A Hoare Logic for SIMT Programs

Kensuke Kojima[1,2] and Atsushi Igarashi[1,2]

[1] Kyoto University, Japan
[2] JST CREST, Japan

Abstract. We study a Hoare Logic to reason about GPU kernels, which are parallel programs executed on GPUs. We consider the SIMT (Single Instruction Multiple Threads) execution model, in which multiple threads execute in lockstep (that is, execute the same instruction at a time). When control branches both branches are executed sequentially but during the execution of each branch only those threads that take it are enabled; after the control converges, all threads are enabled and execute in lockstep again. In this paper we adapt Hoare Logic to the SIMT setting, by adding an extra component representing the set of enabled threads to the usual Hoare triples. It turns out that soundness and relative completeness 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 soundness and relative completeness hold.

1 Introduction

General purpose computing on graphics processing units (GPGPU) has recently become widely available even to end-users, enabling us to utilize computational power of GPUs for solving problems other than graphics processing. Application areas include physics simulation, signal and image processing, etc. [1]. However, writing and optimizing GPU kernels, which are parallel programs executed on GPUs, is still a hard task and error-prone. For example, in programming in CUDA, a parallel computing platform and programming model on GPU [2], we have to care about synchronization and data races so that many threads cooperate correctly. Moreover, to obtain the best performance, we usually have to take into account more low-level mechanisms, to optimize memory access pattern, increase occupancy, etc.

Much effort has recently been made to develop automated verification tools for GPU kernels [3–11]. These tools try to automate detections of synchronization errors, data races, and inefficiency, as well as checking functional correctness and generating test cases. They, although automation is a great advantage, tend 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. The relative completeness of the inference rules guarantee that all correct programs can be proved to be correct, although much effort is often required to complete the correctness proof. Deductive approach has been implemented as tools that can be applied to real-world programs (Why3[3], for example). However, in the context of GPU programming, this approach is not extensively studied yet (at the time of writing, we are only aware of the ongoing work using separation logic by Huisman and Mihelčić [12]).

In this work, we study a deductive verification method for GPU programs. We focus on the 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 few modifications. Our contributions are (1) an extension of Hoare Logic to GPU kernels, and (2) proofs of its soundness and relative completeness for a large class of GPU kernels.

Generally speaking, reasoning about parallel programs requires much more sophisticated techniques than the sequential ones, because threads can interfere with each other through shared resources [13]. Although existing techniques could 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 the soundness and relative completeness hold under a very mild restriction.

In the rest of this section we describe how SIMT works, and how we can extend Hoare Logic to the SIMT setting.

1.1 Overview of the SIMT Execution Model

SIMT (Single Instruction Multiple Threads) is a parallel execution model of GPUs employed by CUDA. A CUDA program is written in CUDA C, an extension of C language, and run on GPUs as specified in the SIMT execution model. In the SIMT execution model, multiple (typically thousands of) threads are launched and execute in lockstep, i.e., execute the same instruction at a time.

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

Therefore, in SIMT, some statements actually may 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) that describe 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.

As an example, let us consider the following program.

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

[3] `http://why3.lri.fr/`

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 whose value 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 4 threads, and `n` equals 6. In the first iteration, the condition `k < n` holds in all threads, so 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, 3 are 4, 5, 6, 7 respectively, so 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 $\{0, 1\}$. After that all threads exit the loop, and program terminates. The final value of `c` is the sum of `a` and `b`.

Although the way SIMT executes threads looks similar to SIMD (Single Instruction Multiple Data) in that a single instruction operates on multiple data, they are different in that parallel operations on vectors are explicitly specified in SIMD while it is not the case for SIMT. Indeed, when programming in CUDA C we only specify a behavior of a single scalar thread, like a usual sequential program written C or C++.

1.2 Extending Hoare Logic

Next we consider a Hoare Logic for the SIMT execution model. The programs we are going to reason about is a single GPU kernel, like the example above.

Actually, we can employ many of the inference rules from the ordinary Hoare Logic without significant changes, although Hoare triples have to be changed. As explained above, in SIMT the effect of the 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 another piece of information, and consider a Hoare *quadruple* of the form $\{\varphi\}\, m \mid P\, \{\psi\}$, where m denotes a mask. Intuitively this quadruple means that "if an initial state satisfies φ, and we execute a program P *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 can handle all programs soundly, we decided to keep simplicity by making some assumption on the program we deal with. As a result we consider a certain class of programs, which we call *regular* programs, and obtain the soundness and relative completeness for regular programs. However, this is not a serious restriction because any program can be transformed into a regular one without changing the behavior (with respect to our operational semantics).

