mathrm{i}} \{C(h, \mathtt{p}) \wedge c(\mathtt{p}) \wedge \mathtt{p} \Downarrow v\}\ \mathtt{p:=}\ a(\mathtt{p})\ \{F(\mathtt{p}) * C(v{::}h, \mathtt{p}) \wedge c(v6)\} \tag{11}$$

$$\bar{R}; \Gamma \vdash_{\mathrm{i}} \{F(\mathtt{p}) * D(h) \wedge c(\mathtt{p})\}\ b(\mathtt{p})\ \{D(\mathtt{p}{::}h) \wedge c(\mathtt{p})\} \tag{12}$$

$$\mathrm{FV}(b(\mathtt{p})) \cap \mathrm{MV}(\mathtt{p:=}\ a(\mathtt{p})) = \mathtt{p} \tag{13}$$

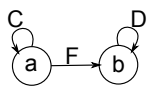
$$\mathrm{FV}(c(\mathtt{p})) \cap \mathrm{MV}(\mathtt{p:=}\ a(\mathtt{p})) = \mathtt{p} \tag{14}$$

$$\mathrm{FV}(c(\mathtt{p})) \cap \mathrm{MV}(b(\mathtt{p})) = \emptyset \tag{15}$$

The program in Figure 4, using the definitions in Figure 10, satisfies properties 9-15 and can be transformed into a parallelized proof. Such properties are fairly typical for programs in the general schema of Figure 3. Our technique applies to programs with a simple dependency structure,  , which is a generalization of more complicated structures: 

Theorem 1. *Given any program instance of Figure 3 with a (sequential) Separation Logic proof given by (9) and (10), and satisfying (11)-(15), there is a Concurrent Separation Logic proof of the correctness of the parallelized program instance in Figure 5.*

4.1 Proof of Theorem 1



The proof in Figure 9 has three separated predicate functions, C, D, and F to model the resources attached to the dependencies flowing from $a(\mathtt{p})$ to $a(\mathtt{p})$, $b(\mathtt{p})$ to $b(\mathtt{p})$, and $a(\mathtt{p})$ to $b(\mathtt{p})$, respectively. The logical variable h is used to reason about past values of $\mathtt{p}$. The association of C to $a(\mathtt{p})$ is supported by property 11, and D to $b(\mathtt{p})$ by property 12. The resources F are shown to be generated by $a(\mathtt{p})$ and transferred to $b(\mathtt{p})$ by properties 11 and 12, following the dependence of $b(\mathtt{p})$ on $a(\mathtt{p})$ through variable $\mathtt{p}$. Property 13 restricts the dependency between $a(\mathtt{p})$ and $b(\mathtt{p})$ to just variable $\mathtt{p}$, and properties 14 and 15 restrict control flow dependencies to just $\mathtt{p}$. The notation $\mathrm{FV}(\iota)$ is the set of free variables in ι and $\mathrm{MV}(\iota)$ is the set of modified variables.

Constructing the parallelized proof. Applying the DSWP optimization to this program results in the program in Figure 5 (for a fresh channel $d \notin \bar{R}$). To construct a proof for the parallelized program, we reuse elements of the original proof. First, we introduce a predicate to help prove that the consumer will receive all values sent.

$$\begin{aligned} T \triangleq \lambda h.\ &\text{match } h \text{ with} \\ &\mid \text{nil} \Rightarrow \text{true} \\ &\mid v{::}h' \Rightarrow c(v) \wedge T(h') \end{aligned}$$

Then, we create a new variable context ($\Gamma' \triangleq \Gamma, \mathtt{p'}, h'$) and new resource invariants $\bar{R}'$ to allow resources $F(\mathtt{p})$ to be produced and consumed through channel d. The new resource invariant satisfies $\forall d'.\ d \neq d' \implies \bar{R}'[d'] = \bar{R}[d']$ and:

$$\begin{aligned} \bar{R}'[d] \triangleq \lambda l_q.\ \lambda l_c.\ &\text{match } l_q, l_c \text{ with} \\ &\mid \text{nil}, \_ \Rightarrow \text{emp} \\ &\mid l'_q @ v{::}\text{nil}, \_ \Rightarrow F(v) * \bar{R}'[d](l'_q, v{::}l_c) \end{aligned}$$

