Certified Reasoning in Memory Hierarchies*

Gilles Barthe[1,2], César Kunz[2], and Jorge Luis Sacchini[2]

[1] IMDEA Software
Gilles.Barthe@imdea.org
[2] INRIA Sophia Antipolis - Méditerranée
{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.

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 [1,7,11], 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. Thus, programming languages for hierarchical memories are designed to exploit the memory hierarchy and are used for programs that require intensive computations on large amounts of data. Languages for hierarchical memories differ from general-purpose concurrent languages in their intent, and in their realization; in particular, such languages are geared towards deterministic programs and do not include explicit primitives for synchronization (typically programs will proceed by dividing computations between a number of cooperating subtasks, that operate on disjoints subsets of the memory).

As programming languages for high performance are gaining wide acceptance, there is an increasing need to provide analysis and verification methods to help

* This work is partially supported by the EU project MOBIUS.

G. Ramalingam (Ed.): APLAS 2008, LNCS 5356, pp. 75–90, 2008.

developers write, maintain, and optimize their high-performance applications. However, verification methods have been seldom considered in the context of high-performance computing. The purpose of this paper is to show for a specific example that existing analysis and verification methods can be adapted to hierarchical languages and are tractable. We focus on the Sequoia programming language [8,12,10], which is designed to program efficient, portable, and reliable applications for hierarchical memories. We adopt the framework of abstract interpretation [5,6], and develop methods to prove and verify the correctness of Sequoia programs. Our methods encompass the usual automated and interactive verification techniques (static analyses and program logics) as well as methods to transform correctness proofs along program optimizations, which are of interest in the context of Proof Carrying Code [14]. In summary, the main technical contributions are: i) a generic, sound, compositional proof system to reason about Sequoia programs (Sect. 3.1); ii) a sound program logic derived as an instance of the generic proof system (Sect. 3.2); iii) algorithms that transform proofs of correctness along with program optimizations such as SPMD distribution or grouping of tasks [12] (Sect. 4).

2 A Primer on Sequoia

Sequoia [8,12,10] is a language for developing portable and efficient parallel programs for hierarchical memories. It is based on a small set of operations over a hierarchical memory, such as communication, memory movement and computation. Computations are organized into self-contained units, called tasks. Tasks can be executed in parallel at the same level of the memory hierarchy, on a dedicated address space, and may rely on subtasks that perform computations on a lower level (and in practice smaller and faster) fragment of the hierarchical memory (i.e., a subtree).

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 ($\mathbb{Z}$) or booleans ($\mathbb{B}$). Besides, locations are either scalar variables, or arrays elements of the form $A[i]$ where A is an array and i is an index. 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 memory modules 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:*

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

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 execution mechanism consists on splitting a task on smaller subtasks that operate on a dedicated copy of a memory subtree. Upon completion of

each task, the initial memory tree is updated with the final state of each locally modified memory copy. However, inconsistencies may arise since, a priori, we cannot require subtasks to modify disjoint portions of the memory. Then, an auxiliary operator $+_\mu$ is defined to capture, as undefined, those portions of the state that are left with inconsistent values. The operator $+_\mu : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, indexed by a memory $\mu \in \mathcal{M}$, is formally defined as:

$$(\mu_1 +_\mu \mu_2)x = \begin{cases} \mu_1 x & \text{if } \mu_2 x = \mu x, \text{ else} \\ \mu_2 x & \text{if } \mu_1 x = \mu x \\ \text{undefined} & \text{otherwise .} \end{cases}$$

Note that the operator $+_\mu$ is partial, and the result is undefined if both μ_1 and μ_2 modify the same variable.

The operator $+_\mu$ is generalized over memory hierarchies in $\mathcal{T}$ and sequences $\vec{\mu} \in \mathcal{M}^\star$, where $\sum^\mu_{1 \le i \le n} \mu_i = (((\mu_1 +_\mu \mu_2) +_\mu \mu_3) +_\mu \ldots +_\mu \mu_n)$.

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:*

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

where a, b are scalar variables, m, n are scalar constants, A, B are array variables, cond is a boolean expression, and H is a dependence graph of programs. We use the operators $\|$ and ; as a syntactic representation (respectively parallel and sequential composition) of the dependence graph composing a Group *task.*

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

Semantics. We now turn to the semantics of programs; in contrast to the original definition [12], our semantics is syntax-directed. We motivate this slight modification at the end of the paragraph.

The semantics of a program G is defined by a judgment $\sigma \vdash G \to \sigma'$ 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. Abusing nomenclature, we refer to elements of $\mathcal{H}$ as memories. The meaning such a judgment is that the evaluation of G with initial memory σ terminates with final memory σ'. Note that for a specific architecture, the shape of the memory hierarchy (that is, the shape of the tree structure) is fixed and does not change with the execution of a program.