Interestingly, the resulting Hoare Logic is quite similar to the ordinary one, despite the parallel nature of GPU programs. It seems that this simplicity is a result of the fact that in SIMT dependency between threads is relatively weak. Threads basically work independently, and only at synchronization points they have to wait for each other. As a result the execution of a SIMT program is very similar to a sequential program.

1.3 Organization of the Paper

The rest of the paper is organized as follows. In Section 2 we formalize the SIMT execution model by extending the usual while-language. Section 3 describes our Hoare Logic. Section 4 introduces the notion of regular programs, and prove soundness and relative completeness of our Hoare Logic for regular programs. In Section 5 we discuss some variants of our system. Section 6 mentions related work and Section 7 concludes the paper.

2 SIMT Execution Model

In this section we formalize SIMT execution model. Our formalization is based on Habermaier and Knapp [14], but there are some differences. First, we omit `break`, function calls, and `return`. Second, we include arrays, which is almost always used in CUDA programs, and barrier synchronization.

In the semantics formalized here, the execution is in complete lockstep, but the actual GPU program is not necessarily executed in this manner. Possible approaches to filling this gap will be discussed in Section 7.

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 as 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 follows:

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

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

Expressions include special constants `tid`, thread identifier, and `ntid`, the number of threads[4]. 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, typically used to avoid data races in CUDA. Although in our formalization a barrier does not play a significant role, we include it so that we can reason about deadlocks (sometimes called barrier divergence) caused by a barrier. The remaining constructs are the same as the usual while-language. Note that we do not have boolean expressions, so we use integer expressions for conditions of `if` and `while` statements, and regard any nonzero value as true.

[4] The name of this constant is taken from a special register in PTX [15]. In our formalization this is the same as the number of threads, although this is not always the case for PTX.

2.2 Operational Semantics

Next we define a formal semantics of SIMT. For simplicity, arrays are represented simply by total maps from tuples of integers to integers, so 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 try to write into the same shared variables simultaneously.

Below we fix a positive integer N which is the number of threads, and therefore is an interpretation of the constant `ntid`. We also assume 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 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:

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

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

$$\sigma[\![f(e_1, \ldots, e_n)]\!](i) = f(\sigma[\![e_1]\!](i), \ldots, \sigma[\![e_n]\!](i))$$

Notation 1. *For a state* σ*, we define* $\sigma[x \mapsto a]$ *to be 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, σ' are states. This relation means that "if P is executed with mask μ and initial state σ, then the resulting state is σ'."

Evaluation rules are listed in Figure 1. The rule E-Inactive means that, if there is no active thread, the execution has no effect. A barrier synchronization succeeds only if all threads are active (or no thread is active, in which case E-Inactive is applicable), hence the set of active thread should be $\mathbb{T}$ in the rule E-Sync. A synchronization does not change the state.

Nondeterministic behavior can arise from E-SAssign; there can be more than one choice of σ', in case of a data race. More precisely, by a data race here we mean a situation that 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

$$P, \emptyset, \sigma \Downarrow \sigma \text{ (E-INACTIVE)} \qquad \texttt{skip}, \mu, \sigma \Downarrow \sigma \text{ (E-SKIP)} \qquad \texttt{sync}, \mathbb{T}, \sigma \Downarrow \sigma \text{ (E-SYNC)}$$

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

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

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

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

$$\frac{P, \mu \cap \sigma \left[\!\left[e\right]\!\right], \sigma \Downarrow \sigma' \qquad \texttt{while } e \texttt{ do } P, \mu \cap \sigma \left[\!\left[e\right]\!\right], \sigma' \Downarrow \sigma''}{\texttt{while } e \texttt{ do } P, \mu, \sigma \Downarrow \sigma''} \text{ (E-WHILE)}$$

Fig. 1. Operational semantics of SIMT programs.

(formally, $\sigma \left[\!\left[\bar{e}\right]\!\right](i) = \sigma \left[\!\left[\bar{e}\right]\!\right](j)$ and $\sigma \left[\!\left[e\right]\!\right](i) \neq \sigma \left[\!\left[e\right]\!\right](j)$). In such a case, following Habermaier and Knapp [14], we allow to choose either $\sigma \left[\!\left[e\right]\!\right](i)$ or $\sigma \left[\!\left[e\right]\!\right](j)$, and set its value to $x\texttt{[}\bar{e}\texttt{]}$. As discussed in Section 5.1, it is possible to define a semantics which raises an error in such cases.

3 Reasoning about SIMT Programs

In this section we describe how to extend Hoare Logic to the SIMT setting 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 as follows:

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

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 contain N (the number of threads) as a constant symbol, and each operation $f \in Op_n$ as an n-ary function symbol.

Extra 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, so has to receive one extra argument as thread identifier to determine its value. The first argument of a local variable 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 $\texttt{ntid}$ 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*, ranged over by ρ, is a map which 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.