We start the parallelized program with the same precondition as the original, but add produce and consume endpoints $d_1\{\text{nil}\}!$ and $d_1\{\text{nil}\}?$ to enable access to the channels. Using rules H-Frame and H-Consequence on property 11, we get:

$$\begin{aligned} \bar{R}'; \Gamma' \vdash_{\mathrm{i}} \ & \{d_1\{\mathtt{p}{::}h\}! * C(h, \mathtt{p}) \wedge T(h) \wedge c(\mathtt{p}) \wedge \mathtt{p} \Downarrow v\} \\ & \mathtt{p} := a(\mathtt{p}) \\ & \{d_1\{\mathtt{p}{::}h\}! * F(\mathtt{p}) * C(v{::}h, \mathtt{p}) \wedge T(h) \wedge c(v)\} \end{aligned}$$

Which, for brevity, we rewrite as:

$$\bar{R}'; \Gamma' \vdash_{\mathrm{i}} \{A(h, \mathtt{p}) \wedge c(\mathtt{p}) \wedge \mathtt{p} \Downarrow v\}\ \mathtt{p} := a(\mathtt{p})\ \{F(\mathtt{p}) * A'(v{::}h, \mathtt{p})\}$$

$$
\begin{array}{l}
\bar{R}'; \Gamma' \vdash_\iota \quad \{C(\mathrm{nil}, \mathtt{p}) * F(\mathtt{p}) * D(\mathrm{nil}) * d_1\{\mathrm{nil}\}! * d_1\{\mathrm{nil}\}? \wedge T(\mathrm{nil})\} \\
\quad \{F(\mathtt{p}) * A'(\mathrm{nil}, \mathtt{p}) * B'(\mathrm{nil})\} \\
\left(\begin{array}{l|l}
\quad \{F(\mathtt{p}) * A'(\mathrm{nil}, \mathtt{p})\} & \quad \{B'(\mathrm{nil})\} \\
\texttt{produce } d\ \texttt{p;} & \texttt{consume } d\ \texttt{p'}\texttt{;} \\
\quad \{\exists h.\ A(h, \mathtt{p})\} & \quad \{F(\mathtt{p'}) * \exists h'.\ B(h', \mathtt{p'})\} \\
\texttt{while } c(\mathtt{p})\ (& \texttt{while } c(\mathtt{p'})\ (\\
\qquad \{\exists h.\ A(h, \mathtt{p}) \wedge c(\mathtt{p})\} & \qquad \{F(\mathtt{p'}) * \exists h'.\ B(h', \mathtt{p'}) \wedge c(\mathtt{p'})\} \\
\qquad \{A(h, \mathtt{p}) \wedge c(\mathtt{p}) \wedge \mathtt{p} \Downarrow v\} & \qquad \{F(\mathtt{p'}) * B(h', \mathtt{p'}) \wedge c(\mathtt{p'})\} \\
\quad \mathtt{p} := a(\mathtt{p})\texttt{;} & \quad b(\mathtt{p'})\texttt{;} \\
\qquad \{F(\mathtt{p}) * A'(v{::}h, \mathtt{p})\} & \qquad \{B'(\mathtt{p'}{::}h') \wedge c(\mathtt{p'})\} \\
\quad \texttt{produce } d\ \texttt{p} & \quad \texttt{consume } d\ \texttt{p'} \\
\qquad \{\exists h.\ A(h, \mathtt{p})\} & \qquad \{F(\mathtt{p'}) * \exists h'.\ B(h', \mathtt{p'})\} \\
) &) \\
\quad \{\exists h.\ A(h, \mathtt{p}) \wedge \neg c(\mathtt{p})\} & \quad \{F(\mathtt{p'}) * \exists h'.\ B(h', \mathtt{p'}) \wedge \neg c(\mathtt{p'})\}
\end{array}\right)\texttt{;} \\
\quad \{A(h, \mathtt{p}) * F(\mathtt{p'}) * B(h', \mathtt{p'}) \wedge \neg c(\mathtt{p'})\} \\
\quad \{C(h, \mathtt{p}) * F(\mathtt{p}) * D(h) * d_1\{\mathtt{p}{::}h\}! * d_1\{\mathtt{p}{::}h\}? \wedge \neg c(\mathtt{p})\} \\
\texttt{assert } C(h, \mathtt{p}) * F(\mathtt{p}) * D(h) \wedge \neg c(\mathtt{p})
\end{array}
$$

Fig. 11: Parallelized while-program with proof

where:

$$
\begin{aligned}
A(h, \mathtt{p}) &\triangleq d_1\{\mathtt{p}{::}h\}! * C(h, \mathtt{p}) \wedge T(h) \\
A'(h, \mathtt{p}) &\triangleq d_1\{h\}! * C(h, \mathtt{p}) \wedge T(h)
\end{aligned}
$$

