A

A Hoare Logic for GPU Kernels

KENSUKE KOJIMA, Kyoto University and CREST, JST
ATSUSHI IGARASHI, Kyoto University and CREST, JST

We study a Hoare Logic to reason about parallel programs executed on graphics processing units (GPUs), called GPU kernels. During the execution of GPU kernels, multiple threads execute in lockstep, that is, execute the same instruction simultaneously. When the control branches, the two branches are executed sequentially, but during the execution of each branch only those threads that take it are enabled; after the control converges, all the threads are enabled and again execute in lockstep. In this article, we first consider a semantics in which all threads execute in lockstep (this semantics simplifies the actual execution model of GPUs), and adapt Hoare Logic to this setting by augmenting the usual Hoare triples with an additional component representing the set of enabled threads. It is determined that the soundness and relative completeness of the logic do not hold for all programs; a difficulty arises from the fact that one thread can invalidate the loop termination condition of another thread through shared memory. We overcome this difficulty by identifying an appropriate class of programs for which the soundness and relative completeness hold. Additionally, we discuss thread interleaving, which is present in the actual execution of GPUs but not in the lockstep semantics mentioned above. We show that, if a program is race-free, then the lockstep and interleaving semantics produce the same result. This implies that our logic is sound and relatively complete for race-free programs, even if the thread interleaving is taken into account.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs*

General Terms: Theory, Verification

Additional Key Words and Phrases: GPU, Hoare Logic

ACM Reference Format:
Kensuke Kojima and Atsushi Igarashi. 2014. A Hoare Logic for GPU Kernels. *ACM Trans. Comput. Logic* V, N, Article A (January YYYY), 42 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

General-purpose computing on graphics processing units (GPGPU) has recently become widely available even to end-users, enabling them to utilize the computational power of GPUs for solving problems other than graphics processing. Application areas include physics simulation, signal and image processing, etc. [Owens et al. 2007]. However, the writing and optimizing of GPU kernels, which are parallel programs executed on GPUs, remain difficult tasks and are error-prone. For example, in programming in CUDA, a parallel computing platform and programming model on a GPU [NVIDIA 2014], attention must be paid to synchronization and data races so that many threads cooperate correctly. Moreover, to obtain the best performance, low-level mechanisms should be considered, to optimize the memory access pattern, increase occupancy, etc.

This is a revised and extended version of Kojima and Igarashi [2013]. Author's addresses: K. Kojima and A. Igarashi, Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University, Yoshida-Honmachi, Sakyo-ku, Kyoto 606-8501, Japan.

DOI:http://dx.doi.org/10.1145/0000000.0000000

Considerable effort has recently been invested in developing automated verification tools for GPU kernels [Betts et al. 2012; Collingbourne et al. 2013; Collingbourne et al. 2011; 2012; Li and Gopalakrishnan 2010; 2012; Li et al. 2012b; Li et al. 2012a; Chiang et al. 2013; Bardsley et al. 2014]. These tools are implemented to attempt to automate the detection of synchronization errors, data races, and inefficiency, as well as to check functional correctness and generate test cases. Although automation is a great advantage, it tends to suffer false positives/negatives because of approximation, as well as combinatorial explosion.

Another approach to formal verification is deductive verification, in which the correctness of a program is verified by formally proving (using a fixed set of deduction rules) that it is indeed correct. Relative completeness of the inference rules guarantees that all correct programs can be proved to be correct, although often frequently a considerable effort is required to complete the correctness proof. Deductive approaches have been implemented as tools that can be applied to real-world programs (Why3[1], for example). However, in the context of GPU programming, this approach has not thus far been extensively studied (at the time of writing, we are aware of only the studies based on permission-based separation logic reported by Blom et al. [2014] and Asakura et al. [2016]).

In this study, we investigated a deductive verification method for GPU programs. In this particle, we focus on the Single-Instruction, Multiple-Thread (SIMT) execution model (described in Section 1.1) and demonstrate that Hoare Logic, one of the traditional approaches to deductive verification, can be applied to GPU kernels with only a few modifications. Although SIMT is a terminology employed by CUDA, this does not mean that our theory is specialized for CUDA. In particular, it applies also to OpenCL.

In general, reasoning about parallel programs requires techniques that are considerably more sophisticated than those used for sequential ones, because parallel threads can interfere with each other through shared resources [Apt et al. 2009]. Although existing techniques can be applied to GPU kernels, we take advantage of the so-called lockstep semantics of SIMT to obtain simpler inference rules. In fact, our inference rules are similar to the usual Hoare Logic, and soundness and relative completeness hold under a very mild restriction regarding the loop guards: a thread does not invalidate other threads' loop guards through shared memory. Any program can be easily transformed into one conforming to this restriction.

This article is an extended version of the authors' previous article [Kojima and Igarashi 2013] in which a Hoare Logic for GPU kernels was introduced, and its soundness and relative completeness for a large class of GPU kernels was proved. In this article, we also consider interleaved thread execution and prove that it does not affect the result of the execution if the program is race-free.

In the rest of this section, we describe the manner in which SIMT operates and how Hoare Logic can be extended to the SIMT setting.

1.1. Overview of the SIMT Execution Model

SIMT is a parallel execution model of GPUs employed by CUDA [NVIDIA 2014]. A CUDA program is written in CUDA C, an extension of the C language, and run on GPUs with multiple (typically thousands of) threads, as specified in the SIMT execution model. In this model, launched threads are divided into groups called *warps*. Each warp consists of a fixed (currently 32) number of threads, and threads belonging to the same warp all execute the same instruction simultaneously. Therefore, the execution of threads in a single warp never interleaves. This type of execution is often called lockstep.

[1] http://why3.lri.fr/

When a conditional branch is encountered during the lockstep execution, and the decisions on which branch to be taken vary among threads within a single warp, then that warp executes both branches sequentially. During the execution of each branch, only those threads that take it are enabled. After all branches are completed, all the threads in the warp are enabled and executed in lockstep again.

Thus, in SIMT, some statements may actually be executed by only some of the threads, depending on the branching. We say that a thread is *active* if it is currently enabled, and *inactive* otherwise. A *mask* is a piece of data (typically a bit mask, but below we often represent a mask as a set) that describes which thread is active. The state of a mask may change during execution and the result of executing a statement may depend on a mask.

For example, let us consider the following program.

```
k = tid;  while (k < n) { c[k] = a[k] + b[k];  k = k + ntid; }
```

Here, we assume that `k` is a thread local variable, `a`, `b`, and `c` are shared arrays of length `n`, and `ntid` is a constant, the value of which is the number of threads. The constant `tid` represents the thread identifier, ranging from 0 to `ntid - 1`. Let us suppose that this program is launched with four threads forming a single warp, and `n` equals 6. In the first iteration, the condition `k < n` holds in all threads, and therefore the mask is $\{0, 1, 2, 3\}$, and all threads execute the loop body. In the second iteration, however, the values of `k` in threads 0, 1, 2, and 3 are 4, 5, 6, and 7, respectively, and therefore the condition `k < n` does not hold in threads 2 and 3. Therefore, these threads are deactivated, and the loop body is executed with mask $\{0, 1\}$. Then, all threads exit the loop, and the program terminates. The final value of `c` is the sum of `a` and `b`.

Although SIMT appears to execute threads in a manner similar to that of single instruction multiple data (SIMD) in that a single instruction operates on multiple data, they differ in that parallel operations on vectors are explicitly specified in SIMD, whereas this is not the case in SIMT. Indeed, when programming in CUDA C only the behavior of a single scalar thread is specified, as in a usual sequential program written in C or C++.

1.2. Extending Hoare Logic

Next, we consider a Hoare Logic for GPU kernels. The programs about which we reason constitute a single GPU kernel, like the example above. In our formalization, we consider mainly a simplified execution model in which *all* threads execute the same instruction simultaneously (in other words, SIMT execution with only one warp). We call this manner of execution *complete lockstep*.

In fact, we can employ most of the inference rules from the ordinary Hoare Logic without significant changes, although the form of Hoare triples has to be changed. As explained above, the effect of the lockstep execution of a statement depends on the mask. Since the usual Hoare triple $\{\varphi\}\, P\, \{\psi\}$ does not contain the information about a mask, it cannot fully specify a program. Therefore, we augment the usual Hoare triple with an additional piece of information, and consider a Hoare *quadruple* of the form $\{\varphi\}\, m \Rightarrow P\, \{\psi\}$, where m denotes a mask. Intuitively, this quadruple means that "if an initial state satisfies φ, and a program P is executed *with a mask denoted by* m, then after termination the state satisfies ψ."

However, a difficulty arises from `while` loops. We found that, in some corner cases, it is difficult to reason about `while` loops correctly. Although it would be possible to modify the inference rule so that we could handle all programs soundly, we decided for the sake of simplicity to make a certain assumption about the programs we re handling. As a result, we consider a certain class of programs and obtain soundness and relative completeness for this class of programs. We consider only loops such that, during their execution, a thread never invalidates the loop termination condition of another thread through shared memory. We call such loops *monotonic*. This is not a serious restriction, because any loop can be transformed into a monotonic one without changing the behavior (with respect to our operational semantics).

Interestingly, our operational semantics and Hoare Logic are quite similar to the ordinary one for sequential programs, despite the parallel nature of GPU programs. It seems that this is a result of the fact that threads work basically independently during the execution of GPU kernels. Although CUDA provides synchronization primitives, their use is allowed only under a certain condition (which will be addressed in Section 5).

1.3. Thread Interleaving

The extension of Hoare Logic above is sound and relatively complete for the semantics in which threads are executed in complete lockstep, but this is not exactly how GPU kernels are executed on real GPUs (we assumed that there is only one warp consisting of all launched threads). This means that our Hoare Logic and its soundness and relative completeness do not immediately apply to actual GPU kernels.

However, even if the actual thread execution is interleaved, if we restricted our attention to race-free programs, the result would not depend on the choice of execution semantics, and therefore, it would be sound to assume that programs are executed in complete lockstep. Therefore, under the assumption of race-freedom, our method can be applied without modification to actual GPU kernels. This assumption would be reasonable, because, as far as we know, many GPU kernels are intended to be race-free.

To investigate this direction, we consider another semantics, which we call *interleaving semantics* (following Collingbourne et al. [2013]), in which the execution of threads is interleaved. The execution in this semantics can be regarded as SIMT execution in which every warp consists of only one thread, whereas the lockstep execution is SIMT execution with only one warp. Intuitively, lockstep and interleaving semantics are under-approximation and over-approximation of the actual SIMT execution, respectively. Interleaving semantics does not necessarily produce the same result as lockstep semantics, but it is possible to show that if a program is race-free, then both lockstep and interleaving semantics produce the same result (Collingbourne et al. [2013] proved a similar result, but our formalization and proof are more formal than theirs; see Section 8 for more detailed comparisons). As a consequence, our Hoare Logic is sound and relatively complete for *race-free* programs with respect to the interleaving semantics. This means that our Hoare Logic can be used to reason about actual GPU kernels provided that the kernel is race-free.

1.4. Organization of the Article

The rest of the article is organized as follows. In Section 2, we introduce the lockstep semantics which is an extension of the usual while-language. Section 3 describes our Hoare Logic. In Section 4 we introduce the notion of monotonic loops, and prove the soundness and relative completeness of our Hoare Logic for programs in which all loops are monotonic. In Section 5, we introduce interleaving semantics and discuss the soundness and relative completeness of our Hoare Logic with respect to this semantics. Section 6 is devoted to the proof of the equivalence between lockstep and interleaving semantics for race-free programs. In Section 7, we discuss a few possible variants and extensions of our system. Section 8 describes related work and Section 9 concludes the article. Some of the detailed proofs are collected in the electronic appendix.

2. LOCKSTEP SEMANTICS

In this section, we formalize the complete lockstep execution. Our formalization is based on Habermaier and Knapp's [2012], but there are some differences. First, we omit `break`, function calls, and `return`. Second, we include arrays, which are almost always used in CUDA programs, and barrier synchronization.

2.1. Formal Syntax

We assume countable, disjoint sets of variables LV_n and SV_n for each nonnegative integer n. Elements of LV_n and SV_n are thread local and shared variables of arrays of dimension n, respectively (when $n = 0$, they are considered scalars). We also fix the set of n-ary operations Op_n for each n. We assume that the standard arithmetic and logical operations, such as `+`, `<`, `&&` and `!`, are included in the language.

Well-formed expressions e and programs P are defined as

$$e ::= \texttt{tid} \mid \texttt{ntid} \mid x_n\,\texttt{[}\bar{e}\texttt{]} \mid f_n(\bar{e})$$
$$P ::= x_n\,\texttt{[}\bar{e}\texttt{]} \texttt{ := } e \mid \texttt{skip} \mid \texttt{sync} \mid P;\ P' \mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P' \mid \texttt{while}\ e\ \texttt{do}\ P.$$

where x_n and f_n range over $LV_n \cup SV_n$ and Op_n, respectively, and $\bar{e}$ stands for the sequence $e_1, \ldots, e_n$.

The expressions include special constants `tid`, thread identifier, and `ntid`, the number of threads.[2] If a variable x is of dimension 0, we write x instead of $x\texttt{[]}$.

$x_n\,\texttt{[}e_1,\ \ldots,\ e_n\texttt{]} \texttt{ := } e$ is an assignment, which is performed by all active threads in parallel. `skip` is a statement that has no effect. `sync` is a barrier synchronization, typically used to avoid data races in CUDA. Although in the semantics defined in this section barrier synchronization does not play an important role, it is essential for the discussion of thread interleaving in Section 5. The remaining constructs are the same as the usual while-language. Note that we do not include Boolean expressions, and therefore we use integer expressions for conditions of `if`- and `while`-statements, and regard any nonzero value as true.

2.2. Operational Semantics

Next, we define a formal semantics for the language introduced above. For simplicity, arrays are represented simply by total maps from tuples of integers to integers. We do not care about array bounds, and negative indices are also allowed. Our operational semantics basically follows the standard evaluation rules, but one of the main differences is that it is nondeterministic, because multiple threads may attempt to write into the same shared variable simultaneously.

Below we fix a positive integer N that specifies the number of threads and therefore is the interpretation of the constant `ntid`. We also assume that for each n-ary operation f_n, a map from $\mathbb{Z}^n$ to $\mathbb{Z}$ (also denoted by f_n) is assigned. We denote the set of threads $\{0, 1, \ldots, N-1\}$ by $\mathbb{T}$.

Definition 2.1. A *state* σ consists of a map $\sigma(x) : \mathbb{T} \to \mathbb{Z}^n \to \mathbb{Z}$ for each $x \in LV_n$, and $\sigma(y) : \mathbb{Z}^n \to \mathbb{Z}$ for each $y \in SV_n$.

Given a state σ, we naturally interpret $\sigma(x)$ as the value of x.

The denotation of an expression e under a state σ is a map $\sigma\,[\![e]\!] : \mathbb{T} \to \mathbb{Z}$ defined by

$$\sigma\,[\![\texttt{tid}]\!]\,(i) = i \qquad\qquad \sigma\,[\![\texttt{ntid}]\!]\,(i) = N$$

[2]The name of this constant is taken from a special register in PTX [NVIDIA 2015]. In our formalization, this is the same as the number of threads, although this is not always the case for PTX.

$$\texttt{skip}, \mu, \sigma \Downarrow \sigma \qquad \text{(E-SKIP)} \qquad\qquad \frac{\mu = \mathbb{T} \text{ or } \mu = \emptyset}{\texttt{sync}, \mu, \sigma \Downarrow \sigma} \quad \text{(E-SYNC)}$$

$$\frac{\begin{array}{c} x \text{ is local} \qquad \sigma'(y) = \sigma(y) \text{ for each variable } y \neq x \\ \sigma'(x)(i) = \sigma(x)(i) \text{ for each } i \notin \mu \\ \sigma'(x)(i) = \sigma(x)(i)\,[\sigma\,[\![\bar{e}]\!]\,(i) \mapsto \sigma\,[\![e]\!]\,(i)] \text{ for each } i \in \mu \end{array}}{x\,\texttt{[}\bar{e}\texttt{]}\ \texttt{:=}\ e, \mu, \sigma \Downarrow \sigma'} \quad \text{(E-LASSIGN)}$$

$$\frac{\begin{array}{c} x \text{ is shared} \qquad \sigma'(y) = \sigma(y) \text{ for each variable } y \neq x \\ \text{for all } \bar{n}, \begin{cases} \text{if } \forall i \in \mu.\sigma\,[\![\bar{e}]\!]\,(i) \neq \bar{n}, \text{ then } \sigma'(x)(\bar{n}) = \sigma(x)(\bar{n}) \\ \text{otherwise, } \exists i \in \mu.\sigma\,[\![\bar{e}]\!]\,(i) = \bar{n} \text{ and } \sigma'(x)(\bar{n}) = \sigma\,[\![e]\!]\,(i) \end{cases} \end{array}}{x\,\texttt{[}\bar{e}\texttt{]}\ \texttt{:=}\ e, \mu, \sigma \Downarrow \sigma'} \quad \text{(E-SASSIGN)}$$

$$\frac{P, \mu, \sigma \Downarrow \sigma' \qquad Q, \mu, \sigma' \Downarrow \sigma''}{P;\ Q, \mu, \sigma \Downarrow \sigma''} \quad \text{(E-SEQ)}$$

$$\frac{P, \mu \cap \sigma\,[\![e]\!]\,, \sigma \Downarrow \sigma' \qquad Q, \mu \setminus \sigma\,[\![e]\!]\,, \sigma' \Downarrow \sigma''}{\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q, \mu, \sigma \Downarrow \sigma''} \quad \text{(E-IF)}$$

$$\frac{\mu \cap \sigma\,[\![e]\!] \neq \emptyset \qquad P, \mu \cap \sigma\,[\![e]\!]\,, \sigma \Downarrow \sigma' \qquad \texttt{while}\ e\ \texttt{do}\ P, \mu \cap \sigma\,[\![e]\!]\,, \sigma' \Downarrow \sigma''}{\texttt{while}\ e\ \texttt{do}\ P, \mu, \sigma \Downarrow \sigma''} \quad \text{(E-WHILETRUE)}$$

$$\frac{\mu \cap \sigma\,[\![e]\!] = \emptyset}{\texttt{while}\ e\ \texttt{do}\ P, \mu, \sigma \Downarrow \sigma} \quad \text{(E-WHILEFALSE)}$$

Fig. 1. Lockstep semantics of GPU kernels.

$$\sigma\,[\![x\,\texttt{[}e_1,\ \ldots,\ e_n\texttt{]}]\!]\,(i) = \begin{cases} \sigma(x)(i)(\sigma\,[\![e_1]\!]\,(i), \ldots, \sigma\,[\![e_n]\!]\,(i)) & \text{if } x \text{ is local} \\ \sigma(x)(\sigma\,[\![e_1]\!]\,(i), \ldots, \sigma\,[\![e_n]\!]\,(i)) & \text{if } x \text{ is shared} \end{cases}$$
$$\sigma\,[\![f(e_1,\ \ldots,\ e_n)]\!]\,(i) = f(\sigma\,[\![e_1]\!]\,(i), \ldots, \sigma\,[\![e_n]\!]\,(i)).$$

NOTATION 2.2. *For a state σ, we define $\sigma[x \mapsto a]$ as the state σ' such that: $\sigma'(x) = a$ and $\sigma'(y) = \sigma(y)$ for each $y \neq x$.*

When an expression is used as a predicate (e.g., the condition part of an `if`-statement), we regard $\sigma\,[\![e]\!]$ as a set of threads satisfying the condition e, that is, the set $\{\, i \in \mathbb{T} \mid \sigma\,[\![e]\!]\,(i) \neq 0 \,\}$. We also use the notation $\sigma\,[\![e]\!]$ to denote this set, when no confusion arises.

The execution of a program is defined as a relation of the form

$$P, \mu, \sigma \Downarrow \sigma',$$

where P is a program, $\mu \subseteq \mathbb{T}$, and σ and σ' are states. This relation means that "if P is executed with mask μ and initial state σ, and if P terminates, then the resulting state is σ'."

The evaluation rules are listed in Figure 1. A barrier synchronization succeeds only if all threads are active or no thread is active, and hence, the set of active threads should be either $\mathbb{T}$ or $\emptyset$ in the rule E-SYNC. A synchronization does not change the state. The case where $\mu \neq \mathbb{T}$ and $\mu \neq \emptyset$ corresponds to *barrier divergence* [Betts et al. 2012], for which there is no evaluation rule. This means that there is no σ' such that $P, \mu, \sigma \Downarrow \sigma'$ if P causes barrier divergence (for given μ and σ), although, on real GPUs, such a program may terminate with an unpredictable state.[3] The rule E-IF means that both branches are executed serially but under different masks: the mask $\mu \cap \sigma [\![e]\!]$ for P is the set of threads where e holds and the second is its (relative) complement (in μ).

Nondeterministic behavior can arise from E-SASSIGN; there can be more than one choice of σ', in the case of a data race. More precisely, by a data race here we mean a situation where there exist two (or more) distinct active threads i and j, where the index $\bar{e}$ takes the same value on i and j, while e does not (formally, $\sigma [\![\bar{e}]\!] (i) = \sigma [\![\bar{e}]\!] (j)$ and $\sigma [\![e]\!] (i) \neq \sigma [\![e]\!] (j)$). In such a case, following Habermaier and Knapp [Habermaier and Knapp 2012], we allow either $\sigma [\![e]\!] (i)$ or $\sigma [\![e]\!] (j)$ to be selected as the value to be set to $x\,[\bar{e}]$.

3. REASONING ABOUT GPU KERNELS

In this section, we describe the extension of Hoare Logic to the language formalized in the previous section.

3.1. Assertion Language

Our assertion language is based on first-order logic with function variables. We assume as many n-ary variables as we want for each nonnegative integer n. Formally, the syntax is

$$\begin{aligned} \text{terms } t &::= c \mid f_n(t_1, ..., t_n) \mid x_n(t_1, ..., t_n) \\ \text{formulas } \varphi &::= p_n(t_1, ..., t_n) \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \to \varphi_2 \mid \neg\varphi \mid \forall x.\varphi \mid \exists x.\varphi. \end{aligned}$$

Here, c ranges over constant symbols, and f_n, x_n, and p_n range over n-ary function symbols, variables, and predicate symbols, respectively.

We assume our assertion language contains N (the number of threads) as a constant symbol, and each operation $f \in Op_n$ as an n-ary function symbol. Additional constants and function symbols are allowed. We also assume that standard predicates on integers such as $\leq$ are included.

We associate a unique variable to each program variable. A variable that is not associated to any program variable is called a specification variable. We denote the variable corresponding to a program variable x again by x. Each $x \in SV_n$ is n-ary, and each $x \in LV_n$ is $(n+1)$-ary. This is because a local variable's value varies among threads: a local variable has to receive a thread identifier as one additional argument to determine its value. We assume that the first argument of a local variable always represents a thread identifier.

An assertion is just a formula of the first-order logic. We briefly describe how to interpret it. First, we fix a model $\mathcal{M}$ of our first-order signature, with domain $\mathbb{Z}$, such that the interpretation of `ntid` is N that we fixed above, and the interpretation of each $f_n \in Op_n$ also equals the function used to define the denotation of an expression. An *assignment* is a map that assigns to (both program and specification) variables of arity n a map $\mathbb{Z}^n \to \mathbb{Z}$. The satisfaction relation $\rho \models \varphi$ for each assignment ρ and a formula φ is defined as usual.

Precisely, we have to distinguish program states from assignments, but for brevity we frequently regard assignments as program states (by restricting their domain to the set of program variables) if no confusion arises. We write $P, \mu, \sigma \Downarrow \sigma'$ when σ and σ' are assignments, the precise meaning of which is the following: it holds that $P, \mu, |\sigma| \Downarrow |\sigma'|$ (where $|\sigma|$ and $|\sigma'|$ denote σ and σ' restricted to the program variables) and that σ and σ' agree on specification variables. We also use the notation $\sigma [\![e]\!]$ for the

[3] The behavior for barrier divergence is indeed undefined in the specification. See Section 7 for more discussions.

set $\{ i \in \mathbb{T} \mid \sigma [\![e]\!] (i) \neq 0 \}$, where σ is an assignment and e is a term that may contain specification variables.

Definition 3.1. A *Hoare quadruple* is of the form $\{\varphi\}\, m \Rightarrow P\, \{\psi\}$, where P is a program, m is an expression built from fresh variables, and φ and ψ are formulas. Note that no variable occurring in m occurs in P.

Definition 3.2. A Hoare quadruple $\{\varphi\}\, m \Rightarrow P\, \{\psi\}$ is *valid* if, for every assignment σ satisfying φ and every σ' such that $P, \sigma [\![m]\!], \sigma \Downarrow \sigma'$, it holds that $\sigma' \models \psi$.

Definition 3.3. For an expression e and a term t, we define a term $e@t$ as

$$\texttt{tid}@t = t \qquad\qquad \texttt{ntid}@t = N$$
$$(x\,\texttt{[}e_1, \dots, e_n\texttt{]})@t = \begin{cases} x(t, e_1@t, \dots, e_n@t) & \text{if } x \text{ is local} \\ x(e_1@t, \dots, e_n@t) & \text{if } x \text{ is shared} \end{cases}$$
$$(f(e_1, \dots, e_n))@t = f(e_1@t, \dots, e_n@t).$$

The intended meaning of $e@t$ is the value of e at thread t.

NOTATION 3.4. *We occasionally use* $\mathbb{T}$ *in place of* m *when* m *is an expression that is nonzero in all threads (*1*, for example).*

Definition 3.5. We use the following abbreviations.

— $\mathit{all}(e) := (\forall i.0 \leq i < N \to e@i \neq 0)$
— $\mathit{none}(e) := (\forall i.0 \leq i < N \to e@i = 0)$
— $i \in m := (m@i \neq 0)$
— $\forall i \in m.\varphi := (\forall i.0 \leq i < N \to m@i \neq 0 \to \varphi)$. Similarly for $\exists$ and other variants.
— If x is a shared variable, $\mathit{assign}(x', m, x, \bar{e}, e)$ is defined as

$$\forall \bar{n}.\, ((\forall i \in m.\bar{e}@i \neq \bar{n}) \wedge x'(\bar{n}) = x(\bar{n})) \vee\ (\exists i \in m.\bar{e}@i = \bar{n} \wedge x'(\bar{n}) = e@i)\,,$$

and, if x is local,

$$\forall \bar{n}, i.\, (i \notin m \vee \bar{e}@i \neq \bar{n} \to x'(i, \bar{n}) = x(i, \bar{n})) \wedge (i \in m \wedge \bar{e}@i = \bar{n} \to x'(i, \bar{n}) = e@i)\,.$$

Intuitively, $\mathit{assign}(x', m, x, \bar{e}, e)$ is true when x' is (one of) the result(s) of executing $x\,\texttt{[}\bar{e}\texttt{]} \texttt{ := } e$ with mask m. If x is shared, this is the case if, for each index $\bar{n}$, either

— no thread modifies $x(\bar{n})$ and $x'(\bar{n})$ equals the original value $x(\bar{n})$, or
— some (possibly multiple) threads try to modify $x(\bar{n})$, and $x'(\bar{n})$ equals a value written by one of these threads.