By abuse of notation we write $P, \mu, \rho \Downarrow \rho'$ if and only if there exists σ' such that $P, \mu, \sigma \Downarrow \sigma'$, where σ is the restriction of ρ and ρ' equals σ' on program variables and ρ on specification variables. We also use the notation $\rho[\![e]\!]$ for the set $\{i \in \mathbb{T} \mid \rho[\![e]\!](i) \neq 0\}$.

Definition 2. *A* Hoare quadruple *is of the form* $\{\varphi\}\, m \mid P\, \{\psi\}$*, where* P *is a program,* m *is a term built from specification variables, and* φ *and* ψ *are formulas. Note that no variable occurring in* m *occurs in* P*.*

Definition 3. *A Hoare quadruple* $\{\varphi\}\, m \mid P\, \{\psi\}$ *is* valid *if, for every assignment* ρ *satisfying* φ *and every* ρ' *such that* $P, \rho[\![m]\!], \rho \Downarrow \rho'$*, it holds that* $\rho' \models \psi$*.*

Precisely speaking we have to distinguish states σ and assignments ρ but for brevity we will not distinguish them, if no confusion arises.

Definition 4. *For an expression* e *and a term* t*, we define a term* $e@t$ *as follows:*

$$\texttt{tid}@t = t \qquad \texttt{ntid}@t = N$$

$$(x\,[\![e_1, \ldots, e_n]\!])@t = \begin{cases} x(t, e_1@t, \ldots, e_n@t) & \text{if } x \text{ is local} \\ x(e_1@t, \ldots, e_n@t) & \text{if } x \text{ is shared} \end{cases}$$

$$(f(e_1, \ldots, e_n))@t = f(e_1@t, \ldots, e_n@t)$$

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

Notation 2. *We occasionally use* $\mathbb{T}$ *in place of* m *when* m *is an expression always nonzero in all threads (*1*, for example).*

Definition 5. *We use the following abbreviations.*

- $all(e) := (\forall i.0 \leq i < N \to e@i \neq 0)$
- $none(e) := (\forall i.0 \leq i < N \to e@i = 0)$

$$\{\varphi\}\, m \mid \texttt{skip}\, \{\varphi\} \qquad \text{(H-Skip)}$$

$$\{all(m) \vee none(m) \to \varphi\}\, m \mid \texttt{sync}\, \{\varphi\} \qquad \text{(H-Sync)}$$

$$\frac{\models \varphi' \to \varphi \qquad \{\varphi\}\, m \mid P\, \{\psi\} \qquad \models \psi \to \psi'}{\{\varphi'\}\, m \mid P\, \{\psi'\}} \qquad \text{(H-Conseq)}$$

$$\frac{\{\varphi\}\, m \mid P\, \{\psi\} \qquad \{\psi\}\, m \mid Q\, \{\chi\}}{\{\varphi\}\, m \mid P;\ Q\, \{\chi\}} \qquad \text{(H-Seq)}$$

$$\{\forall x'.\, assign(x', m, x, \bar{e}, e) \to \varphi[x'/x]\}\, m \mid x\,\texttt{[}\bar{e}\texttt{]}\ \texttt{:=}\ e\, \{\varphi\} \qquad \text{(H-Assign)}$$

$$\frac{\{\varphi \wedge e = z\}\, m\ \texttt{\&\&}\ z \mid P\, \{\psi\} \qquad \{\psi\}\, m\ \texttt{\&\&}\ \texttt{!}\ z \mid Q\, \{\chi\}}{\{\varphi\}\, m \mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q\, \{\chi\}} \qquad \text{(H-If)}$$

$$\frac{\{\varphi \wedge e = z\}\, m\ \texttt{\&\&}\ z \mid P\, \{\varphi\}}{\{\varphi\}\, m \mid \texttt{while}\ e\ \texttt{do}\ P\, \{\varphi \wedge none(m\ \texttt{\&\&}\ e)\}} \qquad \text{(H-While)}$$

Fig. 2. Inference rules.