We do the same for property 12:

$$
\begin{array}{ll}
\bar{R}'; \Gamma' \vdash_{\mathrm{i}} & \{d_\pi\{\mathtt{p}{::}h\}? * F(\mathtt{p}) * D(h) \wedge c(\mathtt{p})\} \\
& b(\mathtt{p}) \\
& \{d_\pi\{\mathtt{p}{::}h\}? * D(\mathtt{p}{::}h) \wedge c(\mathtt{p})\} \\
\bar{R}'; \Gamma' \vdash_{\mathrm{i}} & \{F(\mathtt{p}) * B(h, \mathtt{p}) \wedge c(\mathtt{p})\} \\
& b(\mathtt{p}) \\
& \{B'(\mathtt{p}{::}h) \wedge c(\mathtt{p})\}
\end{array}
$$

where:

$$
\begin{aligned}
B(h, \mathtt{p}) &\triangleq d_1\{\mathtt{p}{::}h\}? * D(h) \\
B'(h) &\triangleq d_1\{h\}? * D(h)
\end{aligned}
$$

Next, we prove the hoare triples for producing values, and rewrite the triple using A and A' for brevity:

$$
\begin{array}{ll}
\bar{R}'; \Gamma' \vdash_{\mathrm{i}} & \{F(\mathtt{p}) * d_1\{v{::}h\}! * C(v{::}h, \mathtt{p}) \wedge T(h)\} \\
& \texttt{produce } d\ \texttt{p} \\
& \{F(\mathtt{p}) * d_1\{\mathtt{p}{::}v{::}h\}! * C(v{::}h, \mathtt{p}) \wedge T(h)\} \\
\bar{R}'; \Gamma' \vdash_{\mathrm{i}} & \{F(\mathtt{p}) * A'(v{::}h, \mathtt{p}))\}\ \texttt{produce } d\ \texttt{p}\ \{\exists h.\ A(h, \mathtt{p})\}
\end{array}
$$

We do the same for consuming values:

$$\begin{aligned}\bar{R}'; \Gamma' \vdash_{\mathrm{i}} & \{d_1\{\mathtt{p'}::h'\}? * D(\mathtt{p'}::h')\} \\ & \mathtt{consume}\ d\ \mathtt{p'} \\ & \{F(\mathtt{p'}) * \exists v.\ d_1\{\mathtt{p'}::v::h'\}? * D(v::h')\} \\ \bar{R}'; \Gamma' \vdash_{\mathrm{i}} & \{B'(\mathtt{p'}::h')\}\ \mathtt{consume}\ d\ \mathtt{p'}\ \{F(\mathtt{p'}) * \exists h'.\ B(h', \mathtt{p'})\}\end{aligned}$$

The last step is to prove:

$$\begin{aligned}& A(h, \mathtt{p}) * F(\mathtt{p'}) * B(h', \mathtt{p'}) \wedge \neg c(\mathtt{p'}) \\ \vdash\ & C(h, \mathtt{p}) * F(\mathtt{p}) * D(h) * d_1\{\mathtt{p}::h\}! * d_1\{\mathtt{p}::h\}? \wedge \neg c(\mathtt{p})\end{aligned}$$

Proof.

$$\begin{aligned}& A(h, \mathtt{p}) * F(\mathtt{p'}) * B(h', \mathtt{p'}) \wedge \neg c(\mathtt{p'}) \\ \vdash\ & \big(d_1\{\mathtt{p}::h\}! * C(h, \mathtt{p}) \wedge T(h)\big) * F(\mathtt{p'}) * \big(d_1\{\mathtt{p'}::h'\}? * D(h')\big) \wedge \neg c(\mathtt{p'})\end{aligned}$$

At this point, we know that since all values consumed were produced at some point, $\exists l.\ \mathtt{p}::h = l@\mathtt{p'}::h'$. (This is encoded in our resources: for all l_q, the intersection of the global produce histories and global consume histories, appended with l_q, cannot be empty). If $l = v::l'$, then $T(l'@\mathtt{p'}::h')$, which implies $c(\mathtt{p'})$. This is a contradiction, so $l = \mathrm{nil}$ and $\mathtt{p}::h = \mathtt{p'}::h'$. Thus $\mathtt{p} = \mathtt{p'}$ and $h = h'$

$$\begin{aligned}& \big(d_1\{\mathtt{p}::h\}! * C(h, \mathtt{p}) \wedge T(h)\big) * F(\mathtt{p}) * \big(d_1\{\mathtt{p}::h\}? * D(h)\big) \wedge \neg c(\mathtt{p}) \\ & \qquad \vdash C(h, \mathtt{p}) * F(\mathtt{p}) * D(h) * d_1\{\mathtt{p}::h\}! * d_1\{\mathtt{p}::h\}? \wedge \neg c(\mathtt{p})\end{aligned}$$

We add the `assert` instruction to the end of the program to ensure partial correctness, encoded as a safety property. If the asserted condition cannot be proven, then the instruction will get stuck and the program will not be safe.