The description is complicated because of possible data races. The case where x is local is similar, but the situation is simpler because there is no data race.

We can state the meaning of assign formally as follows.

LEMMA 3.6. $x\,\texttt{[}\bar{e}\texttt{]} \texttt{ := } e, \sigma [\![m]\!], \sigma \Downarrow \sigma'$ *holds if and only if there exists an assignment* a *such that* $\sigma' = \sigma[x \mapsto a]$*, and* $\sigma[x' \mapsto a] \models \mathit{assign}(x', m, x, \bar{e}, e)$.

3.2. Inference Rules

The inference rules are listed in Figure 2. We write $\vdash \{\varphi\}\, m \Rightarrow P\, \{\psi\}$ if the quadruple $\{\varphi\}\, m \Rightarrow P\, \{\psi\}$ is provable from the rules in Figure 2. The variables x' in H-ASSIGN and z in H-IF and H-WHILE are fresh variables of an appropriate arity. The expression $e = z$ appearing in H-IF and H-WHILE is shorthand for $\forall i \in \mathbb{T}.e@i = z@i$.

Rules H-CONSEQ, H-SKIP, and H-SEQ are standard. H-ASSIGN appears to be different from the standard assignment rule of Hoare Logic, but in view of Lemma 3.6 this would be natural (see also Remark 3.7 below). H-SYNC is also understood similarly.

$$\{\varphi\}\, m \Rightarrow \texttt{skip}\, \{\varphi\} \tag{H-SKIP}$$

$$\{all(m) \vee none(m) \rightarrow \varphi\}\, m \Rightarrow \texttt{sync}\, \{\varphi\} \tag{H-SYNC}$$

$$\frac{\models \varphi' \rightarrow \varphi \qquad \{\varphi\}\, m \Rightarrow P\, \{\psi\} \qquad \models \psi \rightarrow \psi'}{\{\varphi'\}\, m \Rightarrow P\, \{\psi'\}} \tag{H-CONSEQ}$$

$$\frac{\{\varphi\}\, m \Rightarrow P\, \{\psi\} \qquad \{\psi\}\, m \Rightarrow Q\, \{\chi\}}{\{\varphi\}\, m \Rightarrow P;\ Q\, \{\chi\}} \tag{H-SEQ}$$

$$\{\forall x'.\mathit{assign}(x', m, x, \bar{e}, e) \rightarrow \varphi[x'/x]\}\, m \Rightarrow x\,[\bar{e}]\ \texttt{:=}\ e\, \{\varphi\} \tag{H-ASSIGN}$$

$$\frac{\{\varphi \wedge e = z\}\, m\ \texttt{\&\&}\ z \Rightarrow P\, \{\psi\} \qquad \{\psi\}\, m\ \texttt{\&\&}\ \texttt{!}\ z \Rightarrow Q\, \{\chi\}}{\{\varphi\}\, m \Rightarrow \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q\, \{\chi\}} \tag{H-IF}$$

$$\frac{\{\varphi \wedge e = z\}\, m\ \texttt{\&\&}\ z \Rightarrow P\, \{\varphi\}}{\{\varphi\}\, m \Rightarrow \texttt{while}\ e\ \texttt{do}\ P\, \{\varphi \wedge none(m\ \texttt{\&\&}\ e)\}} \tag{H-WHILE}$$

Fig. 2. Inference rules.

Rules H-IF and H-WHILE are more interesting. Since an `if` statement executes both then- and else-branches sequentially, the precondition of the second premise is ψ (the postcondition of the first), not φ. In both rules, we have to remember the initial value of e into a fresh variable z (see Remark 3.8 below). Since the threads in which the condition is false do not execute the body, the mask part of the premises has to be m `&&` z (or m `&& !` z).

Remark 3.7. The standard assignment rule of Hoare Logic

$$\{\varphi[e/x]\}x\ \texttt{:=}\ e\{\varphi\}$$

is equivalent to

$$\{\forall x'.x' = e \rightarrow \varphi[x'/x]\}x\ \texttt{:=}\ e\{\varphi\},$$

which has the same form as H-ASSIGN.

Remark 3.8. We introduce a fresh variable z in rules H-IF and H-WHILE. To see that this is indeed necessary, suppose the rule is of the form

$$\frac{\{\varphi\}\, m\ \texttt{\&\&}\ e \Rightarrow P\, \{\psi\} \qquad \{\psi\}\, m\ \texttt{\&\&}\ \texttt{!}\ e \Rightarrow Q\, \{\chi\}}{\{\varphi\}\, m \Rightarrow \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q\, \{\chi\}}$$

(although this is actually ill-formed because the mask part may contain variables that are not fresh). Let x and y be shared variables and $e = (x\ \texttt{>}\ \texttt{0})$, $P = (x\ \texttt{:=}\ \texttt{0};\ y\ \texttt{:=}\ \texttt{1})$, and $Q = \texttt{skip}$. Then, the following is valid:

$$\{x@0 > 0\}\, \mathbb{T} \Rightarrow \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q\, \{y@0 = 1\}\, .$$

To prove this by using the above rule, we attempt to prove

$$\{x@0 > 0\}\, x\ \texttt{>}\ \texttt{0} \Rightarrow P\, \{y@0 = 1\}\, ,$$

but this is impossible because the verification condition would be

$$x@0 > 0 \rightarrow \forall x'.\mathit{assign}(x', x\ \texttt{>}\ \texttt{0}, x, \cdot, 0) \rightarrow \forall y'.\mathit{assign}(y', x'\ \texttt{>}\ \texttt{0}, y, \cdot, 1) \rightarrow y'@0 = 1,$$

which is not true: $x@0 > 0$ implies $x'@0 = 0$, but we can prove $y'@0 = 1$ only if $x'@0 > 0$.

The problem is that, when executing $y\ \texttt{:=}\ \texttt{1}$, the actual mask is represented by $x\ \texttt{>}\ \texttt{0}$, whereas in the above verification condition it is incorrectly replaced by $x'\ \texttt{>}\ \texttt{0}$. This does not occur in the actual rule H-IF, because, instead of e being directly evaluated, the value of e at the point just before branching is referenced through a fresh variable z.

PROPOSITION 3.9. *Our Hoare Logic admits the disjunction and conjunction rules:*

$$\frac{\{\varphi_1\}\, m \Rightarrow P\, \{\psi_1\} \qquad \{\varphi_2\}\, m \Rightarrow P\, \{\psi_2\}}{\{\varphi_1 \wedge \varphi_2\}\, m \Rightarrow P\, \{\psi_1 \wedge \psi_2\}}\,, \frac{\{\varphi_1\}\, m \Rightarrow P\, \{\psi_1\} \qquad \{\varphi_2\}\, m \Rightarrow P\, \{\psi_2\}}{\{\varphi_1 \vee \varphi_2\}\, m \Rightarrow P\, \{\psi_1 \vee \psi_2\}}\,.$$

PROOF. By induction on the derivations of $\{\varphi_i\}\, m \Rightarrow P\, \{\psi_i\}$. Consider the following cases separately: (1) one of the derivations ends with H-CONSEQ, and (2) both of the derivations end with the same rule (uniquely determined by P), which is other than H-CONSEQ. The proofs are straightforward in both cases. □

3.3. Examples

3.3.1. Vector addition. Let us consider the program that appeared in Section 1.1. When this program is called with N threads, each thread i writes a`[`k`]` + b`[`k`]` into c`[`k`]` for $k = i, N + i, 2N + i, \ldots$ until k exceeds the length n of the arrays. Therefore, after this program terminates, the value of c should be the sum of a and b. More precisely, letting P be the program in Section 1.1, it holds that

$$\{\}\, \mathbb{T} \Rightarrow P\, \{\forall i. 0 \leq i < n \rightarrow c(i) = a(i) + b(i)\}\,.$$

Note that in the postcondition we have to write $c(i)$, not $c@i$, because c is a shared variable and i is the index specified in the program (and similarly for a and b). We can prove this quadruple using the loop invariant

$$\forall i \in \mathbb{T}. \exists l. k@i = lN + i \wedge \forall l'. 0 \leq l' < l \rightarrow c(l'N + i) = a(l'N + i) + b(l'N + i).$$

This formula asserts that at the beginning and end of each iteration, the value of k at thread i is of the form $lN + i$, and all elements at indices $i, N + i, \ldots, (l - 1)N + i$ have been processed correctly. Here, the value of l is the number of iterations that thread i has performed.

3.3.2. Array sum. For simplicity, we assume the number N of threads is a power of 2, and a is an array of length $n = 2N$. Consider the following program P:

```
s = n / 2;
while (s > 0) {
  if (tid < s) a[tid] = a[tid] + a[tid + s];
  s = s / 2;
  sync;
}
```

When this program has been executed, the value of a`[0]` is the sum of all values in the original array a. Intuitively, this program implements the following algorithm. In each iteration, we split a given array into two arrays a_1 and a_2 of equal length (s in the program). Then, we compute the elementwise sum $a_1 + a_2$, and store the result into a_1. This process is continued until the length of the array becomes 1. The final value of the 0-th element is the answer.

The following is an invariant:

$$\exists l \geq 0. \left(\forall i \in \mathbb{T}. s@i = 2^l/2\right) \wedge 2^l/2 \leq N \wedge \forall j. \left(0 \leq j < 2^l \rightarrow a(j) = \textstyle\sum_k a_0(j + 2^l k)\right).$$

Here, a_0 denotes the initial value of a, and the variable k in $\sum_k a_0(j + 2^l k)$ ranges over all nonnegative integers such that $j + 2^l k < n$. The expression $2^l/2$ is interpreted to be 0 when $l = 0$. We can verify that

$$\left\{n = 2N = 2^{t+1} \wedge a = a_0\right\} \mathbb{T} \Rightarrow P \left\{a(0) = \textstyle\sum_{m=0}^{n-1} a_0(m)\right\}.$$

4. SOUNDNESS AND RELATIVE COMPLETENESS

We now prove the soundness and relative completeness. Unfortunately, however, they do *not* hold for all programs. We first describe the situation in which soundness fails and introduce the notion of monotonic loops, based on this observation. Then we prove the soundness and relative completeness for programs containing only monotonic loops.

4.1. Monotonic Loops

As a counterexample for the soundness, let us consider the program

$$e = x\texttt{[tid]} \mathrel{\texttt{==}} \texttt{tid}, \quad P = \texttt{while}\ e\ \texttt{do}\ (x\texttt{[0]} \mathrel{\texttt{:=}} 1;\ x\texttt{[1]} \mathrel{\texttt{:=}} 1),$$

where x is a shared variable and the assertion

$$\varphi = (\exists i \in \mathbb{T}.x(i) = i).$$

It can be verified that φ is an invariant:

$$\{\varphi \land z = e\}\, z \Rightarrow x\texttt{[0]} \mathrel{\texttt{:=}} 1;\ x\texttt{[1]} \mathrel{\texttt{:=}} 1\, \{\varphi\},$$

and therefore, we can prove $\{\varphi\}\, \mathbb{T} \Rightarrow P\, \{\varphi \land \mathit{none}(e)\}$. However, this is not a valid quadruple. Suppose that the initial value of x is $x\texttt{[0]} = x\texttt{[1]} = 0$. Starting from such a state, it is easy to see that P terminates with some state, say σ'. If the quadruple above is valid, it means that σ' satisfies $\varphi \land \mathit{none}(e)$. However, this formula is inconsistent. Therefore the rule H-WHILE is not sound for this example.

The problem is that initially the condition e is false at thread 1, but after the body is executed by thread 0, it becomes true at thread 1. In general, a difficulty arises when

— thread i has already exited the loop,
— another active thread j modifies some shared variable, and
— as a result, the condition e becomes true at thread i.

In fact, this is the only obstacle to proving the soundness and relative completeness. We restrict our attention to programs that do not cause this situation.

First, we define the notion of a stable expression under a given program. We say that e is stable under P, if the value of e at thread i does not change when P is executed with i being disabled. More precisely:

Definition 4.1. Let P be a program and e an expression. We say that e is *stable* under P if for all μ, σ, and σ' such that $P, \mu, \sigma \Downarrow \sigma'$, it holds that $\sigma\,[\![e]\!]\,(i) = \sigma'\,[\![e]\!]\,(i)$ for all $i \notin \mu$.

If e is stable under P, the above difficulty does not arise during the execution of the loop $\texttt{while}\ e\ \texttt{do}\ P$. Formally, this is stated as follows:

LEMMA 4.2. *Suppose e is stable under P. Then, for all μ, σ, and σ' such that $P, \mu \cap \sigma\,[\![e]\!]\,, \sigma \Downarrow \sigma'$, it holds that $\mu \cap \sigma'\,[\![e]\!] \subseteq \mu \cap \sigma\,[\![e]\!]$.*

Definition 4.3. Let us say a loop $\texttt{while}\ e\ \texttt{do}\ P$ is *monotonic* if e is stable under P. A *program with monotonic loops* is a program in which all loops are monotonic.

The following lemma gives a reasonable sufficient condition for the monotonicity.

LEMMA 4.4. *Let P be a program and e an expression. Suppose that any shared variable occurring in e does not occur on the left-hand side of any assignment in P. Then, e is stable under P.*

PROOF. It suffices to show that, if $P, \mu, \sigma \Downarrow \sigma'$, then

— $\sigma(x)(i) = \sigma'(x)(i)$ for all local x and $i \notin \mu$, and

— $\sigma(x) = \sigma'(x)$ for all shared x not occurring on the left-hand side of any assignment in P.

This is proved by induction on the derivation of $P, \mu, \sigma \Downarrow \sigma'$. $\square$

LEMMA 4.5. *Let P be a program, and assume that for any subprogram of the form* $\texttt{while}\ e\ \texttt{do}\ Q$*, e and Q satisfy the condition of Lemma 4.4. Then, P is a program with monotonic loops.*

Below we consider only programs with monotonic loops. However, this is not a limitation in practice, because it is possible to transform a loop into a monotonic one, which is equivalent to the original one (in the sense that, if they are executed under the same state with the same mask, then the set of resulting states is also the same). Indeed, given a program, its subprograms of the form $\texttt{while}\ e\ \texttt{do}\ P$ can be replaced with $z \texttt{ := } e;\ \texttt{while}\ z\ \texttt{do}\ (P;\ z \texttt{ := } e)$, where z is a fresh local variable. The program obtained by this transformation satisfies the condition of Lemma 4.5.

4.2. Soundness and Relative Completeness

After restricting our attention to monotonic loops, we can prove the soundness by verifying that each rule preserves validity. H-WHILE can be checked by induction on the number of iterations (more precisely, the height of the derivation tree of the execution relation $\Downarrow$). For details, see the electronic appendix.

THEOREM 4.6 (SOUNDNESS). *If P is a program with monotonic loops and $\{\varphi\}\, m \Rightarrow P\, \{\psi\}$ is derivable from the rules in Figure 2, then it is valid.*

Next, we consider relative completeness. The statement and proof strategy are standard for the most part, except that masks are involved in the weakest preconditions.

Definition 4.7 (*Weakest Liberal Precondition*). The weakest liberal precondition $wlp(m, P, \varphi)$ is defined as

$$wlp(m, P, \varphi) = \{\, \sigma \mid \forall \sigma'. P, \sigma(m), \sigma \Downarrow \sigma' \implies \sigma' \models \varphi \,\}\,.$$

We use $\mathrm{wlp}(m, P, \varphi)$ to denote a formula defining this set.

To prove the relative completeness, it suffices to show that (1) the weakest liberal precondition is definable in the assertion language, and (2) it holds that $\vdash \{\mathrm{wlp}(m, P, \varphi)\}\, m \Rightarrow P\, \{\varphi\}$. Definability can be checked in a standard manner [Winskel 1993]. The second claim can be proved by induction on P. When P is a while-statement, we can use the formula $\exists z. e = z \wedge \mathrm{wlp}(m \;\texttt{\&\&}\; z, P, \varphi)$ as an invariant. For details, see the electronic appendix.

THEOREM 4.8 (RELATIVE COMPLETENESS). *If P is a program with monotonic loops and $\{\varphi\}\, m \Rightarrow P\, \{\psi\}$ is valid, then it is derivable.*

5. INTERLEAVING SEMANTICS

The first part of this section introduces another semantics, which we call *interleaving semantics*, in which the execution of threads interleaves. In the second part we formalize race-freedom and formally state the soundness and relative completeness of our Hoare Logic with respect to the interleaving semantics. We defer its proof to Section 6, because it is rather long and technical. The basic idea of our formulation of the interleaving semantics is similar to the semantics considered in the literature [Habermaier and Knapp 2012; Collingbourne et al. 2013].

5.1. Definition of Interleaving Semantics

To define interleaving semantics, we slightly extend the program syntax. We add a new construct `endif` and an annotation (which we call a *label*) to each `endif`, `if`-, and `while`-statement. `endif` appears during interleaved execution, but is not supposed to be written by programmers. Labels play an essential role in the semantics to handle `sync` correctly.

The precise syntax is

$$P ::= x_n\,[\bar{e}] \texttt{ := } e \mid \texttt{skip} \mid \texttt{sync} \mid P;\ P' \mid \texttt{if}^{l}\ e\ \texttt{then}\ P\ \texttt{else}\ P' \mid \texttt{while}^{l}\ e\ \texttt{do}\ P \mid \texttt{endif}^{l}.$$

A label l ranges over a fixed set L. We assume that the set of labels L is infinite and totally ordered, and the same label does not appear more than once in a single program.

In our interleaving semantics, we have to keep track of the control flow of the execution of each thread so that we can treat `sync` correctly. According to the NVIDIA CUDA C programming guide [NVIDIA 2014],

> `__syncthreads()` is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects.

This means that if all threads reach a `sync` but under different control flows (`sync`s in different places or `sync` inside a loop with different numbers of iterations), then the execution may fail to proceed correctly. Therefore, we should design an execution rule for `sync` such that the synchronization succeeds only if all threads are in the same control flow. In the case of lockstep semantics, it was sufficient to check that the mask is either $\mathbb{T}$ or $\emptyset$, but this solution is not available in the interleaving semantics.

To this end we introduce an extra component, which we call *stack*, into a configuration of a thread. A stack is the history of branches that a thread has taken. Each element of a stack is a pair (l, k) of a label and a positive integer. If l appears in the stack of a thread configuration, then that thread is executing a statement with label l. When l is a label of an `if`-statement, k determines which of the two branches the thread is executing: $k = 1$ if the thread is executing a `then` part, and $k = 2$ otherwise. If l is a label of a `while`-statement, then (l, k) in the stack means that the thread is executing the k-th iteration of the loop.

Definition 5.1 (*I-configuration*).

— A *stack* is a list of pairs $(l, k) \in L \times (\mathbb{N} \setminus \{0\})$.
— A *thread configuration* is a pair (P, s), where P is a program or a symbol ✓, and s is a stack.
— An *I-configuration*, a configuration of interleaving semantics, is of the form $(P_i, s_i)_i, \sigma$. Here, $(P_i, s_i)_i$ is a family of thread configurations indexed by the set of threads $\mathbb{T}$ and σ is a state.

NOTATION 5.2. *For a predicate $\Phi(i)$ on threads (typically of the form $i \in \mu$), we denote by $(P_i, s_i \mid \Phi(i))$ the family of thread configurations, the i-component of which is (P_i, s_i) if $\Phi(i)$ is true, and $(\checkmark, \varepsilon)$ otherwise. A variant with multiple clauses $\begin{pmatrix} P_i, s_i & \mid \Phi_1(i) \\ Q_i, t_i & \mid \Phi_2(i) \end{pmatrix}$ is used in a similar meaning.*

The evaluation rules are listed in Figures 3 and 4. Figure 3 defines the execution of a single thread and Figure 4 defines the interleaving execution. $\bar{P}$ represents a (possibly empty) list of programs. When $\bar{P}$ is empty, $P;\ \bar{P}$ is understood as P, and the empty list is identified with ✓ if it appears alone. In the rules, we implicitly identify

$$\texttt{skip};\ \bar{P}, s, \sigma \xrightarrow{i} \bar{P}, s, \sigma \qquad \text{(T-SKIP)}$$

$$\frac{x \text{ is local} \qquad \sigma' = \sigma\left[x, i, \sigma\,[\![\bar{e}]\!]\,(i) \mapsto \sigma\,[\![e]\!]\,(i)\right]}{x\,[\bar{e}]\ \texttt{:=}\ e;\ \bar{P}, s, \sigma \xrightarrow{i} \bar{P}, s, \sigma'} \qquad \text{(T-LASSIGN)}$$

$$\frac{x \text{ is shared} \qquad \sigma' = \sigma\left[x, \sigma\,[\![\bar{e}]\!]\,(i) \mapsto \sigma\,[\![e]\!]\,(i)\right]}{x\,[\bar{e}]\ \texttt{:=}\ e;\ \bar{P}, s, \sigma \xrightarrow{i} \bar{P}, s, \sigma'} \qquad \text{(T-SASSIGN)}$$

$$\frac{\sigma\,[\![e]\!]\,(i) \neq 0}{\texttt{if}^l\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2;\ \bar{P}, s, \sigma \xrightarrow{i} P_1;\ \texttt{endif}^l;\ \bar{P}, s \cdot (l, 1), \sigma} \qquad \text{(T-IFTRUE)}$$

$$\frac{\sigma\,[\![e]\!]\,(i) = 0}{\texttt{if}^l\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2;\ \bar{P}, s, \sigma \xrightarrow{i} P_2;\ \texttt{endif}^l;\ \bar{P}, s \cdot (l, 2), \sigma} \qquad \text{(T-IFFALSE)}$$

$$\texttt{endif}^l;\ \bar{P}, s \cdot (l, k), \sigma \xrightarrow{i} \bar{P}, s, \sigma \qquad \text{(T-ENDIF)}$$

$$\frac{\sigma\,[\![e]\!]\,(i) \neq 0}{\texttt{while}^l\ e\ \texttt{do}\ P;\ \bar{P}, s, \sigma \xrightarrow{i} P;\ \texttt{while}^l\ e\ \texttt{do}\ P;\ \bar{P}, s + l, \sigma} \qquad \text{(T-WHILETRUE)}$$

$$\frac{\sigma\,[\![e]\!]\,(i) = 0}{\texttt{while}^l\ e\ \texttt{do}\ P;\ \bar{P}, s, \sigma \xrightarrow{i} \bar{P}, s \setminus l, \sigma} \qquad \text{(T-WHILEFALSE)}$$

Fig. 3. Thread execution of GPU kernels.

$$\frac{P_i, s_i, \sigma \xrightarrow{i} P', s', \sigma'}{(P_i, s_i)_i, \sigma \rightarrow_I (P_i, s_i)_i\,[i \mapsto (P', s')]\,, \sigma'} \qquad \text{(I-THREAD)}$$

$$\frac{\forall i, j. s_i = s_j}{(\texttt{sync};\ \bar{P}_i, s_i)_i, \sigma \rightarrow_I (\bar{P}_i, s_i)_i, \sigma} \qquad \text{(I-SYNC)}$$

Fig. 4. Interleaving semantics of GPU kernels.

programs up to associativity of sequential composition (that is, we identify $(P_1;\ P_2);\ P_3$ with $P_1;\ (P_2;\ P_3)$). The same convention is frequently used hereafter in this article.

In the assignment rules, we used the notation $\sigma[x, i, \bar{n} \mapsto v]$ to denote the state σ' such that $\sigma'(x)(i)(\bar{n}) = v$ and other values are the same as σ.

The operations $+$ and $\setminus$ used in rules T-WHILETRUE and T-WHILEFALSE are defined as

$$s + l = \begin{cases} s' \cdot (l, k+1) & \text{if } s = s' \cdot (l, k) \\ s \cdot (l, 1) & \text{otherwise,} \end{cases} \qquad s \setminus l = \begin{cases} s' & \text{if } s = s' \cdot (l, k) \\ s & \text{otherwise.} \end{cases}$$

The rules for `if`, `while`, and `sync` modify the stacks. T-IFTRUE and T-IFFALSE push $(l, 1)$ and $(l, 2)$, respectively, on the stack, and T-ENDIF pops the element (l, k) out of the stack (if the labels in the statement and the stack agree). T-WHILETRUE increments the second component (the number of iterations) of the stack and T-WHILEFALSE removes (l, k) if it is the top element of the stack.

In I-THREAD, $(P_i, s_i)_i\,[i \mapsto (P', s')]$ denotes the family of thread configurations, the i-component of which is replaced by (P', s'). The rule I-SYNC checks whether the stacks of all threads agree, that is, all threads are in the same control flow.

5.2. Race-Freedom and Equivalence

To define race-freedom, we first define a read set, which describes which part of the shared memory is accessed when an expression is evaluated. The read set is represented by a set of pairs of the form $\langle x, \bar{n} \rangle$, where x is a shared variable and $\bar{n}$ is a sequence of integers of appropriate length (the dimension of x).

Below, by abuse of notation, when $\ell = \langle x, \bar{n} \rangle$ we write $\sigma(\ell)$ for $\sigma(x)(\bar{n})$.

Definition 5.3 (*Read Set, Write Set*). We define the function Rd for expressions as

$$
\begin{aligned}
\mathrm{Rd}(\texttt{tid}, \sigma, i) &= \mathrm{Rd}(\texttt{ntid}, \sigma, i) = \emptyset \\
\mathrm{Rd}(x\,\texttt{[}\bar{e}\texttt{]}, \sigma, i) &= \begin{cases} \{\langle x, \sigma\,[\![\bar{e}]\!]\,(i)\rangle\} \cup \mathrm{Rd}(\bar{e}, \sigma, i) & \text{if } x \text{ is shared} \\ \mathrm{Rd}(\bar{e}, \sigma, i) & \text{if } x \text{ is local} \end{cases} \\
\mathrm{Rd}(f(\bar{e}), \sigma, i) &= \mathrm{Rd}(\bar{e}, \sigma, i) \\
\mathrm{Rd}(\bar{e}, \sigma, i) &= \bigcup_{e \in \bar{e}} \mathrm{Rd}(e, \sigma, i).
\end{aligned}
$$

For programs, we define:

$$
\begin{aligned}
\mathrm{Rd}(x\,\texttt{[}\bar{e}\texttt{]} \texttt{ := } e, \sigma, i) &= \mathrm{Rd}(\bar{e}, \sigma, i) \cup \mathrm{Rd}(e, \sigma, i) \\
\mathrm{Rd}(\texttt{skip}, \sigma, i) = \mathrm{Rd}(\texttt{sync}, \sigma, i) = \mathrm{Rd}(\texttt{endif}^{l}, \sigma, i) &= \mathrm{Rd}(\checkmark, \sigma, i) = \emptyset \\
\mathrm{Rd}(\texttt{if}^{l}\, e\, \texttt{then}\, P\, \texttt{else}\, P', \sigma, i) &= \mathrm{Rd}(e, \sigma, i) \\
\mathrm{Rd}(\texttt{while}^{l}\, e\, \texttt{do}\, P, \sigma, i) &= \mathrm{Rd}(e, \sigma, i).
\end{aligned}
$$

For an assignment to a shared variable x, we define Wr as

$$\mathrm{Wr}(x\,\texttt{[}\bar{e}\texttt{]} \texttt{ := } e, \sigma, i) = (\langle x, \sigma\,[\![\bar{e}]\!]\,(i)\rangle, \sigma\,[\![e]\!]\,(i)).$$

Definition 5.4 (*Race-freedom*).