- $i \in m := (m@i \neq 0)$
- $\forall i \in m.\varphi := (\forall i.0 \leq i < N \to m@i \neq 0 \to \varphi)$. *Similarly for* $\exists$ *and other variants.*
- *If* x *is a shared variable,* $assign(x', m, x, \bar{e}, e)$ *is defined to be*

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

and if x *is local,*

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

The last one of the definitions above would require some explanation. Intuitively, $assign(x', m, x, \bar{e}, e)$ 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 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 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 1. $x\,\texttt{[}\bar{e}\texttt{]}\ \texttt{:=}\ e, \sigma\, [\![m]\!]\, , \sigma \Downarrow \sigma'$ *holds if and only if there exists a such that* $\sigma' = \sigma[x \mapsto a]$*, and* $\sigma[x' \mapsto a] \models assign(x', m, x, \bar{e}, e)$*.*

3.2 Inference Rules

Inference rules are listed in Figure 2. We write $\vdash \{\varphi\}\, m \mid P\, \{\psi\}$ if the quadruple $\{\varphi\}\, m \mid P\, \{\psi\}$ is provable from the rules in Figure 2. The variables x' in

H-Assign and z in H-If and H-While are fresh specification 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 looks different from the standard assignment rule of Hoare Logic, but in view of Lemma 1 this would be natural. H-Sync is also understood in a similar way.

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 1 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 1. We introduce a fresh variable z in rules H-If and H-While. To see that this is indeed necessary, suppose the rule were of the following form (although this is actually ill-formed because the mask part contain a program variable).

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

Let x and y be shared variables and $e = (x \texttt{ > } 0)$, $P = (x \texttt{ := } 0;\ \ y \texttt{ := } 1)$, and $Q = \texttt{skip}$. Then the following is valid:

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

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

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

but this is impossible because the verification condition would be

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

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 `:= 1`, the actual mask is represented by x `> 0`, whereas in the above verification condition it is incorrectly replaced by x' `> 0`. This does not happen in the actual rule H-If because instead of directly evaluating e the value of e at the point of the execution branch is referenced through a fresh variable z.

3.3 Examples

Vector addition. Let us consider the program having 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 above program, the following holds:

$$\{\}\,\mathbb{T} \mid P\,\{\forall i.0 \le i < n \to c(i) = a(i) + b(i)\}\,.$$

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 following loop invariant:

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

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

Array sum. For simplicity we assume the number of threads N 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;
}
```

After executing this program 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 of equal lengths (s in the program), say a_1 and a_2. Then, compute the sum $a_1 + a_2$, and store the result into a_1. Continue this process until the length of the array becomes 1. The final value of 0-th element is the answer.

The following is an invariant:

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

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

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

4 Soundness and Relative Completeness

We are going to prove soundness and relative completeness. Unfortunately, however, they do *not* hold for all programs. We first describe how soundness fails, and introduce the notion of regular programs, being based on this observation. After that we prove soundness and relative completeness for regular programs.

4.1 Regular Programs

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

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

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:

$$\{\varphi \wedge z = e\}\, z \mid x\texttt{[0]} := 1;\ x\texttt{[1]} := 1\, \{\varphi\},$$

and therefore we can prove $\{\varphi\}\, \mathbb{T} \mid P\, \{\varphi \wedge \mathit{none}(e)\}$. 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 $\varphi \wedge \mathit{none}(e)$. However, this formula is inconsistent, so this is a contradiction. It follows that the rule H-WHILE is not sound for this example.

The problem is that initially the condition e is false in 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.

Actually, this is the only obstacle to proving soundness and relative completeness. We will 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 by executing P with i being disabled. More precisely:

Definition 6. *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 would not arise during the execution of the loop $\texttt{while}\, e\, \texttt{do}\, P$. Formally this is stated as follows:

Lemma 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 7. *Let us say a loop $\texttt{while}\, e\, \texttt{do}\, P$ is* regular *if e is stable under P. A program is said to be regular if any `while`-loop contained in it is regular.*

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

Lemma 3. *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 done by induction on the derivation of $P, \mu, \sigma \Downarrow \sigma'$.

Lemma 4. *Let P be a program, and assume that for any subprogram of the form* `while` e `do` Q*, e and Q satisfy the condition of Lemma 3. Then P is regular.*

Below we consider regular programs. However, this is not actually a problem because it is possible to transform a program into a regular one, which is equivalent (in the sense that if they are executed under the same state with the same mask, then the set of resulting states are also the same).

To do this, given a program, replace its subprograms of the form `while` e `do` P with z `:=` e; `while` z `do` $(P;\ z$ `:=` $e)$, where z is a fresh local variable. The program obtained by this transformation satisfies the condition of Lemma 4.

4.2 Soundness and Relative Completeness for Regular Programs

After restricting our attention to regular programs, 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 Section A.2.

Theorem 1 (Soundness). *If P is a regular program and $\{\varphi\}\, m \mid P\, \{\psi\}$ is derivable from the rules in Figure 2, then it is valid.*

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

Definition 8 (Weakest Liberal Precondition). *The weakest liberal precondition $wlp(m, P, \varphi)$ is defined as follows:*

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

If this set is definable in the assertion language, we also use $wlp(m, P, \varphi)$ to denote a formula defining this set.

To prove the relative completeness, by the standard argument it suffices to show that $\vdash \{wlp(m, P, \varphi)\}\, m \mid P\, \{\varphi\}$. We can prove this by induction on P. When P is a `while`-statement, we can use the formula $\exists z.e = z \wedge wlp(m \texttt{ \&\& } z, P, \varphi)$ as an invariant. For the details, see Section A.3.

Theorem 2 (Relative Completeness). *Suppose that the weakest liberal preconditions are definable in the assertion language. If P is a regular program and $\{\varphi\}\, m \mid P\, \{\psi\}$ is valid, then it is derivable.*

5 Extensions

In GPU programs, there are two kinds of errors that are intensively studied: data race and barrier divergence. In the above development we did not consider these errors explicitly. Below we discuss how our framework can be modified to detect these errors.

5.1 A Variant of the Assignment Rule

In rule E-SAssign, conflicting writes on a shared variable result in a nondeterministic behavior. Although this is consistent with NVIDIA's specification [15], such a conflict is often unintended. Thus it would be useful to regard such a situation as an error, so that a Hoare Logic can detect this data race. One of such semantics has been considered by Betts et al. [3]. (Below, we limit our attention to a data race of this type, although other types of data races may arise when lockstep execution is not assumed.)

Let us consider the following variant of E-SAssign:

$$\frac{\begin{array}{c} x \text{ is shared} \qquad \sigma'(y) = \sigma(y) \text{ for each variable } y \neq x \\ (\forall i \in \mu.\sigma\,[\![\bar{e}]\!]\,(i) \neq \bar{n}) \implies \sigma'(x)(\bar{n}) = \sigma(x)(\bar{n}) \\ \forall i \in \mu.(\sigma\,[\![\bar{e}]\!]\,(i) = \bar{n} \implies \sigma'(x)(\bar{n}) = \sigma\,[\![e]\!]\,(i)) \end{array}}{x\,[\bar{e}]\,\texttt{:=}\,e, \mu, \sigma \Downarrow \sigma'} \qquad \text{(E-SAssign')}$$