Figure 11 is the proof for the parallelized program in Figure 5, such that the preconditions and postconditions (modulo the addition of channel endpoints) are preserved.

5 Model & Operational Semantics

5.1 The Separation Algebra of resources

A standard technique [3] for constructing a SL is to first construct a Separation Algebra (SA) on the resources. Our CSL is based on the SA of *worlds*, which are composed of three kinds of resources: the heap, produce endpoints, and consume endpoints. Although the heap is not necessary to demonstrate a CSL for channels, we include it to prove the correctness of parallelized pointer-programs, where channels are used to synchronize access to shared memory and prevent race conditions. We have formalized[1] our SA following Dockins et al. [2].

[1] A Coq formalization of our SA is available at `http://www.cs.princeton.edu/~cbell/cslchannels/`

$$\pi \in \mathsf{Share} \triangleq [0,1] \qquad u \in \mathsf{Location}$$
$$d \in \mathsf{ChannelName} \qquad H \in \mathsf{HistorySet}$$

$$\mathsf{Endpoint} \triangleq \{\, (\pi, H) : \mathsf{Share} \times \mathsf{HistorySet} \mid \pi = 0 \implies H = \{\mathrm{nil}\}\}$$
$$r \;:\; \{\, \mathsf{m} : \mathsf{Location} \to \mathsf{option\ value},$$
$$\mathsf{p} : \mathsf{ChannelName} \to \mathsf{option\ Endpoint},$$
$$\mathsf{c} : \mathsf{ChannelName} \to \mathsf{option\ Endpoint}\}$$

$$\frac{\forall x.\ f_1(x) \oplus f_2(x) \;=\; f_3(x)}{f_1 \oplus_f f_2 \;=\; f_3}\ \text{Join-Function}$$

$$\frac{\begin{array}{c}\pi_1 \oplus_s \pi_2 \;=\; \pi_3 \\ H_1 \between H_2 \;=\; H_3\end{array}}{(\pi_1, H_1) \oplus_e (\pi_2, H_2) \;=\; (\pi_3, H_3)}\ \text{Join-Endpoint} \qquad \frac{\begin{array}{c}r_1.\mathsf{m} \oplus_f r_2.\mathsf{m} \;=\; r_3.\mathsf{m} \\ r_1.\mathsf{p} \oplus_f r_2.\mathsf{p} \;=\; r_3.\mathsf{p} \\ r_1.\mathsf{c} \oplus_f r_2.\mathsf{c} \;=\; r_3.\mathsf{c}\end{array}}{r_1 \oplus_w r_2 \;=\; r_3}\ \text{Join-World}$$

Fig. 12: Channel Resources

A SA is a tuple, $\langle E, \oplus \rangle$, where E is some type and $\oplus$ is a "join" relation that satisfies functionality, associativity, commutativity, cancellation, self-join ($a \oplus a = b \implies a = b$), and has a unit for each element ($\forall a. \exists e.\ e \oplus a = a$).

We first define worlds, r, as a record of the heap (m), produce endpoints (p), and consume endpoints (c). These worlds and the join relation $\oplus_w$ (rule Join-World in Figure 12) form a SA. Dockins et al. have shown how to construct a SA for the heap and $\oplus_f$; we have constructed a SA for channel endpoints using the same approach, using the fact that $\langle \mathsf{Endpoint}, \oplus_e \rangle$ is a SA.

A Share is the set of rationals over $[0,1]$. The relation $\oplus_s$ is the addition operator, and share $\pi = 0$ is the unit for $\oplus_s$. A channel endpoint equal to Some (π, H) implies $\pi > 0$; None implies a share of 0. In our notation, the produce history of channel d, in world r, is $r.\mathsf{p}(d).\mathsf{H}$; its share is $r.\mathsf{p}(d).\pi$; the consume history is $r.\mathsf{c}(d).\mathsf{H}$; and the consume share is $r.\mathsf{c}(d).\pi$. (This notation implies that $r.\mathsf{p}(d) \neq$ None and $r.\mathsf{c}(d) \neq$ None). We write $r.\mathsf{m}(u) =$ Some v if the heap in world r contains value v at address u.

