Certified Reasoning in Memory Hierarchies

Gilles Barthe, César Kunz, and Jorge Luis Sacchini

INRIA Sophia Antipolis-Méditerranée
{Gilles.Barthe,Cesar.Kunz,Jorge-Luis.Sacchini}@inria.fr

Abstract. Parallel programming is rapidly gaining importance as a vector to develop high performance applications that exploit the improved capabilities of modern computer architectures. In consequence, there is a need to develop analysis and verification methods for parallel programs. Sequoia is a language designed to program parallel divide-and-conquer programs over a hierarchical, tree-structured, and explicitly managed memory. Using abstract interpretation, we develop a compositional proof system to analyze Sequoia programs and reason about them. Then, we show that common program optimizations transform provably correct Sequoia programs into provably correct Sequoia programs, provided the specification and the proof of the original program are strengthened by certifying analyzers, an extension of program analyzers that produce a derivation that the results of the analysis are correct.

1 Introduction

As modern computer architectures increasingly offer support for high performance parallel programming, there is a quest to invent adequate programming languages that exploit their capabilities. In order to reflect these new architectures accurately, parallel programming languages are gradually abandoning the traditional memory model, in which memory is viewed as a flat and uniform structure, in favor of a hierarchical memory model, which considers a tree of memories with different bandwidth and latency characteristics. Hierarchical memory is particularly appropriate for divide-and-conquer applications, in which computations are repeatedly fragmented into smaller computations that will be executed lower in the memory hierarchy, and programming languages based on this model perform well for such applications.

As hierarchical memories are gaining broader acceptance [1, 7, 11], there is an interest in adapting formal methods to them, and in particular in developing both automated and interactive methods to prove properties of programs over hierarchical memories. To our best knowledge, there has not been any previous effort to develop program logics or a general theory of program analysis for such a language. The objective of this paper is precisely to provide methods to reason about Sequoia [8, 12, 10], a language designed to program efficient, portable, and reliable applications for the hierarchical memory.

We indicate in red the modifications with respect to the submitted version.

In the first part of the paper, we use the framework of abstract interpretation [5, 6] to develop a compositional proof system to reason about Sequoia programs (Sect. 3). The compositional proof system is proved sound w.r.t. safe programs; our notion of safety enforces that independent subtasks of a program manipulate distinct parts of the memory, and is very similar to the independence notion of strict and-parallelism in the domain of logic programming [9]. In addition, we define a sound and decidable analysis for safety (Sect. 3.2).

In the second part of the paper, we focus on the interplay between program optimization and program verification. To maximize the performance of applications, the Sequoia compiler aggressively performs program optimizations such as code hoisting, instruction scheduling, and SPMD distribution. We show, for common optimizations described in [12], that program optimizations transform provably correct programs into provably correct programs (Sect. 4). More precisely, we provide an algorithm to transform a derivation for the original program into a derivation for the transformed program. For some optimizations, the algorithm relies on a certifying analyzer, that produces a derivation of the correctness of its results. These results find applications in proof carrying code [14], in the context of certificate translation [2, 3].

2 A primer on Sequoia

Sequoia [8, 12, 10] is a language for developing portable and efficient parallel programs for the memory hierarchy. It is based on a small set of constructs that control essential aspects of programming over the memory hierarchy, such as communication, memory movement and computation. Computations are organized into self-contained units, called tasks. Tasks at the same level execute in parallel on a dedicated address space, and may rely on subtasks for performing computations; in this case, each subtask will operate on a smaller (and in practice faster) fragment of the memory.

Hierarchical memory. A hierarchical memory is a tree of memory modules, i.e. of partial functions from a set $\mathcal{L}$ of locations to a set $\mathcal{V}$ of values. In our setting, values are either integers or booleans: the set $\mathcal{V}$ of values is defined as the union of the set $\mathbb{Z}$ of integers and $\mathbb{B}$ of booleans. Besides, locations are either scalar variables, or arrays indices of the form $A[i]$ where A is an array and i is an index: the set $\mathcal{L}$ of locations is defined as the union of the set of identifiers for scalars or array indices. The set of scalar variables is denoted by $\mathcal{N}_{\mathcal{S}}$ and the set of array variables is denoted by $\mathcal{N}_{\mathcal{A}}$. The set of variable names is $\mathcal{N} = \mathcal{N}_{\mathcal{S}} \cup \mathcal{N}_{\mathcal{A}}$.

Definition 1 (States). *The set $\mathcal{M} = \mathcal{L} \rightharpoonup \mathcal{V}$ of memories is defined as the set of partial functions from locations to values. A memory hierarchy representing the machine structure is a memory tree defined as:*

$$\mathcal{T} ::= \langle \mu, \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle \quad k \geq 0,$$

where $\mu \in \mathcal{M}$.

Intuitively, $\langle \mu, \vec{\tau} \rangle \in \mathcal{T}$ represents a memory tree with root memory μ and a possible empty sequence $\vec{\tau}$ of child subtrees. The semantics of programs is given using an operator, $+_\mu : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, indexed by a memory $\mu \in \mathcal{M}$, such that:

$$(\mu_1 +_\mu \mu_2)x = \begin{cases} \mu_1 x & \text{if } \mu_2 x = \mu x \\ \mu_2 x & \text{if } \mu_1 x = \mu x \\ v & \text{where } v \in \{\mu_1 x, \mu_2 x\} \end{cases}$$

Note that the operator $+_\mu$ is not deterministic if both μ_1 and μ_2 modify the same variable. The operator $+_\mu$ is generalized over memory hierarchies in $\mathcal{T}$ and sequences $\vec{\mu} \in \mathcal{M}^\star$. By convention, we set:

$$\textstyle\sum^{\mu}_{m \le i \le n} \mu_i = \begin{cases} \mu & \text{if } n < m \\ \mu_m & \text{if } n = m \\ \mu_m +_\mu \sum^{\mu}_{m+1 \le i \le n} & \text{otherwise} \end{cases}$$

Intuitively, μ denotes a memory in a hierarchical memory, corresponding to a task G to be executed, where G is divided in parallel subtasks $G_1, \ldots, G_n$. Executing each subtask G_i in the initial memory μ returns μ_i as final memory. Since subtasks are intended to operate on pairwise disjoint fragments of the memory μ, it should be the case that, upon termination of the subtasks, the memories $\mu_1 \ldots \mu_n$ are such that $\sum^{\mu}_{m \le i \le n} \mu_i$ is defined.

Syntax. Sequoia features usual constructions as well as specific constructs for parallel execution, for spawning new subtasks, and for grouping computations.

Definition 2 (Sequoia Programs). *The set of programs is defined by the following grammar:*

$$\begin{aligned} G ::= {} & \mathsf{Copy}^{\uparrow}(\vec{A}, \vec{B}) \mid \mathsf{Copy}^{\downarrow}(\vec{A}, \vec{B}) \mid \mathsf{Copy}(\vec{A}, \vec{B}) \\ \mid {} & \mathsf{Kernel}\langle A = f(B_1, \ldots, B_n) \rangle \mid \mathsf{Scalar}\langle a = f(b_1, \ldots, b_n) \rangle \\ \mid {} & \mathsf{Forall}\ i = m : n\ \mathsf{do}\ G \mid \mathsf{Group}(H) \mid \mathsf{Exec}_i(G) \\ \mid {} & \mathsf{If}\ b\ \mathsf{then}\ G_1\ \mathsf{else}\ G_2 \end{aligned}$$

where a, b are scalar variables, m, n are scalar constants, $\vec{A}, \vec{B}$ are array variables and H is a dependence graph of programs.

Atomic statements, i.e. Copy, Kernel, and Scalar statements, are given a specific treatment in the proof system; we let $\mathsf{atomStmt}$ denote the set of atomic statements. A program G in the dependence graph H is maximal if G does not depend on any other program in H.

Semantics. The semantics of a program G is defined by a judgment of the form $\sigma \vdash G \to \sigma'$ where $\sigma, \sigma' \in \mathcal{H}$, and $\mathcal{H} = \mathcal{M} \times \mathcal{T}$. Every $\sigma \in \mathcal{H}$ is a pair $\langle \mu_p, \tau \rangle$ where μ_p is the parent memory and τ is a child memory hierarchy. The meaning of such a judgment is that the evaluation of G with initial memory σ terminates with final memory σ'. To work with memory hierarchies, we define two functions: $\pi_i : \mathcal{H} \to \mathcal{H}$ that returns the i-th child of a memory, and $\oplus_i : \mathcal{H} \times \mathcal{H} \to \mathcal{H}$ that, given two memories σ_1 and σ_2, replaces the i-th child of σ_1 with σ_2. Formally, they are defined as $\pi_i(\mu_p, \langle \mu, \vec{\tau} \rangle) = (\mu, \tau_i)$ and $(\mu_p, \langle \mu, \vec{\tau} \rangle) \oplus_i (\mu', \tau') = (\mu_p, \langle \mu', \vec{\tau_1} \rangle)$, where $\tau_{1i} = \tau'$ and $\tau_{1j} = \tau_j$ for $j \neq i$.

$$\frac{}{\mu_p, \langle \mu, \vec{\tau} \rangle \vdash \mathsf{Copy}^{\uparrow}(\vec{A}, \vec{B}) \rightarrow\ \mu_p[B \mapsto \mu(A)], \langle \mu, \vec{\tau} \rangle}$$

$$\frac{}{\mu_p, \langle \mu, \vec{\tau} \rangle \vdash \mathsf{Copy}^{\downarrow}(\vec{A}, \vec{B}) \rightarrow\ \mu_p, \langle \mu[B \mapsto \mu_p(A)], \vec{\tau} \rangle}$$

$$\frac{}{\mu_p, \langle \mu, \vec{\tau} \rangle \vdash \mathsf{Kernel}\langle A = f(B_1, \dots, B_n) \rangle \rightarrow\ \mu_p, \langle \mu[A \mapsto f(B_1, \dots, B_n)], \vec{\tau} \rangle}$$

$$\frac{}{\mu_p, \langle \mu, \vec{\tau} \rangle \vdash \mathsf{Scalar}\langle a = f(b_1, \dots, b_n) \rangle \rightarrow\ \mu_p, \langle \mu[a \mapsto f(b_1, \dots, b_n)], \vec{\tau} \rangle}$$

$$\frac{\begin{array}{c} X \text{ the set of maximal elements of } H \text{ and } H' = H \setminus X \\ \forall g \in X, \mu, \tau \vdash g \rightarrow\ \mu_g, \tau_g \\ \sum_{g \in X} (\mu_g, \tau_g) \vdash \mathsf{Group}(H') \rightarrow\ \mu', \tau' \end{array}}{\mu, \tau \vdash \mathsf{Group}(H) \rightarrow\ \mu', \tau'}$$

$$\frac{\forall j \in [m, n] \neq \emptyset.\ \mu_p, \langle \mu[i \mapsto j], \vec{\tau} \rangle \vdash G \rightarrow\ \mu_p^j, \langle \mu^j, \vec{\tau}^j \rangle}{\mu_p, \langle \mu, \vec{\tau} \rangle \vdash \mathsf{Forall}\ i = m : n\ \mathsf{do}\ G \rightarrow\ \sum_{j=m}^{n} (\mu_p^j, \langle \mu^j[i \mapsto \mu(i)], \vec{\tau}^j \rangle)}$$

$$\frac{\pi_i(\mu, \tau) \vdash G \rightarrow\ \mu', \tau'}{\mu, \tau \vdash \mathsf{Exec}_i(G) \rightarrow\ (\mu, \tau) \oplus_i (\mu', \tau')}$$

Fig. 1. Sequoia program semantics (excerpt)

Definition 3 (Program semantics). *The semantics of a program G is defined by the rules given in Fig. 1.*

We briefly comment on the rules. The constructs $\mathsf{Copy}^{\uparrow}(\vec{A}, \vec{B})$, and $\mathsf{Copy}^{\downarrow}(\vec{A}, \vec{B})$, and $\mathsf{Copy}(\vec{A}, \vec{B})$ are primitives that enable data to migrate along the tree structure, from a child to its parent, or from the parent to a child, or locally. The constructs $\mathsf{Kernel}\langle A = f(B_1, \dots, B_n) \rangle$ and $\mathsf{Scalar}\langle a = f(b_1, \dots, b_n) \rangle$ execute bulk and scalar computations. We implicitly assume in the rules that array accesses are in-bound. If this condition is not met then there is no applicable rule, and the program is stuck.

The construct $\mathsf{Group}(H)$ executes the maximal elements X of the dependence graph in parallel, and then merges the result before recursively executing $\mathsf{Group}(H \setminus X)$. The semantics of $\mathsf{Group}(H)$ is deterministic if the elements in H manipulate distinct regions of the memory, since otherwise the final memory $\sum_{g \in X} (\mu_g, \tau_g)$ is not well-defined in the overlapping regions. A rule not shown in Fig. 1 states that if $H = \emptyset$ the memory hierarchy is left unchanged.

The construct $\mathsf{Forall}\ i = m : n\ \mathsf{do}\ G$ executes in parallel $n - m$ instances of G with a different value of i, and merges the result. As in the case of group execution, the semantics is deterministic only if iterations manipulate pairwise distinct parts of the memory. The rule in Fig. 1 considers exclusively the case where $m \leq n$, otherwise the memory hierarchy rests unchanged. Finally, the construction $\mathsf{Exec}_i(G)$ spawns a new computation on the i-th subtree of the current memory. We omit the standard semantics for conditional tasks.

The semantics of Fig. 1 is identical to the original semantics defined in [12], to the exception of the rule for $\mathsf{Group}(H)$, which only required X to be a subset of the maximal elements of H. However, both semantics are equivalent for the class of programs where subtasks are independent; that is, that modify different parts of the memory. We say the programs in this class are *safe*. In Section 3.2 we propose a sound analysis to check that a given program is safe, and show that both semantics coincide for safe programs.