If we employ this rule, the execution gets stuck when there are conflicting writes. Indeed, if $\sigma\,[\![\bar{e}]\!]\,(i) = \bar{n}$ holds for multiple i's, with $\sigma\,[\![e]\!]\,(i)$ being distinct, then no σ' satisfy the last line of the premises. In other words, the premises require that all values being written into a certain location must be the same.

If we replace E-SAssign with E-SAssign', then the definition of *assign* in H-Assign has to be modified accordingly. The definition would be as follows (here, we show the definition for shared variables):

$$\begin{aligned} \mathit{assign}'(x', m, x, \bar{e}, e) = \forall \bar{n}.\, ((\forall i \in m.\bar{e}@i \neq \bar{n}) \wedge x'(\bar{n}) = x(\bar{n})) \vee \\ (\forall i \in m.\bar{e}@i = \bar{n} \to x'(\bar{n}) = e@i)\,. \end{aligned}$$

If there exist distinct active threads i, j such that $\bar{e}@i = \bar{e}@j (= \bar{n})$ and $e@i \neq e@j$ (that is, if two threads are trying to write different values to the same location), then there does not exist x' satisfying this formula. For example, if x `:=tid` is executed with mask $\mathbb{T}$, then $\mathit{assign}'(x', \mathbb{T}, x, \cdot, \texttt{tid})$ implies $\forall i \in \mathbb{T}.x' = i$ which is a contradiction (unless $N = 1$).

5.2 Treatment of Erroneous Situations

In the proof rules considered above (including E-SAssign' above), any postcondition can be proved if a program gets stuck. It may be desirable if the rules prevent us from proving such a consequence when a program may get stuck.

To handle such a situation explicitly, we can introduce a special state representing an error, denoted by $\bot$. We extend $\models$ so that $\bot$ do not satisfy any specification; in other words, $\bot \not\models \varphi$ for all φ.

Consider the following rule, which treats a data race as an error.

$$\frac{x \text{ is shared} \qquad i, j \in \mu \qquad \sigma\,[\![\bar{e}]\!]\,(i) = \sigma\,[\![\bar{e}]\!]\,(j) \qquad \sigma\,[\![e]\!]\,(i) \neq \sigma\,[\![e]\!]\,(j)}{x\,\texttt{[}\bar{e}\texttt{]}\ \texttt{:=}\ e, \mu, \sigma \Downarrow \bot} \qquad \text{(E-SAssignRace)}$$

The following axiom would replace the original H-Assign.

$$\{\exists x'.\, assign'(x', m, x, \bar{e}, e) \wedge \varphi[x'/x]\}\, m \mid x\,\texttt{[}\bar{e}\texttt{]}\ \texttt{:=}\ e\,\{\varphi\}$$

Here we use $assign'$ defined in Section 5.1. The precondition of this rule requires that there exists a result of the assignment, so if a program causes a data race, then the precondition becomes inconsistent. Therefore this rule prevents us from proving $\{\varphi\}\, m \mid x\,\texttt{[}\bar{e}\texttt{]}\ \texttt{:=}\ e\,\{\psi\}$, whenever there can be a data race, without φ being inconsistent. Also note that replacing $\forall x'$ in H-Assign with $\exists x'$ does not cause a problem, because E-SAssignRace excludes nondeterminism (there is at most one x' that has to be considered).

Similarly we can treat a so-called barrier divergence (a failure of synchronization) by modifying H-Sync. In the original rule H-Sync, similarly to H-Assign, the precondition is vacuously true for any state σ and a mask m such that $\sigma\,[\![m]\!] \neq \emptyset, \mathbb{T}$ (that is, a barrier divergence).

We add the following evaluation rule

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

and replace H-Sync with

$$\{(all(m) \vee none(m)) \wedge \varphi\}\, m \mid \texttt{sync}\,\{\varphi\}\,.$$

Then, we can prove $\{\varphi\}\, m \mid \texttt{sync}\,\{\psi\}$ only if φ implies $all(m) \vee none(m)$.

6 Related Work

Semantics of GPU programs. Habermaier and Knapp [14] formalized both SIMT and interleaved multi-thread semantics, and discussed relationships between them. In particular, they proved that their SIMT semantics can be simulated by the interleaved semantics with an appropriate scheduling. Collingbourne et al. considered a lockstep execution of an unstructured programs based on control-flow graph [4]. They defined both interleaving and lockstep semantics, and proved that two semantics are equivalent in a certain sense under the assumption of race-freedom and termination. Betts et al. [3] defined another semantics, called synchronous, delayed visibility (SDV) semantics. The major difference from ours is that, in SDV semantics, a conflicting write results in an error, while in our semantics it is not. They developed a verification tool GPUVerify that detects race condition and barrier divergence, based on their SDV semantics.

Verification tools. Verification tools for GPU programs are developed by several authors. Tripakis, Stergiou and Lublinerman developed a method to check determinism and equivalence of SPMD programs based on non-interference [16].

Collingbourne, Cadar and Kelly proposed a method of symbolic execution of SIMD programs based on KLEE symbolic execution tool [5, 6]. Li and Gopalakrishnan developed an SMT-based verification tools PUG [7] and PUG$_{para}$ [8]. They first transform a CUDA program into a first-order formula, and detect assertion failures, barrier divergence and date races by using an SMT solver. Li et al. developed a concolic verification and test generation tool for GPU programs, called GKLEE [9]. Further optimizations and extensions of GKLEE are also considered [10, 11].