To manipulate elements in $\mathcal{H}$, 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

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

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

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

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

$$\frac{\begin{array}{c} \forall j \in [m, n] \neq \emptyset.\ \mu_p, \langle \mu[i \mapsto j], \vec{\tau} \rangle \vdash G \rightarrow \mu_p^j, \langle \mu^j, \vec{\tau}^j \rangle \\ \mu_p, \langle \mu, \vec{\tau} \rangle \vdash \mathsf{Forall}\ i = m : n\ \mathsf{do}\ G \rightarrow \sum_{m \leq j \leq n}^{\mu_p, \langle \mu, \tau \rangle} (\mu_p^j, \langle \mu^j[i \mapsto \mu(i)], \vec{\tau}^j \rangle) \end{array}}{\begin{array}{c} \pi_i(\mu, \tau) \vdash G \rightarrow \mu', \tau' \\ \mu, \tau \vdash \mathsf{Exec}_i(G) \rightarrow (\mu, \tau) \oplus_i (\mu', \tau') \end{array}}$$

Fig. 1. Sequoia program semantics (excerpt)

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 $j \neq i$.

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

We briefly comment on the semantics—the omitted rules are either the usual ones (conditionals) or similar to other rules (copy).

The constructs $\mathsf{Copy}^{\downarrow}(\vec{A}, \vec{B})$ and $\mathsf{Copy}^{\uparrow}(\vec{A}, \vec{B})$ are primitives that enable data to migrate along the tree structure, from the parent memory to the root of the child memory hierarchy and conversely; and $\mathsf{Copy}(\vec{A}, \vec{B})$ represents an intra-memory copy in the root of the child memory hierarchy. Only the rule for $\mathsf{Copy}^{\uparrow}$ is shown in Fig. 1, since the others are similar.

The constructs $\mathsf{Kernel}\langle A = f(B_1, \ldots, B_n) \rangle$ and $\mathsf{Scalar}\langle a = f(b_1, \ldots, b_n) \rangle$ 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 same happens if, in the rules for Group and Forall, the addition of memories is not defined; that is, the program gets stuck.

The construct $\mathsf{Forall}\ i = m : n\ \mathsf{do}\ G$ executes in parallel $n - m + 1$ instances of G with a different value of i, and merges the result. Progress is made only if the instances manipulate pairwise distinct parts of the memory, otherwise the memory after executing the Forall construct is undefined. The rule in Fig. 1 considers exclusively the case where $m \leq n$, otherwise the memory hierarchy remains unchanged. The construct $\mathsf{Exec}_i(G)$ spawns a new computation on the i-th subtree of the current memory.

Finally, the construct $\mathsf{Group}(H)$ executes the set X of maximal elements of the dependence graph in parallel, and then merges the result before recursively

executing $\mathsf{Group}(H \setminus X)$. A rule not shown in Fig. 1 states that if $H = \emptyset$ the state is left unchanged.

We conclude this section with a brief discussion of the difference between our semantics and the original semantics for Sequoia defined in [12]. First, we define the notion of safety. A Sequoia program is safe if its parallel subtasks are independent, i.e. if they modify disjoint regions of the memory hierarchy. [1]

The original semantics and our semantics differ in the rule for $\mathsf{Group}(H)$: whereas we require that X is the complete set of maximal elements, and is thus uniquely determined from the program syntax, the original semantics does not require X to be the complete set of maximal elements, but only a subset of it. Therefore, our semantics is a restriction of the original one. However, the semantics are equivalent in the sense that our semantics can simulate any run of the unrestricted semantics—provided the program is safe.

Checking that a program is safe can be achieved by an analysis that over-approximates the regions of the memory that each subtask reads and writes. A program is safe if, for parallel subtasks, these regions do not overlap. For lack of space, we do not give the complete definition the analysis.

For safe programs, the order of execution of parallel subtasks does not affect the final result and both semantics coincide. Note that the intention of Sequoia is to write safe programs; if a program is not safe, its execution might get stuck because of two (or more) subtasks writing to the same location.

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 a program logic as an instance of our proof system, and show its soundness.

3.1 Program Analysis

We develop our work using a mild generalization of the framework of abstract interpretation, in which the lattice of abstract elements forms a preorder domain.[2] We also have specific operators over the abstract domain for each type of program, as shown below.

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

- $A = \langle A, \sqsubseteq, \sqsupseteq, \sqcup, \sqcap, \top, \bot \rangle$ *is a lattice of abstract states;*

[1] This notion of safety is similar to the notion of strict and-parallelism of logic programming [9].

[2] A preorder over A is a reflexive and transitive binary relation, whereas a partial order is an antisymmetric preorder. We prefer to use preorders 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.