(1) An I-configuration $(P_i, s_i)_i, \sigma$ is said to be *racy* if there exist two distinct threads i and j such that either
  (a) the first statement of P_i is an assignment to a shared variable, $\mathrm{Wr}(P_i, \sigma, i) = (l, v)$, $\sigma(l) \neq v$, and $l \in \mathrm{Rd}(P_j, \sigma, j)$, or
  (b) the first statements of P_i and P_j are both assignments to the same shared variable, $\mathrm{Wr}(P_i, \sigma, i) = (\ell_i, v_i)$, $\mathrm{Wr}(P_j, \sigma, j) = (\ell_j, v_j)$, $\ell_i = \ell_j$ and $v_i \neq v_j$.

(2) An I-configuration is said to be *race-free* if it cannot reach a racy I-configuration.

Note that we do not consider writes by multiple threads as a race if the values being written are the same. This is because this type of race (sometimes called a benign race) is common in practice, and considered tolerable [Betts et al. 2012].

We can prove that the race-freedom defined above implies the equivalence of interleaving and lockstep semantics. The proof is given in Section 6.

THEOREM 5.5. *Let P be a program and μ a mask and suppose that $(P, \varepsilon \mid i \in \mu), \sigma$ is race-free. Then, $P, \mu, \sigma \Downarrow \sigma'$ if and only if $(P, \varepsilon \mid i \in \mu), \sigma \rightarrow_I^* (\checkmark, \varepsilon)_i, \sigma'$.*

From this theorem, together with the results of Section 4, soundness and relative completeness with respect to the interleaving semantics follow.

COROLLARY 5.6. *Let P be a program with monotonic loops and suppose that $\{\varphi\}\, m \Rightarrow P\, \{\psi\}$ is derivable. Let σ be a state such that the I-configuration $(P, \varepsilon \mid i \in \sigma\,[\![m]\!]), \sigma$ is race-free, $\sigma \models \varphi$ holds, and $(P, \varepsilon \mid i \in \sigma\,[\![m]\!]), \sigma \rightarrow_I^* (\checkmark, \varepsilon)_i, \sigma'$. Then, it holds that $\sigma' \models \psi$.*

COROLLARY 5.7. *Let P be a program with monotonic loops such that $(P, \varepsilon \mid i \in \sigma\,[\![m]\!]), \sigma$ is race-free for all σ such that $\sigma \models \varphi$. Then, $\{\varphi\}\, m \Rightarrow P\, \{\psi\}$ is derivable if for all σ and σ' such that $\sigma \models \varphi$ and $(P, \varepsilon \mid i \in \sigma\,[\![m]\!]), \sigma \rightarrow_I^* (\checkmark, \varepsilon)_i, \sigma'$ it holds that $\sigma' \models \psi$.*

6. PROOF OF EQUIVALENCE

This section is devoted to showing that the lockstep and interleaving semantics are equivalent for race-free programs (Theorem 5.5). As the proof is rather long, we outline the proof before providing the details.

We first introduce a derivation search procedure for the lockstep semantics in Section 6.1. This is a procedure to construct a derivation of $P, \mu, \sigma \Downarrow \sigma'$ for some (initially unknown) σ' step by step. The soundness and completeness of this procedure are proved: a derivation produced by this procedure is always valid and any valid derivation can be produced by this procedure. This procedure can be regarded as a small-step version of the lockstep semantics. A small-step semantics is more convenient when comparing lockstep and interleaving semantics, as the latter is defined as a small-step semantics (a similar approach has been considered by Gunter and Rémy [1993] to state and prove the absence of runtime type errors when the language has a big-step semantics).

In Section 6.2 we define a translation from a partial derivation into an I-configuration and prove that this gives a simulation: each step of the derivation search corresponds to an execution of the interleaving semantics (possibly in multiple steps). This fact implies a half of Theorem 5.5. Let us remember the statement of the theorem: under the assumption of race-freedom, it holds that

$$P, \mu, \sigma \Downarrow \sigma' \iff (P, \varepsilon \mid i \in \mu), \sigma \to_I^* (\checkmark, \varepsilon)_i, \sigma'. \tag{1}$$

The left-to-right direction follows from the simulation and the completeness of the derivation search. Additionally, it is easily checked that, if there exists an infinite sequence of derivation search, then there also exists an infinite interleaving execution sequence.

In Section 6.3, we prove that a race-free I-configuration is deterministic, that is, if an I-configuration $\mathcal{C}$ is race-free and there exists a finite sequence $\mathcal{C} \to_I^* \mathcal{C}'$ that is maximal (that is, there exists no $\mathcal{C}''$ such that $\mathcal{C}' \to_I \mathcal{C}''$), then every maximal sequence starting from $\mathcal{C}$ is also finite and ends with $\mathcal{C}'$. This means that, to prove the right-to-left direction of (1), it suffices to show that the lockstep execution terminates with *some* state. Indeed, if the lockstep execution terminates with some σ'' (that is, $P, \mu, \sigma \Downarrow \sigma''$), then by simulation it holds that $(P, \varepsilon \mid i \in \mu), \sigma \to_I^* (\checkmark, \varepsilon)_i, \sigma''$, but from determinacy the resulting state is unique, and hence $\sigma' = \sigma''$.

In Section 6.4, we prove that, if the derivation search barrier-diverges (i.e., fails at a barrier synchronization), then there exists an interleaving execution sequence that does not terminate successfully (i.e., does not end with a configuration of the form $(\checkmark, \varepsilon)_i, \sigma''$). The proof is rather involved; the problem is that, even if the lockstep execution gets stuck at a barrier, it does not mean that the interleaving execution also gets stuck at the same point, since we could choose a thread that is not synchronizing and execute it. The proof is sketched at the beginning of Section 6.4. This result implies the right-to-left direction of (1) as follows. As mentioned above, by determinacy it suffices to show that

$$(P, \varepsilon \mid i \in \mu), \sigma \to_I^* (\checkmark, \varepsilon)_i, \sigma' \implies \exists \sigma''. P, \mu, \sigma \Downarrow \sigma''.$$

Assume the negation of the right-hand side. Then, the derivation search does not terminate or barrier-diverge. In the first case, there is an infinite sequence of the interleaving execution, but this contradicts determinacy. In the second case, the result of Section 6.4 implies that the interleaving execution does not successfully terminate, but this also contradicts determinacy. Therefore, the lockstep execution has to terminate.

Although the proof of Theorem 5.5 was outlined above, a more formal proof of it is given in Section 6.5. The proof does not directly refer to the details of Sections 6.3 and

6.4; only Lemmas 6.16, 6.20, and 6.21 are used. If the reader is not interested in the proofs of these lemmas, the details of these two sections may be skipped.

6.1. Partial Derivation and Derivation Search

We first define partial judgments and partial derivations. We assume an infinite set of *state variables*, ranged over by X. A state variable is used in a partial judgment or derivation as a placeholder, which is eventually replaced with the result of the execution of the statement being executed.

Definition 6.1 (*Partial judgments*). We define *partial judgments* as

$$J ::= P, \mu, \sigma \Downarrow \Sigma \mid P, \mu, X \Downarrow X; \quad \Sigma ::= \sigma \mid X.$$

A partial judgment allows state variables to appear in place of concrete states. However, if the final state is concrete, the initial state has to be concrete.

Definition 6.2 (*Partial derivation*). *Partial derivations* are inductively defined by the rules below. We denote by $\mathscr{P}(J)$ the set of partial derivations with conclusion J. We also assume that D_1 and D_2 below do not contain a common state variable except for Σ' occurring in their conclusions.

(1) $J \in \mathscr{P}(J)$;
(2) If $D_1 \in \mathscr{P}(P_1, \mu, \sigma \Downarrow \Sigma')$ and $D_2 \in \mathscr{P}(P_2, \mu, \Sigma' \Downarrow \Sigma)$, then
$$\frac{D_1 \quad D_2}{P_1;\ P_2, \mu, \sigma \Downarrow \Sigma} \in \mathscr{P}(P_1;\ P_2, \mu, \sigma \Downarrow \Sigma);$$
(3) If $D_1 \in \mathscr{P}(P_1, \mu \cap \sigma\llbracket e \rrbracket, \sigma \Downarrow \Sigma')$ and $D_2 \in \mathscr{P}(P_2, \mu \setminus \sigma\llbracket e \rrbracket, \Sigma' \Downarrow \Sigma)$, then
$$\frac{D_1 \quad D_2}{\mathtt{if}^l\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2, \mu, \sigma \Downarrow \Sigma} \in \mathscr{P}(\mathtt{if}^l\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2, \mu, \sigma \Downarrow \Sigma);$$
(4) If $\mu \cap \sigma\llbracket e \rrbracket \neq \emptyset$, $D_1 \in \mathscr{P}(P, \mu \cap \sigma\llbracket e \rrbracket, \sigma \Downarrow \Sigma')$, and $D_2 \in \mathscr{P}(\mathtt{while}^l\ e\ \mathtt{do}\ P, \mu \cap \sigma\llbracket e \rrbracket, \Sigma' \Downarrow \Sigma)$, then
$$\frac{D_1 \quad D_2}{\mathtt{while}^l\ e\ \mathtt{do}\ P, \mu, \sigma \Downarrow \Sigma} \in \mathscr{P}(\mathtt{while}^l\ e\ \mathtt{do}\ P, \mu, \sigma \Downarrow \Sigma).$$

We say that a derivation D is *total* if it contains no state variable. Occasionally a partial derivation that is not total is said to be *strictly partial*.

Remark 6.3. As is easily seen by case analysis, $\mathscr{P}(P, \mu, X \Downarrow X')$ has actually only one element $P, \mu, X \Downarrow X'$.

Remark 6.4. The names of state variables appearing in a partial derivation are essentially irrelevant, so we implicitly replace state variables with fresh ones so that any two unrelated occurrences of variables have distinct names.

More precisely, relevant occurrences of a state variable is defined as follows. Let D be a partial derivation and X a state variable. We define the relevance of occurrences of X in D as the least equivalence relation such that, for all subderivations of D of the form

$$\frac{\begin{array}{c}\vdots\\ P_1, \mu_1, \sigma_1 \Downarrow X\end{array} \quad P_2, \mu_2, X \Downarrow X'}{J} \quad \text{or} \quad \frac{D' \quad \begin{array}{c}\vdots\\ P_2, \mu_2, \sigma_2 \Downarrow X\end{array}}{P, \mu, \sigma \Downarrow X},$$

the two occurrences of X explicitly indicated are relevant.

$$\frac{P \text{ is either } \mathtt{sync},\, \mathtt{skip}, \text{ or an assignment} \qquad P, \mu, \sigma \Downarrow \sigma'}{E[P, \mu, \sigma \Downarrow X] \longrightarrow E[P, \mu, \sigma \Downarrow X]\{\sigma'/X\}} \quad \text{(S-ATOM)}$$

$$E\left[P_1;\ P_2, \mu, \sigma \Downarrow X\right] \longrightarrow E\left[\frac{P_1, \mu, \sigma \Downarrow X' \quad P_2, \mu, X' \Downarrow X}{P_1;\ P_2, \mu, \sigma \Downarrow X}\right] \quad \text{(S-SEQ)}$$

$$\begin{array}{l} E\left[\mathtt{if}^l\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2, \mu, \sigma \Downarrow X\right] \longrightarrow \\ \quad E\left[\dfrac{P_1, \mu \cap \sigma\,[\![e]\!], \sigma \Downarrow X' \quad P_2, \mu \setminus \sigma\,[\![e]\!], X' \Downarrow X}{\mathtt{if}^l\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2, \mu, \sigma \Downarrow X}\right] \end{array} \quad \text{(S-IF)}$$

$$\frac{\mu \cap \sigma\,[\![e]\!] \neq \emptyset}{\begin{array}{l} E\left[\mathtt{while}^l\ e\ \mathtt{do}\ P, \mu, \sigma \Downarrow X\right] \longrightarrow \\ \quad E\left[\dfrac{P, \mu \cap \sigma\,[\![e]\!], \sigma \Downarrow X' \quad \mathtt{while}^l\ e\ \mathtt{do}\ P, \mu \cap \sigma\,[\![e]\!], X' \Downarrow X}{\mathtt{while}^l\ e\ \mathtt{do}\ P, \mu, \sigma \Downarrow X}\right] \end{array}} \quad \text{(S-WHILETRUE)}$$

$$\frac{\mu \cap \sigma\,[\![e]\!] = \emptyset}{E\left[\mathtt{while}^l\ e\ \mathtt{do}\ P, \mu, \sigma \Downarrow X\right] \longrightarrow E\left[\mathtt{while}^l\ e\ \mathtt{do}\ P, \mu, \sigma \Downarrow X\right]\{\sigma/X\}} \quad \text{(S-WHILEFALSE)}$$

Fig. 5. Derivation search procedure

It is always possible to rename state variables to eliminate irrelevant occurrences of the same variable. Hereafter, unless otherwise specified, we assume that partial derivations have no irrelevant occurrences of the a variable.

Next, we define a derivation search procedure. We describe the procedure as a set of rules, in which we use evaluation contexts. Therefore, we first introduce evaluation contexts.

Definition 6.5 (*Evaluation context*). An *evaluation context*, ranged over by E, is defined by the following syntax.

$$E ::= [] \;\left|\; \frac{E \quad P, \mu, X' \Downarrow X}{J} \;\right|\; \frac{D \quad E}{J}$$

Here, D denotes an arbitrary total derivation.

Application of evaluation contexts, denoted by $E[D]$, is defined as usual.

The derivation search procedure is formally described as a binary relation $\longrightarrow$ on partial derivations. The rules are listed in Figure 5, where X' is a fresh variable. In S-ATOM and S-WHILEFALSE, all occurrences of X have to be replaced by σ', because the variable X is a placeholder for the result of the execution of P.

This procedure is sound and complete in the following sense.

PROPOSITION 6.6 (SOUNDNESS). *If $P, \mu, \sigma \Downarrow X \longrightarrow^* D$ and D is a total derivation, then D is a valid derivation with respect to the rules in Figure 1.*

PROPOSITION 6.7 (COMPLETENESS). *If D_0 is a derivation of $P, \mu, \sigma \Downarrow \sigma'$ constructed from rules in Figure 1, then $P, \mu, \sigma \Downarrow X \longrightarrow^* D_0$.*

For the proofs of these propositions, see the electronic appendix.

6.2. Simulating the Derivation Search Procedure

Having defined the derivation search procedure, we wish to prove that each step of this procedure can be simulated by the interleaving execution. To achieve this, we construct a translation from partial derivations into I-configurations, denoted by $|\cdot|$, and show that this translation is a simulation between lockstep and interleaving semantics.

Unfortunately, however, the desired result is not quite true. For example, consider the program x := x + 1, where x is a shared variable. In the lockstep semantics, this program increments the value of x by 1, but in the interleaving semantics it increments x by the number of active threads. Therefore, we have to work under some assumption that excludes this situation. (Another possible approach is to split the execution rule of assignment into two phases [Habermaier and Knapp 2012]. The first phase calculates the index of the array and the value to be stored, and actual write operation is performed at the second phase.)

Definition 6.8.

(1) Let $A = x\,[\bar{e}]$:= e be an assignment to a shared variable x. An instance of the rule E-SASSIGN with conclusion $A, \mu, \sigma \Downarrow \sigma'$ is said to be *interleavable* if there exists an enumeration $i_1, \ldots, i_m$ of μ and a sequence of states $\sigma_1, \ldots, \sigma_{m-1}$ such that $A, \varepsilon, \sigma_{k-1} \xrightarrow{i_k} \checkmark, \varepsilon, \sigma_k$ for each $1 \le k \le m$, where $\sigma_0 = \sigma$ and $\sigma_m = \sigma'$.
(2) A partial derivation D is said to be *locally interleavable* if every instance of E-SASSIGN appearing in D is interleavable.

We show that the race-freedom implies the local interleavability, in Section 6.3.

In the definition of the translation $|\cdot|$ from partial derivations into I-configurations, we use the following auxiliary operation:

$$l + s = \begin{cases} (l, k+1) \cdot s' & \text{if } s = (l, k) \cdot s' \\ (l, 1) \cdot s & \text{otherwise.} \end{cases}$$

We first define, for an evaluation context E, a transformation $|E|$ of families of thread-configurations. Throughout the following definition, D is a total derivation and $(R_i, t_i) = |E|(Q_i, s_i)_i$.

$$\begin{aligned}
|[]|\,(Q_i, s_i)_i &= (Q_i, s_i)_i \\
\left|\frac{E \quad P_2, \mu, X' \Downarrow X}{P_1;\ P_2, \mu, \sigma \Downarrow X}\right| (Q_i, s_i)_i &= (R_i;\ P_2, t_i \mid i \in \mu) \\
\left|\frac{D \quad E}{P_1;\ P_2, \mu, \sigma \Downarrow X}\right| (Q_i, s_i)_i &= (R_i, t_i \mid i \in \mu) \\
\left|\frac{E \quad P_2, \mu \setminus \sigma\,[\![e]\!]\,, X' \Downarrow X}{\texttt{if}^l\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2, \mu, \sigma \Downarrow X}\right| (Q_i, s_i)_i &= \begin{pmatrix} R_i;\ \texttt{endif}^l, (l,1) \cdot t_i & \mid i \in \mu \cap \sigma\,[\![e]\!] \\ P_2;\ \texttt{endif}^l, (l,2) & \mid i \in \mu \setminus \sigma\,[\![e]\!] \end{pmatrix} \\
\left|\frac{D \quad E}{\texttt{if}^l\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2, \mu, \sigma \Downarrow X}\right| (Q_i, s_i)_i &= \big(R_i;\ \texttt{endif}^l, (l,2) \cdot t_i \mid i \in \mu \setminus \sigma\,[\![e]\!]\big) \\
\left|\frac{E \quad \texttt{while}^l\ e\ \texttt{do}\ P, \mu \cap \sigma\,[\![e]\!]\,, X' \Downarrow X}{\texttt{while}^l\ e\ \texttt{do}\ P, \mu, \sigma \Downarrow X}\right| (Q_i, s_i)_i &= \big(R_i;\ \texttt{while}^l\ e\ \texttt{do}\ P, l + t_i \mid i \in \mu \cap \sigma\,[\![e]\!]\big) \\
\left|\frac{D \quad E}{\texttt{while}^l\ e\ \texttt{do}\ P, \mu, \sigma \Downarrow X}\right| (Q_i, s_i)_i &= (R_i, l + t_i \mid i \in \mu \cap \sigma\,[\![e]\!] \text{ and } R_i \neq \checkmark)
\end{aligned}$$

$|E|$ is basically an operation that appends the continuation denoted by E. Note that, in the last case, l is not added to the stack of thread i if $R_i = \checkmark$. This is because such a thread i has already exited the loop.

We define the transformation from a partial derivation into an I-configuration by

$$|E\,[P, \mu, \sigma \Downarrow X]| = |E|(P, \varepsilon \mid i \in \mu), \sigma \qquad \text{for a strictly partial derivation;}$$

$$\left|\begin{array}{c}\vdots \\ P, \mu, \sigma \Downarrow \sigma'\end{array}\right| = (\checkmark, \varepsilon)_i, \sigma' \qquad\qquad \text{for a total derivation.}$$

PROPOSITION 6.9. *If $D \longrightarrow D'$ and D' is locally interleavable, then $|D| \to_I^* |D'|$. Moreover, if D is of the form $E[P, \mu, \sigma \Downarrow X]$ where P is not a sequencing and $\mu \neq \emptyset$, then $|D| \to_I^+ |D'|$.*

PROOF. By induction on E. For details, see the electronic appendix. $\square$

6.3. Race-Freedom and Determinacy

In this section, we prove that the race-freedom implies the determinacy and local interleavability.

We first prove the determinacy. In the proof, we use several notions about abstract rewriting system.

Definition 6.10 (*Diamond Property*). Let A be a set and $\to$ a binary relation on it.

(1) We say $(A, \to)$ has the *diamond property* if, for all $a, b, c \in A$ such that $a \to b$, $a \to c$, and $b \neq c$, there exists $d \in A$ such that $b \to d$ and $c \to d$.
(2) An element $a \in A$ is said to have the *diamond property* if $\to$ restricted to the set of all elements of A reachable from a has the diamond property.

The above definition of the diamond property is slightly different from the usual one in that we assume $b \neq c$. This assumption is redundant when the relation $\to$ is reflexive, which is often the case. However, here we need this assumption, because the relation we have in mind is $\to_I$, which is not reflexive.

Definition 6.11 (*Determinacy*). Let A be a set and $\to$ a binary relation on it. An element $a \in A$ is said to be

— *normal* if there exists no $b \in A$ such that $a \to b$;
— a *normal form* of $b \in A$ if a is normal and $b \to^* a$;
— *strongly normalizing* if there exists no infinite sequence $a = a_0 \to a_1 \to a_2 \to \dots$;
— *weakly normalizing* if it has a normal form;
— *deterministic* if either (1) it is strongly normalizing and has a unique normal form, or (2) it is not weakly normalizing.

LEMMA 6.12. *Let A be a set and $\to$ a binary relation on it, and suppose $a \in A$ has the diamond property. For any pair of elements $b, c \in A$ such that $a \to^* b$ and $a \to^* c$, there exists $d \in A$ such that $b \to^* d$ and $c \to^* d$.*

PROOF. By induction on $a \to^* b$ and $a \to^* c$. $\square$

LEMMA 6.13. *Let A be a set and $\to$ a binary relation on it. If $a \in A$ has the diamond property and is weakly normalizing, then a is strongly normalizing.*

PROOF. It suffices to show that, if $a \to b$ and b is strongly normalizing, then a is also strongly normalizing, provided that $\to$ has the diamond property. Suppose that b is strongly normalizing but a is not, and take an infinite sequence $a = a_0 \to a_1 \to a_2 \dots$. We inductively construct an infinite sequence $b = b_0 \to b_1 \to b_2 \dots$ such that $a_i \to b_i$ for each i. Suppose we have already constructed such a sequence up to b_i. From the assumption, we have $a_i \to b_i$ and $a_i \to a_{i+1}$. Moreover, $b_i \neq a_{i+1}$, because b_i is strongly normalizing (since it is reachable from b, which is strongly normalizing), while a_{i+1} is not. Therefore, from the diamond property, there exists b_{i+1} such that $b_i \to b_{i+1}$ and $a_{i+1} \to b_{i+1}$, as required. $\square$

The following is an immediate consequence of the lemmas above.

COROLLARY 6.14. *Let A be a set and $\to$ a binary relation on it. If $a \in A$ has the diamond property, then a is deterministic.*

We can apply this corollary to show that race-freedom implies determinacy.

LEMMA 6.15. *Let e be an expression, σ and σ' states such that $\sigma(\ell) = \sigma'(\ell)$ for all $\ell \in \mathrm{Rd}(e, \sigma, i)$, and $\sigma(x)(i) = \sigma'(x)(i)$ for any local variable x. Then $\sigma\, [\![e]\!]\, (i) = \sigma'\, [\![e]\!]\, (i)$.*

PROOF. By induction on e. □

LEMMA 6.16. *A race-free I-configuration has the diamond property. In particular, it is deterministic.*

PROOF. If an I-configuration has two distinct transitions, then both of them have to be derived by I-THREAD. By definition, $\xrightarrow{i}$ is deterministic for each i, and hence it suffices to show that for distinct i and j if $(P_i, s_i)_i, \sigma$ is race-free, $P_i, s_i, \sigma \xrightarrow{i} P'_i, s'_i, \sigma'$, and $P_j, s_j, \sigma \xrightarrow{j} P'_j, s'_j, \sigma''$, then there exists σ''' such that $P_i, s_i, \sigma'' \xrightarrow{i} P'_i, s'_i, \sigma'''$ and $P_j, s_j, \sigma' \xrightarrow{j} P'_j, s'_j, \sigma'''$.

This is verified by case analysis on thread execution rules in Figure 3. If both threads i and j use rules that do not modify the state, the conclusion is obvious. Suppose that at least one of the two threads uses an assignment rule. Without loss of generality, we may assume thread i uses either T-LASSIGN or T-SASSIGN, and $\sigma' = \sigma[\ell \mapsto v]$ (here, ℓ takes the form $\langle x, i, \bar{n}\rangle$ if x is local). From the race-freedom and Lemma 6.15, it follows that $\sigma'\, [\![e]\!]\, (j) = \sigma\, [\![e]\!]\, (j)$ for any expression e that is to be evaluated by thread j. Therefore, if the head of P_j is not an assignment, then $P_j, s_j, \sigma' \xrightarrow{j} P'_j, s'_j, \sigma'$, as required. If P_j is also an assignment, we have $\sigma'' = \sigma[\ell' \mapsto v']$ for some ℓ' and v'. For these ℓ' and v' we have $P_j, s_j, \sigma' \xrightarrow{j} P'_j, s'_j, \sigma'[\ell' \mapsto v']$, and therefore it suffices to show that $\sigma'[\ell' \mapsto v'] = \sigma''[\ell \mapsto v]$, that is, $\sigma[\ell \mapsto v][\ell' \mapsto v'] = \sigma[\ell' \mapsto v'][\ell \mapsto v]$ but this follows from the race-freedom. □

Next, we show that any $D \in \mathscr{P}(P, \mu, \sigma \Downarrow X)$ is locally interleavable, if the I-configuration $(P, \varepsilon \mid i \in \mu), \sigma$ (which corresponds to $P, \mu, \sigma \Downarrow X$) is race-free. In fact, we prove a stronger assertion that every judgment of the form $A, \mu, \sigma \Downarrow \sigma'$, where A is an assignment to a shared variable, is race-free in the following sense.

Definition 6.17 (*Race-free assignment*). Consider an assignment to a shared variable $A = x\,[\bar{e}]$:= e. For brevity, let us write R_i, ℓ_i, and v_i for $\mathrm{Rd}(A, \sigma, i)$, $\langle x, \sigma\, [\![\bar{e}]\!]\, (i)\rangle$, and $\sigma\, [\![e]\!]\, (i)$, respectively. Then, (A, μ, σ) is said to be *race-free* if, for any $i, j \in \mu$,

(1) if $\ell_i = \ell_j$ then $v_i = v_j$, and
(2) if $i \neq j$ and $v_i \neq \sigma(\ell_i)$ then $\ell_i \notin R_j$.