Deductive verification. Huisman and Mihelčić suggest permission-based separation logic [12] for deductive verification of GPU programs. They demonstrated how they can verify race-freedom and functional correctness by separation logic. They consider an assignment of resources to threads, and use it to prove race-freedom. As discussed in Section 5, our approach can also be used to detect data races, although we did not consider explicit resource assignments. This is because in our language there is no pointers, and no two arrays can ovarlap. Moreover, in our semantics the execution is in lockstep, which implies that there is no data races between different instructions. Under these assumptions, absence of data races can be expressed without introducing a new construct like points-to relation in separation logic. Since Huisman and Mihelčić do not assume lockstep execution, the same method would not apply to their setting.

7 Conclusions and Future Work

We have formalized the SIMT execution model for while-language extended with arrays and SIMT constructs, and defined a Hoare Logic for this language. We also proved that the inference rules are sound and relatively complete for a regular program. This restriction is, as discussed above, not significant, because it is always possible to transform a given program into a regular one without changing the meaning of the program.

In our semantics, each program is executed in all threads in complete lockstep. However, actual execution on GPUs do not necessarily proceed in such a way. For example, CUDA has a thread hierarchy in its programming model, and the execution of threads may be interleaved [2]. One possible future direction would be to extend our framework so that it can handle this thread hierarchy.

However, there is another possible approach to fill the gap. Even if the actual thread execution is interleaved, if we restrict our attention to programs that are scheduling independent (that is, programs that produce the same result regardless of which scheduling is selected), it would be sound to assume that programs are executed in complete lockstep. So under such an assumption, our method can be applied to a more realistic programs such as CUDA without significant changes. Since, as far as we know, many GPU programs are intended to be scheduling independent, a detailed investigation is left for future work.

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

References

1. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26**(1) (2007) 80–113
2. NVIDIA: NVIDIA CUDA C Programming Guide. (2012)
3. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA '12, New York, NY, USA, ACM (2012) 113–132
4. Collingbourne, P., Donaldson, A.F., Ketema, J., Qadeer, S.: Interleaving and lock-step semantics for analysis and verification of GPU kernels. In: Proc. of European Symposium on Programming (ESOP'13). Volume 7792 of LNCS., Springer Verlag (2013) 270–289
5. Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic crosschecking of floating-point and SIMD code. In: Proc. of the sixth conference on Computer systems. EuroSys '11, New York, NY, USA, ACM (2011) 315–328
6. Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic testing of OpenCL code. In Eder, K., Lourenço, J.a., Shehory, O., eds.: Proc. of Hardware and Software: Verification and Testing. Volume 7261 of LNCS., Springer Verlag (2012) 203–218
7. Li, G., Gopalakrishnan, G.: 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
8. Li, G., Gopalakrishnan, G.: Parameterized verification of GPU kernel programs. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IEEE (May 2012) 2450–2459
9. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: 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, New York, NY, USA, ACM (2012) 215–224
10. Li, P., Li, G., Gopalakrishnan, G.: 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 (2012) 29:1–29:10
11. Chiang, W.F., Gopalakrishnan, G., Li, G., Rakamarić, Z.: Formal analysis of GPU programs with atomics via conflict-directed delay-bounding. In: Proc. of the 5th NASA Formal Methods Symposium (NFM 2013). Volume 7871 of LNCS., Springer Verlag (2013) 213–228
12. Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs using permission-based separation logic. Bytecode 2013, 8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation, available as `http://hgpu.org/?p=9099` (2013)
13. Apt, K.R., de Boer, F., Olderog, E.R.: Verification of Sequential and Concurrent Programs. 3rd edn. Springer Publishing Company, Incorporated (2009)
14. Habermaier, A., Knapp, A.: On the correctness of the SIMT execution model of GPUs. In: Proc. of European Symposium on Programming (ESOP'12). Volume 7211 of LNCS., Springer Verlag (2012) 316–335
15. NVIDIA: Parallel Thread Execution ISA Version 3.1. (2012)
16. Tripakis, S., Stergiou, C., Lublinerman, R.: Checking equivalence of SPMD programs using non-interference. Technical Report UCB/EECS-2010-11, EECS Department, University of California, Berkeley (Jan 2010)

A Proofs of Theorems

A.1 Auxiliary Lemmas

First, we list some basic lemmas for later use.

Lemma 5. *The following holds for any state σ.*

- $\sigma \models \mathit{all}(e) \iff \sigma[\![e]\!] = \mathbb{T}$
- $\sigma \models \mathit{none}(e) \iff \sigma[\![e]\!] = \emptyset$

Lemma 6. *If x' is a variable not occurring in φ, then*

$$\sigma[x' \mapsto a] \models \varphi[x'/x] \iff \sigma[x \mapsto a] \models \varphi.$$

Lemma 7. *Suppose that m consists of specification variables. Then,*

$$P, \sigma[\![m]\!], \sigma \Downarrow \sigma' \implies \sigma[\![m]\!] = \sigma'[\![m]\!].$$

Lemma 8. *If $P, \emptyset, \sigma \Downarrow \sigma'$, then $\sigma = \sigma'$.*

Lemma 9. *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 $W, \mu', \sigma \Downarrow \sigma'$.*

A.2 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 $\sigma \not\models \mathit{all}(m) \vee \mathit{none}(m)$.

To show that H-Assign is sound, suppose $x\,\texttt{[}\bar{e}\texttt{]}\ \texttt{:=}\ e, \sigma[\![m]\!], \sigma \Downarrow \sigma'$. From Lemma 1, σ' is of the form $\sigma[x \mapsto a]$ where a satisfies $\sigma[x' \mapsto a] \models \mathit{assign}(x', m, x, \bar{e}, e)$. So if we have $\sigma \models \forall x'.\mathit{assign}(x', m, x, \bar{e}, e) \to \varphi[x'/x]$, then $\sigma[x' \mapsto a] \models \varphi[x'/x]$. By using Lemma 6, 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 $Q, \sigma_0'[\![m\ \texttt{\&\&}\ \texttt{!}\ z]\!], \sigma_0' \Downarrow \sigma_0''$. 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 $\{\varphi \wedge e = z\}\ m\ \texttt{\&\&}\ z\ \mid$