- *for each* $s \in \mathsf{atomStmt}$*, a relation* $T_s \subseteq A \times A$*;*
- $+_A : A \times A \to A$*;*
- *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* $\oplus_i^A : A \times A \to A$*;*
- $\rho : A \times \mathsf{Bool} \to A$*, where* Bool *is the set of boolean expressions.*

Intuitively, for each rule of the semantics, we have a corresponding operator that reflect the changes of the memory on the abstract domain. For each atomic statement s, the relation T_s characterizes the effect of the atomic semantic operation on the abstract domain. A particular instance of T that we usually consider is when s is a scalar assignment, i.e., $T_{i:=j}$, where $i \in \mathcal{N}_\mathcal{S}$ and $j \in \mathbb{Z}$. Note that we don't use the common *transfer functions* to define the abstract operators regarding atomic statements. Instead, we use relations, which encompasses the use of the more typical *backward* or *forward* functions. We can consider backward transfer functions by defining $a\ T_s\ b$ as $a = f_s(b)$ for a suitable f_s (in fact, this is the case for our definition of the verification framework), and forward transfer functions by defining $a\ T_s\ b$ as $b = g_s(a)$ for a suitable g_s. Also, atomic statements include Kernel and Scalar operations that can be arbitrarily complex and whose behavior can be better abstracted in a relation. For instance, in the case of verification, we will require that these atomic statements be specified with pre and postconditions that define the relation.

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 value of the iteration variable is not relevant. To give more intuition about this operator, consider, for instance, that A is the lattice of first-order formulae (as is the case of the verification framework of Sect. 3.2), then $\mathsf{weak}_i(a)$ is defined as $\exists i.a$. If A has the form $\mathcal{N}_\mathcal{S} \to D$, where D is a lattice, then $\mathsf{weak}_i(a)$ can be defined as $a \sqcup \{i \to \top\}$, effectively removing any condition on i.

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. It can be simply defined as $\rho(a, b) = a$, but this definition does not take advantage of knowing that b is true. If we have an expressive domain we can find a value that express this; for instance, in the lattice of logic formulae, we can define $\rho(a, b) = a \wedge b$.

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. Basically, it states that satisfiability is preserved for each operator of the abstract interpretation. Note that we can also restate these lemmas and definitions in terms

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

$$\frac{}{\langle a\rangle \vdash \mathsf{Group}(\emptyset)\,\langle a\rangle}\ [\mathbf{G}_\emptyset] \qquad \frac{s \in \mathsf{atomStmt} \quad a\ T_s\ a'}{\langle a\rangle \vdash s\,\langle a'\rangle}\ [\mathbf{A}] \qquad \frac{\langle \pi_i(a)\rangle \vdash G\,\langle a'\rangle}{\langle a\rangle \vdash \mathsf{Exec}_i(G)\,\langle a \oplus_i a'\rangle}\ [\mathbf{E}]$$

$$\frac{\forall j,\ m \le j \le n \qquad a\ T_{i:=j}\ a_j \qquad \langle a_j\rangle \vdash G\,\langle a'_j\rangle}{\langle a\rangle \vdash \mathsf{Forall}\ i = m : n\ \mathsf{do}\ G\,\langle \textstyle\sum_{j=m}^{n} \mathsf{weak}_i(a'_j)\rangle}\ [\mathbf{F}]$$

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

Fig. 2. Program analysis rules

of Galois connections, since we can define a Galois connection from the relation $\models$ by defining $\gamma : A \to \mathcal{H}$ as $\gamma(a) = \{\sigma \in \mathcal{H} : \sigma \models a\}$.

Definition 5. *The abstract interpretation $I = \langle A, T, +, \mathsf{weak}, \pi, \oplus, \rho\rangle$ is consistent if the following holds for every $\sigma, \sigma' \in \mathcal{H}$, $a, a_1, a_2 \in A$, $\mu, \mu_p \in \mathcal{M}$, $\tau \in \mathcal{T}$ and $cond \in \mathsf{Bool}$:*

- *for every $s \in \mathsf{atomStmt}$, if $\sigma \vdash s \to \sigma'$, $\sigma \models a$ and $a\ T_s\ a'$, then $\sigma' \models a'$;*
- *if $\sigma_1 \models a_1$ and $\sigma_2 \models a_2$ then $\sigma_1 +_\sigma \sigma_2 \models a_1 + a_2$;*
- *if $\mu_p, \langle \mu, \tau\rangle \models a$, then for all $k \in \mathbb{Z}$ we have $\mu_p, \langle \mu[i \mapsto k], \tau\rangle \models \mathsf{weak}_i(a)$;*
- *if $\sigma \models a$ then $\pi_i(\sigma) \models \pi_i(a)$;*
- *if $\sigma \models a$ and $\sigma' \models a'$, then $\sigma \oplus_i \sigma' \models a \oplus_i a'$;*
- *if $\sigma \models a$ and $\sigma \models_{\mathsf{Bool}} cond$, then $\sigma \models \rho(a, cond)$.*[3]