We conclude this section by introducing useful notations. The extended set of scalar names, ${\mathcal{N}_\mathcal{S}}^+$, is defined as

$${\mathcal{N}_\mathcal{S}}^+ = \mathcal{N}_\mathcal{S} \cup \{x^{\uparrow} : x \in \mathcal{N}_\mathcal{S}\} \cup \{x^{\downarrow^{i_1} \cdots \downarrow^{i_k}} : x \in \mathcal{N}_\mathcal{S} \wedge k \geq 0 \wedge i_1, \ldots, i_k \in \mathbb{N}\}$$

We define, in a similar way, the sets ${\mathcal{N}_\mathcal{A}}^+$, $\mathcal{N}^+$, and $\mathcal{L}^+$ of extended locations. These sets allow us to refer to variables at all levels of a memory hierarchy.

Given $\sigma \in \mathcal{H}$, with $\sigma = \mu_p, \langle \mu, \tau \rangle$, and $l \in \mathcal{L}^+$, we define $\sigma(l)$ with the following rules:

$$\sigma(l) = \begin{cases} \mu_p(x) & \text{if } l = x^{\uparrow} \\ \mu(x) & \text{if } l = x \\ (\mu, \tau_{i_1})(x^{\downarrow^{i_2} \cdots \downarrow^{i_k}}) & \text{if } l = x^{\downarrow^{i_1} \downarrow^{i_2} \cdots \downarrow^{i_k}} \end{cases}$$

We also define the functions $\uparrow^i, \downarrow^i : {\mathcal{N}_\mathcal{S}}^+ \to {\mathcal{N}_\mathcal{S}}^+$ with the following rules:

$$\begin{array}{ll} \downarrow^i(x) = x^{\downarrow^i} & \uparrow^i(x) = x^{\uparrow} \\ \downarrow^i(x^{\downarrow^{j_1} \cdots \downarrow^{j_n}}) = x^{\downarrow^i \downarrow^{j_1} \cdots \downarrow^{j_n}} & \uparrow^i(x^{\downarrow^i \downarrow^{j_1} \cdots \downarrow^{j_k}}) = x^{\downarrow^{j_1} \cdots \downarrow^{j_k}} \\ \downarrow^i(x^{\uparrow}) = x & \end{array}$$

Note that $\downarrow^i$ is a total function, while $\uparrow^i$ is undefined in $x^{\uparrow}$ and $x^{\downarrow^j \downarrow^{j_1} \cdots \downarrow^{j_k}}$ if $j \neq i$. These functions are defined likewise for ${\mathcal{N}_\mathcal{A}}^+$, $\mathcal{N}^+$, and $\mathcal{L}^+$.

3 Analyzing and reasoning about Sequoia programs

This section presents a proof system for reasoning about Sequoia programs. We start by generalizing the basics of abstract interpretation to our setting, using a sound, compositional proof system. Then, we define as an instance of our setting a program analysis for safety, and show its soundness. Finally, we define a program logic as an instance of our proof system, and show its soundness for safe programs.

3.1 Program Analysis

We develop our work using a mild generalization of the framework of abstract interpretation, in which abstract elements form a pre-lattice.[1] We also have

[1] A pre-order over A is a reflexive and transitive binary relation, whereas a partial order is an anti-symmetric pre-order. We prefer to use pre-orders instead of partial orders because one instance of an abstract interpretation is that of propositions; we do not want to view it as a partial order since it implies that logically equivalent formulae are equal, which is not appropriate in the setting of Proof Carrying Code.

specific operators over the abstract domain for each type of program, as shown below.

Definition 4. *An abstract interpretation is a tuple* $I = \langle A, f, T, +_A, \mathsf{weak}, \pi, \oplus, \rho \rangle$ *where:*

- $A = \langle A, \sqsubseteq, \sqsupseteq, \sqcup, \sqcap, \top, \bot \rangle$ *is a pre-lattice of abstract states;*
- f *is the flow sense, either forward* ($f = \downarrow$)*, or backward* ($f = \uparrow$)*;*
- *for each* $s \in \mathsf{atomStmt}$*,* $T_s : A \to A$*;*
- $+_A : A \times A \to A$
- *for each* $i \in \mathcal{N}_{\mathcal{S}}$*,* $\mathsf{weak}_i : A \to A$
- *for each* $i \in \mathbb{N}$*,* $\pi_i^A : A \to A$ *and* $\oplus_i^A : A \times A \to A$*.*
- $\rho : A \times \mathsf{Bool} \to A$*;*

Intuitively, for each operation and rule of the semantics, we have a corresponding operator on the abstract domain that reflects the changes to the memory on the abstract domain. The set of functions $\{T_s\}_{s \in \mathsf{atomStmt}}$ is the set of transfer functions corresponding to atomic statements. The operator $+_A$ abstracts the operator $+$ for memories (we omit the reference to the domain when it is clear from the context). Given an $i \in \mathcal{N}_{\mathcal{S}}$ and $a \in A$, the function $\mathsf{weak}_i(a)$ removes any condition on the scalar variable i from a. It is used when processing a Forall task, with i being the iteration variable, to show that after execution, the iteration variable has an undefined value. For each $i \in \mathbb{N}$, the operators $\{\pi_i^A\}_{i \in \mathbb{N}}$ and $\{\oplus_i^A\}_{i \in \mathbb{N}}$ abstract the operations π_i and $\oplus_i$ for memories (we omit the reference to the domain when it is clear from the context). Finally, the function $\rho : A \times \mathsf{Bool} \to A$ is a transfer function used in an If task to update an abstract value depending on the test condition.

To formalize the connection between the memory states and the abstract states, we assume a satisfaction relation $\models \subseteq \mathcal{H} \times A$ that is an approximation order, i.e., for all $\sigma \in \mathcal{H}$ and $a_1, a_2 \in A$, if $\sigma \models a_1$ and $a_1 \sqsubseteq a_2$ then $\sigma \models a_2$. The next definition formalizes the intuition given about the relation between the operators of an abstract interpretation and the semantics of programs.

Definition 5. *The abstract interpretation* $I = \langle A, f, T, +, \mathsf{weak}, \pi, \oplus, \rho \rangle$ *is consistent if the following holds:*

- *for every statement* $s \in \mathsf{atomStmt}$*, and* $\sigma, \sigma' \in \mathcal{H}$ *s.t.* $\sigma \vdash s \to \sigma'$*,*
  - *if* $f = \downarrow$ *and* $\sigma \models a$*, then* $\sigma' \models T_s(a)$*;*
  - *if* $f = \uparrow$ *and* $\sigma \models T_s(a)$*, then* $\sigma' \models a$*.*
- *for all* $\sigma, \sigma_1, \sigma_2 \in \mathcal{H}$ *and* $a_1, a_2 \in A$*, if* $\sigma_1 \models a_1$ *and* $\sigma_2 \models a_2$ *then* $\sigma_1 +_\sigma \sigma_2 \models a_1 + a_2$*.*
- *for all* $i \in \mathcal{N}$*,* $a \in A$*, and* $\mu_p, \langle \mu, \tau \rangle \in \mathcal{H}$*, if* $\mu_p, \langle \mu, \tau \rangle \models a$*, then for all* $k \in \mathbb{Z}$ $\mu_p, \langle \mu[i \mapsto k], \tau \rangle \models \mathsf{weak}_i(a)$*;*
- *for all* $\sigma \in \mathcal{H}$*, if* $\sigma \models a$ *then* $\pi_i(\sigma) \models \pi_i(a)$*;*
- *for all* $\sigma, \sigma' \in \mathcal{H}$*, and* $a, a' \in A$*, if* $\sigma \models a$ *and* $\sigma' \models a'$*, then* $\sigma \oplus_i \sigma' \models a \oplus_i a'$*;*

$$\frac{\begin{array}{c}X \text{ the set of maximal elements of } H \text{ and } H' = H \setminus X: \\ \forall\, g \in X. \langle a \rangle \vdash g \ \langle a_g \rangle \qquad \langle \sum_{g \in X} a_g \rangle \vdash \mathsf{Group}(H')\ \langle a' \rangle\end{array}}{\langle a \rangle \vdash \mathsf{Group}(H)\ \langle a' \rangle}\textbf{[G]}$$

$$\frac{f = \uparrow \qquad \forall m \le j \le n.\ \langle a'_j \rangle \vdash G\ \langle a_j \rangle}{\langle \prod_{j=m}^{n} T_{i:=j}(a'_j) \rangle \vdash \mathsf{Forall}\ i = m : n\ \mathsf{do}\ G\ \langle \sum_{j=m}^{n} \mathsf{weak}_i(a_j) \rangle}\textbf{[FB]}$$

$$\frac{f = \downarrow \qquad \forall m \le j \le n.\ \langle T_{i:=j}(a) \rangle \vdash G\ \langle a_j \rangle}{\langle a \rangle \vdash \mathsf{Forall}\ i = m : n\ \mathsf{do}\ G\ \langle \sum_{j=m}^{n} \mathsf{weak}_i(a_j) \rangle}\textbf{[FF]} \qquad \frac{\langle \pi_i(a) \rangle \vdash G\ \langle a' \rangle}{\langle a \rangle \vdash \mathsf{Exec}_i(G)\ \langle a \oplus_i a' \rangle}\textbf{[E]}$$

$$\frac{f = \uparrow, s \in \mathsf{atomStmt}}{\langle T_s(a) \rangle \vdash s\ \langle a \rangle}\textbf{[AB]} \qquad \frac{f = \downarrow, s \in \mathsf{atomStmt}}{\langle a \rangle \vdash s\ \langle T_s(a) \rangle}\textbf{[AF]}$$

$$\frac{b \sqsubseteq a \quad \langle a \rangle \vdash G\ \langle a' \rangle \quad a' \sqsubseteq b'}{\langle b \rangle \vdash G\ \langle b' \rangle}\textbf{[SS]} \qquad \frac{\langle \rho(a, cond) \rangle \vdash G_1\ \langle a' \rangle \quad \langle \rho(a, \neg cond) \rangle \vdash G_2\ \langle a' \rangle}{\langle a \rangle \vdash \mathsf{If}\ cond\ \mathsf{then}\ G_1\ \mathsf{else}\ G_2\ \langle a' \rangle}\textbf{[I]}$$

Fig. 2. Program analysis rules (excerpt)

– *for all* $\sigma \in \mathcal{H}$, $a \in A$ *and* $cond \in \mathsf{Bool}$, *if* $\sigma \models a$ *and* $\sigma \models_{\mathsf{Bool}} cond$, *then* $\sigma \models \rho(a, cond)$. [2]

Definition 6 (Valid Analysis Judgment). *Let* $I = \langle A, f, T, +, \mathsf{weak}, \pi, \oplus, \rho \rangle$ *be an abstract interpretation. A judgment is a tuple* $\langle a \rangle \vdash G\ \langle a' \rangle$*, where G is a program and* $a, a' \in A$*. A judgment is valid if it is the root of a derivation tree built using the rules in Fig. 2.*

Lemma 1 (Analysis Soundness). *Let G be a Sequoia program and suppose* $I = \langle A, f, T, +, \mathsf{weak}, \pi, \oplus, \rho \rangle$ *is a consistent abstract interpretation. For every* $a, a' \in A$ *and* $\sigma, \sigma' \in \mathcal{H}$*, if the judgment* $\langle a \rangle \vdash G\ \langle a' \rangle$ *is valid and* $\sigma \vdash G \to\ \sigma'$ *and* $\sigma \models a$ *then* $\sigma' \models a'$*.*

Proof. By induction on the judgment $\vdash$. Since the abstract interpretation I is consistent, we have the required conditions for each case of the judgment.

3.2 Program Safety

Intuitively, a program is safe if the subtasks that can be executed in parallel are independent, i.e., they modify different parts of the memory. This means that independent subtasks can be executed in any order without affecting the final result, which justifies the use of our semantics (cf. Proposition 1).

To check safety for a given program, we define an abstract interpretation that over-approximates the regions of the memory that are read and written by each

[2] Given a memory σ and a boolean condition *cond*, the judgment $\sigma \models_{\mathsf{Bool}} cond$ states that the condition is valid in the memory σ. The definition is standard so we omit it.

subtask (region analysis). A program is safe if these regions do not overlap for subtasks that can be executed in parallel. The presence of arrays makes things more complicated, since when a program of the form Forall $j = m : n$ do G is executed, we must check that, for different values of j, G writes non-overlapping portions of the arrays. This ensures that the final memory does not depend on the order of execution of the subtasks composing a program. Note that the compiler for Sequoia described in [12] assumes that programs are safe, but does not check it. For our purposes of verification, checking this property is essential.

We first define an interval analysis that over-approximates the values of scalar variables, with domain D_I, where

$$D_I = \mathcal{N_S}^+ \to \mathit{Interval},$$
$$\mathit{Interval} = \{(a,b) : a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\}_\perp.$$

The complete definition is given in Appendix A.

The region analysis is described by two abstract interpretations I^R (reading region) and I^W (writing region), both defined over a domain D, where

$$D = (\mathcal{N_A}^+ \to \mathit{Interval}) \times \mathcal{P}(\mathcal{N_S}^+).$$

Each element $d \in D$ is a pair (a, s), where, for each $n \in \mathcal{N_A}^+$, $a(n)$ represents the range of indices of array a that are accessed (read or write) by a subtask, and s is the set of scalar variables that are accessed. The interval analysis described above is used to determine which part of a given array is accessed by each subtask.

The analyses I^R and I^W define the judgments $\langle a \rangle \vdash_R G \langle a' \rangle$ and $\langle a \rangle \vdash_W G \langle a' \rangle$, where G is a program, using the rules of Fig. 2. The meaning of these judgments is stated in Lemma 2 and Lemma 3 below. We first give some definitions.