A $\mathsf{HistorySet}$ is a nonempty set of histories. The merge function ($\between$) satisfies all the properties of a SA except self-join. Constructing a SA from the tuple of two SAs is straightforward, but without the property of self-join for $\between$, we must add the condition that for any $\mathsf{Endpoint}$ (π, H), if the share empty ($\pi = 0$), then the set of histories is $\{\mathrm{nil}\}$.

5.2 Predicate formulae

In Figure 13, we write $[\![e]\!]_s$ to evaluate e within environment s, $r[\mathsf{p}\colon d \mathrel{+}= v]$ to append value v to the local produce histories of channel d in r, and $r[\mathsf{c}\colon d \mathrel{+}= v]$ to similarly update the local consume histories. An environment, s, is a finite partial map from variables to values.

$$
\begin{array}{rcl}
r; s \models d_\pi H! & \text{iff} & r.\mathsf{p}(d) = \text{Some } (\pi, [\![H]\!]_s) \wedge \forall d' \neq d.\ r.\mathsf{p}(d') = \text{None} \\
& & \quad \wedge \forall d.\ r.\mathsf{c}(d) = \text{None} \wedge \forall l.\ r.\mathsf{m}(l) = \text{None} \\
r; s \models d_\pi H? & \text{iff} & r.\mathsf{c}(d) = \text{Some } (\pi, [\![H]\!]_s) \wedge \forall d' \neq d.\ r.\mathsf{c}(d') = \text{None} \\
& & \quad \wedge \forall d.\ r.\mathsf{p}(d) = \text{None} \wedge \forall l.\ r.\mathsf{m}(l) = \text{None} \\
r; s \models A[\mathsf{p}\text{: } d \mathrel{+}= v] & \text{iff} & \exists r'.\ r = r'[\mathsf{p}\text{: } d \mathrel{+}= v] \wedge r'; s \models A \\
r; s \models A[\mathsf{c}\text{: } d \mathrel{+}= v] & \text{iff} & \exists r'.\ r = r'[\mathsf{c}\text{: } d \mathrel{+}= v] \wedge r'; s \models A \\
r; s \models \text{PHist } d\ H & \text{iff} & r.\mathsf{p}(d).\mathsf{H} = [\![H]\!]_s \\
r; s \models \text{CHist } d\ H & \text{iff} & r.\mathsf{c}(d).\mathsf{H} = [\![H]\!]_s \\
r; s \models e_1 \mapsto e_2 & \text{iff} & r.\mathsf{m}([\![e_1]\!]_s) = \text{Some } [\![e_2]\!]_s \wedge \forall l \neq [\![e_1]\!]_s.\ r.\mathsf{m}(l) = \text{None} \\
& & \quad \wedge \forall d.\ r.\mathsf{p}(d) = \text{None} \wedge r.\mathsf{c}(d) = \text{None} \\
r; s \models A_1 * A_2 & \text{iff} & \exists r_1, r_2.\ r_1 \oplus r_2 = r \wedge r_1; s \models A_1 \wedge r_2; s \models A_2 \\
r; s \models A_1 \mathrel{-\!\!*} A_2 & \text{iff} & \forall r_1, r_2.\ r_1; s \models A_1 \implies r \oplus r_1 = r_2 \implies r_2; s \models A_2 \\
r; s \models A_1 \implies A_2 & \text{iff} & r; s \models A_1 \implies r; s \models A_2 \\
r; s \models e_1 = e_2 & \text{iff} & [\![e_1]\!]_s = [\![e_2]\!]_s \\
r; s \models \text{emp} & \text{iff} & \forall l.\ r.\mathsf{m}(l) = \text{None} \wedge \forall d.\ r.\mathsf{p}(d) = \text{None} \wedge r.\mathsf{c}(d) = \text{None}
\end{array}
$$

Fig. 13: Predicate Formulae

5.3 Operational semantics

We present the operational semantics in Figure 14. The machine state is the tuple $S = (\bar{c}, \bar{P})$, where $\bar{c}$ is a channel-indexed list of tuples which contain information about each channel's state. Specifically, $\bar{c}[d] = (R, r, l_q, l_c)$, where R is the resource invariant, r are the resources contained within the queue, l_q is the queue, and l_c is the list of consumed values. $\bar{P}$ is a list of processes, where for any process k, $\bar{P}[k] = (r, s, z)$, r are the process' resources, s is its environment, and z is the current instruction. Stepping process k from state S to state S' is written as $S \rightarrow_k S'$.