Remark 6.18. The above definition of race-freedom is consistent with Definition 5.4 in the following sense: (A, μ, σ) is race-free in the sense of Definition 6.17 if and only if $(A, \varepsilon \mid i \in \mu), \sigma$ is not racy in the sense of Definition 5.4.

We first show that a race-free assignment is interleavable.

LEMMA 6.19. *Consider an assignment to a shared variable $A = x\,[\bar{e}]$:= e, and suppose that (A, μ, σ) is race-free. Then, σ' for which $A, \mu, \sigma \Downarrow \sigma'$ is valid is unique, and this judgment is interleavable.*

PROOF. We use R_i, ℓ_i, and v_i in the same meaning as Definition 6.17. Let $\{i_1, \ldots, i_m\}$ be an arbitrary enumeration of μ.

We first check the uniqueness. Let us suppose $A, \mu, \sigma \Downarrow \sigma'$, and show that $\sigma'(\ell)$ is uniquely determined for each ℓ. If $\ell \neq \ell_{i_j}$ for every j, then $\sigma'(\ell)$ necessarily equals $\sigma(\ell)$, and hence it is unique. Otherwise, $\sigma(\ell) = v_{i_j}$ for some j with $\ell = \ell_{i_j}$. From the race-freedom, if $\ell_{i_j} = \ell_{i_k}$ then $v_{i_j} = v_{i_k}$, and hence $\sigma'(\ell)$ is unique.

Let us define

$$\sigma_k = \sigma[\ell_{i_1} \mapsto v_{i_1}] \dots [\ell_{i_k} \mapsto v_{i_k}]$$

for $0 \leq k \leq m$. It is easy to check that $A, \mu, \sigma \Downarrow \sigma_m$. To prove the interleavability, it is sufficient to show that $A, \varepsilon, \sigma_{k-1} \xrightarrow{i_k} \checkmark, \varepsilon, \sigma_k$ for $1 \leq k \leq m$. By T-SASSIGN rule, we have

$$A, \varepsilon, \sigma_{k-1} \xrightarrow{i_k} \checkmark, \varepsilon, \sigma_{k-1}[\langle x, \sigma_{k-1} [\![\bar{e}]\!] (i_k)\rangle \mapsto \sigma_{k-1} [\![e]\!] (i_k)].$$

It suffices to show that the state on the right equals σ_k. Therefore, what we have to show is $\sigma_{k-1} [\![\bar{e}]\!] (i_k) = \sigma [\![\bar{e}]\!] (i_k)$ and $\sigma_{k-1} [\![e]\!] (i_k) = \sigma [\![e]\!] (i_k)$. To prove this, by Lemma 6.15 it suffices to check that $\sigma = \sigma_{k-1}$ on R_{i_k} and $\sigma(x)(i_k) = \sigma_{k-1}(x)(i_k)$ for all local variables x. The latter part is immediate from the definition of σ_k's. Consider $\ell \in R_{i_k}$, and suppose $\sigma(\ell) \neq \sigma_{k-1}(\ell)$. Then, since the only differences between σ and σ_{k-1} are the values at $\ell_{i_1}, \dots, \ell_{i_{k-1}}$, we have $\ell = \ell_{i_j}$ for some j with $1 \leq j \leq k-1$. Then, by definition of σ_{k-1}, we have $\sigma(\ell_{i_j}) \neq \sigma_{k-1}(\ell_{i_j}) = v_{i_j}$, and hence it follows from the race-freedom that $\ell_{i_j} \notin R_{i_k}$ (note that $j \neq k$, hence $i_j \neq i_k$), a contradiction. $\square$

LEMMA 6.20. *Suppose $(P, \varepsilon \mid i \in \mu), \sigma$ is race-free, and let D be a partial derivation reachable from $P, \mu, \sigma \Downarrow X$. Then, D is locally interleavable.*

PROOF. By Lemma 6.19, it suffices to show that, for every instance $A, \mu, \sigma \Downarrow \sigma'$ of T-SASSIGN in D, (A, μ, σ) is race-free. We prove this by induction on $\longrightarrow^*$. The base case is obvious, because there is no such rule instance. For the induction step, consider D and D' such that $(P, \mu, \sigma \Downarrow X) \longrightarrow^* D \longrightarrow D'$, and suppose that D is locally interleavable. If D' contains an instance of E-SASSIGN that does not appear in D, then it has to be the case that $D \longrightarrow D'$ is obtained by S-ATOM; note that the substitution performed by S-WHILEFALSE does not produce a new instance of E-SASSIGN, because it does not replace an occurrence of a state variable on a leaf except for the leaf to which E-WHILEFALSE is applied. Therefore, D takes the form $E[A, \mu', \sigma' \Downarrow X']$ where A is an assignment to a shared variable, and $D' = D\{\sigma''/X'\}$.

By the induction hypothesis, D is locally interleavable, and hence by Proposition 6.9, $|D|$ is reachable from $(P, \varepsilon \mid i \in \mu), \sigma$. To show that (A, μ', σ') is race-free, by Remark 6.18 it suffices to show that $(A, \varepsilon \mid i \in \mu'), \sigma'$ is not racy. Here, we use the following facts: from the assumption of race-freedom, $|D|$ is not racy, and $|D|$ has the form $(Q_i, s_i)_i, \sigma'$ where $Q_i = (A;\ Q'_i)$ if $i \in \mu'$. First, if $(Q_i, s_i)_i, \sigma'$ is not racy, then neither is a configuration $(Q_i, s_i \mid i \in \mu')_i, \sigma'$ with fewer active threads (that is, i with $Q_i \neq \checkmark$). This is because the definition of a race can be written as an existential statement on active threads (there exists two active threads such that...). Since $Q_i = A;\ Q'_i$, this means that $(A;\ Q'_i, s_i \mid i \in \mu')_i, \sigma'$ is not racy. Second, this implies that $(A, \varepsilon \mid i \in \mu')_i, \sigma'$ is not racy, because the definition of a race only mentions the first statement of each program, and hence Q_i and s_i are irrelevant. $\square$

6.4. Barrier Divergence

In this section, we prove the following Lemma.

LEMMA 6.21. *If $P_0, \mu_0, \sigma_0 \Downarrow X_0 \longrightarrow^* D = E[\texttt{sync}, \mu, \sigma \Downarrow X]$ and $\mu \neq \emptyset, \mathbb{T}$, then there exists no σ' such that $|D| \rightarrow_I^* (\checkmark, \varepsilon)_i, \sigma'$.*

The basic strategy of the proof is to define an ordering $<_{\mathsf{L}}$ on thread configurations so that

— if $(P_i, s_i)_i, \sigma \to_I (P_i', s_i')_i, \sigma'$, then $(P_i, s_i) \leq_{\mathsf{L}} (P_i', s_i')$ for each i,
— under the same assumptions as Lemma 6.21, if $j \in \mu$, $k \notin \mu$, and $|D| = (P_i, s_i)_i, \sigma$, then $(P_j, s_j) <_{\mathsf{L}} (P_k, s_k)$, and
— if $(P_i, s_i)_i, \sigma$ is an I-configuration to which I-SYNC applies, then $(P_j, s_j) \not<_{\mathsf{L}} (P_k, s_k)$ for every pair of threads j, k.

We call this order L-order (L stands for lockstep). The intuition behind this order is that to simulate the lockstep execution in the interleaving semantics, the least thread with respect to this order has to be executed first. The first and third clauses mean that a smaller configuration becomes larger as its execution proceeds and, by the time a barrier synchronization succeeds, all configurations are incomparable. The second clause means that, if threads are barrier-divergent, then a thread that has reached `sync` is strictly smaller. Therefore, being strictly smaller is an invariant condition (by the first clause), and the threads are never ready for synchronization. The lemma follows from this observation, because $|D|$ cannot terminate without using I-SYNC.

The precise definition of L-order (Definition 6.33) is complicated, but basically it compares program counters (which we introduce below) of both sides, so that the state increases as the execution of the program proceeds (this implies the first condition above). A difficulty arises from the existence of loops: when the execution returns to the beginning of a loop, the program counter decreases. To handle such a case correctly, L-order takes the state of stacks into account, and this complicates the definition of L-order.

The proof of Lemma 6.21 is as follows. First, we introduce counters and modify the notions we have introduced thus far, such as derivation search and interleaving semantics; then, we show that Lemma 6.21 follows from its variant, Lemma 6.31 (Section 6.4.1). The latter states that Lemma 6.21 holds for the annotated versions of partial derivations and I-configurations. Then, it remains to prove Lemma 6.31. To prove this, we define L-order (Section 6.4.2) and show that it is transitive on reachable thread configurations (Lemma 6.48 in Section 6.4.3). To prove the transitivity, we have to introduce several auxiliary notions and lemmas. This machinery is used only in the proof of the transitivity, and is not used in the remaining part of the proof, except in Lemmas 6.41 and 6.46. After the transitivity has been proved, in Section 6.4.4, the first and second properties of L-order listed above are proved in Lemma 6.51 and Lemma 6.53, respectively. The third property is almost immediate from the definition of L-order. Finally, at the end of this section, we put these results together to prove Lemma 6.31.

6.4.1. Interleaving Semantics with Program Counters. First, we introduce a program counter into our program syntax:

$$\begin{aligned} P ::= {} & c : x_n\,[\bar{e}]\;\texttt{:=}\;e \mid c : \texttt{skip} \mid c : \texttt{sync} \mid P;\ P' \mid c : \texttt{if}^l\ e\ \texttt{then}\ P\ \texttt{else}\ P' \\ & \mid c : \texttt{while}^l\ e\ \texttt{do}\ P \mid c : \texttt{endif}^l \end{aligned}$$

Counters c range over natural numbers.

Since c represents a position of each statement in the initial program, it is natural to assume that c is annotated from left to right. In addition, we assume a certain condition on counters, which we specify below.

Definition 6.22. Let P be a program. Then we denote the sequence of labels appearing in P by $labs(P)$. Similarly the counters appearing in P is denoted by $cts(P)$. More formally,

$$\begin{aligned} labs(c : x\,[\bar{e}]\;\texttt{:=}\;e) &= \varepsilon \\ labs(c : \texttt{skip}) &= \varepsilon \\ labs(c : \texttt{sync}) &= \varepsilon \end{aligned}$$

$$\begin{aligned}
labs(P;\ P') &= labs(P) \cdot labs(P') \\
labs(c : \mathtt{if}^l\ e\ \mathtt{then}\ P\ \mathtt{else}\ P') &= l \cdot labs(P) \cdot labs(P') \\
labs(c : \mathtt{while}^l\ e\ \mathtt{do}\ P) &= l \cdot labs(P) \\
labs(c : \mathtt{endif}^l) &= \varepsilon \\
cts(c : x\,[\bar{e}]\ \mathtt{:=}\ e) &= c \\
cts(c : \mathtt{skip}) &= c \\
cts(c : \mathtt{sync}) &= c \\
cts(P;\ P') &= cts(P) \cdot cts(P') \\
cts(c : \mathtt{if}^l\ e\ \mathtt{then}\ P\ \mathtt{else}\ P') &= c \cdot cts(P) \cdot cts(P') \\
cts(c : \mathtt{while}^l\ e\ \mathtt{do}\ P) &= c \cdot cts(P) \\
cts(c : \mathtt{endif}^l) &= c,
\end{aligned}$$

where $\cdot$ represents the concatenation of sequences.

Although $labs(P)$ and $cts(P)$ are sequences, we sometimes regard them as sets. For example, $l \in labs(P)$ means that l appears in $labs(P)$. Also, $ct(P)$ denotes the first element of $cts(P)$. Since $cts(P)$ is always non-empty, $ct(P)$ is well-defined. For convenience, we also define $cts(\checkmark) = labs(\checkmark) = \varepsilon$ and $ct(\checkmark) = \infty$.

Definition 6.23. Let P be a program. A subprogram P' of P *with label* l is a subprogram of P of the form either $c : \mathtt{if}^l\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$ or $c : \mathtt{while}^l\ e\ \mathtt{do}\ P_1$.

Definition 6.24. Let P be a program such that $labs(P)$ contains no multiple occurrences of the same label. We define bgn_P for such a program as a map from $labs(P)$ (regarded as a set) to $\mathbb{N}$ such that for each label $l \in labs(P)$ and *the* subprogram P' of P with label l, $\mathsf{bgn}_P(l) = ct(P')$ (that is, $\mathsf{bgn}_p(l)$ is the counter annotated to the statement with label l).

Definition 6.25. A program P is said to be *well-annotated* if

— $cts(P)$ and $labs(P)$ are strictly increasing, and
— there exists a function $\mathsf{end} : labs(P) \to \mathbb{N}$ such that, for all subprograms P' of P, if P' has a label l, then
  — $\mathsf{end}(l') < \mathsf{end}(l)$ for all $l' \in labs(P')$,
  — $c < \mathsf{end}(l)$ for all $c \in cts(P')$, and
  — $\mathsf{end}(l) < c$ for all $c \in cts(P) \setminus cts(P')$ such that $\mathsf{bgn}(l) < c$.

The function end is used to define the semantics (see Figure 7). As an example, consider the programs in Figure 6. Although both programs satisfy the first condition of Definition 6.25, the program on the left is well-annotated while the one on the right is not. The function end for the program on the left is given by $\mathsf{end}(l) = 8$ and $\mathsf{end}(m) = 5$, whereas for the one on the right, we have to choose $\mathsf{end}(m)$ so that $4 < \mathsf{end}(m) < 5$, which is impossible.

Hereafter, when we consider a well-annotated program, we implicitly assume that a function end (which is sometimes denoted by end_P, making P explicit) is specified.

NOTATION 6.26. *We denote by* $\lfloor \cdot \rfloor$ *the operation that removes all counters, which applies to both programs and I-configurations. Below we often follow the convention that if we have to treat both annotated and unannotated programs, we use* P *for unannotated one and* $\tilde{P}$ *for annotated one (and similarly for partial derivations and I-configurations).*

```
1:s = n / 2;
2:while^l (s > 0) {
  3:if^m (tid < s)
    4:a[tid] = a[tid] + a[tid + s];
  6:s = s / 2;
  7:sync;
}
```

```
1:s = n / 2;
2:while^l (s > 0) {
  3:if^m (tid < s)
    4:a[tid] = a[tid] + a[tid + s];
  5:s = s / 2;
  6:sync;
}
```

Fig. 6. Examples of well- and ill-annotated programs.

The following lemma shows that any program can be made well-annotated.

LEMMA 6.27. *For any program P in the sense of Section 5 (i.e., without counters), there exists a well-annotated program $\tilde{P}$ such that P_0 and $\lfloor \tilde{P} \rfloor$ are the same except for labels.*

PROOF. We may assume that $labsP$ is increasing (if not, rename the labels). Then, define counter annotation and end by the following procedure:

— $\mathrm{annot}(P_1;\, P_2, c) = (P_1';\, P_2', c_2, m_1 \uplus m_2)$, where $(P_1', c_1, m_1) = \mathrm{annot}(P_1, c)$ and $(P_2', c_2, m_2) = \mathrm{annot}(P_2, c_1)$;
— $\mathrm{annot}(\texttt{if}^l\, e \,\texttt{then}\, P_1 \,\texttt{else}\, P_2, c) = (c : \texttt{if}^l\, e \,\texttt{then}\, P_1' \,\texttt{else}\, P_2', c_2 + 1, m_1 \uplus m_2 \uplus \{(l, c_2)\})$ where $(P_1', c_1, m_1) = \mathrm{annot}(P_1, c + 1)$ and $(P_2', c_2, m_2) = \mathrm{annot}(P_2, c_1)$;
— $\mathrm{annot}(\texttt{while}^l\, e \,\texttt{do}\, P, c) = (c : \texttt{while}^l\, e \,\texttt{do}\, P', c_1 + 1, m_1 \uplus \{(l, c_1)\})$ where $(P', c_1, m_1) = \mathrm{annot}(P, c + 1)$;
— if none of the above clauses applies, $\mathrm{annot}(P, c) = (c : P, c + 1, \emptyset)$.

annot receives a program P and an auxiliary natural number c that represents the least value of the counter that may be used to annotate P, and returns an annotated program $\tilde{P}$, a natural number c that is greater than any counter appearing in $\tilde{P}$, and a partial function m that constitutes end for $\tilde{P}$. Let $(\tilde{P}, c', m) = \mathrm{annot}(P, c)$. Then it holds that $\tilde{P}$ is well-annotated with end $= m$. $\square$

Having introduced the counters and well-annotated programs, we now adapt some of the arguments in the current and the previous sections to the new setting. Figures 7 and 8 show how to modify the interleaving semantics introduced in Section 5.1 to annotated programs (differences are highlighted). Most part of the modification is straightforward, except that every $\texttt{endif}^l$ appearing on the right-hand side is annotated by end(l), and similarly for a while-statement in T-WHILETRUE. Because we expect counters to increase from left to right, we have to annotate them with some number larger than the counters of P_i or P, but smaller than those of $\bar{P}$. The second condition of Definition 6.25 is exactly what is needed here. I-SYNC allows the counters to vary among threads. Although it seems more natural to require that they are uniform, this relaxed version makes the proofs below simpler.

In the interleaving semantics defined here, we assume a fixed, well-annotated initial program P_0. The functions bgn and end appearing in these rules are considered as bgn_{P_0} and end_{P_0}, respectively. Below, unless otherwise specified we implicitly assume that a well-annotated initial program is fixed, and we omit the subscripts of bgn and end for brevity.

Since the new rules only add counters to programs and change nothing else, $\to_{Ic}$ and $\to_I$ are almost equivalent.

LEMMA 6.28. *Let $\tilde{C}$ be an annotated I-configuration.*

(1) *If $\tilde{C}'$ is another annotated I-configuration and $\tilde{C} \to_{Ic} \tilde{C}'$, then $\lfloor \tilde{C} \rfloor \to_I \lfloor \tilde{C}' \rfloor$.*

$$\boxed{c}:\texttt{skip};\ \bar{P}, s, \sigma \xrightarrow{i}_c \bar{P}, s, \sigma \qquad \text{(T-SKIP)}$$

$$\frac{x \text{ is local} \qquad \sigma' = \sigma\left[x, i, \sigma\,[\![\bar{e}]\!]\,(i) \mapsto \sigma\,[\![e]\!]\,(i)\right]}{\boxed{c}: x\,[\bar{e}]\ \texttt{:=}\ e;\ \bar{P}, s, \sigma \xrightarrow{i}_c \bar{P}, s, \sigma'} \qquad \text{(T-LASSIGN)}$$

$$\frac{x \text{ is shared} \qquad \sigma' = \sigma\left[x, \sigma\,[\![\bar{e}]\!]\,(i) \mapsto \sigma\,[\![e]\!]\,(i)\right]}{\boxed{c}: x\,[\bar{e}]\ \texttt{:=}\ e;\ \bar{P}, s, \sigma \xrightarrow{i}_c \bar{P}, s, \sigma'} \qquad \text{(T-SASSIGN)}$$

$$\frac{\sigma\,[\![e]\!]\,(i) \neq 0}{\boxed{c}: \texttt{if}^l\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2;\ \bar{P}, s, \sigma \xrightarrow{i}_c P_1;\ \boxed{\mathsf{end}(l)}: \texttt{endif}^l;\ \bar{P}, s \cdot (l, 1), \sigma} \qquad \text{(T-IFTRUE)}$$

$$\frac{\sigma\,[\![e]\!]\,(i) = 0}{\boxed{c}: \texttt{if}^l\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2;\ \bar{P}, s, \sigma \xrightarrow{i}_c P_2;\ \boxed{\mathsf{end}(l)}: \texttt{endif}^l;\ \bar{P}, s \cdot (l, 2), \sigma} \qquad \text{(T-IFFALSE)}$$

$$\boxed{c}: \texttt{endif}^l;\ \bar{P}, s \cdot (l, k), \sigma \xrightarrow{i}_c \bar{P}, s, \sigma \qquad \text{(T-ENDIF)}$$

$$\frac{\sigma\,[\![e]\!]\,(i) \neq 0}{\boxed{c}: \texttt{while}^l\ e\ \texttt{do}\ P;\ \bar{P}, s, \sigma \xrightarrow{i}_c P;\ \boxed{\mathsf{end}(l)}: \texttt{while}^l\ e\ \texttt{do}\ P;\ \bar{P}, s + l, \sigma} \qquad \text{(T-WHILETRUE)}$$

$$\frac{\sigma\,[\![e]\!]\,(i) = 0}{\boxed{c}: \texttt{while}^l\ e\ \texttt{do}\ P;\ \bar{P}, s, \sigma \xrightarrow{i}_c \bar{P}, s \setminus l, \sigma} \qquad \text{(T-WHILEFALSE)}$$

Fig. 7. Thread execution of GPU kernels with counters.

$$\frac{P_i, s_i, \sigma \xrightarrow{i}_c P', s', \sigma'}{(P_i, s_i)_i, \sigma \to_{Ic} (P_i, s_i)_i\,[i \mapsto (P', s')]\,, \sigma'} \qquad \text{(I-THREAD)}$$

$$\frac{\forall i, j. s_i = s_j}{(\boxed{c_i}: \texttt{sync};\ \bar{P}_i, s_i)_i, \sigma \to_{Ic} (\bar{P}_i, s_i)_i, \sigma} \qquad \text{(I-SYNC)}$$

Fig. 8. Interleaving semantics of GPU kernels with counters.

(2) *If $\mathcal{C}'$ is an unannotated I-configuration satisfying $\lfloor \tilde{\mathcal{C}} \rfloor \to_I \mathcal{C}'$, then there exists a unique annotated I-configuration $\tilde{\mathcal{C}}'$ such that $\tilde{\mathcal{C}} \to_{Ic} \tilde{\mathcal{C}}'$ and $\lfloor \tilde{\mathcal{C}}' \rfloor = \mathcal{C}'$.*

We next annotate partial derivations and the derivation search procedure defined in Section 6.1. The new definition of partial derivations is mostly the same as Definition 6.2. The only nontrivial case is (4) of Definition 6.2, where the counter of the `while`-statement of D_2 has to be $\mathsf{end}(l)$, while the counter of the statement at the bottom is arbitrary. The other clauses are exactly the same as before, except that P, P_1, and P_2 denote annotated programs. This means that, for example, in clause (4) the three occurrences of P in the conclusions of D_1, D_2, and the whole partial derivation have to be identical including counters. The derivation search procedure defined in Figure 5 is adapted in a straightforward manner. We occasionally use $\longrightarrow_c$ to denote the resulting relation for clarity.

LEMMA 6.29. *Let $\tilde{D}$ be an annotated partial derivation.*

(1) *If $\tilde{D}'$ is another annotated partial derivation and $\tilde{D} \longrightarrow_c \tilde{D}'$, then $\lfloor \tilde{D} \rfloor \longrightarrow \lfloor \tilde{D}' \rfloor$.*
(2) *If D' is an unannotated partial derivation satisfying $\lfloor \tilde{D} \rfloor \longrightarrow D'$, then there exists a unique annotated partial derivation $\tilde{D}'$ such that $\tilde{D} \longrightarrow_c \tilde{D}'$ and $\lfloor \tilde{D}' \rfloor = D'$.*

We can prove analogues of the results in Sections 6.2 and 6.3 in the same way as before. We omit the details because they are mostly straightforward. Below we denote the translation from annotated partial derivations into annotated I-configurations by $|\cdot|_c$.

LEMMA 6.30. *For an annotated partial derivation $\tilde{D}$, it holds that $\lfloor|\tilde{D}|_c\rfloor = |\lfloor\tilde{D}\rfloor|$.*

Now we can reduce our goal, Lemma 6.21, to the following lemma.

LEMMA 6.31. *If $\tilde{P}_0, \mu_0, \sigma_0 \Downarrow X_0 \longrightarrow_c^* \tilde{D} = \tilde{E}[c : \texttt{sync}, \mu, \sigma \Downarrow X]$ and $\mu \neq \emptyset, \mathbb{T}$, then there exists no σ' such that $|\tilde{D}|_c \rightarrow_{Ic}^* (\checkmark, \varepsilon)_i, \sigma'$.*

LEMMA 6.32. *Lemma 6.31 implies Lemma 6.21.*

PROOF. We may assume $labs P_0$ is increasing, without affecting the premises of Lemma 6.21, because the interleaving semantics do not rely on the ordering over labels. Then, by Lemma 6.27 we have a well-annotated program $\tilde{P}_0$ such that $\lfloor\tilde{P}_0\rfloor = P_0$. By Lemma 6.29, the assumption of Lemma 6.21 implies the existence of $\tilde{D}$ satisfying the assumption of Lemma 6.31 and the equation $\lfloor\tilde{D}\rfloor = D$. Therefore, there is no σ' such that $|\tilde{D}|_c \rightarrow_{Ic}^* (\checkmark, \varepsilon)_i, \sigma'$. Let us assume there exists σ' such that $|D| \rightarrow_I^* (\checkmark, \varepsilon)_i, \sigma'$. Then, by using $D = \lfloor\tilde{D}\rfloor$ and Lemma 6.30, we obtain $\lfloor|\tilde{D}|_c\rfloor \rightarrow_I^* (\checkmark, \varepsilon)_i, \sigma'$. Therefore, by Lemma 6.28, there exists an annotated I-configuration $\tilde{\mathcal{C}}$ such that $|\tilde{D}|_c \rightarrow_{Ic}^* \tilde{\mathcal{C}}$ and $\lfloor\tilde{\mathcal{C}}\rfloor = (\checkmark, \varepsilon)_i, \sigma'$. However, by definition of $\lfloor\tilde{\mathcal{C}}\rfloor$ this equality implies $\tilde{\mathcal{C}} = (\checkmark, \varepsilon)_i, \sigma'$, a contradiction. □

In the rest of this section, we prove Lemma 6.31. In what follows, we mostly consider annotated programs and configurations. Therefore, for brevity, we usually omit the word "annotated" and use P and $\mathcal{C}$ rather than $\tilde{P}$ and $\tilde{\mathcal{C}}$. Counters are also omitted, if they are not important.

6.4.2. L-order

Definition 6.33 (*L-order*).

— The partial order $\preceq$ on stacks is the prefix relation, that is, $s \preceq s'$ if and only if there exists s'' (which is possibly empty) such that $s' = s \cdot s''$.
— The relation $\parallel$ on stacks is defined as: $s \parallel s'$ if and only if neither $s \preceq s'$ nor $s' \preceq s$. We use $\nparallel$ for the negation of $\parallel$.
— The partial order $\leq_{\mathsf{s}}$ on stacks is the lexicographical order, where elements are also ordered lexicographically. More precisely, $s \leq_{\mathsf{s}} s'$ if and only if either
  — $s \preceq s'$, or
  — $s = s_0 \cdot (l, k) \cdot s_1, s' = s_0 \cdot (l', k') \cdot s_2$ and $(l, k) < (l', k')$,
  where $(l, k) < (l', k')$ if and only if either $l < l'$, or $l = l'$ and $k < k'$.
— The relation $<_{\mathsf{L}}$ on thread configurations is such that $(P, s) <_{\mathsf{L}} (P', s')$ if and only if either
  — $s \parallel s'$ and $s <_{\mathsf{s}} s'$, or
  — $s \nparallel s'$ and $ct(P) < ct(P')$.
  In particular, $(P, s) <_{\mathsf{L}} (\checkmark, \varepsilon)$ for all $P \neq \checkmark$ and s since $ct(\checkmark) = \infty$. We call this relation *L-order*.