Given a set of locations $L \subseteq \mathcal{L}$, and memories $\mu, \mu' \in \mathcal{M}$ we write $\mu \approx_L \mu'$ if μ and μ' have the same values at the variables in L. This definition is extended to regions (i.e. elements of D) and memory hierarchies. Given two memories $\mu, \mu' \in \mathcal{M}$, we denote with $\mathit{Modified}(\mu, \mu')$ the set of locations that have different values in μ and μ'. This definition is also extended to memory hierarchies.

Lemma 2 (Soundness of I^R). *Assume a program G, $R_1, R_2 \in D^R$ with $\langle R_1 \rangle \vdash_R G \langle R_2 \rangle$, and $\sigma_1, \sigma_2 \in \mathcal{H}$ with $\sigma_1 \approx_{R_2} \sigma_2$. If $\sigma_1 \vdash G \to \sigma_1'$ and $\sigma_2 \vdash G \to \sigma_2'$, then $\sigma_1' \approx_R \sigma_2'$, where $R = \mathit{Modified}(\sigma_1, \sigma_1') \cup \mathit{Modified}(\sigma_2, \sigma_2')$.*

Lemma 3 (Soundness of I^W). *Assume a program G, $W_1, W_2 \in D^W$ with $\langle W_1 \rangle \vdash_W G \langle W_2 \rangle$, and $\sigma, \sigma' \in \mathcal{H}$. If $\sigma \vdash G \to \sigma'$ then $\mathit{Modified}(\sigma, \sigma') \subseteq W_2$.*

Using the region analysis described above, we can determine whether a program is safe, using the safety judgment, $\vdash_{\mathsf{Safe}} G$, defined by the rules shown in Fig. 3. The interesting rules are the rules for Group and Forall. We check that subtasks that can be executed in parallel have non-overlapping regions. More specifically, the writing region of one task cannot overlap with neither the reading nor the writing region of another independent task. This ensures that the final memory does not depend on the order in which tasks are executed. In the

$$\frac{G \in \mathsf{atomStmt}}{\vdash_{\mathrm{Safe}} G} \qquad \frac{\pi_i(s) \vdash_{\mathrm{Safe}} G}{s \vdash_{\mathrm{Safe}} \mathsf{Exec}_i(G)} \qquad \frac{s \vdash_{\mathrm{Safe}} G_1 \qquad s \vdash_{\mathrm{Safe}} G_2}{s \vdash_{\mathrm{Safe}} \mathsf{If}\ b\ \mathsf{then}\ G_1\ \mathsf{else}\ G_2}$$

$$\frac{\begin{array}{c}\forall G, G' \in H \text{ with } G \text{ and } G' \text{ not related in } H \qquad \langle\bot\rangle \vdash_W G\langle w_G\rangle \qquad \langle\bot\rangle \vdash_W G'\langle w_{G'}\rangle \\ \langle\bot\rangle \vdash_R G\langle r_G\rangle \qquad \langle\bot\rangle \vdash_R G'\langle r_{G'}\rangle \qquad (r_G \cup w_G) \cap w'_G = \emptyset \qquad (r_{G'} \cup w_{G'}) \cap w_G = \emptyset\end{array}}{\vdash_{\mathrm{Safe}} \mathsf{Group}(H)}$$

$$\frac{\begin{array}{c}\forall k, \langle T^R_{j:=k}(\bot)\rangle \vdash_R G\langle r_k\rangle \\ \forall k, \langle T^W_{j:=k}(\bot)\rangle \vdash_W G\langle w_k\rangle \qquad \forall k, k', (r_k \cup w_k) \cap w'_k = \{j\}\end{array}}{\vdash_{\mathrm{Safe}} \mathsf{Forall}\ j = s : e\ \mathsf{do}\ G}$$

Fig. 3. Program Safety

case of Forall, there is an overlap between the variables written by each subtask, namely the iteration variable (since this variable is written at the beginning of execution). However, in the semantics we restore the previous value of it after execution of the Forall, since we consider it a local variable. Hence, the final memory does not depend on the order of execution of parallel subtasks.

The next result justifies the change we made to the Sequoia semantics. In the original semantics defined in [12], there is a different rule for Group, where the set X is *any* set of maximal elements, as opposed to the semantics defined in Fig. 1 where we take X to be the complete set of maximal elements. This means that, in the original semantics, the order of execution of parallel subtasks is not deterministic. However, for safe programs, both semantics give the same final result, as stated in the proposition below. To differentiate, we use $\to_O$ to refer to the original semantics.

Proposition 1. *Assume a program G s.t. $\vdash_{\mathrm{Safe}} G$, and a memory hierarchy $\sigma \in \mathcal{H}$. If $\sigma \vdash G \to \sigma_1$ and $\sigma \vdash G \to_O \sigma_2$, then $\sigma_1 = \sigma_2$.*

Before proving this proposition, we introduce a useful lemma.

Lemma 4. *Assume programs G_1 and G_2, and regions R_1, R_2, W_1 and W_2 s.t.*

$$\begin{array}{cc}\langle\bot\rangle \vdash_R G_1\langle R_1\rangle & \langle\bot\rangle \vdash_R G_2\langle R_2\rangle \\ \langle\bot\rangle \vdash_R G_1\langle W_1\rangle & \langle\bot\rangle \vdash_R G_2\langle W_2\rangle,\end{array}$$

and $W_1 \cap (R_2 \cup W_2) = \emptyset$ and $W_2 \cap (R_1 \cup W_1) = \emptyset$. If $\sigma, \sigma_1, \sigma_2, \sigma_{12}, \sigma_{21} \in \mathcal{H}$ are memories s.t.

$$\begin{array}{cc}\sigma \vdash G_1 \to \ \sigma_1 & \sigma_1 \vdash G_2 \to \ \sigma_{12} \\ \sigma \vdash G_2 \to \ \sigma_2 & \sigma_2 \vdash G_1 \to \ \sigma_{21},\end{array}$$

then $\sigma_{12} = \sigma_{21}$

Proof. Using soundness of I^R and I^W.

Proof (Proposition 1). We want to prove that the order of execution of independent subtasks does not matter for safe programs. While the original semantics seems to be non-deterministic because of the choice we can make in the Group rule, for safe programs, this choice does not matter as the final result will always be the same.

We consider yet another rule for Group, where we execute all subtasks in sequential order (we use the symbol $\rightarrow_1$ to differentiate from other semantics):

$$\frac{\begin{array}{c}\{g_1, g_2, \ldots, g_n\} = H \qquad \sigma = \sigma_1 \\ \forall i \in \{1, \ldots, n\}, \sigma_i \vdash g_i \rightarrow_1 \ \sigma_{i+1}\end{array}}{\sigma \vdash \mathsf{Group}(H) \rightarrow_1 \ \sigma_{n+1}}$$

The condition in the previous rule is that the order $g_1, \ldots, g_n$ respects the dependencies of the graph H. Note that this order is not unique.

Given a safe program G s.t. $\sigma \vdash G \rightarrow \ \sigma'$ we can show by induction on the semantics that $\sigma \vdash G \rightarrow_1 \ \sigma'$. In fact, we can show, using the previous lemma, that the order in which we execute the subtasks of a Group does not affect the final memory.

Using a similar reasoning, we can show that if $\sigma \vdash G \rightarrow_O \ \sigma''$, then $\sigma \vdash G \rightarrow_1 \ \sigma''$. Combining both results, we infer that $\sigma' = \sigma''$. $\square$

3.3 Program Verification

We now define a verification infrastructure as an instance of the abstract interpretation $I = \langle \mathsf{Prop}, \uparrow, T, +_{\mathsf{Prop}}, \mathsf{weak}, \pi, \oplus, \rho \rangle$, where Prop is the pre-lattice of first-order formulae over variables from $\mathcal{N}^+$. Before giving the complete definition of I, we need some preliminary definitions.

Given a formula ϕ, the formula $\downarrow^i \phi$ is obtained by substituting every free variable $v \in \mathcal{N}^+$ of ϕ with $\downarrow^i v$. In the same way, the formula $\uparrow^i \phi$ is obtained by substituting every free variable $v \in \mathcal{N}^+$ of ϕ by $\uparrow^i v$; if $\uparrow^i v$ is not defined, we substitute v by a fresh variable, and quantify existentially over all the introduced fresh variables.

Definition of $+$. We need some care when defining the operator $+$ on independent subtasks. For instance, assume two tasks G_1 and G_2 that execute in parallel (this means that the memories σ_1 and σ_2 after execution are added together) with post-conditions Q_1 and Q_2. Intuitively, we want to verify that each G_i satisfies the post-condition Q_i and then conclude that after executing both tasks, the resulting memory satisfies $Q_1 \wedge Q_2$. However, this may be false, since Q_1 can refer to variables that are modified by G_2. [3] Hence, while Q_1 is true after executing G_1, it may be false after executing G_1 and G_2 in parallel. The solution is to give a weaker formula than $Q_1 \wedge Q_2$. We will have that after execution of G_1 and G_2, a formula of the form $Q'_1 \wedge Q'_2$ is valid, where Q'_1 (resp. Q'_2) is a modification of Q_1 (resp. Q_2) that does not refer to variables that are modified by G_2 (resp. G_1). In fact, Q'_1 (resp. Q'_2) is defined as an existential quantification over the

[3] Remember that we consider only safe programs, so there are no variables which are modified by both G_1 and G_2

variables that are modified by G_2 (resp. G_1). Therefore, to define the operator $+$, we require that subtasks are annotated with the set of variables that they modify. In our case, we use the region analysis given in the previous section. For two subtasks G_1 and G_2 that respectively modify the scalar variables in the sets SW_1 and SW_2, and the array ranges in AW_1 and AW_2, we define the merging operator $+$ as $\phi_1 + \phi_2 = \phi_1' \wedge \phi_2'$ where ϕ_1' is a weaker version of ϕ_1 defined as $\exists X'.\ \phi_1[^{X'}/_{X}] \wedge \bigwedge_{A[m,n] \in AW_1} A'[m,n] = A[m,n]$, X' representing the set of scalar and array variables in $SW_2 \cup AW_2$ (and similarly with ϕ_2). We generalize the operator $+$ for a family of post-conditions $\{\phi_i\}^{i\in I}$ and a family of specifications of modified variables $\{SW_i\}_{i\in I}$ and $\{AW_i\}_{i\in I}$, by defining $\sum_{i\in I} \phi_i$ as $\bigwedge_{i\in I} \phi_i'$ where for every $i \in I$, $\phi' = \exists X'.\ \phi[^{X'}/_{X}] \wedge \bigwedge_{A[m,n]\in AW_i} A[m,n] = A'[m,n]$, s.t. X' represents every scalar or array variable in $\{SW_j\}_{j\neq i \in I}$ or $\{AW_j\}_{j\neq i\in I}$.

In practice, if an assertion ϕ_i refers only to scalar and array variables that are not declared as modifiable by other member $j \neq i$, we have $\phi_i' \Rightarrow \phi_i$.

Definition of other components of I. They are defined as follows:

- $\{T_s\}_{s\in \mathsf{atomStmt}}$ are the weakest pre-condition transformers (the complete definition is given in Appendix B);
- $\mathsf{weak}_i(\phi) = \exists i.\ \phi$, where $i \in \mathcal{N_S}^+$;
- $\pi_i(\phi) = \uparrow^i \phi$, where $i \in \mathbb{N}$;
- $\phi_1 \oplus_i \phi_2 = \overline{\phi_1} \wedge \downarrow^i \phi_2$, where $i \in \mathbb{N}$, and $\overline{\phi_1}$ is obtained from ϕ_1 by replacing every variable of the form x or $x^{\downarrow^i \downarrow^{j_1} \ldots \downarrow^{j_k}}$ with a fresh variable and then quantifying existentially all the introduced fresh variables.
- $\rho(\phi, cond) = \phi \wedge cond$;

The satisfaction relation $\sigma \models \phi$ is defined as the validity of the interpretation of the formula ϕ in the memory hierarchy σ. To appropriately adapt a standard semantics $[\![.]\!]$ to a hierarchy of memories, it suffices to extend the interpretation for the extended set of variables $\mathcal{N}^+$, where $[\![n]\!]\sigma = \sigma(n)$, for $n \in \mathcal{N}^+$.

In the rest of the paper, we denote as $\{P\} \vdash G\ \{Q\}$ the judgments in the domain of logical formulae, and P and Q are said to be pre- and post-conditions of G respectively. If the judgment $\{P\} \vdash G\ \{Q\}$ is valid, and the program starts in a memory σ that satisfies P and finishes in a memory σ', then σ' satisfies Q. The proposition below formalizes this result.

Proposition 2 (Verification Soundness). *Assume that $\langle P\rangle \vdash G\langle Q\rangle$ is a valid judgment and that $\sigma \vdash G \rightarrow \sigma'$, where G is a program, P, Q are assertions, and $\sigma, \sigma' \in \mathcal{H}$. If $\sigma \models P$ then $\sigma' \models Q$.*

3.4 Example Program

We show an example of program verification. The program G_{Add} (shown in Fig. 4) adds two arrays producing an output array. This task is divided in a number of independent subtasks that operate on different parts of the arrays. We use the operators $\|$ and ; to describe the dependence graph composing a Group task, that represent parallel and sequential composition respectively.

$$
\begin{aligned}
G_{\mathrm{Add}} &= \mathsf{Group}(G_1 \parallel \ldots \parallel G_n) \\
G_i &= \mathsf{Exec}_i(\mathsf{Group}(\mathrm{InitArgs}; \\
&\qquad \mathsf{Kernel}\langle Z[0,S] = \mathrm{VectAdd}(X[0,S], Y[0,S])\rangle; \\
&\qquad \mathsf{Copy}^{\uparrow}(Z[0,S], C[i\,S,(i+1)S]))) \\
\mathrm{InitArgs} &= (\mathsf{Copy}^{\downarrow}(A[i\,S,(i+1)S], X[0,S]) \parallel \\
&\qquad \mathsf{Copy}^{\downarrow}(B[i\,S,(i+1)S], Y[0,S]))
\end{aligned}
$$

Fig. 4. Program code for addition of two arrays

We assume that the top-level of the memory hierarchy has at least n child memories. The arrays are divided in chunks of size S. Note that the optimum value of S may depend on the underlying architecture.