To produce a value, we use rule S-Produce. We start by appending the value, $[\![e]\!]_s = v$, to the local produce histories of the process' resources r, which we write as $r[\mathsf{p}\text{: } d \mathrel{+}= v]$. Then, we push this value onto the back of the channel's queue, l_q. Finally, we split r into resources r_1 and r_2 such that r_1 can be transfered to the channel while satisfying its resource invariant for the updated queue (v::l_q), while the resources r_2 remain with the producer.

We append the value to the producer's initial resources before splitting them because either r_1 or r_2 can record the value to its local produce history (although our Hoare rules currently prevent r_1 from containing any history). The channel's resource invariant, $\bar{R}[d]$, is enough to determine r_1 because we know that the updated channel is satisfied by $r_q \oplus r_1;. \models \bar{R}[d](v{::}l_q, l_c)$ and that this resource invariant is precise (a requirement of $\vdash \bar{R}[d]$ `R-okay`).

If consuming from a channel and its queue is empty, the consuming process will block (S-BlockingConsume). To progress using rule S-Consume, there must be a value v to receive from the front of the queue (l_q@v::nil) and we must show

$$\frac{x \in \mathrm{dom}(s) \qquad \bar{P}[k] = (r, s, x \texttt{:= } e\texttt{;}\iota)}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P}[k = (r, s[x = [\![e]\!]_s], \iota)])}\ \text{S-Assign}$$

$$\frac{x \in \mathrm{dom}(s) \qquad \bar{P}[k] = (r, s, x \texttt{:= [}e\texttt{];}\iota) \qquad r.\mathsf{m}([\![e]\!]_s) = \mathrm{Some}\ v}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P}[k = (r, s[x = v], \iota)])}\ \text{S-Fetch}$$

$$\frac{\bar{P}[k] = (r, s, \texttt{[}e_1\texttt{]:= } e_2\texttt{;}\iota) \qquad r.\mathsf{m}([\![e]\!]_s) = \mathrm{Some}\ v}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P}[k = (r[h : [\![e_1]\!]_s \leftarrow [\![e_2]\!]_s], s, \iota)])}\ \text{S-Store}$$

$$\frac{[\![e]\!]_s = \mathrm{true} \qquad \bar{P}[k] = (r, s, \texttt{while}\ e\ \iota\texttt{;}\iota')}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P}[k = (r, s, \iota\texttt{;while}\ e\ \iota\texttt{;}\iota')])}\ \text{S-WhileTrue}$$

$$\frac{[\![e]\!]_s = \mathrm{false} \qquad \bar{P}[k] = (r, s, \texttt{while}\ e\ \iota\texttt{;}\iota')}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P}[k = (r, s, \iota')])}\ \text{S-WhileFalse}$$

$$\frac{\begin{array}{ll}\bar{P}[k] = (r, s, \texttt{produce}\ d\ e\texttt{;}\iota) & \bar{c}[d] = (R, r_q, l_q, l_c) \\ r_q \oplus r_1; . \models R(v{::}l_q, l_c) & r[\mathsf{p}\colon d \mathrel{+}= v] = r_1 \oplus r_2 \qquad [\![e]\!]_s = v\end{array}}{(\bar{c}, \bar{P}) \to_k (\bar{c}[d = (R, r_q \oplus r_1, v{::}l_q, l_c)], \bar{P}[k = (r_2, s, \iota)])}\ \text{S-Produce}$$

$$\frac{\begin{array}{ll}r.\mathsf{c}(d) = \mathrm{Some}\ \_ & \\ \bar{P}[k] = (r, s, \texttt{consume}\ d\ x\texttt{;}\iota) & x \in \mathrm{dom}(s) \\ \bar{c}[d] = (R, r_q \oplus r', l_q @ v{::}\mathrm{nil}, l_c) & r_q; . \models R(l_q, v{::}l_c)\end{array}}{(\bar{c}, \bar{P}) \to_k (\bar{c}[d = (R, r_q, l_q, v{::}l_c)], \bar{P}[k = ((r \oplus r')[\mathsf{p}\colon d \mathrel{+}= v], s[x = v], \iota)])}\ \text{S-Consume}$$

$$\frac{\bar{P}[k] = (r, s, \texttt{consume}\ d\ x\texttt{;}\iota) \qquad \bar{c}[d] = (R, r_q, \mathrm{nil}, l_c)}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P})}\ \text{S-BlockingConsume}$$

$$\frac{\bar{P}[k] = (r, s, \texttt{assert}\ A\texttt{;}\iota) \qquad r; s \models A * \mathrm{true}}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P}[k = (r, s, \iota)])}\ \text{S-Assert}$$

$$\frac{\bar{P}[k] = (r, s, \texttt{skip;}\iota)}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P}[k = (r, s, \iota)])}\ \text{S-Skip}$$