As usual, we write $s \prec s'$ when $s \preceq s'$ and $s \neq s'$, and similarly for $<_{\mathsf{s}}$. We also write $\leq_{\mathsf{L}}$ for the reflexive closure of $<_{\mathsf{L}}$.

Intuitively, two thread configurations T and T' satisfy $T <_{\mathsf{L}} T'$ when T is at an earlier stage of execution than T'. To see that this applies when the execution branches on `if`-statement, remember that our semantics executes the `then` part first. Thus, the `then` part is considered an earlier stage of execution than the `else` part. Taking this into account, T-IFTRUE and T-IFFALSE push $(l, 1)$ and $(l, 2)$ to the stacks, respectively, so that $s \cdot (l, 1) <_{\mathsf{L}} s \cdot (l, 2)$.

Therefore, we basically compare counters of both configurations, as in the second clause of the definition of L-order. However, it is not sufficient to compare counters only, if a loop is involved. For example, consider a program $\mathtt{while}^l\, e\, \mathtt{do}\, (\mathtt{1} : P_1;\, \mathtt{2} : P_2)$ and two thread configurations

$$T = (\mathtt{2} : P_2;\, \mathtt{3} : \mathtt{while}^l\, e\, \mathtt{do}\, (\mathtt{1} : P_1;\, \mathtt{2} : P_2), (l, 1)),$$
$$T' = (\mathtt{1} : P_1;\, \mathtt{2} : P_2;\, \mathtt{3} : \mathtt{while}^l\, e\, \mathtt{do}\, (\mathtt{1} : P_1;\, \mathtt{2} : P_2), (l, 2)).$$

T is executing P_2 in the first iteration, and T' is executing P_1 in the second iteration. We expect that $T <_{\mathsf{L}} T'$, and this is indeed the case for the actual definition of L-order (apply the first clause), but if we compare only counters, we would have $T' <_{\mathsf{L}} T$. To treat this situation correctly, we have to take the stack into account.

6.4.3. Transitivity of L-order. L-order defined above is not transitive, when considered as a relation over the set of *all* thread configurations. For example, consider the three thread configurations $(\mathtt{1} : P_1, \varepsilon)$, $(\mathtt{2} : P_2, (l, 1))$ and $(\mathtt{1} : P_1, (l, 2))$. Then we have: $(\mathtt{1} : P_1, \varepsilon) <_{\mathsf{L}} (\mathtt{2} : P_2, (l, 1))$ because $\varepsilon \nparallel (l, 1)$ and $1 < 2$ (the second clause of the definition of L-order); $(\mathtt{2} : P_2, (l, 1)) <_{\mathsf{L}} (\mathtt{1} : P_1, (l, 2))$ because $(l, 1) \parallel (l, 2)$ and $(l, 1) < (l, 2)$ (the first clause); but it is not the case that $(\mathtt{1} : P_1, \varepsilon) <_{\mathsf{L}} (\mathtt{1} : P_1, (l, 2))$ because $\varepsilon \nparallel (l, 2)$ but $1 \not< 1$.

However, we do not need the transitivity on all thread configurations. Since we are interested in configurations that are reachable from the initial configuration, it is sufficient to have the transitivity on such configurations. We show that this is indeed the case (Lemma 6.48). The precise definition of reachabe configuration is as follows:

Definition 6.34 (*Reachability*). We say an I-configuration $\mathcal{C}$ is *reachable* from an initial configuration $\mathcal{C}_0$ if $\mathcal{C}_0 \to_{Ic}^* \mathcal{C}$. We also say (P, s) is reachable if it is a thread configuration of some reachable I-configuration, that is, $(P, s) = (P_i, s_i)$ for some i and a reachable configuration $(P_i, s_i)_i, \sigma$.

To prove the transitivity, we analyze the relationship between P and s when (P, s) is a reachable thread configuration. In particular, we prove Lemma 6.45, which says that $\mathrm{dom}(s)$ is actually determined by P. We first define the function p used in Lemma 6.45.

Definition 6.35 (*Context*). We define a (one-hole) *context* as follows:

$$\begin{aligned} C ::= [\,] \mid C;\, P \mid P;\, C \mid c : \mathtt{if}^l\, e\, \mathtt{then}\, C\, \mathtt{else}\, P \mid c : \mathtt{if}^l\, e\, \mathtt{then}\, P\, \mathtt{else}\, C \\ \mid c : \mathtt{while}^l\, e\, \mathtt{do}\, C. \end{aligned}$$

Definition 6.36 (*Path*). For each context C, we define the *path* to the hole in C as follows:

$$p([\,]) = \varepsilon; \quad p(C;\, P) = p(P;\, C) = p(C);$$
$$p(c : \mathtt{if}^l\, e\, \mathtt{then}\, P\, \mathtt{else}\, C) = p(c : \mathtt{if}^l\, e\, \mathtt{then}\, C\, \mathtt{else}\, P) = l \cdot p(C);$$
$$p(c : \mathtt{while}^l\, e\, \mathtt{do}\, C) = l \cdot p(C).$$

Also, for a well-annotated program P, we define a map p_P from $cts(P) \cup \mathsf{end}_P(labs(P))$ to L^* as follows:

— given $c \in cts(P)$, there exists a unique context C and program P' such that $P = C[c : P']$. Define $p_P(c) = p(C)$;
— given $l \in labs(P)$, define $p_P(\mathsf{end}_P(l)) = p_P(\mathsf{bgn}_P(l)) \cdot l$ (note that $\mathsf{bgn}_P(l) \in labs(P)$).

Below we denote the set $cts(P) \cup \mathsf{end}_P(labs(P))$ by $\mathrm{dom}(p_P)$.

Remember that we work under some initial program P_0. Below we omit the subscript P_0 of p_{P_0}, and similarly for bgn and end.

LEMMA 6.37. bgn *and* end *are injective, and* bgn *is strictly monotone.*

PROOF. By definition of bgn and end. □

LEMMA 6.38. *Take any pair of labels* $l, l' \in labs(P_0)$*, and consider two intervals* $[\mathsf{bgn}(l), \mathsf{end}(l)]$ *and* $[\mathsf{bgn}(l'), \mathsf{end}(l')]$*. Then either they are disjoint, or one of them is contained in the other. Equivalently, if* $\mathsf{bgn}(l) < \mathsf{bgn}(l')$ *then either* $\mathsf{end}(l) < \mathsf{bgn}(l')$ *or* $\mathsf{end}(l') < \mathsf{end}(l)$.

PROOF. Let $l, l' \in labs(P_0)$ and suppose $\mathsf{bgn}(l) < \mathsf{bgn}(l')$. Let P' be the program with label l. If $l' \in labs(P')$ then by Definition 6.25 we have $\mathsf{end}(l') < \mathsf{end}(l)$. Otherwise, $l' \notin labs(P')$ implies $\mathsf{bgn}(l') \notin cts(P')$, therefore by the last clause of Definition 6.25 (put $c = \mathsf{bgn}(l')$) we obtain $\mathsf{end}(l) < \mathsf{bgn}(l')$, as required. □

LEMMA 6.39. *Let* $l \in labs(P_0)$ *and* $c \in \mathrm{dom}(p)$*. Then* $l \in p(c)$ *if and only if* $\mathsf{bgn}(l) < c \leq \mathsf{end}(l)$.

PROOF. We first consider the case $c \in cts(P_0)$. Let C be the context such that $P_0 = C[c : P]$. Then $p(c) = p(C)$. Let P_l be the subprogram of P_0 with label l, and take C_1 so that $P_0 = C'[P_l]$. Then by Definition 6.25 we have $c \in cts(P_l)$ if and only if $\mathsf{bgn}(l) \leq c \leq \mathsf{end}(l)$. It is also easy to see that $c \in cts(P_l)$ if and only if there exists C_2 such that $P_l = C_2[c : P]$. Therefore it suffices to show that

$$l \in p(C) \iff P_l = C_2[c : P] \text{ for some } C_2 \text{, and } c \neq \mathsf{bgn}(l) \qquad (2)$$

where $P_0 = C[c : P] = C'[P_l]$ and P_l has label l. First, note that

$$l \in p(C) \iff C = C_1[C_2] \text{ for some } C_1, C_2 \text{ where } C_2 \neq [] \text{ has label } l$$

(here by abuse of terminology we apply the predicate "has label l" to a context when the context is non-empty) since in general $p(C_1[C_2]) = p(C_1) \cdot p(C_2)$ and $p(C)$ starts with l if and only if C has label l. If $C = C_1[C_2]$ and C_2 has label l, then $P_l = C_2[c : P]$. In this case we also have $c \neq \mathsf{bgn}(l)$ since otherwise $c = \mathsf{bgn}(l) = ct(P_l)$ and therefore C_2 has to be empty. This shows the left-to-right direction of (2). For the converse, suppose $P_l = C_2[c : P]$ and $c \neq \mathsf{bgn}(l)$. Then, because $P_0 = C[c : P] = C'[P_l]$, we have $P_0 = C'[C_2[c : P]]$. Therefore $C = C'[C_2]$. Also, we can check that $C_2 \neq []$ in the same way as above, hence $l \in p(C_2) \subseteq p(C)$. This proves the lemma for $c \in cts(P_0)$.

If $c = \mathsf{end}(l')$, then $p(c) = p(\mathsf{bgn}(l')) \cdot l'$, so

$$\begin{aligned} l \in p(c) &\iff l = l' \text{ or } l \in p(\mathsf{bgn}(l')) \\ &\iff l = l' \text{ or } \mathsf{bgn}(l) < \mathsf{bgn}(l') \leq \mathsf{end}(l) \\ &\iff \mathsf{bgn}(l) < \mathsf{end}(l') \leq \mathsf{end}(l). \end{aligned}$$

The second equivalence follows from this lemma for $c = \mathsf{bgn}(l')$ which is already proved above. The last equivalence follows from Lemmas 6.37 and 6.38. □

LEMMA 6.40. $p(c)$ *is strictly increasing for all* $c \in cts(P_0)$*, where* P_0 *is well-annotated.*

PROOF. It suffices to prove that $p(c)$ is a subsequence of $labs(P_0)$. It is easy prove that if $P_0 = C[c : P]$ then $p(C)$ is a subsequence of $labs(P_0)$, by induction on P_0. This immediately implies the lemma for c not of the form $\mathsf{end}(l)$.

Consider the other case: $c = \mathsf{end}(l)$. Suppose that l is a label of an `if`-statement, and define C and C' so that $P_0 = C[\mathtt{if}^l\, e\, \mathtt{then}\, P_1\, \mathtt{else}\, P_2]$ and $C' = C[\mathtt{if}^l\, e\, \mathtt{then}\, []\, \mathtt{else}\, P_2]$.

Then $p(\mathsf{end}(l)) = p(C')$ since both sides equal $p(C) \cdot l$, and therefore $p(\mathsf{end}(l))$ is a subsequence of $labs(P_0)$. The case l is a label of a `while`-statement is similar. $\square$

LEMMA 6.41. *Let $c_1, c_2, c_3 \in \mathrm{dom}(p)$. If $c_1 \leq c_2 \leq c_3$ then $p(c_1) \cap p(c_3) \subseteq p(c_2)$.*

PROOF. If $l \in p(c_1) \cap p(c_3)$, by Lemma 6.39 we have $\mathsf{bgn}(l) < c_1, c_3 \leq \mathsf{end}(l)$. Since $c_1 \leq c_2 \leq c_3$ it has to be the case that $\mathsf{bgn}(l) < c_2 \leq \mathsf{end}(l)$. Again applying Lemma 6.39 we obtain $l \in p(c_2)$ as required. $\square$

In the proof of Lemma 6.45, we use an auxiliary function rm. This function receives a program and returns the list of labels which will be removed from the stack when the program is executed.

Definition 6.42. Define a mapping rm from programs to L^* by

— $\mathrm{rm}(P_1;\, P_2) = \mathrm{rm}(P_2) \cdot \mathrm{rm}(P_1)$,
— $\mathrm{rm}(\mathsf{end}(l) : \mathtt{endif}^l) = \mathrm{rm}(\mathsf{end}(l) : \mathtt{while}^l\, e\, \mathtt{do}\, P) = l$, and
— $\mathrm{rm}(P) = \varepsilon$ in other cases.

LEMMA 6.43. *If (P, s) is reachable for some s, and P' is a subprogram of P that does not occur in the top level of P (precisely, there exists no (sequences of) programs $\bar{P}$ and $\bar{P}'$ such that $P = \bar{P};\, P';\, \bar{P}'$), then P' is a subprogram of the initial program.*

PROOF. By induction on $\rightarrow^*_{Ic}$. $\square$

LEMMA 6.44. *If P is a subprogram of the initial program, then $\mathrm{rm}(P) = \varepsilon$.*

PROOF. By straightforward induction on P, using the fact that neither $\mathsf{end}(l) : \mathtt{endif}^l$ nor $\mathsf{end}(l) : \mathtt{while}^l\, e\, \mathtt{do}\, P'$ appears in the initial program. Note that $\mathsf{end}(l)$ is defined as a label that does not appear in the initial program. $\square$

LEMMA 6.45. *Let (P, s) be a reachable thread configuration. Then $p(ct(P)) = \mathrm{dom}(s)$.*

PROOF. By induction on $\rightarrow^*_{Ic}$ we prove that if (P, s) is reachable, then for any $\bar{Q}_1$ and $\bar{Q}_2$ such that $P = \bar{Q}_1;\, \bar{Q}_2$ (here we allow $\bar{Q}_1$ to be empty) it holds that $\mathrm{dom}(s) = p(ct(\bar{Q}_2)) \cdot \mathrm{rm}(\bar{Q}_1)$. The base case is obvious as both sides are empty (the right-hand side is empty by the above two lemmas). Below we say that such $\bar{Q}_2$ is a *tail* of P.

Consider the case of T-IFTRUE. We have $(c : \mathtt{if}^l\, e\, \mathtt{then}\, P_1\, \mathtt{else}\, P_2;\, \bar{P}, s) \xrightarrow{i}_c (P_1;\, \mathsf{end}(l) : \mathtt{endif}^l;\, \bar{P}, s \cdot (l, 1))$. If $P_1;\, \mathsf{end}(l) : \mathtt{endif}^l;\, \bar{P} = \bar{Q}_1;\, \bar{Q}_2$, then there are three cases: (1) $\bar{Q}_2$ is a tail of $\bar{P}$, (2) $\bar{Q}_2 = \mathsf{end}(l) : \mathtt{endif}^l;\, \bar{P}$, or (3) $\bar{Q}_2$ starts with a tail of P_1. Case (1) is immediate from the induction hypothesis, using the fact that $\mathrm{rm}(P_1) = \varepsilon$ (because P_1 is a subprogram of the initial program). In case (2) we have $\mathrm{rm}(\bar{Q}_1) = \mathrm{rm}(P_1) = \varepsilon$. Moreover $p(ct(\bar{Q}_2)) = p(\mathsf{end}(l)) = p(\mathsf{bgn}(l)) \cdot l$. By the induction hypothesis and the fact that the counter c is actually $\mathsf{bgn}(l)$ (which is easy to prove that in general the counter of this statement is always $\mathsf{bgn}(l)$), this sequence equals $\mathrm{dom}(s) \cdot l = \mathrm{dom}(s \cdot (l, 1))$, as required. The case (3) can be treated similarly, if we notice that $p(ct(\bar{Q}_2)) = p(\mathsf{bgn}(l)) \cdot l = p(\mathsf{end}(l))$. The second equality is by definition, and the first is checked as follows. Since P_1 has the form $\bar{Q}_1;\, \bar{Q}$, and therefore if C is the context such that $P_0 = C[\mathtt{if}^l\, e\, \mathtt{then}\, P_1\, \mathtt{else}\, P_2]$, then $p(\mathsf{bgn}(l)) = p(C)$, and $p(ct(\bar{Q}_2)) = p(C[\mathtt{if}^l\, e\, \mathtt{then}\, (\bar{Q}_1;\, [])\, \mathtt{else}\, P_2]) = p(C) \cdot l$, as required. The case of T-IFFALSE is proved in the same way.

Next we consider T-WHILETRUE. In this case we have

$$(c : \mathtt{while}^l\, e\, \mathtt{do}\, P;\, \bar{P}, s) \xrightarrow{i}_c (P;\, \mathsf{end}(l) : \mathtt{while}^l\, e\, \mathtt{do}\, P;\, \bar{P}, s + l).$$

First, by the induction hypothesis, we have $p(c) = \mathrm{dom}(s)$. It is easily checked by induction on $\to_{Ic}$ that c is either $\mathsf{bgn}(l)$ or $\mathsf{end}(l)$. If $c = \mathsf{bgn}(l)$, then $l \notin p(c)$, since $p(\mathsf{end}(l)) = p(\mathsf{bgn}(l)) \cdot l$ by definition, and by Lemma 6.40 this sequence is strictly increasing. Therefore $l \notin \mathrm{dom}(s)$, so s does not end with (l, k) (for any k). If $c = \mathsf{end}(l)$, then $p(c) = p(\mathsf{bgn}(l)) \cdot l$. Therefore $s = s' \cdot (l, k)$ for some s' and k. To summarize the argument above, s ends with (l, k) for some k if and only if $c = \mathsf{end}(l)$.

With this in mind, we consider three cases, similarly to the case of T-IFTRUE. First, consider the case $\bar{Q}_2$ is a tail of $\bar{P}$, and write $\bar{P} = \bar{P}';\, \bar{Q}_2$. By the induction hypothesis we have

$$\begin{aligned}\mathrm{dom}(s) &= p(ct(\bar{Q}_2)) \cdot \mathrm{rm}(c : \mathtt{while}^l\, e\, \mathtt{do}\, P;\, \bar{P}') \\ &= p(ct(\bar{Q}_2)) \cdot \mathrm{rm}(\bar{P}') \cdot \mathrm{rm}(c : \mathtt{while}^l\, e\, \mathtt{do}\, P) \\ &= \begin{cases} p(ct(\bar{Q}_2)) \cdot \mathrm{rm}(\bar{P}') & c = \mathsf{bgn}(l) \\ p(ct(\bar{Q}_2)) \cdot \mathrm{rm}(\bar{P}') \cdot l & c = \mathsf{end}(l). \end{cases}\end{aligned}$$

What we have to prove is

$$\mathrm{dom}(s + l) = p(ct(\bar{Q}_2)) \cdot \mathrm{rm}(P;\, \mathsf{end}(l) : \mathtt{while}^l\, e\, \mathtt{do}\, P;\, \bar{P}').$$

Rewriting the right-hand side using the previous equation we obtain

$$\begin{aligned}\mathrm{RHS} &= p(ct(\bar{Q}_2)) \cdot \mathrm{rm}(\bar{P}') \cdot l \\ &= \begin{cases} \mathrm{dom}(s) \cdot l & c = \mathsf{bgn}(l) \\ \mathrm{dom}(s) & c = \mathsf{end}(l). \end{cases}\end{aligned}$$

This indeed equals $\mathrm{dom}(s+l)$, since s ends with (l, k) for some k if and only if $c = \mathsf{end}(l)$, as we have proved above. Consider the second case, where $\bar{Q}_1 = P$ and $\bar{Q}_2 = \mathsf{end}(l) : \mathtt{while}^l\, e\, \mathtt{do}\, P;\, \bar{P}$. In this case $p(ct(\bar{Q}_2)) \cdot \mathrm{rm}(\bar{Q}_1) = p(\mathsf{end}(l)) \cdot \mathrm{rm}(P) = p(\mathsf{end}(l))$. By a case splitting similar to the previous case we can verify that this indeed equals $\mathrm{dom}(s + l)$. In the third case, $\bar{Q}_2$ starts with a tail of P, by an argument similar to (3) of T-IFTRUE case, we obtain $p(ct(\bar{Q}_2)) = p(\mathsf{end}(l))$. The rest of the proof is the same as the previous case.

Other cases are almost straightforward. We only mention T-ENDIF, in which case we have $(\mathsf{end}(l) : \mathtt{endif}^l;\, \bar{P}, s \cdot (l, k)) \xrightarrow{i}_c (\bar{P}, s)$. If we split $\bar{P}$ as $\bar{Q}_1;\, \bar{Q}_2$, by the induction hypothesis we have $p(ct(\bar{Q}_2)) \cdot \mathrm{rm}(\mathsf{end}(l) : \mathtt{endif}^l;\, \bar{Q}_1) = \mathrm{dom}(s \cdot (l, k))$, and hence $p(ct(\bar{Q}_2)) \cdot \mathrm{rm}(\bar{Q}_1) \cdot l = \mathrm{dom}(s) \cdot l$. By canceling l we obtain the conclusion. □

LEMMA 6.46. *If (P, s) is reachable, then $\mathrm{dom}(s)$ is strictly increasing.*

PROOF. Immediate from Lemmas 6.40 and 6.45. □

LEMMA 6.47. *Let (P_i, s_i) for $i = 1, 2, 3$ be reachable configurations, and let $c_i = ct(P_i)$. If $s_1 \preceq s_2$, $s_1 \preceq s_3$, $s_2 \parallel s_3$, and c_1 is in the closed interval spanned by c_2 and c_3, then $s_2 <_{\mathsf{s}} s_3$ if and only if $c_2 < c_3$.*

PROOF. From the assumption we have either $c_2 \le c_1 \le c_3$ or $c_3 \le c_1 \le c_2$. In either case we have $p(c_2) \cap p(c_3) \subseteq p(c_1)$ by Lemma 6.41, and therefore $\mathrm{dom}(s_2) \cap \mathrm{dom}(s_3) \subseteq \mathrm{dom}(s_1)$ by Lemma 6.45. The converse of this inclusion also holds since s_1 is a common prefix. Since if we write $s_2 = s_1 \cdot (l_2, k_2) \cdots$ and $s_3 = s_1 \cdot (l_3, k_3) \cdots$, then $l_2 \neq l_3$ (for otherwise $l_2 = l_3 \in \mathrm{dom}(s_2) \cap \mathrm{dom}(s_3) = \mathrm{dom}(s_1)$, so l_2 appears in s_1, and hence l_2 appears more than once in $\mathrm{dom}(s_2)$, but this contradicts Lemma 6.46).

From the argument above, we have $l_2 \in \mathrm{dom}(s_2) \setminus \mathrm{dom}(s_3)$ and $l_3 \in \mathrm{dom}(s_3) \setminus \mathrm{dom}(s_2)$. By using an equivalence

$$l \in \mathrm{dom}(s_i) \iff \mathsf{bgn}(l) < c_i \le \mathsf{end}(l)$$

which follows from Lemma 6.45 and Lemma 6.39, we can obtain

$$c_2 \in (\mathsf{bgn}(l_2), \mathsf{end}(l_2)] \setminus (\mathsf{bgn}(l_3), \mathsf{end}(l_3)]\,, \text{ and}$$
$$c_3 \in (\mathsf{bgn}(l_3), \mathsf{end}(l_3)] \setminus (\mathsf{bgn}(l_2), \mathsf{end}(l_2)]\,.$$

Therefore by Lemma 6.38 two intervals $(\mathsf{bgn}(l_2), \mathsf{end}(l_2)]$ and $(\mathsf{bgn}(l_3), \mathsf{end}(l_3)]$ are disjoint. Therefore $c_2 < c_3$ if and only if $\mathsf{bgn}(l_2) < \mathsf{bgn}(l_3)$. since bgn is strictly monotone map between linear orders (Lemma 6.37), $\mathsf{bgn}(l_2) < \mathsf{bgn}(l_3)$ if and only if $l_2 < l_3$. So it only remains to show that $l_2 < l_3$ if and only if $s_2 <_{\mathsf{s}} s_3$, but this is immediate from the definition of $<_{\mathsf{s}}$ and the choice of l_2 and l_3 (also note that $l_2 \neq l_3$). $\square$

LEMMA 6.48. *L-order $<_{\mathsf{L}}$, when restricted to the set of all reachable thread configurations, is transitive.*

PROOF. Suppose $(P_1, s_1) <_{\mathsf{L}} (P_2, s_2) <_{\mathsf{L}} (P_3, s_3)$. Let $c_i = ct(P_i)$ for $i = 1, 2, 3$, and consider the conditions

(1) $s_1 \parallel s_2$ and $s_1 <_{\mathsf{s}} s_2$,
(2) $s_1 \preceq s_2$ and $c_1 < c_2$,
(3) $s_2 \preceq s_1$ and $c_1 < c_2$,
(4) $s_2 \parallel s_3$ and $s_2 <_{\mathsf{s}} s_3$,
(5) $s_2 \preceq s_3$ and $c_2 < c_3$, and
(6) $s_3 \preceq s_2$ and $c_2 < c_3$.

Then we have $(1 \vee 2 \vee 3) \wedge (4 \vee 5 \vee 6)$, hence $(1 \wedge 4) \vee (1 \wedge 5) \vee \cdots \vee (3 \wedge 6)$.

First, consider the case $1 \wedge 4$. Since $s_1 <_{\mathsf{s}} s_3$ is clear, it suffices to show that $s_1 \parallel s_3$. Suppose otherwise. It does not hold that $s_3 \preceq s_1$, since this implies $s_3 \leq_{\mathsf{s}} s_1$ but this contradicts $s_1 <_{\mathsf{s}} s_3$. So we have $s_1 \preceq s_3$. However, together with $s_1 <_{\mathsf{s}} s_2$ and $s_1 \parallel s_2$ this implies $s_3 <_{\mathsf{s}} s_2$ (which is a contradiction) as follows. From $s_1 \parallel s_2$ we have $s_1 = s_0 \cdot q \cdots$ and $s_2 = s_0 \cdot q' \cdots$ where s_0 is the longest common prefix. As $s_1 <_{\mathsf{s}} s_2$, we have $q < q'$. If $s_1 \preceq s_3$ then s_3 also has the form $s_0 \cdot q \cdots$, therefore $s_3 <_{\mathsf{s}} s_2$.

If $1 \wedge 5$ is the case, since $s_1 <_{\mathsf{s}} s_3$ it suffices to check that $s_1 \parallel s_3$. If $s_1 \preceq s_3$, then both s_1 and s_2 are prefixes of s_3, but this implies $s_1 \not\parallel s_2$, a contradiction. If $s_3 \preceq s_1$, then $s_2 \preceq s_3 \preceq s_1$, but this also implies $s_1 \not\parallel s_2$, a contradiction.