The program G_{Add} is safe, since the subtasks $G_1,\ldots,G_n$ are independent (they modify different parts of array C). We want to derive the following judgment:

$$\{\mathsf{true}\} \vdash G_{\mathrm{Add}}\ \{\forall k, 0 \le k < n\,S \Rightarrow C[k] = A[k] + B[k]\} \qquad (1)$$

First, we derive, for each i, the following judgment:

$$
\begin{gathered}
\{\mathsf{true}\} \vdash G_i\ \{Q_i\} \\
Q_i = \forall k, i\,S \le k < (i+1)S \Rightarrow C[k] = A[k] + B[k]
\end{gathered}
$$

That is, we show that each subtask G_i computes the addition of arrays A and B for indices between $i\,S$ and $(i+1)S$. Now, using the rule for Group we derive

$$\{\mathsf{true}\} \vdash G_{\mathrm{Add}}\ \{\sum_i Q_i\},$$

where $\sum_i Q_i = \bigwedge_i Q_i'$ and Q_i' is obtained from Q_i by quantifying existentially over the variables that are not modified by Q_i, as explained in the previous section. However, since Q_i only refers to variables that are not modified by other subtasks (besides G_i), we have $Q_i' \Rightarrow Q_i$. Using the subsumption rule we derive (1).

4 Certificate translation

In this section, we show that for the common optimizations considered in [8, 12] we can transform proof of correctness of the original program into proof of correctness of the optimized program. The problem of certificate translation along program optimizations in this setting is motivated by research in Proof Carrying Code (PCC) [15, 14], and in particular by our earlier work on certificate translation [2, 3].

Strictly, the results of this section are not immediately applicable to PCC and to certificate translation, since we have not considered formal certificates. However, it is easy to adapt the notion of valid judgment, including an abstract notion of certificates, as in [3], and to extend our algorithms to transform certified Sequoia programs into certified Sequoia programs.

Due to space limitations, we consider simplified versions of the optimizations.

Certified setting. For the purpose of certificate translation, we capture the notion of certificate infrastructure by an abstract proof algebra, and we assume that certificates are closed under specific operations of the algebra.

Definition 7 (Certificate infrastructure). *A certificate infrastructure consists on a proof algebra $\mathcal{P}$ that assigns to every $\phi \in \mathsf{Prop}$ a set of certificates $\mathcal{P}(\vdash \phi)$ s.t.:*

- *$\mathcal{P}$ is closed under the operations of Fig. 5, all implicitly quantified in the obvious way;*
- *$\mathcal{P}$ is sound, i.e. for every $\phi \in \mathsf{Prop}$, if ϕ is not valid, then $\mathcal{P}(\phi) = \emptyset$.*

In the sequel, we write $c :\vdash \phi$ instead of $c \in \mathcal{P}(\phi)$.

The operations of the proof algebra are standard, to the exception of the operator subst that allows to substitute selected instances of equals by equals, and of the operator *ring*, which establishes ring equalities that will be used to justify the optimizations.

$$
\begin{array}{lll}
\mathsf{intro}_{\mathsf{True}} & : & \mathcal{P}(\Gamma \vdash \mathsf{True}) \\
\mathsf{axiom} & : & \mathcal{P}(\Gamma; A; \Delta \vdash A) \\
\mathsf{ring} & : & \mathcal{P}(\Gamma \vdash n_1 = n_2) \quad \text{if } n_1 = n_2 \text{ is a ring equality} \\
\\
\mathsf{intro}_{\wedge} & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma \vdash B) \to \mathcal{P}(\Gamma \vdash A \wedge B) \\
\mathsf{elim}^{\mathsf{l}}_{\wedge} & : & \mathcal{P}(\Gamma \vdash A \wedge B) \to \mathcal{P}(\Gamma \vdash A) \\
\mathsf{elim}^{\mathsf{r}}_{\wedge} & : & \mathcal{P}(\Gamma \vdash A \wedge B) \to \mathcal{P}(\Gamma \vdash B) \\
\mathsf{intro}_{\Rightarrow} & : & \mathcal{P}(\Gamma; A \vdash B) \to \mathcal{P}(\Gamma \vdash A \Rightarrow B) \\
\mathsf{elim}_{\Rightarrow} & : & \mathcal{P}(\Gamma \vdash A \Rightarrow B) \to \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma \vdash B) \\
\mathsf{elim}_{=} & : & \mathcal{P}(\Gamma \vdash e_1 = e_2) \to \mathcal{P}(\Gamma \vdash A[^{e_1}/_{r}]) \to \mathcal{P}(\Gamma \vdash A[^{e_2}/_{r}]) \\
\mathsf{subst} & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma[^{e}/_{r}] \vdash A[^{e}/_{r}]) \\
\mathsf{weak} & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma; \Delta \vdash A) \\
\mathsf{intro}_{\forall} & : & \mathcal{P}(\Gamma \vdash A) \to \mathcal{P}(\Gamma \vdash \forall r. A) \qquad \text{if } r \text{ is not in } \Gamma \\
\mathsf{elim}_{\forall} & : & \mathcal{P}(\Gamma \vdash \forall r. A) \to \mathcal{P}(\Gamma \vdash A)
\end{array}
$$

Fig. 5. Proof Algebra (excerpt)

Definition 8 (Certified Analysis Solution). *We say that the verification judgment* $\{\Phi\} \vdash G \{\Psi\}$ *is* certified *if it is the root of a derivation tree, built from the rules in Fig. 2, such that every application of the subsumption rule*

$$\frac{\varphi \Rightarrow \phi \quad \{\phi\} \vdash G \{\phi'\} \quad \phi' \Rightarrow \varphi'}{\{\varphi\} \vdash G \{\varphi'\}} \textbf{[CSS]}$$

is accompanied with certificates c *and* c' *s.t.* $\mathsf{c} : \vdash \varphi \Rightarrow \phi$ *and* $\mathsf{c}' : \vdash \phi' \Rightarrow \varphi'$*.*

We call a *certificate* for the judgment $\{\Phi\} \vdash G \{\Psi\}$ a derivation tree together with a tuple of certificates for each application of the subsumption rule.

4.1 General framework

In this section, we consider the translation of certified verification judgments when a program G' is derived from the original program G by a structure preserving transformation that is justifiable by a valid analysis judgment. This category of transformations covers several optimizations that improve the efficiency of an atomic sub-program exploiting conditions ensured by previously executed statements, including standard optimizations such as constant propagation or common sub-expression elimination (which are not treated here, but refer to [2, 3]). We illustrate the transformation with copy propagation.

Certifying analyzers. Certificate translation along a program transformation may require that the analysis that justifies the transformation is certified, i.e. that there is a certificate of the result of the analysis expressed as logical formulae. In this section, we provide sufficient conditions for the existence of such extended analyzers, called certifying analyzers.

Consider an abstract interpretation $I = \langle A, f, T, +, \mathsf{weak}, \pi, \oplus, \rho \rangle$ and assume that $\langle a \rangle \vdash G \langle a' \rangle$ is a valid judgment of the analysis. Let $\gamma : A \to \mathsf{Prop}$ be a concretization function, i.e. a function that for any $a \in A$, returns an interpretation of a as a logic formula.

We provide sufficient conditions to generate a certified verification judgment $\{\gamma(a)\} \vdash G \{\gamma(a')\}$ from the valid analysis judgment $\langle a \rangle \vdash G \langle a' \rangle$.

Definition 9. *An abstract interpretation* $I = \langle A, f, T, +, \mathsf{weak}, \pi, \oplus, \rho \rangle$ *is consistent with the verification infrastructure if we have*

- *for every* $a_1, a_2 \in A$ *s.t.* $a_1 \sqsubseteq a_2$*, a certificate* $\mathsf{monot}_\gamma : \gamma(a_1) \Rightarrow \gamma(a_2)$
- *for every* $a \in D, s \in \mathsf{atomStmt}$*,*
  - $f = \uparrow$ *and* $\mathsf{cons}_s(a) : \gamma(T_s(a)) \Rightarrow \mathsf{wp}(\gamma(a))$*, or*
  - $f = \downarrow$ *and* $\mathsf{cons}_s(a) : \gamma(a) \Rightarrow \mathsf{wp}(\gamma(T_s(a)))$*.*
- *for every* $a, a' \in A$*, the certificates* $\mathsf{distrib}_{+,\gamma} : \vdash \gamma(a) + \gamma(a') \Rightarrow \gamma(a + a')$ *and* $\mathsf{distrib}_{\sqcap,\gamma} : \vdash \gamma(a \sqcap a') \Rightarrow \gamma(a) \wedge \gamma(a')$*;*
- *for every* $a \in A$*,* $i \in \mathcal{N}_{\mathcal{S}}$*,* $\mathsf{c}_{\mathsf{weak}} : \vdash \mathsf{weak}_i(\gamma(a)) \Rightarrow \gamma(\mathsf{weak}_i(a))$*;*
- *for every* $i \in \mathbb{Z}$ $a, a' \in A$*, the certificates* $\mathsf{c}_{\pi_i} : \vdash \pi_i(\gamma(a)) \Rightarrow \gamma(\pi_i(a))$ *and* $\mathsf{c}_{\oplus_i} : \vdash \gamma(a) \oplus_i \gamma(a') \Rightarrow \gamma(a \oplus_i a')$*; and*

– *for every $a \in A$ and $b \in \mathsf{Bool}$, a certificate $\mathsf{c}_\rho : \gamma(a) \wedge b \Rightarrow \gamma(\rho(a, b))$.*

The following result states that a valid analysis judgment that motivates a program transformation is certifiable, as long as the analysis I is consistent with the verification environment. This result is fundamental, since for several program transformations the certification of the analysis is a component of the certificate translation.

Lemma 5. *Let $\langle a \rangle \vdash G \ \langle a' \rangle$ be a valid analysis judgment. Then, provided I is consistent with the verification environment, $\{\gamma(a)\} \vdash G \ \{\gamma(a')\}$ is a certified verification judgment*

Proof. The proof is by induction on the derivation of $\langle a \rangle \vdash G \ \langle a' \rangle$. We consider only some representative cases.

– *Base case. Last rule applied is* **[AB]**. *Then $a = T_s(a')$, with $s = G$. By application of the corresponding rule* **[AB]** *in the domain of the verification environment, we get the judgment $\{\mathsf{wp}_s(\gamma(a'))\} \vdash s \ \{\gamma(a')\}$. By consistency of the analysis I, we have a certificate $\mathsf{cons} : \vdash \gamma(T_s(a')) \Rightarrow \mathsf{wp}_s(\gamma(a'))$, and then, by application of the subsumption rule* **[CSS]** *we get the certified judgment $\{\gamma(T_s(a'))\} \vdash s \ \{\gamma(a')\}$.*
– *Last rule applied is* **[SS]**. *Then we have a sub-derivation tree for the judgment $\langle b \rangle \vdash G \ \langle b' \rangle$ and $a \sqsubseteq b$ and $b' \sqsubseteq a'$. We know, by I.H., that the judgment $\{\gamma(b)\} \vdash G \ \{\gamma(b')\}$ is certified. By monotonicity of γ we have certificates for $\gamma(a) \Rightarrow \gamma(b)$ and $\gamma(b') \Rightarrow \gamma(a')$ and then, by subsumption, a certified judgment $\{\gamma(a)\} \vdash G \ \{\gamma(a')\}$*
– *Last rule applied is* **[FB]**. *Then G has the form $\mathsf{Forall}\ i = m : n\ \mathsf{do}\ G'$ and a are a' are necessarily equal to $\prod_{j=m}^{n} T_{i:=j}(a_j)$ and $\sum_{j=m}^{n} \mathsf{weak}_i(a'_j)$ respectively, s.t. for every $j \in [m, n]$ there is a sub-derivation tree for the judgment $\langle a_j \rangle \vdash G' \ \langle a'_j \rangle$. By I.H., we know the judgments $\{\gamma(a_j)\} \vdash G' \ \{\gamma(a'_j)\}$ are certified and, thus, by application of the rule* **[FB]**, *we get the certified judgment $\{\bigwedge_{j=m}^{n} T_{i:=j}(\gamma(a_j))\} \vdash G' \ \{\sum_{j=m}^{n} \mathsf{weak}_i(\gamma(a'_j))\}$. It remains to show that we have a certificate for $\gamma(\prod_{j=m}^{n} T_{i:=j}(a_j)) \Rightarrow \bigwedge_{j=m}^{n} \mathsf{wp}_{i:=j}(\gamma(a_j))$ and a certificate for $\sum_{j=m}^{n} \mathsf{weak}_i(\gamma(a'_j)) \Rightarrow \gamma(\sum_{j=m}^{n} \mathsf{weak}_i(a'_j))$. The former follows from cons and $\mathsf{distrib}_{\wedge,\gamma}$, and the latter from $\mathsf{distrib}_{+,\gamma}$ and $\mathsf{c}_{\mathsf{weak}}$.*
– *Last rule applied is* **[E]**. *We know that G is of the form $\mathsf{Exec}_i(G')$, a' is equal to $a \oplus_i a''$ for some $a'' \in A$, and that there sub-derivation tree for the judgment $\langle \pi_i(a) \rangle \vdash G' \ \langle a'' \rangle$. Then, by I.H., the judgment $\langle \gamma(\pi_i(a)) \rangle \vdash G' \ \langle \gamma(a'') \rangle$ is certified. By application of c_{π_i}, we get a certificate for $\pi_i(\gamma(a)) \Rightarrow \gamma(\pi_i(a))$ and then, by subsumption, a certified judgment $\langle \pi_i(\gamma(a)) \rangle \vdash G' \ \langle \gamma(a'') \rangle$. Applying the rule* **[E]**, *we get $\langle \gamma(a) \rangle \vdash G' \ \langle \gamma(a) \oplus_i \gamma(a'') \rangle$. Finally, from $\mathsf{c}_{\oplus_i}$ and application of the subsumption rule* **[CSS]** *we have the certified judgment $\langle \gamma(a) \rangle \vdash G' \ \langle \gamma(a \oplus_i a'') \rangle$.*

Certificate translation. Let G' be a program derived from G by a structure preserving transformation, i.e. G and G' have the same structure but differ on the leaves of the abstract syntax tree.