$$\frac{\begin{array}{ll}\bar{P}[k] = (r, s, [G_1; A_1]\ \iota_1 \parallel [G_2; A_2]\ \iota_2\texttt{;}\iota) & s_1 \# s_2 \\ r = r_1 \oplus r_2 & s = s_1 + s_2 \\ G_1 = dom(s_1) & G_2 = dom(s_2) \\ r_1; s_1 \models A_1 * \mathrm{true} & r_2; s_2 \models A_2 \\ k_1 = \mathrm{length}(\bar{P}) & k_2 = \mathrm{length}(\bar{P}) + 1\end{array}}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P}[k = (., ., \texttt{wait}\ k_1\ k_2\texttt{;}\ \iota)] + (r_1, s_1, \iota_1\texttt{;exit}) + (r_2, s_2, \iota_2\texttt{;exit}))}\ \text{S-Fork}$$

$$\frac{\bar{P}[k] = (., ., \texttt{wait}\ k_1\ k_2\texttt{;}\ \iota) \qquad \bar{P}[k_1] = (r_1, s_1, \texttt{exit}) \qquad \bar{P}[k_2] = (r_2, s_2, \texttt{exit})}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P}[k = (r_1 \oplus r_2, s_1 + s_2, \iota)][k_1 = (., ., \texttt{halted})][k_2 = (., ., \texttt{halted})])}\ \text{S-Wait}$$

$$\frac{\begin{array}{ll}\bar{P}[k] = (., ., \texttt{wait}\ k_1\ k_2\texttt{;}\ \iota) & \\ \bar{P}[k_1] = (r_1, s_1, \iota_1) & \bar{P}[k_2] = (r_2, s_2, \iota_2) \\ \iota_1 = \texttt{exit} \vee \iota_2 = \texttt{exit} & \end{array}}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P})}\ \text{S-BlockingWait}$$

$$\frac{\bar{P}[k] = (r, s, (\iota_1\texttt{;}\iota_2)\texttt{;}\iota_3)}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P}[k = (r, s, \iota_1\texttt{;}(\iota_2\texttt{;}\iota_3))])}\ \text{S-Seq}$$

$$\frac{\bar{P}[k] = (r, s, \texttt{exit})}{(\bar{c}, \bar{P}) \to_k (\bar{c}, \bar{P})}\ \text{S-Exit}$$

Fig. 14: Operational Semantics

that the channel's resource invariant will be satisfied by the channel's subsequent state. As a result, the consuming process receives the value and resources r', which are tied to the value, from the queue. The value is popped from the front of the queue and pushed onto the list of consumed values. Finally, the consumer's local histories are appended with the value $((r \oplus r')[\mathsf{c}: d \mathrel{+}= v])$ and the value is bound to the program variable x in the environment $(s[x = v])$.

Note that our operational semantics is not computable because we use logic formulae in the judgments. We believe that the forcing relationship can eventually be erased from our operational semantics during compilation.

Our operational semantics use permissions to guarantee that programs are well-synchronized, so that any process will get stuck if it attempts to access a resource without permission. Crucially, two processes may not have permission to mutate the same memory at the same time. Well-synchronized programs are race free, and it is generally understood that such programs have equivalent executions in both strong and weak memory models. Thus, while our operational semantics define a strong memory model, using a weak memory model would not change its behavior.

6 Soundness

We prove[2] soundness for our logic using progress and preservation. In our proofs, we consider only well-formed machine states S such that $\vdash S$ `well-formed`. A well-formed machine state requires each channel state to satisfy its resource invariant; that the resource invariant R for each channel is $\vdash R$ `R-okay`; that each process is well-formed; and that no two processes share any of the same environment variables. A process is well-formed only if its instructions have a Hoare triple derivation and if the precondition of this triple is exactly satisfied by the process' current resources and environment.

For any machine-states S and S',

Theorem 2 (Preservation). *For all processes k, if $\vdash S$ `well-formed` and $S \rightarrow_k S'$, then $\vdash S'$ `well-formed`.*

Theorem 3 (Progress). *If $\vdash S$ `well-formed`, then for all processes k, there either exists a state S' such that $S \rightarrow_k S'$, or process k in S is halted.*

7 Related Work

Ottoni informally proved that when parallelizing a program, if the dependencies (the Program Dependency Graph) are preserved, then the generated program is observationally equivalent [10]. Yet this is not enough for a certified compiler because no implementation of the algorithm is proven.