Suppose $1 \wedge 6$ is the case. Then $s_1 \not\preceq s_3$, since otherwise the transitivity of $\preceq$ implies $s_1 \preceq s_2$. Let us first consider the case $s_3 \not\preceq s_1$. In this case we can show that $s_1 <_{\mathsf{s}} s_3$. Let s_0 be the longest common prefix of s_1 and s_2. Then because $s_1 \parallel s_2$ we have $s_1 = s_0 \cdot q \cdots$ and $s_2 = s_0 \cdot q' \cdots$ with $q < q'$. Since $s_3 \preceq s_2$ but $s_3 \not\preceq s_1$, s_3 also has the form $s_0 \cdot q' \cdots$, therefore $s_1 <_{\mathsf{s}} s_3$. Next consider the case $s_3 \preceq s_1$. In this case $c_1 < c_3$ holds, because otherwise $c_2 < c_3 \leq c_1$, so by Lemma 6.47 we have $s_2 <_{\mathsf{s}} s_1$, but this contradicts 1.

Consider the case $2 \wedge 4$. Since $s_1 <_{\mathsf{s}} s_3$, it suffices to consider the case $s_1 \parallel s_3$. It is clear that $s_3 \not\preceq s_1$, so suppose $s_1 \preceq s_3$. We prove that $c_1 < c_3$. Otherwise, $c_3 \leq c_1 < c_2$. Because $s_1 \preceq s_2$, $s_1 \preceq s_3$, and $s_2 \parallel s_3$, by Lemma 6.47 we have $s_3 <_{\mathsf{s}} s_2$. However this contradicts 4.

The case $2 \wedge 5$ holds is easy since $\preceq$ and $<$ are transitive.

If $2 \wedge 6$ holds, then both s_1 and s_3 are prefixes of s_2. Therefore one of s_1 and s_3 is a prefix of the other, that is, $s_1 \not\parallel s_3$. Moreover we have $c_1 < c_2 < c_3$, so $(P_1, s_1) <_{\mathsf{L}} (P_3, s_3)$ as required.

Consider the case $3 \wedge 4$ holds. First, note that $s_2 \preceq s_1$, $s_2 <_{\mathsf{s}} s_3$, and $s_2 \not\preceq s_3$ implies $s_1 <_{\mathsf{s}} s_3$, so it suffices to show that $s_1 \parallel s_3$. Clearly $s_3 \not\preceq s_1$ since $s_1 <_{\mathsf{s}} s_3$, while $s_1 \preceq s_3$ implies $s_2 \preceq s_3$, a contradiction.

If $3 \wedge 5$ is the case, it suffices to show that $s_1 \parallel s_3$ implies $s_1 <_{\mathsf{s}} s_3$. Because s_2 is a common prefix of s_1 and s_3, and $c_1 < c_2 < c_3$, this follows from Lemma 6.47.

Finally, the case of $3 \wedge 6$ is similar to $2 \wedge 5$. This completes the proof. $\square$

6.4.4. More Properties of L-order. Having proved the transitivity of L-order, we next show that the thread configuration keeps increasing (with respect to L-order) during the execution. To prove this we use the fact that for all reachable thread configurations of the form $(c : P;\ c' : P';\ \bar{P}, s)$ it holds that $c < c'$ (which follows from Lemma 6.50). However, to prove this by induction, we need a stronger induction hypothesis, which is stated in terms of the following functions.

Definition 6.49. We define $allcts$ and $allcts^*$ as follows:

$$\begin{aligned}
allcts(P;\ P') &= allcts(P) \cdot allcts(P') \\
allcts(c : \mathtt{if}^l\ e\ \mathtt{then}\ P\ \mathtt{else}\ P') &= c \cdot allcts(P) \cdot allcts(P') \cdot \mathsf{end}(l) \\
allcts(c : \mathtt{while}^l\ e\ \mathtt{do}\ P) &= c \cdot allcts(P) \cdot \mathsf{end}(l) \\
allcts^*(P;\ P') &= allcts^*(P) \cdot allcts^*(P') \\
allcts^*(c : \mathtt{if}^l\ e\ \mathtt{then}\ P\ \mathtt{else}\ P') &= c \cdot allcts(P) \cdot allcts(P') \cdot \mathsf{end}(l) \\
allcts^*(c : \mathtt{while}^l\ e\ \mathtt{do}\ P) &= \begin{cases} \mathsf{end}(l) & c = \mathsf{end}(l) \\ c \cdot allcts(P) \cdot \mathsf{end}(l) & \text{otherwise} \end{cases}
\end{aligned}$$

and, if none of the above applies,

$$allcts^*(c : P) = allcts(c : P) = c.$$

$allcts(P)$ is the list of all counters appearing in P and $\mathsf{end}(l)$ for all $l \in labs(P)$, sorted in ascending order. $allcts^*(P)$ is similar, but the body of a `while`-statement is ignored if its counter is $\mathsf{end}(l)$ (that is, that loop is currently being executed).

LEMMA 6.50. *If (P, s) is reachable, then $allcts^*(P)$ is strictly increasing.*

PROOF. We prove that

(1) $allcts^*(P)$ is strictly increasing, and
(2) for each tail of P the form $\mathtt{while}^l\ e\ \mathtt{do}\ P';\ \bar{P}$, the sequence $allcts(P') \cdot \mathsf{end}(l) \cdot allcts^*(\bar{P})$ is also strictly increasing

by induction on $\rightarrow^*_{Ic}$. This holds for initial configurations by definition of well-annotated programs. For the induction step, we only check T-IFTRUE, T-IFFALSE, and T-WHILETRUE since other cases are easy. In the first two cases, we have

$$c : \mathtt{if}^l\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2;\ \bar{P} \xrightarrow{i}_c P_k;\ \mathsf{end}(l) : \mathtt{endif}^l;\ \bar{P}$$

for $k = 1$ or $k = 2$, hence the $allcts^*$ of the right-hand side is a subsequence of that of the left-hand side, so the first claim is immediate from the induction hypothesis. For the second claim, consider a tail $\bar{Q}$ of the right-hand side with the specified form. It suffices to consider the case where the tail contains a tail of P_k, since otherwise $\bar{Q}$ is a tail of $\bar{P}$ in which case the conclusion is immediate from the induction hypothesis. If $\bar{Q}$ contains a tail of P_k, split P_k as $P_k = \bar{Q}_1;\ \mathtt{while}^{l'}\ e\ \mathtt{do}\ P';\ \bar{Q}_2$ so that $\bar{Q} = \mathtt{while}^{l'}\ e\ \mathtt{do}\ P';\ \bar{Q}_2;\ \mathsf{end}(l) : \mathtt{endif}^l;\ \bar{P}$. Then what we have to show is that

$$allcts(P') \cdot \mathsf{end}(l') \cdot allcts^*(\bar{Q}_2) \cdot \mathsf{end}(l) \cdot allcts^*(\bar{P})$$

is increasing. Notice that $allcts(P') \cdot \mathsf{end}(l') \cdot allcts^*(\bar{Q}_2)$ is a subsequence of $allcts(\mathtt{while}^{l'}\ e\ \mathtt{do}\ P';\ \bar{Q}_2)$, which is a subsequence of $allcts(P_k)$. Therefore the whole sequence is a subsequence of $allcts^*(c : \mathtt{if}^l\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2;\ \bar{P})$, which is increasing by the induction hypothesis.

Consider T-WHILETRUE

$$c : \texttt{while}^l\, e\, \mathsf{do}\, P;\ \bar{P} \longrightarrow_c P;\ \mathsf{end}(l) : \texttt{while}^l\, e\, \mathsf{do}\, P;\ \bar{P}.$$

The second claim of the induction hypothesis implies that $allcts(P) \cdot \mathsf{end}(l) \cdot allcts^*(\bar{P})$ is strictly increasing. Since this sequence is a subsequence of $allcts^*$ of the right-hand side, the first claim is verified. For the second claim, we have three cases: the tail either (1) contains a tail of P, (2) equals $\mathsf{end}(l) : \texttt{while}^l\, e\, \mathsf{do}\, P;\ \bar{P}$, and (3) is a tail of $\bar{P}$. The last case is immediate from the induction hypothesis. In case (1), let the tail be $\texttt{while}^{l'}\, e'\, \mathsf{do}\, P';\ \bar{P}'$. Then the sequence we have to consider is $allcts(P') \cdot \mathsf{end}(l') \cdot allcts^*(\bar{P}') \cdot \mathsf{end}(l) \cdot allcts^*(\bar{P})$. This is indeed strictly increasing as it is a subsequence of $allcts(P) \cdot \mathsf{end}(l) \cdot allcts^*(\bar{P})$. In case (2), what we have to show is that $allcts(P) \cdot \mathsf{end}(l) \cdot allcts^*(\bar{P})$ is strictly increasing, which is the induction hypothesis. $\Box$

LEMMA 6.51. *Let $(P_i, s_i)_i, \sigma$ be a reachable I-configuration and suppose $(P_i, s_i)_i, \sigma \to_{Ic} (P'_i, s'_i)_i, \sigma'$. Then $(P_i, s_i) \leq_{\mathsf{L}} (P'_i, s'_i)$ for each i.*

PROOF. We check that the assertion holds for each rule. We omit the state part of the I-configurations, because it is irrelevant to the proof of this lemma.

First, consider I-SYNC. By Lemma 6.50 and the fact that $ct(\bar{P})$ is the first element of $allcts^*(\bar{P})$, we have $c < ct(\bar{P})$. Therefore $(c : \texttt{sync};\ \bar{P}, s) <_{\mathsf{L}} (\bar{P}, s)$.

Other cases uses I-THREAD, so we check that $(P, s) \xrightarrow{i}_c (P', s')$ implies $(P, s) <_{\mathsf{L}} (P', s')$ for each i.

The cases of T-SKIP, T-LASSIGN, and T-SASSIGN are similar to the case of I-SYNC. The cases of T-IFTRUE, T-IFFALSE, T-ENDIF, and T-WHILEFALSE are also similar. Although in these cases the stack is modified, in any cases we have $s \nparallel s'$: In the first two cases we have $s \preceq s'$, and the other two cases we have $s' \preceq s$.

The remaining case is T-WHILETRUE. If s is of the form $s_0 \cdot (l, k)$, then we have $s' = s_0 \cdot (l, k+1)$, so we have $s \parallel s'$ and $s <_{\mathsf{s}} s'$, from which the conclusion follows. Otherwise, we have

$$c : \texttt{while}^l\, e\, \mathsf{do}\, P;\ \bar{P}, s \xrightarrow{i}_c P;\ \mathsf{end}(l) : \texttt{while}^l\, e\, \mathsf{do}\, P;\ \bar{P}, s \cdot (l, 1).$$

If $c = \mathsf{bgn}(l)$ then this case is treated in the same way as other cases, using Lemma 6.50. Therefore it suffices to show that $c = \mathsf{bgn}(l)$. Suppose otherwise. Then it has to be the case that $c = \mathsf{end}(l)$, because a counter of a statement with label l is necessarily one of $\mathsf{bgn}(l)$ and $\mathsf{end}(l)$, which is easily proved by induction on the interleaving execution. However, by an easy induction we can also show that for any reachable thread configuration (P, s), if $\mathsf{end}(l)$ appears in P then l has to appear in $\mathrm{dom}(s)$. However, this is impossible since $\mathrm{dom}(s \cdot (l, 1)) = \mathrm{dom}(s) \cdot l$ is strictly increasing by Lemma 6.46. $\Box$

LEMMA 6.52. *Suppose $D = E[P, \mu, \sigma \Downarrow X] \in \mathscr{P}(P_0, \mu_0, \sigma_0 \Downarrow X_0)$, and $|D| = (P_i, s_i)_i, \sigma$. Then*

(*1*) *for each i, P_i is a sequence of subprograms of P;*
(*2*) $\mu \subseteq \mu_0$*;*
(*3*) *if $i \in \mu$, then $P_i \neq \checkmark$;*
(*4*) *if $i \notin \mu_0$, then $P_i = \checkmark$;*
(*5*) *if $P_i = \checkmark$, then $s_i = \varepsilon$.*

PROOF. By induction on D. $\Box$

LEMMA 6.53. *Suppose $D = E[P, \mu, \sigma \Downarrow X] \in \mathscr{P}(P_0, \mu_0, \sigma_0 \Downarrow X_0)$, and $|D| = (P_i, s_i)_i, \sigma$. Then for all $i \in \mu$ and $j \in \mathbb{T} \setminus \mu$, it holds that $(P_i, s_i) <_{\mathsf{L}} (P_j, s_j)$. Moreover, if $j \in \mu_0$, then $s_i \npreceq s_j$.*

PROOF. First, note that if $i \in \mu$ then $P_i \neq \checkmark$, and if $j \notin \mu_0$ then $P_j = \checkmark$ and $s_j = \varepsilon$. In such a case the lemma is obvious. Therefore, below we assume $j \in \mu_0 \setminus \mu$.

We prove the lemma by induction on the construction of D. (Notice that if P' is a subprogram of a well-annotated program P, then P' is also well-annotated with $\mathsf{end}_{P'}$ being the restriction of end_P.)

In the base case, we have $D = (P_0, \mu_0, \sigma_0 \Downarrow X_0)$, hence $\mu_0 = \mu$. Therefore we have nothing to prove.

Consider the case D is of the form

$$D = \frac{D_1 \quad P_2; \mu_0, X' \Downarrow X_0}{P_1;\ P_2, \mu_0, \sigma_0, \Downarrow X_0}$$

for non-total D_1. Let $(Q_i, s_i)_i = |D_1|$ (in the proof below, we omit the state part of an I-configuration because it is irrelevant). Then by definition of $|\cdot|$ we have $|D| = (Q_i;\ P_2, s_i \mid i \in \mu_0)$, and by the induction hypothesis $(Q_i, s_i) <_{\mathsf{L}} (Q_j, s_j)$. We have to show that $(Q_i;\ P_2, s_i) <_{\mathsf{L}} (Q_j;\ P_2, s_j)$ and $s_i \not\preceq s_j$; the latter is immediate from the induction hypothesis. If $s_i \parallel s_j$, then by the induction hypothesis we have $s_i <_{\mathsf{s}} s_j$, hence the conclusion follows. Otherwise, the induction hypothesis implies $ct(Q_i) < ct(Q_j)$. If $Q_j \neq \checkmark$, this implies $ct(Q_i;\ P_2) = ct(Q_i) < ct(Q_j) = ct(Q_j;\ P_2)$, so the conclusion follows. If $Q_j = \checkmark$, we have to show that $ct(Q_i) < ct(P_2)$. To see this, note that Q_i consists of subprograms of P_1 and $P_1;\ P_2$ is well-annotated. This means that any counter appearing in P_2 is greater than any counter in Q_i, hence $ct(Q_i) < ct(P_2)$.

If D is of the form

$$D = \frac{D_1 \quad D_2}{P_1;\ P_2, \mu_0, \sigma_0, \Downarrow X_0}$$

and D_1 is total, we have $|D| = |D_2|$, so the conclusion is immediate from the induction hypothesis.

Next, consider the case

$$D = \frac{D_1 \quad P_2, \mu_0 \setminus \sigma_0 [\![e]\!], X \Downarrow X_0}{\mathtt{if}^l\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2, \mu_0, \sigma_0 \Downarrow X_0}$$

where D_1 is non-total. Let $(Q_i, s_i)_i = |D_1|$. We have

$$|D| = \begin{pmatrix} Q_i;\ \mathtt{endif}^l, (l,1)\cdot s_i & \mid i \in \mu_0 \cap \sigma_0 [\![e]\!] \\ P_2;\ \mathtt{endif}^l, (l,2) & \mid i \in \mu_0 \setminus \sigma_0 [\![e]\!] \end{pmatrix}.$$

In case $j \in \mu_0 \setminus \sigma_0 [\![e]\!]$, the claim is that

$$(Q_i;\ \mathtt{endif}^l, (l,1)\cdot s_i) <_{\mathsf{L}} (P_2;\ \mathtt{endif}^l, (l,2)) \text{ and } (l,1)\cdot s_i \not\preceq (l,2).$$

The latter is obvious. The former follows from $(l,1)\cdot s_i \parallel (l,2)$, $Q_i \neq \checkmark$, and $ct(Q_i) < ct(P_2)$. The last inequality can be checked by an argument similar to the first case of sequencing. If $j \in (\mu_0 \cap \sigma_0 [\![e]\!]) \setminus \mu$, we have to show

$$(Q_i;\ \mathtt{endif}^l, (l,1)\cdot s_i) <_{\mathsf{L}} (Q_j;\ \mathtt{endif}^l, (l,1)\cdot s_j) \text{ and } (l,1)\cdot s_i \not\preceq (l,1)\cdot s_j$$

which follows from the induction hypothesis and, in case of $Q_j = \checkmark$, the fact that $ct(Q_i) < \mathsf{end}(l)$.

Next, consider the case

$$D = \frac{D_1 \quad D_2}{\mathtt{if}^l\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2, \mu_0, \sigma_0 \Downarrow X_0}$$

where D_1 is total. Let $(Q_i, s_i)_i = |D_2|$. In this case we have $j \in (\mu_0 \setminus \sigma_0 [\![e]\!]) \setminus \mu$, and

$$|D| = (Q_i;\ \mathtt{endif}^l, (l,2)\cdot s_i \mid i \in \mu \setminus \sigma_0 [\![e]\!]).$$

The claim, $(Q_i; \texttt{endif}^l, (l,2)\cdot s_i) <_{\mathsf{L}} (Q_j; \texttt{endif}^l, (l,2)\cdot s_j)$ and $(l,2)\cdot s_i \not\preceq (l,2)\cdot s_j$, is checked in a similar way to the previous case.

The remaining cases are while-statement. We first consider the case D has the form

$$\dfrac{P_0, \mu_1, \sigma_0 \Downarrow \sigma_1 \qquad \dfrac{\vdots}{\dfrac{D_1 \quad \texttt{while}^l\, e\, \texttt{do}\, P_0, \mu_k, X_1 \Downarrow X_0}{\texttt{while}^l\, e\, \texttt{do}\, P_0, \mu_{k-1}, \sigma_{k-1} \Downarrow X_0}}\; \texttt{while}^l\, e\, \texttt{do}\, P_0, \mu_1, \sigma_1 \Downarrow X_0}{\texttt{while}^l\, e\, \texttt{do}\, P_0, \mu_0, \sigma_0 \Downarrow X_0}$$

where $k \geq 1$ and D_1 is non-total. Let $(Q_i, s_i)_i = |D_1|$. Then s_i does not contain l since $D_1 \in \mathscr{P}(P_0, \mu_k, \sigma_{k-1} \Downarrow X_1)$ and P_0 does not contain l. Therefore

$$|D| = (Q_i; \mathsf{end}(l) : \texttt{while}^l\, e\, \texttt{do}\, P, (l,k)\cdot s_i \mid i \in \mu_k).$$

If $j \notin \mu_k$ the claim is obvious, so suppose $j \in \mu_k$. Note that $(l,k)\cdot s_i \parallel (l,k)\cdot s_j$ if and only if $s_i \parallel s_j$. In case $s_i \parallel s_j$, we have $s_i <_{\mathsf{s}} s_j$ by the induction hypothesis, and hence $(l,k)\cdot s_i <_{\mathsf{s}} (l,k)\cdot s_j$. Next, suppose $s_i \nparallel s_j$. Then we have $ct(Q_i) < ct(Q_j)$, and the claim follows in a similar way to other cases (e.g. the case of sequencing). $(l,k)\cdot s_i \not\preceq (l,k)\cdot s_j$ easily follows from the induction hypothesis.

Finally, we consider the case D is of the form

$$\dfrac{P_0, \mu_1, \sigma_0 \Downarrow \sigma_1 \qquad \dfrac{\vdots}{\dfrac{D_1 \quad \texttt{while}^l\, e\, \texttt{do}\, P_0, \mu_k, \sigma_k \Downarrow X_0}{\texttt{while}^l\, e\, \texttt{do}\, P_0, \mu_{k-1}, \sigma_{k-1} \Downarrow X_0}}\; \texttt{while}^l\, e\, \texttt{do}\, P_0, \mu_1, \sigma_1 \Downarrow X_0}{\texttt{while}^l\, e\, \texttt{do}\, P_0, \mu_0, \sigma_0 \Downarrow X_0}$$

where $k \geq 1$ and D_1 is total. We have

$$|D| = (\mathsf{end}(l) : \texttt{while}^l\, e\, \texttt{do}\, P_0, (l,k) \mid i \in \mu_k).$$

In this case $\mu = \mu_k$, so the claim is obvious. $\square$

PROOF OF LEMMA 6.31. Suppose $P_0, \mu_0, \sigma_0 \Downarrow X_0 \longrightarrow_c^* D = E[\mathsf{sync}, \mu, \sigma \Downarrow X]$ and $\mu \neq \emptyset, \mathbb{T}$. Then P_i is of the form $\mathsf{sync}; P$ for every $i \in \mu \neq \emptyset$. Therefore $|D|$ never terminates without using I-SYNC, since the execution of thread i does not proceed by other rules. So it suffices to show that I-SYNC is not applicable to any I-configuration reachable from $|D|$.

Suppose otherwise: there exists an I-configuration $(P_i', s_i')_i, \sigma'$ to which I-SYNC is applicable and is reachable from $|D|$. Then from the premise of I-SYNC we obtain $s_i' = s_j'$ for any pair of threads i, j. Without loss of generality we may assume that I-SYNC is not used in the transition $|D| = (P_i, s_i)_i, \sigma \to_{Ic}^* (P_i', s_i')_i\sigma'$. Then we also have $(P_i, s_i) = (P_i', s_i')$ for every $i \in \mu$.

Take $i \in \mu$ and $j \in \mathbb{T} \setminus \mu$. If $j \notin \mu_0$, then it follows that $P_j = \checkmark$, and in that case it is easy to see the conclusion: since $P_j = \checkmark$ the rule I-SYNC is never applicable. Therefore, we may assume $j \in \mu_0$. Then, by Lemmas 6.48, 6.51, and 6.53, we have

$$(P_i, s_i) <_{\mathsf{s}} (P_j, s_j) \leq_{\mathsf{s}} (P_j', s_i) \quad \text{and} \quad s_i \not\preceq s_j.$$

We will show that this leads to a contradiction. From the above inequalities we have

— if $s_i \parallel s_j$ then $s_i <_{\mathsf{s}} s_j$, and otherwise $ct(P_i) < ct(P_j)$, and
— if $s_j \parallel s_i$ then $s_j <_{\mathsf{s}} s_i$, and otherwise $ct(P_j) \leq ct(P_j')$.

Since $\|$ is symmetric while $s_i <_{\mathsf{s}} s_j$ and $s_j <_{\mathsf{s}} s_i$ are exclusive, it has to be the case that $s_i \nparallel s_j$ and $ct(P_i) < ct(P_j) \leq ct(P'_j)$. By Lemma 6.41, the latter implies $\mathrm{dom}(s_i) \subseteq \mathrm{dom}(s_j)$. On the other hand, since $s_i \not\preceq s_j$ as mentioned above, $s_i \nparallel s_j$ implies $s_j \prec s_i$. Therefore $s_j \cdot t = s_i$ for some $t \neq \varepsilon$. Then clearly $\mathrm{dom}(t) \subseteq \mathrm{dom}(s_i)$, but because $\mathrm{dom}(s_i) \subseteq \mathrm{dom}(s_j)$, we have $\mathrm{dom}(t) \subseteq \mathrm{dom}(s_j)$. However this is impossible because $\mathrm{dom}(s_j) \cdot \mathrm{dom}(t) = \mathrm{dom}(s_i)$ has to be strictly increasing and t is non-empty. □

6.5. Proof of the Equivalence

We now prove the equivalence between lockstep and interleaving semantics under race-freedom.

THEOREM. *Let P be a program and μ a mask and suppose that $(P, \varepsilon \mid i \in \mu), \sigma$ is race-free. Then, $P, \mu, \sigma \Downarrow \sigma'$ if and only if $(P, \varepsilon \mid i \in \mu), \sigma \rightarrow_I^* (\checkmark, \varepsilon)_i, \sigma'$.*

PROOF. Below, $\mathcal{C}_0$ denotes the initial I-configuration $(P, \varepsilon \mid i \in \mu), \sigma$.

Suppose that $P, \mu, \sigma \Downarrow \sigma'$ has a derivation D. From completeness of the derivation search procedure (Proposition 6.7), we have $P, \mu, \sigma \Downarrow X \longrightarrow^* D$. By Lemma 6.20, the assumption of race-freedom implies that D is locally interleavable, so by Proposition 6.9 we conclude that $\mathcal{C}_0 \rightarrow_I^* |D| = (\checkmark, \varepsilon)_i, \sigma'$.

For the converse, suppose that $\mathcal{C}_0 \rightarrow_I^* (\checkmark, \varepsilon)_i, \sigma'$. Notice that, by Lemma 6.16, any execution sequence from $\mathcal{C}_0$ is finite and ends with $(\checkmark, \varepsilon)_i, \sigma'$.

First, suppose that the lockstep execution terminates, that is, there exists σ'' such that $P, \mu, \sigma \Downarrow \sigma''$. Then by the same argument as above we obtain $\mathcal{C}_0 \rightarrow_I^* (\checkmark, \varepsilon)_i, \sigma''$, so by determinacy $\sigma' = \sigma''$. Therefore, if the lockstep execution terminates, the final state necessarily equals σ', that is $P, \mu, \sigma \Downarrow \sigma'$ holds, as required. This argument means that it is sufficient to show the termination of the lockstep execution.

Below we suppose that there does not exist σ'' such that $P, \mu, \sigma \Downarrow \sigma''$, and derive a contradiction. From this assumption, at least one of the following holds: (1) there is an infinite sequence via $\longrightarrow$ from $P, \mu, \sigma \Downarrow X$, or (2) $P, \mu, \sigma \Downarrow X \longrightarrow^* E[\mathtt{sync}, \mu_1, \sigma_1 \Downarrow X_1]$ and $\mu_1 \neq \emptyset, \mathbb{T}$.

In case (1), let $D_0 = (P, \mu, \sigma \Downarrow X) \longrightarrow D_1 \longrightarrow \ldots$ be an infinite sequence. Then by Proposition 6.9 we obtain a sequence $\mathcal{C}_0 = |D_0| \rightarrow_I^* |D_1| \rightarrow_I^* \ldots$. It suffices to show that this sequence is also infinite, since the existence of such a sequence contradicts the determinacy mentioned above. To this end, we show that $|D_n| \rightarrow_I^+ |D_{n+1}|$ for infinitely many n. Let us write $D_n = E_n[P_n, \mu_n, \sigma_n \Downarrow X_n]$, and $D_n \Longrightarrow D_{n+1}$ if either P_n is a sequencing or $\mu_n = \emptyset$. Then, from Proposition 6.9, $|D_n| = |D_{n+1}|$ if and only if $D_n \Longrightarrow D_{n+1}$. It is easy to check that there is no infinite sequence using only $\Longrightarrow$ (the length of such a sequence can be bound by the size of the program). Therefore $|D_n| \rightarrow_I^+ |D_{n+1}|$ for infinitely many n.