Definition 10 (justified transformation). *Let s and s' be atomic statements and R a logic formula. The substitution of s by s' is justified by R if for every assertion ϕ, we have a certificate* $\mathsf{justif} : \vdash R \wedge \mathsf{wp}_s(\phi) \Rightarrow \mathsf{wp}_{s'}(\phi)$. *We say that a derivation tree for the judgment $\{P\} \vdash G\ \{Q\}$ justifies a structure preserving transformation from G to G', if the substitution of every atomic subprogram g in G by g' is justified by a pre-condition R s.t. $\{R\} \vdash g\ \{R'\}$ is the corresponding derivation sub-tree of $\{P\} \vdash G\ \{Q\}$.*

The following result, in combination with Lemma 5, states that certificate translation from G to G' is feasible if G' is derived from G by a structure preserving transformation that is justified by a valid analysis judgment.

Lemma 6. *Let G' by a program derived from G by a structure preserving transformation, justified by the certified judgment $\{R\} \vdash G\ \{R'\}$. Then we can build a derivation tree for the judgment $\{P \wedge R\} \vdash G'\ \{Q \wedge R'\}$ from the certificate of the original judgment $\{P\} \vdash G\ \{Q\}$.*

Proof. The proof is by induction on the derivation of $\{P\} \vdash G\ \{Q\}$. We consider only some representative cases, and assume w.l.g. (by trivial applications of the subsumption rule) that $\{P\} \vdash G\ \{Q\}$ and $\{R\} \vdash G\ \{R'\}$ are derived applying exactly the same rules. For simplicity, we assume the existence of the following extra certificates, built by application of the operators of the proof algebra, and by definition of wp, weak, *π_i and $\oplus_i$:*

1. $\mathsf{distrib}_{\wedge,\mathsf{wp}} : \vdash \mathsf{wp}_s(\phi) \wedge \mathsf{wp}_s(\psi) \Rightarrow \mathsf{wp}_s(\phi \wedge \psi)$
2. $\mathsf{distrib}_{\wedge,\mathsf{weak}} : \vdash \mathsf{weak}_i(\phi \wedge \psi) \Rightarrow \mathsf{weak}_i(\phi) \wedge \mathsf{weak}_i(\psi)$
3. $\mathsf{distrib}_{\wedge,\pi_i} : \vdash \pi_i(\phi \wedge \psi) \Rightarrow \pi_i(\phi) \wedge \pi_i(\psi)$
4. $\mathsf{distrib}_{\wedge,\oplus_i} : \vdash (\phi \wedge \psi) \oplus_i (\phi_i \wedge \psi_i) \Rightarrow (\phi \oplus_i \phi_i) \wedge (\psi \oplus_i \psi_i)$
5. $\mathsf{commut}_{\wedge} : \vdash \phi \wedge \psi \Rightarrow \psi \wedge \phi$
6. $\mathsf{assoc}_{\wedge} : \vdash (\phi \wedge \psi) \wedge \varphi \Rightarrow \phi \wedge (\psi \wedge \varphi)$

- *Base case: last rule applied is* $[\mathbf{AB}]$. *Let $s = G$ and $s' = G'$, then we have $R = \mathsf{wp}_s(R')$ and $P = \mathsf{wp}_s(Q)$. Since R justifies the substitution of s by s', there exist a certificate for $R \wedge \mathsf{wp}_s(R' \wedge Q) \Rightarrow \mathsf{wp}_{s'}(R' \wedge Q)$. By definition of R and* $\mathsf{distrib}_{\wedge,\mathsf{wp}}$, *we have a certificate for $R \wedge P \Rightarrow \mathsf{wp}_{s'}(R' \wedge Q)$, then, by subsumption we get the certified judgment $\{R \wedge P\} \vdash G'\ \{R' \wedge Q\}$.*
- *If the last rule applied is* $[\mathbf{CSS}]$, *we know that the judgments $\langle P_1 \rangle \vdash G\ \langle Q_1 \rangle$ and $\langle R_1 \rangle \vdash G\ \langle R'_1 \rangle$ are certified, and there exists certificates for $P \Rightarrow P_1$, $Q_1 \Rightarrow Q$, $R \Rightarrow R_1$ and $R'_1 \Rightarrow R_1$. The judgment $\langle P_1 \wedge R_1 \rangle \vdash G'\ \langle Q_1 \wedge R'_1 \rangle$ is certified, by I.H. Besides, using rules of the proof algebra, we get certificates for the judgments $P \wedge R \Rightarrow P_1 \wedge R_1$, $Q_1 \wedge R'_1 \Rightarrow Q \wedge R$. By application of the subsumption rule* $[\mathbf{CSS}]$, *we get the certified judgment $\{P \wedge R\} \vdash G'\ \{Q \wedge R'\}$.*
- *If the last rule applied is* $[\mathbf{FB}]$, *we have that G is equal to the program* $\mathsf{Forall}\ i = m : n\ \mathsf{do}\ G_1$, *that G' is equal to* $\mathsf{Forall}\ i = m : n\ \mathsf{do}\ G'_1$, *$P = \bigwedge_{j=m}^{n} \mathsf{wp}_{i:=j}(P_j)$, $R = \bigwedge_{j=m}^{n} \mathsf{wp}_{i:=j}(R_j)$, $Q = \sum_{j=m}^{n} \mathsf{weak}_i(Q_j)$ and $R' = \sum_{j=m}^{n} \mathsf{weak}_i(R'_j)$. In addition, we know the judgments $\{P_j\} \vdash G_1\ \{Q_j\}$ and $\{R_j\} \vdash G_1\ \{R'_j\}$ are certified, for every $j \in [m,n]$. By I.H., we get the*

certified judgments $\{P_j \wedge R_j\} \vdash G'_1 \; \{Q_j \wedge R'_j\}$, and therefore, by application of the rule **[FB]**, *we have the certified judgment*

$$\{\textstyle\bigwedge_{j=m}^{n} \mathsf{wp}_{i:=j}(P_j \wedge R_j)\} \vdash G'_1 \; \{\textstyle\sum_{j=m}^{n} \mathsf{weak}_i(Q_j \wedge R'_j)\}$$

From $\mathsf{commut}_\wedge$, $\mathsf{assoc}_\wedge$ *and* $\mathsf{distrib}_{\wedge,\mathsf{wp}}$, *we can define a certificate for $P \wedge R \Rightarrow \bigwedge_{j=m}^{n} \mathsf{wp}_{i:=j}(P_j \wedge R_j)$. From* $\mathsf{distrib}_{\wedge,\mathsf{weak}}$ *and* assoc_+ *we have a certificate for $\sum_{j=m}^{n} \mathsf{weak}_i(Q_j \wedge R'_j) \Rightarrow Q \wedge R'$. Therefore, by application of the subsumption rule* **[CSS]** *we get the desired result.*

– *If the last rule applied is* **[E]**, *then $G = \mathsf{Exec}_i(G_1)$, $G' = \mathsf{Exec}_i(G'_1)$, for some G_1, G'_1, and Q and R' are equal to $P \oplus_i Q_i$ and $R \oplus_i R_i$, respectively, for some Q_i, R_i. In addition we know that the judgments $\{\pi_i(P)\} \vdash G_1 \; \{Q_i\}$ and $\{\pi_i(R)\} \vdash G_1 \; \{R_i\}$ are certified. By I.H., we have the certified judgment $\{\pi_i(P) \wedge \pi_i(R)\} \vdash G'_1 \; \{Q_i \wedge R_i\}$. From* $\mathsf{distrib}_{\wedge,\pi_i}$ *we get the certificate $\pi_i(P \wedge R) \Rightarrow \pi_i(P) \wedge \pi_i(R)$, and hence, by subsumption the certified judgment $\{P \wedge R\} \vdash G' \; \{(P \wedge R) \oplus_i (Q_i \wedge R_i)\}$. In addition, by application of* $\mathsf{distrib}_{\wedge,\oplus_i}$ *we have a certificate for $(P \wedge R) \oplus_i (Q_i \wedge R_i) \Rightarrow (P \oplus_i Q_i) \wedge (R \oplus_i R_i)$. Therefore, by application of the subsumption rule* **[CSS]**, *we get the desired result.*

Copy propagation for arrays. Since array operations are common in programs targeting data intensive applications, it is of interest to extend traditional copy propagation to consider copy operations between arrays. Naturally, this transformation requires a richer analysis domain to relate not simply arrays but array intervals.

Consider an abstract interpretation $I = \langle A, \downarrow, +, \mathsf{weak}, \pi, \oplus, \rho \rangle$ with domain $A = \mathcal{P}(\mathcal{N}_\mathcal{A} \times \mathit{Interval} \times \mathcal{N}_\mathcal{A} \times \mathit{Interval})$. An element $(A, [m,n], A', [m',n'])$ in $\mathcal{N}_\mathcal{A} \times \mathit{Interval} \times \mathcal{N}_\mathcal{A} \times \mathit{Interval}$, denoted $A[m,n] = A'[m',n']$ for readability, is satisfied by a memory hierarchy $\sigma \in \mathcal{H}$ if the intervals $[m,n]$ and $[m',n']$ are equally sized and the arrays A and A' coincide on that ranges. An element a of the abstract domain A is satisfied by a memory hierarchy $\sigma \in \mathcal{H}$ if for all $A[m,n] \in S$, σ satisfies $A[m,n]$.

Definition 11. *Let a be an abstract element such that $A[m,n] = A'[m',n'] \in a$. Consider an atomic statement s that reads an array range $A[x,y]$ such that $[x,y] \subseteq [m,n]$. Then, the judgment $\langle a \rangle \vdash s \; \langle a' \rangle$ induces a transformation of s into s' if s' is the result of substituting every access of $A[i]$ in s by $A'[i-m+m']$.*

For instance, let s be the statement $\mathsf{Kernel}\langle Z = \mathit{VectAdd}(B[0,k], C[0,k]) \rangle$ and $a \in A$ such that $B[0,k] = A[m, m+k] \in a$. Then, for any $a' \in A$ the judgment $\langle a \rangle \vdash s \; \langle a' \rangle$ induces the substitution of the atomic statement s by the statement $\mathsf{Kernel}\langle Z = \mathit{VectAdd}(A[m,m+k], C[0,k]) \rangle$.

Let the program G and $a, a' \in A$ s.t. there is a derivation of the judgment $\langle a \rangle \vdash G \; \langle a' \rangle$. We say that the derivation tree of $\langle a \rangle \vdash G \; \langle a' \rangle$ induces a structure preserving transformation of G into G' if every substitution of an atomic statement s in G by s' in G', is induced by a judgment $\langle b \rangle \vdash s \; \langle b' \rangle$, root of a derivation subtree of $\langle a \rangle \vdash G \; \langle a' \rangle$.

Consider the certified judgment $\{\Phi\} \vdash G \ \{\Psi\}$. A first requirement to translate the judgment along array copy propagation is a certificate of the analysis judgment $\langle a \rangle \vdash G \ \langle a' \rangle$. To this end, we interpret the result of the analysis with a concretization function $\gamma : A \to \mathsf{Prop}$ defined as

$$\gamma(A[m,n] = A'[m',n']) \doteq (n - m = n' - m') \wedge (\forall_{m \leq i \leq n}.\ A[i] = A'[i + m' - m])$$

Lemma 7. *Consider a program G and a valid analysis judgment $\langle a \rangle \vdash G \ \langle a' \rangle$. Assume that the abstract interpretation I is consistent with the verification environment, and that the judgment $\{\gamma(a)\} \vdash G \ \{\gamma(a')\}$ justifies a copy propagation transformation from G to a program G'. Then, for every certified judgment $\{\Phi\} \vdash G \ \{\Psi\}$ we have a certified judgment $\{\gamma(a) \wedge \Phi\} \vdash G' \ \{\gamma(a') \wedge \Psi\}$.*

Consider again the substitution of $\mathsf{Kernel}\langle Z = VectAdd(B[0,k], C[0,k]\rangle$ by the statement $\mathsf{Kernel}\langle Z = VectAdd(A[m, m+k], C[0,k]\rangle$ induced by $\langle a \rangle \vdash s \ \langle a' \rangle$ such that $B[0,k] = A[m, m+k] \in a$. The interpretation $\gamma(a)$ is such that $\gamma(a) \Rightarrow \gamma(B[0,k] = A[m, m+k])$. Hence, to show that this atomic substitution is justified by the analysis consists on requiring the validity of the following proposition:

$$\gamma(B[0,k] = A[m, m+k]) \wedge \phi[{}^{B[0,k],C[0,k]}\!/\!{}_{Z}] \Rightarrow \phi[{}^{A[m,m+k],C[0,k]}\!/\!{}_{Z}]$$

4.2 SPMD Distribution

SPMD distribution is a common parallelization technique that spreads over multiple processors the execution of the same program over independent fragments of data.

Consider a subprogram $\mathsf{Exec}_i(\mathsf{Forall}\ j = 0 : k.n\ \mathsf{do}\ g)$ executing in parallel $k.n$ instances of g in the i-th child memory. Since we are considering safe programs, we can assume that each instance of g operates over independent data.