$P\{\varphi\}$ 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 $\sigma' \models \varphi \wedge \mathit{none}(m \,\texttt{\&\&}\, e)$.

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

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

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

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 9 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 9 is indeed satisfied: the regularity and Lemma 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 $\sigma_2 \models \varphi \wedge \mathit{none}(m \,\texttt{\&\&}\, e)$. Since the antecedent is already proved, we have $\sigma_2 \models \varphi \wedge \mathit{none}(m \,\texttt{\&\&}\, e)$. Moreover, z does not occur in φ, m nor e, which this implies $\sigma' \models \varphi \wedge \mathit{none}(m \,\texttt{\&\&}\, e)$. This completes the proof.

A.3 Proof of Relative Completeness

By the standard argument, it suffices to show that

$$\vdash \{\mathit{wlp}(m, P, \varphi)\}\, m \mid P\, \{\varphi\}\,.$$

We proceed by induction on P

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

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

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

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

So it suffices to show that

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

Suppose $\sigma \models wlp(m, x\,\texttt{[}\bar{e}\texttt{]}\ \texttt{:=}\ e, \varphi)$ and $\sigma[x' \mapsto a] \models assign(x', m, x, \bar{e}, e)$. Then from Lemma 1, 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 6 we obtain $\sigma[x' \mapsto a] \models \varphi[x'/x]$.

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

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

So it suffices to show that

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

Suppose $\sigma \models wlp(m, P_1;\ P_2, \varphi)$, and consider σ' such that $P_1, \sigma\,[\![m]\!], \sigma \Downarrow \sigma'$. We have to show that $\sigma' \models wlp(m, P_2, \varphi)$, 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 $\chi = wlp(m\ \texttt{\&\&}\ z, P_1, wlp(m\ \texttt{\&\&}\ \texttt{!}\ z, P_2, \varphi))$. Then by the induction hypotheses we have

$$\vdash \{\chi\}\ m\ \texttt{\&\&}\ z \mid P_1\, \{wlp(m\ \texttt{\&\&}\ \texttt{!}\ z, P_2, \varphi)\}\,,$$
$$\vdash \{wlp(m\ \texttt{\&\&}\ \texttt{!}\ z, P_2, \varphi)\}\ m\ \texttt{\&\&}\ \texttt{!}\ z \mid P_2\, \{\varphi\}\,.$$

Since

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

we have

$$\vdash \{(\exists z.e = z \wedge \chi) \wedge e = z\}\ m\ \texttt{\&\&}\ z \mid P_1\, \{\varphi\}\,.$$

Therefore, by H-If,

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

So our goal is to prove

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

Suppose $\sigma \models wlp(\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2, m, \varphi)$, 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

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

Then, since z and variables in m are specification variables, we also have

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

By E-If and the equality $\sigma_0(z) = \sigma[\![e]\!]$ we have

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

On the other hand, we assumed that $\sigma \models \mathit{wlp}(\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2, m, \varphi)$ 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 $\psi = \exists z.e = z \wedge \mathit{wlp}(m\ \texttt{\&\&}\ z, P, \varphi)$. We prove

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

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

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

which is, by definition of ψ, equivalent to

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

So 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 easily follows that $\sigma \models \mathit{wlp}(m\ \texttt{\&\&}\ z, P, \varphi)$, so $\sigma_0'' \models \varphi$ follows from

$$P, \sigma[\![m\ \texttt{\&\&}\ z]\!], \sigma \Downarrow \sigma_0''.$$

To prove this, we first show that

$$Q, \sigma[\![m\ \texttt{\&\&}\ z]\!] \cap \sigma[\![e]\!], \sigma \Downarrow \sigma', \quad P, \sigma[\![m\ \texttt{\&\&}\ z]\!] \cap \sigma[\![e]\!], \sigma' \Downarrow \sigma''.$$

and then apply E-While. The former follows from $\sigma[\![m\ \texttt{\&\&}\ z]\!] \cap \sigma[\![e]\!] = \sigma[\![m\ \texttt{\&\&}\ z]\!]$, which is a consequence of the assumption $\sigma \models e = z$. For the latter, note that

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

holds from assumption and the fact that z is fresh. In view of Lemma 9, it suffices to show that

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

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 regularity and Lemma 2. This completes the proof of (1).

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

Finally we prove (3). Suppose $\sigma \models wlp(m, P, \varphi)$, and let $\sigma_0 = \sigma[z \mapsto \sigma [\![e]\!]]$. Then clearly $\sigma_0 \models e = z$. We will prove $\sigma_0 \models wlp(m \texttt{ \&\& } z, P, \varphi)$. To do this suppose $P, \sigma_0 [\![m \texttt{ \&\& } z]\!], \sigma_0 \Downarrow \sigma_0'$. Then, since $\sigma_0 [\![m \texttt{ \&\& } z]\!] = \sigma [\![m]\!] \cap \sigma [\![e]\!]$, by Lemma 9 we have $P, \sigma [\![m]\!], \sigma_0 \Downarrow \sigma_0'$. Since $wlp(m, P, \varphi)$ does not depend on z and σ satisfies this formula, so does σ_0. Therefore $\sigma_0' \models \varphi$.