In case (2), let $D = E[\mathtt{sync}, \mu_1, \sigma_1 \Downarrow X_1]$. Then by Lemma 6.20, D is locally interleavable, hence by Proposition 6.9, we obtain $\mathcal{C}_0 \rightarrow_I^* |D|$. This means that any execution sequence from $|D|$ is a suffix of an execution sequence from $\mathcal{C}_0$, which eventually has to terminate with $(\checkmark, \varepsilon)_i, \sigma'$ by determinacy. Therefore $|D| \rightarrow_I^* (\checkmark, \varepsilon)_i, \sigma'$, but this contradicts Lemma 6.21. □

7. ADDITIONAL REMARKS

Treatment of synchronization failure. As already mentioned, barrier divergence and non-termination are identified in our lockstep semantics, although the behavior of barrier divergence is typically given as undefined [NVIDIA 2014] and, in fact, barrier divergence may cause the program execution to terminate, but with an unpredictable result on a real GPU. Since our logic is for *partial* correctness assertions, any assertion can be proved for a program that causes barrier divergence. Therefore, in order for our verification technique to be useful in practice, the absence of barrier divergence should be verified separately, e.g., by using other verification techniques studied elsewhere [Li and Gopalakrishnan 2010; 2012; Li et al. 2012b; Betts et al. 2012; Bardsley et al. 2014].

As we elaborate below, it would not be difficult to adapt our lock-step semantics and logic so that barrier divergence results in a special, erroneous state, but correspondence with interleaving execution (under the race-freedom assumption) would be rather subtle.

The lock-step semantics could be modified by introducing a special symbol $\bot$ and a new rule

$$\frac{\mu \neq \emptyset \qquad \mu \neq \mathbb{T}}{\mathtt{sync}, \mu, \sigma \Downarrow \bot} \qquad \text{(E-SyncDiv)}$$

which models barrier divergence. The Hoare quadruple $\{\varphi\}\, m \Rightarrow P\, \{\psi\}$ would be defined to be valid if and only if $P, \sigma\, [\![m]\!]\, , \sigma \Downarrow \sigma'$ implies $\sigma' \neq \bot$ and $\sigma' \models \psi$; H-SYNC would be replaced with

$$\{(\mathit{all}(m) \vee \mathit{none}(m)) \wedge \varphi\}\, m \Rightarrow \mathtt{sync}\, \{\varphi\}\, .$$

(Other rules have to be adapted to handle $\bot$.) Then, we could prove $\{\varphi\}\, m \Rightarrow \mathtt{sync}\, \{\psi\}$ only if φ implies $\mathit{all}(m) \vee \mathit{none}(m)$. Both soundness and relative completeness would hold.

Can we modify the interleaving semantics so that Theorem 5.5 holds as it is? Such interleaving semantics has to simulate E-SYNCDIV for the left-to-right direction of the theorem to hold, but we find it difficult to define such semantics, because there is no obvious way of detecting barrier-divergence when threads interleave.

One possible trick would be to use the L-order introduced in Section 6.4: we conjecture that we can recover the equivalence between lockstep and interleaving semantics by extending the interleaving semantics with the following execution rule, which handles barrier divergence.

$$\frac{P_j = \mathtt{sync};\ P'_j \qquad (P_j, s_j) <_{\mathsf{L}} (P_k, s_k)}{((P_i, s_i)_i, \sigma) \to_I \bot}$$

The second premise $(P_j, s_j) <_{\mathsf{L}} (P_k, s_k)$ above means that thread j is waiting at a barrier, but thread k cannot reach this location. Note that, in this case, we have to work with programs annotated with program counters, as in Section 6.4. We conjecture that $P, \mu, \sigma \Downarrow \bot$ if and only if $(P, \varepsilon \mid i \in \mu), \sigma \to_I^* \bot$; both directions would be proved as in Section 6. Soundness and relative completeness with respect to the new interleaving semantics would be stated as follows:

CONJECTURE 7.1. *Let P be a program with monotonic loops and suppose that $\{\varphi\}\, m \Rightarrow P\, \{\psi\}$ is derivable. Let σ be a state such that the I-configuration $(P, \varepsilon \mid i \in \sigma\, [\![m]\!]), \sigma$ is race-free, $\sigma \models \varphi$ holds, and $(P, \varepsilon \mid i \in \sigma\, [\![m]\!]), \sigma \to_I^* \mathcal{C} \not\to_I$. Then, it holds that $\mathcal{C}$ is of the form $(\checkmark, \varepsilon)_i, \sigma'$ and $\sigma' \models \psi$.*

CONJECTURE 7.2. *Let P be a program with monotonic loops such that $(P, \varepsilon \mid i \in \sigma\, [\![m]\!]), \sigma$ is race-free for all σ such that $\sigma \models \varphi$. Then, $\{\varphi\}\, m \Rightarrow P\, \{\psi\}$ is derivable if for all σ and σ' such that $\sigma \models \varphi$ and $(P, \varepsilon \mid i \in \sigma\, [\![m]\!]), \sigma \to_I^* \mathcal{C} \not\to_I$, it holds that $\mathcal{C}$ is of the form $(\checkmark, \varepsilon)_i, \sigma'$ and $\sigma' \models \psi$.*

(Here, $\mathcal{C} \not\to_I$ means that there is no $\mathcal{C}'$ such that $\mathcal{C} \to_I \mathcal{C}'$.) Note that under the new semantics a terminating configuration is of the form either $(\checkmark, \varepsilon)_i, \sigma'$ or $\bot$. The changes in the statements correspond to those in the definition of valid quadruples (under the new lockstep semantics).

Even without changing the interleaving semantics, a slightly weakened version of Theorem 5.5 still holds by modifying the statement to "... Then, $P, \mu, \sigma \Downarrow \sigma'$ and $\sigma' \neq \bot$ if and only if $(P, \varepsilon \mid i \in \mu), \sigma \rightarrow_I^* (\checkmark, \varepsilon)_i, \sigma'$." The proof of Section 6 would still work (notice that $\sigma' \neq \bot$ if and only if $P, \mu, \sigma \Downarrow \sigma'$ is derived without using E-SYNCDIV). Soundness (Corollary 5.6) would hold as it is, but relative completeness (Corollary 5.7) would not. A counterexample can be easily constructed from a program that results in barrier divergence. We conjecture that relative completeness is recovered by assuming that P does not result in barrier divergence (in the lockstep semantics), i.e., $P, \sigma \llbracket m \rrbracket, \sigma \not\Downarrow \bot$ whenever $\sigma \models \varphi$.

Multiple warps. The actual execution of GPUs is a hybrid of our interleaving and lockstep semantics. Instead of scheduling each thread individually, we treat each warp as a unit of interleaving execution. Each warp executes a program as in our lockstep semantics, and warps interleave as in our interleaving semantics. Equivalence between this semantics and complete lockstep semantics would be proved similarly. Race-freedom can be relaxed so that a race involving two threads belonging to the same warp occurs only within a single assignment.

Function calls. We did not include function calls in our formalism, but we conjecture that we can follow the existing extension of Hoare Logic with function calls, and this would not be technically difficult. However, a complication would stem from function parameters. If a function is called from the host code, then the arguments are uniform (i.e., all threads receive the same value), but if it is called from the device code the arguments may vary among threads. Therefore, we would have to treat these two types of function calls differently.

In addition, masks have to be taken into account to write the specifications of functions. Currently, we do not have to introduce a mask into pre- and postconditions, because we assume that the program is a complete device code, and therefore all threads are enabled at the beginning of the execution. If we extend our system with function calls, then a function may have to specify a formula referring to the state of the mask. Thus, we have to slightly extend the assertion language.

8. RELATED WORK

Semantics of GPU programs. Habermaier and Knapp [2012] formalized both SIMT (lockstep) and interleaved multi-thread semantics and discussed the relationships between them. In particular, they proved that their SIMT semantics can be simulated by the interleaved semantics with appropriate scheduling. Collingbourne et al. [2013] considered a lockstep execution of an unstructured program based on a control-flow graph. They defined both interleaving and lockstep semantics and proved that the two semantics are equivalent in a certain sense under the assumption of race-freedom and termination. Betts et al. [2012] defined another semantics, called synchronous, delayed visibility (SDV) semantics. The main difference between this and other semantics (including ours) is that it keeps track of the accesses to shared memory, and raises an error if a race is detected. Based on this semantics, they developed a verification tool GPUVerify that automatically detects race condition and barrier divergence. These three semantics are all small-step, and it appears that ours is the first big-step semantics for lockstep execution.

Deductive verification. The Owicki–Gries method [Owicki and Gries 1976] and rely/guarantee reasoning [Jones 1981] are well-known approaches for deductive verification of concurrent programs. Their main concern is to reason about interference. The difficulty is that an assertion can be invalidated by other threads through shared variables. To solve this problem, the Owicki-Gries method verifies that each assertion is not invalidated by other threads; rely/guarantee reasoning specifies an assumption on the behavior of an environment as a rely condition. In contrast, in this work we did not need to handle such an interference. This is because we assumed lockstep execution, in which threads cannot interleave. Although a race can occur when assigning to a shared variable, such a race does not affect the soundness of our logic because the assertion language can express such a nondeterminism (as in the rule H-ASSIGN).

For deductive verification of GPU programs, Blom, Huisman, and Mihelčić suggested using permission-based separation logic [Blom et al. 2014]. They demonstrated that they could verify race-freedom and functional correctness by using separation logic. They considered an assignment of resources to thread, and used it to prove race-freedom. As compared to their approach, our framework cannot prove race-freedom, but provides a simpler proof system for verifying functional correctness relying on the assumption of race-freedom. An extension of Blom et al.'s system with a frame rule has recently been proposed by Asakura et al. [2016]. The soundness of their system was proved on the Coq proof assistant.

Equivalence between lockstep and interleaving semantics. Collingbourne et al. [2013] also proved an equivalence result between lockstep and interleaving execution. We discuss several differences between their work and ours, and what is improved from theirs.

First, our semantics treats barrier synchronizations more formally than that of Collingbourne et al., by introducing a stack into a thread configuration. Collingbourne et al. did not do this formally. They introduced special thread-local variables v_{barrier}, which are set to the id of the barrier when a barrier is reached, and v_L for every loop labeled by L, which counts the number of iterations of the loop L. These variables are called barrier variables and play a role similar to that of the stack in our semantics. It is assumed that these variables are modified appropriately when a thread executes a loop or reaches a barrier, but this is stated informally only in prose English. The formal execution rules do not mention barrier variables, and thus, the rules do not specify when and to what value the contents of barrier variables should be changed.

Second, our proof given in Section 6 is more formal than that provided in the full version of [Collingbourne et al. 2013]. This is partly because barrier variables are not fully formalized in their execution rules. For example, to prove that the interleaving semantics can simulate the lockstep semantics, they had to show that if the lockstep semantics succeeds synchronization, then so does the interleaving semantics (the first part of the claim in the proof of Theorem B.20). To do this, they argued that the barrier variables satisfy the premise of the execution rule for synchronization, but this argument does not appear to be formal. In contrast, we have made our arguments as formal as possible throughout the proof of equivalence. We believe that most parts of our arguments are sufficiently formal that one can mechanize them in a proof assistant without nontrivial modifications.

Third, the statement of our equivalence theorem is simpler. The equivalence stated in Collingbourne et al. [2013], unlike ours, guarantees the equivalence on shared variables only, and the statement explicitly mentions the termination of the program. Further, their lockstep semantics was not directly defined. It was given by a translation from a GPU program P (which is executed in an interleaving semantics) into a sequential vector program $\phi(P)$ encoding the lockstep execution of P.

Verification tools. Verification tools for GPU programs have been developed by several authors. Tripakis, Stergiou, and Lublinerman [2010] developed a method to check the determinism and equivalence of SPMD programs based on non-interference. Collingbourne, Cadar, and Kelly [2011; 2012] proposed a method of symbolic execution of SIMD programs based on the KLEE symbolic execution tool. Li and Gopalakrishnan [2010; 2012] developed SMT-based verification tools. Li et al. [2012b] developed a concolic verification and test generation tool for GPU programs, called GKLEE. Further optimizations and extensions of GKLEE have also been considered [Li et al. 2012a; Chiang et al. 2013]. Bardsley et al. developed a tool, GPUVerify [Betts et al. 2012; Bardsley et al. 2014; Betts et al. 2015], which statically checks the race freedom of OpenCL and CUDA kernels.

9. CONCLUSION

We extended the while-language with arrays and several features of GPU kernels, and defined a Hoare Logic for this language. We formalized the execution model of our language using two semantics, lockstep and interleaving, and we first proved that our Hoare Logic is sound and relatively complete for the lockstep semantics. Although in the proof we worked under the assumption that the program contains only monotonic loops, this additional assumption is not a serious limitation, because we can transform any program into an equivalent one conforming to this condition. We also considered the relationship between lockstep and interleaving semantics. We proved that for race-free programs the two semantics produce the same result. This means that, as far as race-free programs are concerned, our Hoare Logic is sound and relatively complete with respect to the interleaving semantics. This implies that we can separate the verification of GPU kernels into two problems, race-freedom and functional correctness, and our framework can be used to solve the latter, assuming that the former has already been verified.

We are currently implementing an automated verifier based on this work. It successfully verifies a matrix multiplication program with shared-memory optimization [Kojima et al. 2016]. We have also mechanized our Hoare Logic on Coq, and manually verified several implementations of prefix-sum algorithms [Okumura et al. 2016], which are more complicated than the examples shown in Section 3.3.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We thank Kohei Suenaga and the anonymous reviewers for valuable comments.

REFERENCES

Krzysztof R. Apt, Frank de Boer, and Ernst-Rdiger Olderog. 2009. *Verification of Sequential and Concurrent Programs* (3rd ed.). Springer Publishing Company, Incorporated.

Izumi Asakura, Hidehiko Masuhara, and Tomoyuki Aotani. 2016. Proof of Soundness of Concurrent Separation Logic for GPGPU in Coq. *Journal of Information Processing* 24, 1 (2016), 132–140.

Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Quadeer. 2014. Engineering a Static Verification Tool for GPU Kernels. In *Proc. of the 26th International Conference on Computer Aided Verification, CAV 2014 (LNCS)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer Verlag, 226–242.

Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 113–132. DOI:http://dx.doi.org/10.1145/2384616.2384625

Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. 2015. The Design and Implementation of a Verification Technique for GPU Kernels. *ACM Trans. Program. Lang. Syst.* 37, 3, Article 10 (May 2015), 49 pages. DOI:http://dx.doi.org/10.1145/2743017

Stefan Blom, Marieke Huisman, and Matej Mihelčić. 2014. Specification and verification of GPGPU programs. *Science of Computer Programming* 95, 3 (12 2014), 376–388.

Wei-Fan Chiang, Ganesh Gopalakrishnan, Guodong Li, and Zvonimir Rakamarić. 2013. Formal Analysis of GPU Programs with Atomics via Conflict-Directed Delay-Bounding. In *Proc. of the 5th NASA Formal Methods Symposium (NFM 2013) (LNCS)*, Vol. 7871. Springer Verlag, 213–228.

Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic crosschecking of floating-point and SIMD code. In *Proc. of the sixth conference on Computer systems (EuroSys '11)*. ACM, New York, NY, USA, 315–328. DOI:http://dx.doi.org/10.1145/1966445.1966475

Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2012. Symbolic Testing of OpenCL Code. In *Proc. of Hardware and Software: Verification and Testing (LNCS)*, Kerstin Eder, João Lourenço, and Onn Shehory (Eds.), Vol. 7261. Springer Verlag, 203–218. DOI:http://dx.doi.org/10.1007/978-3-642-34188-5_18

Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. 2013. Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels. In *Proc. of European Symposium on Programming (ESOP'13) (LNCS)*, Vol. 7792. Springer Verlag, 270–289. full version is available at http://multicore.doc.ic.ac.uk/tools/GPUVerify/ESOP2013/.

Carl A. Gunter and Didier Rémy. 1993. *A Proof-Theoretic Assessment of Runtime Type Errors*. Technical Report. AT&T Bell Laboratories Technical Memo 11261-921230-43TM.

Axel Habermaier and Alexander Knapp. 2012. On the Correctness of the SIMT Execution Model of GPUs. In *Proc. of European Symposium on Programming (ESOP'12) (LNCS)*, Vol. 7211. Springer Verlag, 316–335.

C. B. Jones. 1981. *Development methods for computer programs including a notion of interference*. Ph.D. Dissertation. Oxford University. Printed as: Programming Research Group, Technical Monograph 25.

Kensuke Kojima and Atsushi Igarashi. 2013. A Hoare Logic for SIMT Programs. In *Proc. of Asian Symposium on Programming Languages and Systems (LNCS)*, Chung chieh Shan (Ed.), Vol. 8301. Springer Verlag, 58–73.

Kensuke Kojima, Akifumi Imanishi, and Atsushi Igarashi. 2016. Automated Verification of Functional Correctness of Race-Free GPU Programs. (2016). draft.

Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *Proc. of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. ACM, 187–196. DOI:http://dx.doi.org/10.1145/1882291.1882320

Guodong Li and Ganesh Gopalakrishnan. 2012. Parameterized Verification of GPU Kernel Programs. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2450–2459.

Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012b. GKLEE: concolic verification and test generation for GPUs. In *Proc. of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 215–224. DOI:http://dx.doi.org/10.1145/2145816.2145844

Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2012a. Parametric flows: automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, Article 29, 10 pages.

NVIDIA 2014. *NVIDIA CUDA C Programming Guide*. NVIDIA.

NVIDIA 2015. *Parallel Thread Execution ISA Version 4.3*. NVIDIA.

Kentaro Okumura, Kensuke Kojima, and Atsushi Igarashi. 2016. Mechanization of Hoare Logic for SIMT in Coq and Verification of Parallel Prefix-Sum Algorithms. In *Proceedings of the 18th JSSST Workshop on Programming and Programming Languages (PPL2016)*. in Japanese.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. Purcell. 2007. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum* 26, 1 (2007), 80–113.

Susan Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* 6 (1976), 319–340.

Stavros Tripakis, Christos Stergiou, and Roberto Lublinerman. 2010. *Checking Equivalence of SPMD Programs Using Non-Interference*. Technical Report UCB/EECS-2010-11. EECS Department, University of California, Berkeley.

Glynn Winskel. 1993. *The Formal Semantics of Programming Languages*. The MIT Press.

Online Appendix to: A Hoare Logic for GPU Kernels

KENSUKE KOJIMA, Kyoto University and CREST, JST
ATSUSHI IGARASHI, Kyoto University and CREST, JST

A. AUXILIARY LEMMAS

LEMMA A.1. *Let σ be a state. Then $\sigma \models all(e)$ if and only of $\sigma\,[\![e]\!] = \mathbb{T}$, and $\sigma \models none(e)$ if and only if $\sigma\,[\![e]\!] = \emptyset$.*

LEMMA A.2. *If x' is a variable not occurring in φ, then $\sigma[x' \mapsto a] \models \varphi[x'/x]$ if and only if $\sigma[x \mapsto a] \models \varphi$.*

LEMMA A.3. *Suppose that m does not contain variables occurring in P. Then $P, \sigma\,[\![m]\!]\,, \sigma \Downarrow \sigma'$ implies $\sigma\,[\![m]\!] = \sigma'\,[\![m]\!]$.*

LEMMA A.4. *If $P, \emptyset, \sigma \Downarrow \sigma'$, then $\sigma = \sigma'$.*

LEMMA A.5. *Let $W = \texttt{while}\, e\, \texttt{do}\, P$. Suppose $\mu \cap \sigma\,[\![e]\!] = \mu' \cap \sigma\,[\![e]\!]$ and there is a derivation of $W, \mu, \sigma \Downarrow \sigma'$. Then there is a derivation of the same size (the number of nodes) with conclusion $W, \mu', \sigma \Downarrow \sigma'$.*

B. PROOF OF SOUNDNESS

Soundness of H-CONSEQ, H-SKIP, and H-SEQ are obvious. H-SYNC is also easy; `sync` gets stuck if and only if $\sigma \not\models all(m) \vee none(m)$.

To show that H-ASSIGN is sound, suppose $x\,[\bar{e}]\ \texttt{:=}\ e, \sigma\,[\![m]\!]\,, \sigma \Downarrow \sigma'$. From Lemma 3.6, σ' is of the form $\sigma[x \mapsto a]$ where a satisfies $\sigma[x' \mapsto a] \models assign(x', m, x, \bar{e}, e)$. So if we have $\sigma \models \forall x'.assign(x', m, x, \bar{e}, e) \to \varphi[x'/x]$, then $\sigma[x' \mapsto a] \models \varphi[x'/x]$. By using Lemma A.2, we obtain $\sigma[x \mapsto a] \models \varphi$, and therefore $\sigma' \models \varphi$ (because $\sigma' = \sigma[x \mapsto a]$), as required.

Next we check H-IF. Suppose $\sigma \models \varphi$ and $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q, \sigma\,[\![m]\!]\,, \sigma \Downarrow \sigma''$. Then there exists σ' such that $P, \sigma\,[\![m]\!] \cap \sigma\,[\![e]\!]\,, \sigma \Downarrow \sigma'$ and $Q, \sigma\,[\![m]\!] \setminus \sigma\,[\![e]\!]\,, \sigma' \Downarrow \sigma''$. We have to show $\sigma'' \models \chi$. Let $\sigma_0 = \sigma[z \mapsto \sigma\,[\![e]\!]]$, $\sigma_0' = \sigma'[z \mapsto \sigma\,[\![e]\!]]$, and $\sigma_0'' = \sigma''[z \mapsto \sigma\,[\![e]\!]]$. Then, since z does not occur in P and $\sigma\,[\![e]\!] = \sigma_0(z)$ it holds that $P, \sigma_0\,[\![m\ \texttt{\&\&}\ z]\!]\,, \sigma_0 \Downarrow \sigma_0'$. Similarly, we also have $Q, \sigma_0'\,[\![m\ \texttt{\&\&}\ !\ z]\!]\,, \sigma_0' \Downarrow \sigma_0''$. Then from the induction hypotheses we have $\sigma_0' \models \psi$ and $\sigma_0'' \models \chi$. Since z does not occur in χ, and σ_0'' and σ'' differ only in z, it holds that $\sigma'' \models \chi$.

Finally we show that H-WHILE is sound by induction on the size of the derivation of $\Downarrow$. Precisely, by induction we prove that if $\{\varphi \wedge e = z\}\ m\ \texttt{\&\&}\ z \Rightarrow P\ \{\varphi\}$ is valid, then for all σ and σ' such that $\texttt{while}\ e\ \texttt{do}\ P, \sigma\,[\![m]\!]\,, \sigma \Downarrow \sigma'$ and $\sigma \models \varphi$, it holds that $\sigma' \models \varphi \wedge none(m\ \texttt{\&\&}\ e)$.

The base case is the rule E-WHILEFALSE, which is obvious. For the induction step, let us assume the derivation has the form

$$\dfrac{P, \sigma\,[\![m]\!] \cap \sigma\,[\![e]\!]\,, \sigma \Downarrow \sigma' \qquad \begin{array}{c}\vdots\ \mathcal{D}\\ \texttt{while}\ e\ \texttt{do}\ P, \sigma\,[\![m]\!] \cap \sigma\,[\![e]\!]\,, \sigma' \Downarrow \sigma''\end{array}}{\texttt{while}\ e\ \texttt{do}\ P, \sigma\,[\![m]\!]\,, \sigma \Downarrow \sigma'}$$

and suppose $\sigma \models \varphi$. We have to show that $\sigma'' \models \varphi \wedge none(m\ \texttt{\&\&}\ e)$.

DOI: http://dx.doi.org/10.1145/0000000.0000000

Let $\sigma_0 = \sigma[z \mapsto \sigma\,[\![e]\!]]$ and $\sigma'_0 = \sigma'[z \mapsto \sigma\,[\![e]\!]]$. Then, since z is fresh, we have

$$P, \sigma_0\,[\![m]\!] \cap \sigma_0\,[\![e]\!], \sigma_0 \Downarrow \sigma'_0,$$

and $\sigma_0 \models \varphi \wedge e = z$. Since $\sigma_0\,[\![m]\!] \cap \sigma_0\,[\![e]\!] = \sigma_0\,[\![m \;\texttt{\&\&}\; z]\!]$, by assumption we obtain $\sigma'_0 \models \varphi$.

Let $\sigma''_0 = \sigma''[z \mapsto \sigma_0\,[\![e]\!]]$. Then we have a derivation of

$$\texttt{while}\; e\; \texttt{do}\; P, \sigma\,[\![m]\!] \cap \sigma\,[\![e]\!], \sigma'_0 \Downarrow \sigma''_0$$

with the same size as $\mathcal{D}$. Now we are going to use Lemma A.5 to obtain a derivation of

$$\texttt{while}\; e\; \texttt{do}\; P, \sigma'_0\,[\![m]\!], \sigma'_0 \Downarrow \sigma''_0,$$

again with the same size as $\mathcal{D}$. Here the assumption of Lemma A.5 is indeed satisfied: the monotonicity and Lemma 4.2 implies $\sigma\,[\![m]\!] \cap \sigma'\,[\![e]\!] \subseteq \sigma\,[\![m]\!] \cap \sigma\,[\![e]\!]$, so by definition of σ'_0 we have $(\sigma\,[\![m]\!] \cap \sigma\,[\![e]\!]) \cap \sigma'_0\,[\![e]\!] = \sigma'_0\,[\![m]\!] \cap \sigma'_0\,[\![e]\!]$.

Then we can apply the induction hypothesis, therefore $\sigma'_0 \models \varphi$ implies $\sigma''_0 \models \varphi \wedge \mathit{none}(m \;\texttt{\&\&}\; e)$. Since the antecedent is already proved, we have $\sigma''_0 \models \varphi \wedge \mathit{none}(m \;\texttt{\&\&}\; e)$. Moreover, z does not occur in φ, m nor e, which implies $\sigma'' \models \varphi \wedge \mathit{none}(m \;\texttt{\&\&}\; e)$. This completes the proof.

C. PROOF OF RELATIVE COMPLETENESS

By the standard argument, it suffices to show that

$$\vdash \{\mathrm{wlp}(m, P, \varphi)\}\, m \Rightarrow P\, \{\varphi\}\,.$$

We proceed by induction on P

When $P = \texttt{skip}$, by H-SKIP we have $\vdash \{\varphi\}\, m \Rightarrow \texttt{skip}\, \{\varphi\}$. So it suffices to show that $\models \mathrm{wlp}(m, \texttt{skip}, \varphi) \to \varphi$. Suppose $\sigma \models \mathrm{wlp}(m, \texttt{skip}, \varphi)$. Then, since $\texttt{skip}, m, \sigma \Downarrow \sigma$, we conclude $\sigma \models \varphi$.

When $P = \texttt{sync}$, by H-SYNC we have $\vdash \{\mathit{all}(m) \vee \mathit{none}(m) \to \varphi\}\, m \Rightarrow \texttt{sync}\, \{\varphi\}$, so it suffices to show that $\models \mathrm{wlp}(m, \texttt{sync}, \varphi) \to \mathit{all}(m) \vee \mathit{none}(m) \to \varphi$. This is clear from E-SYNC.