G' is transformed from G by applying SPMD distribution if G' is the result of substituting every subprogram $\mathsf{Exec}_i(\mathsf{Forall}\ j = 1 : k.n\ \mathsf{do}\ g)$ by the equivalent subprogram $\mathsf{Group}(G_1 \ || \ \dots \ || \ G_k)$ where for any $i \in [1,k]$, the program G_i is defined as $\mathsf{Exec}_i(\mathsf{Forall}\ j = 1 + (i-1)n : i.n\ \mathsf{do}\ g)$.

The following result follows from the commutativity and associativity of the operator $+$ and the distributivity of the function $\downarrow$ w.r.t. the operator $+$, in the domain of logic formulae.

Lemma 8. *Let G' be a program transformed from G by SPMD distribution. Then it is possible to translate a certified verification judgment $\{\Phi\} \vdash G \ \{\Psi\}$ into a certified judgment $\{\Phi\} \vdash G' \ \{\Psi\}$.*

Proof. *To prove this, we show that a certificate of the judgment $\{P\} \vdash G \ \{Q\}$ corresponding to a subprogram of the form $G = \mathsf{Exec}_i(\mathsf{Forall}\ j = 1 : k.n\ \mathsf{do}\ g)$ can be transformed into a certificate of the judgment $\{P\} \vdash G' \ \{Q\}$, where $G' = \mathsf{Group}(G_1 \ || \ \dots \ || \ G_k)$ with $G_i = \mathsf{Exec}_i(\mathsf{Forall}\ j = 1 + (i-1)n : n.i\ \mathsf{do}\ G)$. For simplicity, we refrain from considering the application of the subsumption rule* **[CSS]**. *Then, by application of rule* **[E]**, *Q is equal to $\downarrow^i(Q')$ for some*

Q', and the judgment $\{\uparrow^i(P)\} \vdash$ Forall $j = 0 : k.n$ do G $\{Q'\}$ is certified. We know then that, by application of the rule [**FB**], *$Q' = \sum_{j'=1}^{k.n} \mathsf{weak}_j(Q_{j'})$, that $\uparrow^i(P) = \bigwedge_{j'=1}^{k.n} \mathsf{wp}_{j:=j'}(P_{j'})$ and that for each $j' \in [1, k.n]$ we have a certified judgment $\{P_{j'}\} \vdash G$ $\{Q_{j'}\}$. For each $i \in [1, k]$ we construct a derivation for the judgment $\{\uparrow^i(P)\} \vdash$ Forall $j = 1 + (i-1)n : n.i$ do G $\{Q'_i\}$ where $Q'_i = \sum_{j'=1+(i-1)n}^{n.i} \mathsf{weak}_j(Q_{j'})$, by application of the rule* [**FB**] *and* [**CSS**] *with a certificate of $\uparrow^i(P) \Rightarrow \bigwedge_{j'=1+(i-1)n}^{i.n} \mathsf{wp}_{j:=j'}(P_{j'})$. By application of rule* [**E**] *we have a derivation for $\{P\} \vdash \mathsf{Exec}_i(\text{Forall } j = 1 + (i-1)n : n.i \text{ do } G)$ $\{\downarrow^i(Q'_i)\}$. Finally, by a simple application of the rule RGroup we get the certified judgment $\{P\} \vdash \mathsf{Group}(G_1 \mid\mid \ldots \mid\mid G_k)$ $\{\sum_{i=1}^{k}(\downarrow^i(Q'_i))\}$. To complete the proof notice that by definition of $\sum$ and $\downarrow^i$, $\sum_{i=1}^{k}(\downarrow^i(\sum_{j'=1+(i-1)n}^{n.i} \mathsf{weak}_j(Q_{j'})))$ is equal to $\downarrow^i(\sum_{j'=1}^{k.n} \mathsf{weak}_j(Q_{j'}))$.*

4.3 Exec Grouping

The execution of the task $\mathsf{Exec}_i(G)$ starts the computation of the subprogram G on the i-th child memory. Since the operation of delegating execution to one of the child nodes may be costly for a parent processor, there is an interest on merging relatively small and independent Exec operations executing in the same node in a single Exec operation.

G' is the result of applying Exec grouping to a program G, if it is defined by substituting every subprogram $\mathsf{Group}(\{\mathsf{Exec}_i(G_1), \ldots, \mathsf{Exec}_i(G_k)\} \cup H)$ in G, where $\{G_1, \ldots, G_k\}$ are independent and maximal subprograms, with the subprogram $\mathsf{Group}(\{\mathsf{Exec}_i(\mathsf{Group}(\{G_1, \ldots, G_k\}))\} \cup H)$.

The commutativity and associativity of the operator $+ : \mathsf{Prop} \rightarrow \mathsf{Prop}$ and the distributivity of the operation $\downarrow^i : \mathsf{Prop} \rightarrow \mathsf{Prop}$ w.r.t. $+$ enable us to prove the following result.

Lemma 9. *Let G' be a program transformed from G by* Exec *grouping. Then, it is possible to translate a certified judgment $\{\Phi\} \vdash G$ $\{\Psi\}$ into the certified judgment $\{\Phi\} \vdash G'$ $\{\Psi\}$.*

Proof. To prove this lemma we show that for a subprogram G of the form $\mathsf{Group}(\{\mathsf{Exec}_i(G_1), \ldots, \mathsf{Exec}_i(G_k)\} \cup H)$ with G_j maximal elements, and certified judgment $\{P\} \vdash G$ $\{Q\}$, the judgment $\{P\} \vdash G'$ $\{Q\}$ is also certified, with G' equal to $\mathsf{Group}(\{\mathsf{Exec}_i(\mathsf{Group}(\{G_1, \ldots, G_k\}))\} \cup H)$. For simplicity, we do not consider the application of the subsumption rule [**CSS**]*, and we assume the existence of the certificate $\mathsf{distrib}_{+,\oplus_i} : \vdash \phi \oplus_i (\psi + \varphi) \Rightarrow (\phi \oplus_i \psi) + (\phi \oplus_i \varphi)$, built by application of the operators of the proof algebra, and by definition of $\oplus_i$. If we consider the last application of rule* [**G**]*, we know there are sets X and H' such that $X \cup \{\mathsf{Exec}_i(G_j) \mid j \in [1, k]\}$ are the maximal elements in H, $H' = H \backslash (X \cup \{G_i\})$, and we have the certified judgments $\{P\} \vdash G_j$ $\{Q_j\}$ for $j \in [1, k]$, $\{P\} \vdash G$ $\{Q_g\}$ for $g \in X$, and $\{\sum_{j \in [1,k]} Q_j + \sum_{g \in X} Q_g\} \vdash \mathsf{Group}(H')$ $\{Q\}$. It can be seen that if $X \cup \{\mathsf{Exec}_i(G_j) \mid j \in [1, k]\}$ is the set of maximal elements in H, then $X \cup \{\mathsf{Exec}_i(\mathsf{Group}(\{G_j) \mid j \in [1, k]\}))\}$ are also the maximal elements in*

$H' \cup X \cup \{\mathsf{Exec}_i(\mathsf{Group}(\{G_j) \mid j \in [1,k]\}))\}$. Hence, if we show that the judgment $\{P\} \vdash \mathsf{Exec}_i(\mathsf{Group}(\{G_j) \mid j \in [1,k]\}))\ \{\sum_{j\in[1,k]} Q_j\}$ is certified, by definition of $\sum$, and application of the rule $[\mathbf{G}]$ we get the desired result. To show that $\{P\} \vdash \mathsf{Exec}_i(\mathsf{Group}(\{G_j) \mid j \in [1,k]\}))\ \{\sum_{j\in[1,k]} Q_j\}$ is a certified judgment, we analyze the derivation of $\{P\} \vdash \mathsf{Exec}_i(G_j)\ \{Q_j\}$ for each $j \in [1,k]$. We have then that $Q_j = P \oplus_i Q'_j$ for some Q'_j, and the certified judgment $\{\uparrow^i(P)\} \vdash G_j\ \{Q'_j\}$. Since G_j are independent subprograms, by application of the rule $[\mathbf{G}]$ we get the certified judgment $\{\uparrow^i(P)\} \vdash \mathsf{Group}(\{G_j \mid j \in [1,k]\})\ \{\sum_{j\in[1,k]} Q'_j\}$, and by application of $[\mathbf{E}]$, the certified judgment $\{P\} \vdash \mathsf{Exec}_i(\mathsf{Group}(\{G_j \mid j \in [1,k]\}))\ \{P \oplus_i \sum_{j\in[1,k]} Q'_j\}$. Finally, from $\mathsf{distrib}_{+,\oplus_i}$ we have a certificate for $P \oplus_i \sum_{j\in[1,k]} Q'_j \Rightarrow \sum_{j\in[1,k]}(P \oplus_i Q'_j)$, which together with an application of rule $[\mathbf{CSS}]$ enables us to certify the judgment $\{P\} \vdash \mathsf{Exec}_i(\mathsf{Group}(\{G_j \mid j \in [1,k]\}))\ \{\sum_{j\in[1,k]} Q_j\}$.

4.4 Copy Grouping

A common property of the execution environments targeted by Sequoia programs is that scheduling several memory transfers simultaneously in a single copy operation is more efficient than executing them independently.

Copy grouping is an optimization that takes advantage of this fact and consists on clustering multiple independent copy operations in a single copy operation.

More precisely, we say that G' is the result of applying copy grouping to a program G, if every subprogram g of the form $\mathsf{Group}(H \cup \{g_1, g_2\})$ is replaced by $g' = \mathsf{Group}(H \cup \{g_{1,2}\})$, such that $g_1 = \mathsf{Copy}(A_1, B_1)$ and $g_2 = \mathsf{Copy}(A_2, B_2)$ are independent and maximal and $g_{1,2}$ merges both copy operations in a single operation $\mathsf{Copy}(A_1, A_2, B_1, B_2)$.

Lemma 10. *Let G' the result of applying copy grouping to a program G. Assume that $\{P\} \vdash G\ \{Q\}$ is a certified judgment. Then, assuming we have a certificate $\mathsf{c} :\vdash T_{g_1}(Q_1) \wedge T_{g_2}(Q_2) \Rightarrow T_{g_{1,2}}(Q_1 + Q_2)$, we can give a certificate for the judgment $\{P\} \vdash G'\ \{Q\}$.*

Proof. *To prove this lemma we show that for every subprogram G of the form $\mathsf{Group}(H \cup \{g_1, g_2\})$ transformed into the program $G' = \mathsf{Group}(H \cup \{g_{1,2}\})$, we can certify the judgment $\{P\} \vdash G'\ \{Q\}$ from a certificate of the judgment $\{P\} \vdash G\ \{Q\}$. For simplicity, we refrain ourselves from considering applications of the subsumption rule $[\mathbf{CSS}]$. If we do not consider $[\mathbf{CSS}]$, the last rule applied for the derivation of $\{P\} \vdash G\ \{Q\}$ is $[\mathbf{G}]$. Then, for some sets X and H', $X \cup \{g_1, g_2\}$ is the set of maximal elements in $H \cup \{g_1, g_2\}$, and $H' = H \setminus X$. In addition, we have the judgments $\{P\} \vdash g_1\ \{Q_1\}$, $\{P\} \vdash g_2\ \{Q_2\}$, and $\{Q_1 + Q_2 + \sum_{g\in X} Q_g\} \vdash \mathsf{Group}(H')\ \{Q\}$ and a judgment $\{P\} \vdash g\ \{Q_g\}$ for every g in X. It can be seen that the dependence conditions do not change when merging the copy operations g_1 and g_2. Hence, $X \cup \{g_{1,2}\}$ is the set of maximal elements in $H \cup \{g_{1,2}\}$. Based on this conditions, and by application of rule $[\mathbf{G}]$ and by associativity of $+$, it sufficient to show that we can certify the judgment $\{P\} \vdash T_{g_{1,2}}\ \{Q_1 + Q_2\}$. To* *this end, since we have certificates for $P \Rightarrow T_{g_1}(Q_1)$ and $P \Rightarrow T_{g_2}(Q_2)$ (the only applicable rules are* **[AB]** *and* **[CSS]***) and the certificate* c*, we can construct a certificate for $P \Rightarrow T_{g_{1,2}}(Q_1 + Q_2)$ to certify the judgment $\{P\} \vdash g_{1,2}\ \{Q_1 + Q_2\}$.*

5 Conclusion

We have used the framework of abstract interpretation to develop a sound proof system to reason about Sequoia programs, and to provide sufficient conditions for the existence of certificate translators. Then, we have instantiated these results to common optimizations described in [12]. Our results lay the foundations for extending Proof-Carrying Code to parallel languages.

There are several directions for future work. First, it would be of interest to investigate weaker definitions of safe programs. The semantics of Sequoia programs, and the definition of safety, rules out the possibility of races, in which two subtasks can write on the same scalar variable or on the same interval of an array. While there may be limited interest to extend the definition of Sequoia programs to support non-deterministic computations, it would seem worthwhile to allow for benign data races, where parallel subtasks are allowed to modify the same variables, with the condition that they do it in an identical manner. This notion of benign data races [13] is also closely related to non-strict and-parallelism, as studied in [9]. In future work, we intend to develop a static analysis that supports a relaxed notion of safety that allows for such benign races, and adapts our analysis and verification framework accordingly.

In a different direction, it would be interesting to develop a prototype implementation of the proof system and to use it for verifying examples of Sequoia programs. We also intend to extend our results on certificate translation by using Sequoia to support parallel executions of sequential programs.

Finally, it would be interesting to see how separation logic compares with our work. In particular, if we can replace the region analysis with classical separation logic [16] or permission-accounting separation logic [4].