Our CSL with channels was partially modeled after Concurrent C Minor [6]; specifically the style of resource invariants. There are close similarities between

[2] Our proofs can be found at `http://www.cs.princeton.edu/~cbell/cslchannels/`

our while-programs with channels and pi-calculus, enough so that our CSL is similar to the logic proposed by Hoare and O'Hearn [5]. Turon et al. have extended that work with multiple producers and consumers [14] independently from us. Neither of these works have shared memory or an equivalent to histories, so are not applicable to reasoning about the correctness of DSWP.

Hurlin shows how to parallelize a sequential program's proof, using rewrite rules on its derivation tree, for the DOALL optimization [7]. We demonstrate how to prove a DSWP-parallelized program without using the proof's derivation tree, but we assume a proof structure that may not characterize all proofs.

8 Future Work & Conclusion

We have developed an operational semantics and CSL for asynchronous channels and proved the logic sound. This logic features histories that are used to overcome limitations in local reasoning and to prove the correctness of parallelized programs in the presence of asynchronous communication. We demonstrated how an existing proof of correctness (of a particular structure) for a sequential program can be used to generate a related proof for the parallelized output of DSWP. By maintaining the preconditions and postconditions, we can prove partial correctness of the optimization.

Our CSL and operational semantics are thus potential targets for an automatic method of generating parallelized proofs for parallelizing optimizations. Using such a method, we could prove that the optimization preserves all *specified* behaviors of a program.

First class channels are unnecessary for our proofs about DSWP. Some separation logics have first class objects [4][6], others do not permit passing objects this way [9]. We would need allocation, deallocation, and a more complicated definition of histories in order to support first class channels. The first two are not possible in first-order logic, but we believe it is straightforward to add following the approach used by Gotsman et al. and Hobor et al. The more complicated history model requires histories to have shared pasts that, upon merging, would retain their original ordering. For example, breaking the history $\{[123]\}$ into histories $\{[12]\}$ and $\{[3]\}$ would result in $\{[123]\}$ upon re-merging rather than $\{[123], [132], [312]\}$. (With such histories, side conditions involving PHist and CHist could be dropped from the Hoare rules).

We are now investigating methods of manipulating an existing arbitrary proof of a program using facts acquired from shape analysis, with the goal of eventually proving the correctness of DSWP and other parallelizing optimizations.

Acknowledgements. This research is funded in part by NSF awards CNS-0627650 and IIS-0612147. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

1. Bridges, M.J., Vachharajani, N., Zhang, Y., Jablin, T., August, D.I.: Revisiting the Sequential Programming Model for Multi-Core. Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO), December 2007.
2. Dockins, R., Hobor, A, Appel, A.W.: A Fresh Look at Separation Algebras and Share Accounting. 7th Asian Symposium on Programming Languages and Systems, December 2009.
3. Calcagno, C., O'Hearn, P., Yang, H.: Local actions and abstract separation logic. Proceeding of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS), pp. 353–367, 2008.
4. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. 5th Asian Symposium on Programming Languages and Systems (APLAS'07), 2007.
5. Hoare, T., O'Hearn, P.: Separation Logic Semantics for Communicating Processes. Electronic Notes in Theoretical Computer Science, vol. 212, pp. 3–25, 2008.
6. Hobor, A.: Oracle Semantics. Ph.D. Thesis, Princeton TR-836-08, October 2008.
7. Hurlin, C.: Automatic Parallelization and Optimization of Programs by Proof Rewriting. Static Analysis Symposium, Lecture Notes in Computer Science. Springer-Verlag, August 2009.
8. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. 33rd ACM symposium on Principles of Programming Languages (POPL), pp. 42-54. ACM Press, 2006
9. O'Hearn, P.: Resources, Concurrency, and Local Reasoning. Theoretical Computer Science, 375(1-3), pp. 271–307. Elsevier, 2007.
10. Ottoni, G.: Ph.D. Thesis, Department of Computer Science, Princeton University, September 2008.
11. Parkinson, M.: Ph.D. Thesis, The Computer Laboratory, University of Cambridge, 2005.
12. Rangan, R., Vachharajani, N., Vachharajani, M., August, D.I.: Decoupled Software Pipelining with the Synchronization Array. Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2004.
13. Rangan, R.: Pipelined Multithreading Transformations and Support Mechanisms. Ph.D. Thesis, Department of Computer Science, Princeton University, June 2004.
14. Turon, A., Wand, M.: A separation logic for the pi-calculus. `http://www.ccs.neu.edu/home/turon/pi-sep-logic.pdf`
15. Vachharajani, N., Rangan, R., Raman, E., Bridges, M.J., Ottoni, G., August, D.I.: Speculative Decoupled Software Pipelining. Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2007.