When $P = x\,\texttt{[}\bar{e}\texttt{]} \;\texttt{:=}\; e$, by H-ASSIGN we have

$$\vdash \{\forall x'.\mathit{assign}(x', m, x, \bar{e}, e) \to \varphi[x'/x]\}\, m \Rightarrow x\,\texttt{[}\bar{e}\texttt{]} \;\texttt{:=}\; e\, \{\varphi\}\,.$$

So it suffices to show that

$$\models \mathrm{wlp}(m, x\,\texttt{[}\bar{e}\texttt{]} \;\texttt{:=}\; e, \varphi) \to \forall x'.\mathit{assign}(x', m, x, \bar{e}, e) \to \varphi[x'/x].$$

Suppose $\sigma \models \mathrm{wlp}(m, x\,\texttt{[}\bar{e}\texttt{]} \;\texttt{:=}\; e, \varphi)$ and $\sigma[x' \mapsto a] \models \mathit{assign}(x', m, x, \bar{e}, e)$. Then from Lemma 3.6, we have $x\,\texttt{[}\bar{e}\texttt{]} \;\texttt{:=}\; e, \sigma\,[\![m]\!], \sigma \Downarrow \sigma[x \mapsto a]$. Therefore $\sigma[x \mapsto a] \models \varphi$, hence by Lemma A.2 we obtain $\sigma[x' \mapsto a] \models \varphi[x'/x]$.

When $P = P_1;\, P_2$, by the induction hypotheses we have $\vdash \{\mathrm{wlp}(m, P_1, \psi)\}\, m \Rightarrow P_1\, \{\psi\}$ and $\vdash \{\mathrm{wlp}(m, P_2, \varphi)\}\, m \Rightarrow P_2\, \{\varphi\}$ for all ψ and φ. Therefore by H-SEQ

$$\vdash \{\mathrm{wlp}(m, P_1, \mathrm{wlp}(m, P_2, \varphi))\}\, m \Rightarrow P_1; P_2\, \{\varphi\}\,.$$

So it suffices to show that

$$\models \mathrm{wlp}(m, P_1;\, P_2, \varphi) \to \mathrm{wlp}(m, P_1, \mathrm{wlp}(m, P_2, \varphi)).$$

Suppose $\sigma \models \mathrm{wlp}(m, P_1;\, P_2, \varphi)$, and consider σ' such that $P_1, \sigma\,[\![m]\!], \sigma \Downarrow \sigma'$. We have to show that $\sigma' \models \mathrm{wlp}(m, P_2, \varphi)$, that is, $\sigma'' \models \varphi$ for all σ'' with $P_2, \sigma'\,[\![m]\!], \sigma' \Downarrow \sigma''$. This is immediate from $P_1;\, P_2, \sigma\,[\![m]\!], \sigma \Downarrow \sigma''$ which follows from assumptions and E-SEQ.

When $P = \texttt{if}\; e\; \texttt{then}\; P_1\; \texttt{else}\; P_2$, let $\chi = \mathrm{wlp}(m \;\texttt{\&\&}\; z, P_1, \mathrm{wlp}(m \;\texttt{\&\&}\; \texttt{!}\; z, P_2, \varphi))$. Then by the induction hypotheses we have

$$\vdash \{\chi\}\, m \;\texttt{\&\&}\; z \Rightarrow P_1\, \{\mathrm{wlp}(m \;\texttt{\&\&}\; \texttt{!}\; z, P_2, \varphi)\}\,,$$

$$\vdash \{\mathrm{wlp}(m \,\texttt{\&\&}\, \texttt{!}\, z, P_2, \varphi)\}\; m \,\texttt{\&\&}\, \texttt{!}\, z \Rightarrow P_2 \,\{\varphi\}\,.$$

Since

$$\models (\exists z.e = z \wedge \chi) \wedge e = z \rightarrow \chi,$$

we have

$$\vdash \{(\exists z.e = z \wedge \chi) \wedge e = z\}\; m \,\texttt{\&\&}\, z \Rightarrow P_1 \,\{\mathrm{wlp}(m \,\texttt{\&\&}\, \texttt{!}\, z, P_2, \varphi)\}$$

by H-CONSEQ. Therefore, by H-IF,

$$\vdash \{\exists z.e = z \wedge \chi\}\; m \Rightarrow \texttt{if}\; e \;\texttt{then}\; P_1 \;\texttt{else}\; P_2 \,\{\varphi\}\,.$$

So our goal is to prove

$$\models \mathrm{wlp}(\texttt{if}\; e \;\texttt{then}\; P_1 \;\texttt{else}\; P_2, m, \varphi) \rightarrow \exists z.e = z \wedge \chi.$$

Suppose $\sigma \models \mathrm{wlp}(\texttt{if}\; e \;\texttt{then}\; P_1 \;\texttt{else}\; P_2, m, \varphi)$, and let $\sigma_0 = \sigma[z \mapsto \sigma\,[\![e]\!]]$. It suffices to show that $\sigma_0 \models e = z \wedge \chi$. It is obvious that $\sigma_0 \models e = z$. To prove $\sigma_0 \models \chi$, suppose

$$\begin{aligned} &P_1, \sigma_0\,[\![m \,\texttt{\&\&}\, z]\!]\,, \sigma_0 \Downarrow \sigma', \\ &P_2, \sigma'\,[\![m \,\texttt{\&\&}\, \texttt{!}\, z]\!]\,, \sigma' \Downarrow \sigma''. \end{aligned}$$

Then, since z and variables in m are fresh, we have $\sigma'\,[\![m \,\texttt{\&\&}\, \texttt{!}\, z]\!] = \sigma_0\,[\![m \,\texttt{\&\&}\, \texttt{!}\, z]\!]$, hence $P_2, \sigma_0\,[\![m \,\texttt{\&\&}\, \texttt{!}\, z]\!]\,, \sigma' \Downarrow \sigma''$. Therefore by E-IF and the equality $\sigma_0(z) = \sigma\,[\![e]\!] = \sigma_0\,[\![e]\!]$ we obtain

$$\texttt{if}\; e \;\texttt{then}\; P_1 \;\texttt{else}\; P_2, \sigma_0\,[\![m]\!]\,, \sigma_0 \Downarrow \sigma''.$$

On the other hand, we assumed that $\sigma \models \mathrm{wlp}(\texttt{if}\; e \;\texttt{then}\; P_1 \;\texttt{else}\; P_2, m, \varphi)$ and this formula does not depend on z, so σ_0 satisfies the same formula. Hence $\sigma'' \models \varphi$, as required.

When $P = \texttt{while}\; e \;\texttt{do}\; Q$, let $\psi = \exists z.e = z \wedge \mathrm{wlp}(m \,\texttt{\&\&}\, z, P, \varphi)$. We prove

(1) $\vdash \{\psi \wedge e = z\}\; m \,\texttt{\&\&}\, z \Rightarrow Q \,\{\psi\}$,
(2) $\models \psi \wedge \mathit{none}(m \,\texttt{\&\&}\, e) \rightarrow \varphi$, and
(3) $\models \mathrm{wlp}(m, P, \varphi) \rightarrow \psi$.

The conclusion follows from them by H-WHILE and H-CONSEQ.

First we prove (1). By the induction hypothesis it suffices to prove the validity instead of the provability. So our goal is

$$\sigma \models \psi \wedge e = z \text{ and } Q, \sigma\,[\![m \,\texttt{\&\&}\, z]\!]\,, \sigma \Downarrow \sigma' \implies \sigma' \models \psi.$$

If $\sigma\,[\![m \,\texttt{\&\&}\, z]\!] = \emptyset$, then by Lemma A.4 we have $\sigma' = \sigma$, hence this is clear. Below we assume $\sigma\,[\![m \,\texttt{\&\&}\, z]\!] \neq \emptyset$. By definition of ψ, the above statement is equivalent to

$$\begin{aligned} &\sigma \models \psi \wedge e = z,\;\; Q, \sigma\,[\![m \,\texttt{\&\&}\, z]\!]\,, \sigma \Downarrow \sigma', \text{ and} \\ &P, (\sigma'[z \mapsto \sigma'\,[\![e]\!]])\,[\![m \,\texttt{\&\&}\, z]\!]\,, \sigma'[z \mapsto \sigma'\,[\![e]\!]] \Downarrow \sigma'' \\ &\implies \sigma'' \models \varphi. \end{aligned}$$

Suppose the premises hold for σ, σ' and σ''. Let $\sigma''_0 = \sigma''[z \mapsto \sigma'(z)]$. Then it suffices to show that $\sigma''_0 \models \varphi$.

First, from $\sigma \models \psi \wedge e = z$ it follows that $\sigma \models \mathrm{wlp}(m \,\texttt{\&\&}\, z, P, \varphi)$, so to prove $\sigma''_0 \models \varphi$ it suffices to show that

$$P, \sigma\,[\![m \,\texttt{\&\&}\, z]\!]\,, \sigma \Downarrow \sigma''_0.$$

To prove this, we first show that

$$Q, \sigma\,[\![m \,\texttt{\&\&}\, z]\!] \cap \sigma\,[\![e]\!]\,, \sigma \Downarrow \sigma' \text{ and } P, \sigma\,[\![m \,\texttt{\&\&}\, z]\!] \cap \sigma\,[\![e]\!]\,, \sigma' \Downarrow \sigma''_0,$$

and then apply E-WHILETRUE. Note that the rule is applicable because the assumption $\sigma \models e = z$ implies $\sigma [\![m \mathbin{\&\&} z]\!] \cap \sigma [\![e]\!] = \sigma [\![m \mathbin{\&\&} z]\!]$. The first assertion also follows from $\sigma [\![m \mathbin{\&\&} z]\!] \cap \sigma [\![e]\!] = \sigma [\![m \mathbin{\&\&} z]\!]$ together with the assumption. For the second one, note that

$$P, (\sigma'[z \mapsto \sigma' [\![e]\!]]) [\![m \mathbin{\&\&} z]\!], \sigma' \Downarrow \sigma_0''$$

holds from assumption and the fact that z is fresh. In view of Lemma A.5, it suffices to show that

$$(\sigma [\![m \mathbin{\&\&} z]\!] \cap \sigma [\![e]\!]) \cap \sigma' [\![e]\!] = ((\sigma'[z \mapsto \sigma' [\![e]\!]]) [\![m \mathbin{\&\&} z]\!]) \cap \sigma' [\![e]\!].$$

From $\sigma [\![e]\!] = \sigma [\![z]\!]$ this reduces to

$$(\sigma [\![m]\!] \cap \sigma [\![e]\!]) \cap \sigma' [\![e]\!] = \sigma [\![m]\!] \cap \sigma' [\![e]\!].$$

This follows from the assumption of monotonicity and Lemma 4.2. This completes the proof of (1).

Next we prove (2). Suppose $\sigma \models \psi \wedge \mathit{none}(m \mathbin{\&\&} e)$ and let $\sigma_0 = \sigma[z \mapsto \sigma [\![e]\!]]$. Then by definition of ψ we have $\sigma_0 \models \mathrm{wlp}(m \mathbin{\&\&} z, P, \varphi)$. Moreover, $\sigma_0 [\![z]\!] = \sigma [\![e]\!]$ and $\sigma \models \mathit{none}(m \mathbin{\&\&} e)$ imply that $\sigma_0 [\![m \mathbin{\&\&} z]\!] = \emptyset$, therefore $P, \sigma_0 [\![m \mathbin{\&\&} z]\!], \sigma_0 \Downarrow \sigma_0$. Hence $\sigma_0 \models \varphi$, and φ does not contain z, so $\sigma \models \varphi$ as required.

Finally we prove (3). Suppose $\sigma \models \mathrm{wlp}(m, P, \varphi)$, and let $\sigma_0 = \sigma[z \mapsto \sigma [\![e]\!]]$. Then clearly $\sigma_0 \models e = z$. We will prove $\sigma_0 \models \mathrm{wlp}(m \mathbin{\&\&} z, P, \varphi)$. To do this suppose $P, \sigma_0 [\![m \mathbin{\&\&} z]\!], \sigma_0 \Downarrow \sigma_0'$. Then, since $\sigma_0 [\![m \mathbin{\&\&} z]\!] = \sigma [\![m]\!] \cap \sigma [\![e]\!]$, by Lemma A.5 we have $P, \sigma [\![m]\!], \sigma_0 \Downarrow \sigma_0'$. Since $\mathrm{wlp}(m, P, \varphi)$ does not depend on z and σ satisfies this formula, so does σ_0. Therefore $\sigma_0' \models \varphi$.

D. PROOF OF THE SOUNDNESS OF DERIVATION SEARCH

Let us say a partial derivation D to be *admissible* if, for every substitution $\{\bar{\sigma}/\bar{X}\}$ such that $\bar{X}$ is the list of all state variables occurring in D and every leaf of $D\{\bar{\sigma}/\bar{X}\}$ is derivable, then $D\{\bar{\sigma}/\bar{X}\}$ is obtained by truncating several branches of some valid derivation.

It suffices to prove that all partial derivations are admissible: a total derivation D is a partial derivation that does not contain state variables, hence its admissibility implies that D itself is a valid derivation.

To prove this it suffices to prove that $\longrightarrow$ preserves admissibility, since $P, \mu, \sigma \Downarrow X$ is clearly admissible. The cases of S-ATOM and S-WHILEFALSE are obvious. Consider the case of S-SEQ:

$$D = E\left[P_1;\ P_2, \mu, \sigma \Downarrow X\right] \longrightarrow E\left[\frac{P_1, \mu, \sigma \Downarrow X' \quad P_2, \mu, X' \Downarrow X}{P_1;\ P_2, \mu, \sigma \Downarrow X}\right] = D'.$$

Suppose D is admissible, and let $\{\bar{\sigma}/\bar{X}\}$ be a substitution such that every leaf of $D'\{\bar{\sigma}/\bar{X}\}$ is a valid judgment. Let us denote by σ_Y the state corresponding to a variable Y appearing in $\bar{X}$. Then the assumption means that

$$D'\{\bar{\sigma}/\bar{X}\} = E\{\bar{\sigma}/\bar{X}\}\left[\frac{P_1, \mu, \sigma \Downarrow \sigma_{X'} \quad P_2, \mu, \sigma_{X'} \Downarrow \sigma_X}{P_1;\ P_2, \mu, \sigma \Downarrow \sigma_X}\right]$$

has valid judgments as its leaves.[4] Then, its truncation

$$D\{\bar{\sigma}/\bar{X}\} = E\{\bar{\sigma}/\bar{X}\}\left[P_1;\ P_2, \mu, \sigma \Downarrow \sigma_X\right]$$

[4]Here $E\{\bar{\sigma}/\bar{X}\}$ does not belong to any syntactic category that we have defined so far, but we believe the meaning of the whole expression is clear.

also has valid judgments as its leaves: leaves appearing in $E\{\bar{\sigma}/\bar{X}\}$ are valid since they are leaves of $D'\{\bar{\sigma}/\bar{X}\}$, and $P_1;\ P_2, \mu, \sigma \Downarrow \sigma_X$ is valid by rule E-SEQ and the assumptions that both $P_1, \mu, \sigma \Downarrow \sigma_{X'}$ and $P_2, \mu, \sigma_{X'} \Downarrow \sigma_X$ are valid. Therefore, by admissibility of D it follows that $D\{\bar{\sigma}/\bar{X}\}$ is a truncation of some valid derivation, say D_0. The node of D_0 corresponding to $P_1;\ P_2, \mu, \sigma \Downarrow \sigma_X$ in place of the hole of E has to be extended as

$$\frac{P_1, \mu, \sigma \Downarrow \sigma'' \quad P_2, \mu, \sigma'' \Downarrow \sigma_X}{P_1;\ P_2, \mu, \sigma \Downarrow \sigma_X}$$

in D_0. Although σ'' does not necessarily coincide with $\sigma_{X'}$, it is possible to replace this node with another one. This is because the premises of

$$\frac{P_1, \mu, \sigma \Downarrow \sigma_{X'} \quad P_2, \mu, \sigma_{X'} \Downarrow \sigma_X}{P_1;\ P_2, \mu, \sigma \Downarrow \sigma_X}$$

are both valid, hence have some derivations. Therefore $D'\{\bar{\sigma}/\bar{X}\}$ is also a truncation of some derivation. S-IF and S-WHILETRUE can be treated in the same way.

E. PROOF OF THE COMPLETENESS OF DERIVATION SEARCH

Let us say a partial derivation D *approximates* a total derivation D_0 if there exists a substitution $\{\bar{\sigma}/\bar{X}\}$ such that $D\{\bar{\sigma}/\bar{X}\}$ is obtained by truncating several branches of D_0. We write $D \sqsubseteq D_0$ if D approximates D_0.

Clearly if D_0 is a derivation of $P, \mu, \sigma \Downarrow \sigma'$ then $(P, \mu, \sigma \Downarrow X) \sqsubseteq D_0$, so it suffices to show that

(1) for every partial derivation D, if $D \sqsubseteq D_0$ and $D \neq D_0$, then there exists D' such that $D \longrightarrow D'$ and $D' \sqsubseteq D_0$, and
(2) there exists no infinite sequence $D_1 \longrightarrow D_2 \longrightarrow \dots$ such that $D_i \sqsubseteq D_0$ for all $i > 0$.

To prove the first claim, note that if $D \sqsubseteq D_0$ and $D \neq D_0$, then D contains at least one state variable, and hence of the form $E[P, \mu, \sigma \Downarrow X]$. If this cannot be extended, then it has to be the case that $P = \mathtt{sync}$ and $\mu \neq \emptyset, \mathbb{T}$. However, since D approximates D_0, for some σ_X the judgment $P, \mu, \sigma \Downarrow \sigma_X$ has to appear in D_0, and hence this has to be a valid judgment, a contradiction. Therefore there exists some D' such that $D \longrightarrow D'$. It remains to check that D' can be chosen so that $D' \sqsubseteq D_0$. If S-ATOM is applicable to D, take σ' so that $P, \mu, \sigma \Downarrow \sigma'$ is the corresponding rule instance in D_0, and let $D' = D\{\sigma'/X\}$. Then we have $D \longrightarrow D'$, and D' is a truncation of D_0. The case of S-WHILEFALSE is similar. Next, consider S-SEQ:

$$D = E[P_1;\ P_2, \mu, \sigma \Downarrow X] \longrightarrow E\left[\frac{P_1, \mu, \sigma \Downarrow X_1 \quad P_2, \mu, X_1 \Downarrow X}{P_1;\ P_2, \mu, \sigma \Downarrow X}\right] = D'.$$

The state corresponding to X_1 in D_0 determines σ_{X_1}, so that $D'\{\bar{\sigma}, \sigma_{X_1}/\bar{X}, X_1\}$ is a truncation of D_0. Other two rules are similar.

For the second claim, let $n(D)$ be the number of nodes of a partial derivation D, and $v(D)$ the number of state variables occurring in D. For each D such that $D \sqsubseteq D_0$, consider the pair $m(D) = (n(D_0) - n(D), v(D)) \in \mathbb{N} \times \mathbb{N}$. Then it is easy to check that if $D \longrightarrow D'$ then $m(D') < m(D)$ where $<$ is the lexicographic order. The absence of an infinite sequence follows from the well-foundedness of $(\mathbb{N} \times \mathbb{N}, <)$.

F. PROOF OF SIMULATION

LEMMA F.1. *Suppose* $(P_i, s_i \mid i \in \mu), \sigma \rightarrow_I (P'_i, s'_i \mid i \in \mu), \sigma'$, *and consider families of programs* Q_i *indexed by* $i \in \mu$ *and a stack* t. *In case* $t \neq \varepsilon$ *we additionally assume that, for each* i *such that* $s_i = \varepsilon$, *the last element of* $\mathrm{dom}(t)$ *does not appear in* P_i. *Then,* $(P_i;\ Q_i, t \cdot s_i \mid i \in \mu), \sigma \rightarrow_I (P'_i;\ Q_i, t \cdot s'_i \mid i \in \mu), \sigma'$.

PROOF. By induction on the derivation of $\xrightarrow{i}$. In the case of T-WHILEFALSE, we need to check that if s_i does not end with an element of the form (l,k) then neither does $t \cdot s_i$, which is a consequence of the assumption that if $s_i = \varepsilon$ the last element of $\mathrm{dom}(t)$ does not appear in P_i. □

LEMMA F.2. *Let $D \in \mathscr{P}(P,\mu,\sigma \Downarrow X)$ and $(Q_i,s_i)_i,\sigma' = |D|$. Then all labels appearing in Q_i appear in P.*

PROOF. By induction on D. □

PROOF OF PROPOSITION 6.9. By induction on the size of E. First, consider the case $E = []$. Then we have $D = (P,\mu,\sigma \Downarrow X)$. If P is a sequencing, $|D| = |D'|$. Otherwise, we can obtain $|D| \to_I^* |D'|$ by applying I-THREAD $|\mu|$ times, so if $\mu \neq \emptyset$ we have $|D| \to_I^+ |D'|$. Note that if P is an assignment to a shared variable, we use the assumption that D' is locally interleavable.

Suppose $E \neq []$ and the conclusion of E is $\texttt{while}^l\, e\, \texttt{do}\, P_0$. Then E is either

$$\dfrac{P_0,\mu_1,\sigma_0 \Downarrow \sigma_1 \qquad \dfrac{\vdots}{\dfrac{E' \quad \texttt{while}^l\, e\, \texttt{do}\, P_0,\mu_k,X_1 \Downarrow X_0}{\texttt{while}^l\, e\, \texttt{do}\, P_0,\mu_{k-1},\sigma_{k-1} \Downarrow X_0}} \atop \texttt{while}^l\, e\, \texttt{do}\, P_0,\mu_1,\sigma_1 \Downarrow X_0}{\texttt{while}^l\, e\, \texttt{do}\, P_0,\mu_0,\sigma_0 \Downarrow X_0} \tag{3}$$

where $k \geq 1$, or

$$\dfrac{P_0,\mu_1,\sigma_0 \Downarrow \sigma_1 \qquad \dfrac{\vdots}{\dfrac{P_0,\mu_k,\sigma_{k-1} \Downarrow \sigma_k \quad []}{\texttt{while}^l\, e\, \texttt{do}\, P_0,\mu_{k-1},\sigma_{k-1} \Downarrow X_0}} \atop \texttt{while}^l\, e\, \texttt{do}\, P_0,\mu_1,\sigma_1 \Downarrow X_0}{\texttt{while}^l\, e\, \texttt{do}\, P_0,\mu_0,\sigma_0 \Downarrow X_0} \tag{4}$$

where $k \geq 1$. In both cases $\mu_j = \mu_{j-1} \cap \sigma_{j-1} [\![e]\!]$ for each $1 \leq j \leq k$, and $\mu_k \neq \emptyset$. Below we abbreviate $\texttt{while}^l\, e\, \texttt{do}\, P_0$ as R.

Consider the case (3), and let $D_1 = E'[P,\mu,\sigma \Downarrow X]$. Let E'' be a context enclosing D_1 in D, that is, $E = E''[E']$ and therefore $D = E''[D_1]$. Since E'' is an evaluation context, we have either $D' = E''[D_1']$ for some D_1' with $D_1 \longrightarrow D_1'$, or $D' = E''[D_1]\{\sigma'/X_1\}$ for some σ' with $D_1 \longrightarrow D_1\{\sigma'/X_1\}$. In each of these cases, by the induction hypothesis we have $|D_1| \to_I^* |D_1'|$ and $|D_1| \to_I^* (\checkmark,\varepsilon)_i,\sigma'$, respectively. In the first case, let $(Q_i,s_i)_i,\sigma = |D_1|$ and $(Q_i',s_i')_i,\sigma' = |D_1'|$. Then what we have to show is

$$|E''|(Q_i,s_i)_i,\sigma \to_I^* |E''|(Q_i',s_i')_i,\sigma',$$

that is

$$(Q_i;\ R,(l,k)\cdot s_i \mid i \in \mu_k),\sigma \to_I^* (Q_i';\ R,(l,k)\cdot s_i' \mid i \in \mu_k),\sigma'.$$

This follows from Lemma F.1 and the induction hypothesis, but to apply Lemma F.1 we have to check that $l \notin \mathit{labs}(Q_i)$. This follows from the fact that Q_i is a program part of $|D_1|$ and D_1 has a conclusion of the form $P_0,\mu_k,\sigma_{k-1} \Downarrow X_1$. Here P_0 is the body of a while-statement with label l, hence does not contain l (because we assume the same label does not appear twice in a single program). Therefore by Lemma F.2, Q_i does not contain l. In the second case, where $|D_1| \to_I^* (\checkmark,\varepsilon)_i,\sigma'$, we have $D' = E''[D_1]\{\sigma'/X_1\}$ so $|D'| = (R,(l,k) \mid i \in \mu_k),\sigma'$. Since $|D|$ is of the form $(Q_i;\ R,(l,k)\cdot s_i)_i,\sigma$ and by induction hypothesis we have $(Q_i,s_i)_i,\sigma \to_I^* (\checkmark,\varepsilon)_i,\sigma'$, the conclusion follows from Lemma F.1, using $l \notin \mathit{labs}(Q_i)$ which is verified in the same way as above.

Next, consider the case (4). In this case $D = E[R, \mu_k, \sigma_k \Downarrow X_0]$ and there are two possibilities:

$$D' = D\{\sigma_k/X_0\}, \text{ or } D' = E\left[\frac{P_0, \mu_{k+1}, \sigma_k \Downarrow X_1 \quad R, \mu_{k+1}, X_1 \Downarrow X_0}{R, \mu_k, \sigma_k \Downarrow X_0}\right].$$

In the first case, we have $|D'| = (\checkmark, \varepsilon)_i, \sigma_k$ and $\mu_k \cap \sigma_k [\![e]\!] = \emptyset$, and in the second case $|D'| = (P_0;\ R, (l, k+1) \mid i \in \mu_{k+1}), \sigma_k$ and $\mu_{k+1} = \mu_k \cap \sigma_k [\![e]\!]$. In both cases it is easy to see that $|D| = (R, (l, k) \mid i \in \mu_k), \sigma \rightarrow_I^* |D'|$.

Next we consider the case of sequencing:

$$E = \frac{D_1 \quad E'}{P_1;\ P_2, \mu_0, \sigma_0 \Downarrow X_0} \quad \text{or} \quad E = \frac{E' \quad P_2, \mu_0, X_1 \Downarrow X_0}{P_1;\ P_2, \mu_0, \sigma_0 \Downarrow X_0}.$$

Then we have either

$$|D| = (Q_i, s_i)_i, \sigma \ \text{ or } \ |D| = (Q_i;\ P_2, s_i), \sigma.$$

The conclusion follows by an argument similar to the previous case, using Lemma F.1. The case of if-statement is also similar. □

Received February 2007; revised March 2009; accepted June 2009