References

1. Alpern, B., Carter, L., and Ferrante, J. Modeling parallel computers as memory hierarchies. In *Proc. Programming Models for Massively Parallel Computers*, 1993.
2. G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2006.
3. G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In S. Drossopoulou, editor, *ESOP*, Lecture Notes in Computer Science. Springer, 2008. To appear.
4. R. Bornat, P. O'Hearn, C. Calcagno, and M. Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages*, pages 259–270, New York, NY, USA, 2005. ACM Press.

5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages*, pages 269–282, 1979.
7. W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J. Ho Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *SC*, page 35. ACM, 2003.
8. K. Fatahalian, D. Reiter Horn, T. J. Knight, L. Leem, M. Houston, J. Young Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *SC*, page 83. ACM Press, 2006.
9. M. V. Hermenegildo and F. Rossi. Strict and nonstrict independent and-parallelism in logic programs: Correctness, efficiency, and compile-time conditions. *J. Log. Program.*, 22(1):1–45, 1995.
10. M. Houston, J. Young Park, M. Ren, T. Knight, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. A portable runtime interface for multi-level memory hierarchies. In M. L. Scott, editor, *PPOPP*. ACM, 2008.
11. Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter R. Mattson, and John D. Owens. Programmable stream processors. *IEEE Computer*, 36(8):54–62, 2003.
12. T. J. Knight, J. Young Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In K. A. Yelick and J. M. Mellor-Crummey, editors, *PPOPP*, pages 226–236. ACM, 2007.
13. Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 22–31. ACM, 2007.
14. G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.
15. G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, pages 229–243. Usenix, 1996.
16. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, Copenhagen, Denmark, July 2002. IEEE Press.

A Region analysis

In this section we give the complete definition of the region analyses I^R and I^W, described in Sect. 3.2. First, we need some preliminary definitions.

Given a function $f : {\mathcal{N}_\mathcal{S}}^+ \rightarrow X$ for some set X, we define the functions $\downarrow^i f : {\mathcal{N}_\mathcal{S}}^+ \rightarrow X$ as $(\downarrow^i f)(x) = f(\downarrow^i x)$, and $\uparrow^i f : {\mathcal{N}_\mathcal{S}}^+ \rightarrow X$ as

$$(\uparrow^i f)(x) = \begin{cases} f(\uparrow^i x) & \text{if } x \in \mathrm{dom}(\uparrow^i) \\ \bot & \text{otherwise} \end{cases}$$

Given a set of variables $V \subseteq {\mathcal{N}_\mathcal{S}}^+$ we define $\downarrow^i V$ as $\downarrow^i V = \{\downarrow^i x : x \in V\}$ and $\uparrow^i V$ as $\uparrow^i V = \{\uparrow^i x : x \in V \cap \mathrm{dom}(\uparrow^i)\}$. These definitions of $\uparrow^i$ and $\downarrow^i$ can be extended to functions with domains ${\mathcal{N}_\mathcal{A}}^+$ and $\mathcal{N}^+$, and also to subsets of ${\mathcal{N}_\mathcal{A}}^+$ and $\mathcal{N}^+$.

Interval analysis. We define an interval analysis that assigns to each scalar variable the range of possible values. It serves as a basis for defining the region analysis below, where the results given by this analysis are used to approximate which part of an array is read (or written) by each subtask. That way we can check that two parallel subtasks are not writing in the same part of an array.

We use an instance of the abstract interpretation

$$I^{\mathrm{I}} = \langle A^{\mathrm{I}}, \downarrow, T^{\mathrm{I}}, +, \mathsf{weak}^{\mathrm{I}}, \pi^{\mathrm{I}}, \oplus^{\mathrm{I}}, \rho^{\mathrm{I}} \rangle,$$

where the domain A^{I} is defined as:

$$\begin{aligned} A^{\mathrm{I}} &= (\mathcal{N_S}^+ \to \mathit{Interval}), \\ \mathit{Interval} &= \{(a,b) : a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\}_\perp. \end{aligned}$$

To define the transfer functions, we assume, for simplicity, that scalar functions only take one parameter. However, we also consider a special case of Scalar, where the function is a binary operation, $op \in \{+, -, *\}$. This way we can obtain more accurate results for operations that are common for computing array indices. The transfer functions T_s^{I} are defined as follows:

$$T_s^{\mathrm{I}}(f) = \begin{cases} f \oplus \{a \mapsto (-\infty, \infty)\} & \text{if } s = \mathsf{Scalar}\langle a = f(b) \rangle \\ f \oplus \{c \mapsto [a_1 \; op \; b_1, a_2 \; op \; b_2]\} & \begin{array}{l} \text{if } s = \mathsf{Scalar}\langle c = a \; op \; b \rangle, \\ f(a) = [a_1, a_2], f(b) = [b_1, b_2] \end{array} \\ f & \text{otherwise} \end{cases}$$

The operator $+$ is $\sqcup$, the standard join operator for the interval domain; the operator $\mathsf{weak}_i^{\mathrm{I}}$ is defined as $\mathsf{weak}_i^{\mathrm{I}}(f) = f \oplus \{i \mapsto (-\infty, \infty)\}$; the function ρ^{I} is defined as $\rho^{\mathrm{I}}(f, b) = f$. The operator π_i^{I} is defined as $\pi_i^{\mathrm{I}}(f) = \downarrow^i f$, and the operator $\oplus_i^{\mathrm{I}}$ is defined as $f_1 \oplus_i^{\mathrm{I}} f_2 = \overline{f_1}^i \sqcup \uparrow^i f_2$, where

$$\overline{f}^i(v) = \begin{cases} \perp & \text{if } v = x^{\downarrow^i \downarrow^{j_1} \dots \downarrow^{j_k}}, \text{or } v = x \\ f(v) & \text{otherwise} \end{cases}$$

A satisfaction relation $\models^R$ for I^{I} is defined as follows:

$$\sigma \models^R a = \forall v \in \mathcal{N_S}^+, a_1 \leq \sigma(v) \leq a_2,$$

where $a(v) = [a_1, a_2]$. Note that if $a = \perp$ or $a(v) = \perp$ for some $v \in \mathcal{N_S}^+$, then there is no σ such that $\sigma \models^R a$.

Lemma 11. *The abstract interpretation* $I^{\mathrm{I}} = \langle A^{\mathrm{I}}, \downarrow, T^{\mathrm{I}}, +, \mathsf{weak}^{\mathrm{I}}, \pi^{\mathrm{I}}, \oplus^{\mathrm{I}}, \rho^{\mathrm{I}} \rangle$ *is sound.*

Proof. We need to prove that the I^{I} is consistent. This means proving that the operators defined in I^{I} satisfy the conditions stated in Def. 5. We treat here the cases of $+$, π and $\oplus$:

($+$)**:** Given memories $\sigma, \sigma' \in \mathcal{H}$, and values $a, a' \in A^{\mathrm{I}}$, such that $\sigma \models a$ and $\sigma' \models a'$, we want to prove that $\sigma + \sigma' \models a \sqcup a'$ (since the operator $+$ is defined as $\sqcup$). Notice that for every scalar variable $v \in \mathcal{N_S}^+$, $(\sigma + \sigma')(v) \in \{\sigma(v), \sigma'(v)\}$. The result follows from the fact that $\models$ is an approximation order.

(π)**:** Given a memory $\sigma \in \mathcal{H}$, we have that $\pi_i(\sigma) = \downarrow^i \sigma$. It is easy to see that if $\sigma \models a$, then $\downarrow^i \sigma \models \downarrow^i a$.

($\oplus$)**:** Similar to the previous case. Given memories $\sigma, \sigma' \in \mathcal{H}$, we have that $\sigma \oplus_i \sigma' = \overline{\sigma}^i \sqcup \uparrow^i \sigma'$, where $\overline{\sigma}^i$ is defined the same as above and extended to locations in the obvious way. It is easy to see that if $\sigma \models a$, and $\sigma' \models a'$, then $\overline{\sigma}^i \sqcup \uparrow^i \sigma' \models \overline{a}^i \sqcup \uparrow^i a'$.

□

Region analysis. To define the read and write regions of a task, we will assume that programs are annotated with the results computed with I^{I}. That is, for a given program G, we assume we have a derivation of $\langle \top \rangle \vdash G\langle a \rangle$, and we annotate each subtask G' of G with a_1, written in subscript as G'_{a_1}, if we have a sub-derivation $\langle a_1 \rangle \vdash G'\langle a_2 \rangle$ for some $a_2 \in A^{\mathrm{I}}$. [4] For our region analysis, only the first component of the interval analysis is of interest, since it contains the possible values of the scalar variables before executing the statement.

The read and write regions of a task are defined by two abstract interpretations

$$I^R = \langle D^R, \downarrow, T^R, +, \mathsf{weak}^R, \pi^R, \oplus^R, \rho^R \rangle,$$
$$I^W = \langle D^W, \downarrow, T^W, +, \mathsf{weak}^W, \pi^W, \oplus^W, \rho^W \rangle,$$

given below. The domains D^R and D^W are the same for both frameworks, namely

$$D^R = D^W = (\mathcal{N_A}^+ \to \mathit{Interval}) \times \mathcal{P}(\mathcal{N_S}^+),$$

and $\mathcal{P}(\mathcal{N_S}^+)$ is the lattice of subsets of variables from $\mathcal{N_S}^+$. The first component represents, for each array variable, the range used; and the second component represents the set of scalar variables used. The transfer functions T_s^R and T_s^W are defined below. For each $s \in \mathsf{atomStmt}$, we assume we have $a \in A^{\mathrm{I}}$ computed using I^{I} as mentioned above. For each scalar variable m, we write m^1 and m^2 to mean the possible range of m, i.e., $a(m) = [m^1, m^2]$. The functions T_s^R and T_s^W are defined depending on the statement s with the following rules. Again, for simplicity, we consider that Scalar and Kernel functions take only one parameter, and we only consider one case of Copy, since the others are similar.

$$T_s^R(d_a, d_s) = \begin{cases} (d_a, d_s \cup \{b\}) & \text{if } s = \mathsf{Scalar}\langle a = f(b)\rangle \\ (d_a \sqcup \{A \mapsto [s_A^1, e_A^2]\}, d_s \cup \{s_A, e_A, s_B\}) & \text{if } s = \mathsf{Copy}^{\uparrow}(A[s_A, e_A], B[s_B]) \\ (d_a \sqcup \{B \mapsto [s_B^1, e_B^2]\}, d_s \cup \{s_A, e_A, s_B, e_B\}) & \text{if } s = \mathsf{Kernel}\langle A[s_A, e_A] = f(B[s_B, e_B])\rangle \end{cases}$$

[4] Note that this is only a technical detail of our presentation. We could have as well defined the region analysis and the interval analysis simultaneously.

$$T_s^W(d_a, d_s) = \begin{cases} (d_a, d_s \cup \{a\}) & \text{if } s = \mathsf{Scalar}\langle a = f(b)\rangle \\ (d_a \sqcup \{B^{\uparrow} \mapsto [s_B^1, s']\}, d_s) & \text{if } s = \mathsf{Copy}^{\uparrow}(A[s_A, e_A], B[s_B]), \text{ and} \\ & s' = s_B^2 + e_A^2 - s_A^1 \\ (d_a \sqcup \{A \mapsto [s_A^1, e_A^2]\}, d_s) & \text{if } s = \mathsf{Kernel}\langle A[s_A, e_A] = f(B[s_B, e_B])\rangle \end{cases}$$

The operator $+$ is $\sqcup$, weak_i^R and weak_i^W are defined as $\mathsf{weak}_i^R(d_a, d_s) = \mathsf{weak}_i^W(d_a, d_s) = (d_a, d_s \setminus \{i\})$; and finally $\rho^R(d, b) = \rho^W(d, b) = d$. The functions π_i^R and π_i^W are defined as $\pi_i^R(d_a, d_s) = \pi_i^W(d_a, d_s) = (\downarrow^i d_a, \uparrow^i d_s)$, and $(d_a^1, d_s^1) \oplus_i^R (d_a^2, d_s^2) = (d_a^1, d_s^1) \oplus_i^W (d_a^2, d_s^2) = (\overline{d_a^1}^i \sqcup \uparrow^i d_a^2, d_s^1 \cup d_s^2)$.

We prove soundness of the abstract interpretations I^R and I^W, as stated in Lemma 2 and Lemma 3.

Proof (Lemma 2). Given a program G, regions $R_1, R_2 \in D^R$ with $\langle R_1 \rangle \vdash_R G \ \langle R_2 \rangle$, and $\sigma_1, \sigma_2 \in \mathcal{H}$ with $\sigma_1 \approx_{R_2} \sigma_2$. If $\sigma_1 \vdash G \rightarrow \sigma_1'$ and $\sigma_2 \vdash G \rightarrow \sigma_2'$, we want to prove that

I. $R_1 \sqsubseteq R_2$,
II. $\mathit{Modified}(\sigma_1, \sigma_1') = \mathit{Modified}(\sigma_2, \sigma_2')$, and
III. $\sigma_1' \approx_R \sigma_2'$, where $R = R_2 \cup \mathit{Modified}(\sigma_1, \sigma_1')$.

From this, the conclusion of Lemma 2 follows immediately.

We prove I by induction on the derivation of $\langle R_1 \rangle \vdash_R G \ \langle R_2 \rangle$. All cases are simple.

II and III are proved simultaneously by induction on the derivation of $\langle R_1 \rangle \vdash_R G \ \langle R_2 \rangle$. We show a few cases.

- Last rule applied is **[G]**. Let us assume that $G = \mathsf{Group}(H)$, and X is the set of maximal elements in H with $H = X \cup H'$. For every $g \in X$ there exists $R_g \in D^R$ such that $\langle R_1 \rangle \vdash_R g \ \langle R_g \rangle$. Also, $\langle \sum_{g \in X} R_g \rangle \vdash_R \mathsf{Group}(H') \ \langle R_2 \rangle$. From the derivation of $\sigma_1 \vdash \mathsf{Group}(H) \rightarrow \ \sigma_1'$, we have that, for each $g \in X$, there exists $\sigma_1^g \in \mathcal{H}$ such that $\sigma_1 \vdash g \rightarrow \ \sigma_1^g$. Analogously, for each $g \in X$, there exists $\sigma_2^g \in \mathcal{H}$ such that $\sigma_2 \vdash g \rightarrow \ \sigma_2^g$. Since $R_1 \sqsubseteq R_g \sqsubseteq R_2$, we have that $\sigma_1 \approx_{R_g} \sigma_2$. By IH, we have, for each $g \in X$, $\mathit{Modified}(\sigma_1, \sigma_1^g) = \mathit{Modified}(\sigma_2, \sigma_2^g)$, and $\sigma_1^g \approx_{S_g} \sigma_2^g$, where $S_g = R_g \cup \mathit{Modified}(\sigma_1, \sigma_1^g)$. This means that $\sigma_1^g \approx_{T_g} \sigma_2^g$, where $T_g = R_2 \cup \mathit{Modified}(\sigma_1, \sigma_1^g)$, since initially $\sigma_1 \approx_{R_2} \sigma_2$.
  Hence, $\sum_{g \in X} \sigma_1^g \approx_R \sum_{g \in X} \sigma_2^g$, where $R = R_2 \cup \bigcup_{g \in X} \mathit{Modified}(\sigma_1, \sigma_1^g)$. The result follows from applying IH to $\mathsf{Group}(H')$.
- Last rule applied is **[FF]**. For each $k \in [m, n]$ we have $\langle T_{j:=k}(R_1) \rangle \vdash_R G \langle R_k \rangle$. From the semantics, we have $\sigma_1[j \mapsto k] \vdash G \rightarrow \ \sigma_1^k$ and $\sigma_2[j \mapsto k] \vdash G \rightarrow \ \sigma_2^k$. By IH, and proceed similarly as with the case for **[G]**.
- Last rule applied is **[AF]**. We only consider the case where $G = \mathsf{Copy}(A[s_A, e_A], B[s_B])$, with f_G being the result of the interval analysis for the statement G. We have that $R_2 = R_1 \sqcup \{A \mapsto [s_A^1, e_A^2]\}$, where $f_G(s_A) = [s_A^1, s_A^2]$ and $f_G(e_A) = [e_A^1, e_A^2]$. The range $[s_A^1, e_A^2]$ is an over-approximation of the actual range that is read from array A, hence the results follows.

Proof (Lemma 3). Assume a program G, regions $W_1, W_2 \in D^W$ with $\langle W_1 \rangle \vdash_W G \langle W_2 \rangle$. If $\sigma \vdash G \to \sigma'$, we want to prove that

I. $W_1 \sqsubseteq W_2$,
II. $\mathit{Modified}(\sigma, \sigma') \subseteq W_2$

To prove I, we proceed by induction on $\langle W_1 \rangle \vdash_W g \langle W_2 \rangle$. All cases are simple.

To prove I, we also proceed by induction on $\langle W_1 \rangle \vdash_W g \langle W_2 \rangle$. We consider a few cases:

– Last rule applied is [**G**]. Let us assume that $G = \mathsf{Group}(H)$, and X is the set of maximal elements in H with $H = X \cup H'$. For every $g \in X$ there exists $W_g \in D^W$ such that $\langle W_1 \rangle \vdash_R g \langle W_g \rangle$. From the derivation of $\sigma \vdash \mathsf{Group}(H) \to \ \sigma_g$, we have that, for each $g \in X$, there exists $\sigma_g \in \mathcal{H}$ such that $\sigma \vdash g \to \ \sigma_g$. By IH, we have $\mathit{Modified}(\sigma, \sigma_g) \subseteq W_g$, for all $g \in X$. Again, by IH, $\mathit{Modified}(\sigma', \sigma_g) \subseteq W_2$. Since $W_1 \sqsubseteq W_g \sqsubseteq W_2$ for each $g \in X$, the result follows from the fact that

$$\mathit{Modified}(\sigma, \sigma') \subseteq \bigcup_{g \in X} (\mathit{Modified}(\sigma, \sigma_g) \cup \mathit{Modified}(\sigma_g, \sigma')) \ .$$

– Last rule applied is [**AF**]. We only consider the case where $G = \mathsf{Copy}(A[s_A, e_A], B[s_B])$, with f_G being the result of the interval analysis for the statement G. We have that $W_2 = W_1 \sqcup \{B \mapsto [s_B^1, s_B^2 + e_A^2 - s_A 1]\}$. The range $[s_B^1, s_B^2 + e_A^2 - s_A 1]$ is an over-approximation of the actual range that is written from array B, hence the results follows.

B Weakest pre-condition

In this section we present the complete definition of the weakest pre-condition transformers for atomic statements presented in Sect. 3.3.

The transfer function for copy to the parent memory is

$$\{P\} \vdash \mathsf{Copy}^{\uparrow}(A[m,n], B[k]) \ \{Q\},$$

where $P = Q[^{B^{\uparrow} \oplus [k, k+n-m] \mapsto A[m]} /_{B^{\uparrow}}]$. Given two arrays A and B and scalars m, n and k, we write $B \oplus [m,n] \mapsto A[k]$ to mean the array obtained from B by replacing the range $[m,n]$ with values from $A[k, k+n-m]$. The transfer functions for other types of copy (from the parent memory, and intra-memory) are treated similarly.

The case of Kernel and Scalar are similar, so we will only consider the former. For each function appearing in a Kernel or Scalar statement, we assume that we have a pre- and a post-condition. To illustrate, let us consider a function f with one parameter of array-type, and also returning and array. The pre- and post-condition of f are first order formulae. Also, the pre-condition can refer to the input variable (with arg), to the range of indices read of the argument (with $\mathsf{arg_{start}}$ and $\mathsf{arg_{end}}$), and to the range of indices written of the result array (with

$\mathsf{res}_{\mathsf{start}}$ and $\mathsf{res}_{\mathsf{end}}$). The post-condition can also refer to the result variable (with res).

Lets assume that f doubles each value of the input array, so the pre- and post-condition could be written as follows:

$$\text{Pre} = (\mathsf{res}_{\mathsf{end}} - \mathsf{res}_{\mathsf{start}} = \mathsf{arg}_{\mathsf{end}} - \mathsf{arg}_{\mathsf{start}})$$
$$\text{Post} = \forall x, \mathsf{res}_{\mathsf{start}} \leq x \leq \mathsf{res}_{\mathsf{end}} \Rightarrow \mathsf{res}[x] = 2 \cdot \mathsf{arg}[x - \mathsf{res}_{\mathsf{start}} + \mathsf{arg}_{\mathsf{start}}]$$

The pre-condition states that the input and output array must have the same length, and the post-condition states that all values of the result array are the double of the values of the argument, for the corresponding ranges.

The transfer function for a function f with pre-condition *Pre* and post-condition *Post* is the following:

$$\{P\} \vdash \mathsf{Kernel}\langle A[k,l] = f(B[m,n])\rangle \ \{Q\},$$

where

$$\begin{aligned} P &= P' \land \forall \mathsf{res}, Q' \Rightarrow Q[{}^{\mathsf{res}}\!/_{A}] \\ P' &= \mathit{Pre}[{}^{B}\!/_{\mathsf{arg}}][{}^{m}\!/_{\mathsf{arg}_{\mathsf{start}}}][{}^{n}\!/_{\mathsf{arg}_{\mathsf{end}}}][{}^{k}\!/_{\mathsf{res}_{\mathsf{start}}}][{}^{l}\!/_{\mathsf{res}_{\mathsf{end}}}] \\ Q' &= \mathit{Post}[{}^{B}\!/_{\mathsf{arg}}][{}^{m}\!/_{\mathsf{arg}_{\mathsf{start}}}][{}^{n}\!/_{\mathsf{arg}_{\mathsf{end}}}][{}^{k}\!/_{\mathsf{res}_{\mathsf{start}}}][{}^{l}\!/_{\mathsf{res}_{\mathsf{end}}}] \end{aligned}$$

C Example program

In this section we present the complete verification of the program presented in Sect. 3.4. We recall the code of the program:

$$G_{\text{Add}} = \mathsf{Group}(G_1 \parallel \ldots \parallel G_n)$$

$$\begin{aligned} G_i &= \mathsf{Exec}_i(\mathsf{Group}(\text{InitArgs}; \text{Add}; \text{CopyZC})) \\ \text{InitArgs} &= \text{CopyAX} \parallel \text{CopyBY} \\ \text{CopyAX} &= \mathsf{Copy}^{\downarrow}(A[i\,S, (i+1)S], X[0,S]) \\ \text{CopyBY} &= \mathsf{Copy}^{\downarrow}(B[i\,S, (i+1)S], Y[0,S])) \\ \text{Add} &= \mathsf{Kernel}\langle Z[0,S] = \text{VectAdd}(X[0,S], Y[0,S])\rangle \\ \text{CopyZC} &= \mathsf{Copy}^{\uparrow}(Z[0,S], C[i\,S, (i+1)S]) \end{aligned}$$

As we mention in Appendix B, we assume a pre- and a post-condition for each function used in a Kernel or Scalar. In the case of VectAdd we have the following:

$$\text{Pre}_{\text{VectAdd}} = (\mathsf{res}_{\mathsf{end}} - \mathsf{res}_{\mathsf{start}} = \mathsf{arg}_{1,\mathsf{end}} - \mathsf{arg}_{1,\mathsf{start}} = \mathsf{arg}_{2,\mathsf{end}} - \mathsf{arg}_{2,\mathsf{start}})$$

$$\begin{aligned} \text{Post}_{\text{VectAdd}} = \forall k.\ &\mathsf{res}_{\mathsf{start}} \leq k < \mathsf{res}_{\mathsf{end}} \Rightarrow \\ &\mathsf{res}[k] = \mathsf{arg}_1[k - \mathsf{res}_{\mathsf{start}} + \mathsf{arg}_{1,\mathsf{start}}] + \mathsf{arg}_2[k - \mathsf{res}_{\mathsf{start}} + \mathsf{arg}_{2,\mathsf{start}}] \end{aligned}$$

Note that we require that the array arguments have the same length. We want to verify that the program satisfies the following judgment:

$$\{\mathsf{true}\} \vdash G_{\text{Add}}\ \{\text{Post}\} \tag{2}$$

$$\text{Post} = \forall k.\ 0 \leq k < n\,S \Rightarrow C[k] = A[k] + B[k] \tag{3}$$

First, we derive, for each G_i, the following judgment:

$$\{\mathsf{true}\} \vdash G_i\ \{\text{Post}_i\} \tag{4}$$

$$\text{Post}_i = \forall k, i\,S \leq k < (i+1)S \Rightarrow C[k] = A[k] + B[k]$$

Applying the rule for atomic statement to the last action in the program we have:

$$\{Q_2\} \vdash \text{CopyZC}\ \{\text{Post}_i\}$$
$$Q_2 = \forall k, i\ S \leq k < (i+1)S \Rightarrow (C \oplus [i\,S, (i+1)S] \mapsto Z[0])[k] = A[k] + B[k]$$

Applying subsumption we obtain the judgment:

$$\{Q_2'\} \vdash \text{CopyZC}\ \{\text{Post}_i\} \tag{5}$$

$$Q_2' = \forall k.\ 0 \leq k < S \Rightarrow Z[k] = A[k + i\,S] + B[k + i, S]$$

since $Q_2' \Rightarrow Q_2$. Applying the atomic statement rule once again:

$$\{Q_3\} \vdash \text{Add}\ \{Q_2'\} \tag{6}$$

$$\begin{aligned} Q_3 = (S = S \wedge S = S \wedge S = S) \wedge \\ &(\forall R.\ (\forall k.\ 0 \leq k < S \Rightarrow R[k] = X[k] + Y[k]) \Rightarrow \\ &\qquad (\forall k.\ 0 \leq k < S \Rightarrow R[k] = A[k + i\,S] + B[k + i\,S])) \end{aligned}$$

Applying the rule for Group to (5) and (6), we obtain:

$$\{Q_3\} \vdash \mathsf{Group}(\text{Add}; \text{CopyZC})\ \{\text{Post}_i\} \tag{7}$$

On the other hand we have that

$$\{Q_5\} \vdash \text{CopyAX}\ \{Q_4\} \tag{8}$$

$$Q_5 = \forall k.\ 0 \leq k < S \Rightarrow (X \oplus [0, S] \mapsto A[i\,S])[k] = A[k + i\,S]$$
$$Q_4 = \forall k.\ 0 \leq k < S \Rightarrow X[k] = A[k + i\,S]$$

Applying the subsumption rule we obtain

$$\{\mathsf{true}\} \vdash \text{CopyAX}\ \{Q_4\} \tag{9}$$

since $\mathsf{true} \Rightarrow Q_5$. Analogously, we have the following judgment:

$$\{\mathsf{true}\} \vdash \text{CopyBY } \{Q_6\} \tag{10}$$

$$Q_6 = \forall k.\ 0 \leq k < S \Rightarrow Y[k] = B[k + i\ S]$$

Since $Q_4 \wedge Q_6 \Rightarrow Q_3$, we have, applying the subsumption rule on (7),

$$\{Q_4 \wedge Q_6\} \vdash \mathsf{Group}(\text{Add}; \text{CopyZC})\ \{\text{Post}_i\} \tag{11}$$

We have $Q_4 + Q_6 \Rightarrow Q_4 \wedge Q_6$. Hence, applying the rule for Group to (9), (10), and (11) we obtain

$$\{\mathsf{true}\} \vdash \mathsf{Group}(\text{InitArgs} \parallel (\text{Add}; \text{CopyZC}))\ \{\text{Post}_i\}$$

which is the desired result.