Given an abstract interpretation I, a judgment is a tuple $\langle a\rangle \vdash_I G\,\langle a'\rangle$, where G is a program and $a, a' \in A$. We will omit the reference to I when it is clear from the context. A judgment is *valid* if it is the root of a derivation tree built using the rules in Fig. 2. The interpretation of a valid judgment $\langle a\rangle \vdash G\,\langle a'\rangle$ is that executing G in a memory that satisfies a, we end up in a memory satisfying a'. The following lemma claims that this is case, provided I is consistent.

Lemma 1 (Analysis Soundness). *Let G be a Sequoia program and assume that $I = \langle A, T, +, \mathsf{weak}, \pi, \oplus, \rho\rangle$ 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 $\sigma' \models a'$.*

3.2 Program Verification

We now define a verification framework $I = \langle \mathsf{Prop}, T, +_{\mathsf{Prop}}, \mathsf{weak}, \pi, \oplus, \rho\rangle$ as an instance of the abstract interpretation, where Prop is the lattice of first-order formulae. Before defining I, we need some preliminary definitions.

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

The extended set of scalar names, ${\mathcal{N}_{\mathcal{S}}}^+$, is defined as

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

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, as is shown by the following definition. 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:

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

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

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

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

Given a formula ϕ, we obtain $\downarrow^i\phi$ 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 +. To define the operator + we require that each subprogram comes annotated with the sets SW and AW specifying, respectively, the scalar variables and the array ranges that it may modify. Given two programs G_1 and G_2 annotated, respectively, with the modifiable regions SW_1, AW_1 and SW_2, AW_2, and the postconditions Q_1 and Q_2, we define $Q_1 + Q_2$ as $Q_1' \wedge Q_2'$, where Q_1' is the result of existentially quantifying in Q_1 the variables that may be modified by G_2. More precisely, $Q_1' = \exists X'.\ Q_1[{}^{X'}/_{X}] \wedge \bigwedge_{A[m,n] \in AW_1} A'[m,n] = A[m,n]$, X representing the set of scalar and array variables in $SW_2 \cup AW_2$ and X' a set of fresh variables (and similarly with Q_2).

To explain the intuition behind this definition, assume two tasks G_1 and G_2 that execute in parallel with postconditions Q_1 and Q_2. After verifying that each G_i satisfies the postcondition Q_i, one may be tempted to conclude that after executing both tasks, the resulting state satisfies $Q_1 \wedge Q_2$. The reason for which we do not define $Q_1 + Q_2$ simply as $Q_1 \wedge Q_2$ is that while Q_1 may be true after executing G_1, Q_1 may state conditions over variables that are not modified by G_1 but are modified by G_2. Then, since from the definition of the operator + in the semantic domain the value of a variable not modified by G_1 is overwritten with a new value if modified by G_2, Q_1 may be false in the final memory state after executing G_1 and G_2 in parallel.

For the definition of $Q_1 + Q_2$ to be sound we require the annotations SW_1, AW_1 and SW_2, AW_2 to be correct. For this, we can use a static analysis or generate additional proof obligations to validate the program annotations. However, for space constraints and since such analysis can be applied earlier and independently of the verification framework, we do not consider this issue. Certainly, the applicability of the logic is limited by the precision of such static analysis.

We generalize the operator + for a set of postconditions $\{\phi_i\}_{i\in I}$ and a set 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 $\phi'_i = \exists X'.\ \phi_i[{}^{X'}/_{X}] \wedge \bigwedge_{A[m,n]\in AW_i} A[m,n] = A'[m,n]$, s.t. X represents every scalar or array variable in $\{SW_j\}_{j\neq i\in I}$ or $\{AW_j\}_{j\neq i\in I}$, and X' a set of fresh variables. 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:

- for each $s \in \mathsf{atomStmt}$, the relation T_s is defined from the weakest precondition transformer wp_s, as $\mathsf{wp}_s(\phi)\ T_s\ \phi$ for every logic formula ϕ. For Kernel and Scalar statements, we assume that we have a pre and postcondition specifying their behavior (the definition of $\{\mathsf{wp}_s\}_{s\in\mathsf{atomStmt}}$ is standard);
- $\mathsf{weak}_i(\phi) = \exists i.\phi$, where $i \in \mathcal{N}_{\mathcal{S}}{}^{+}$;
- $\pi_i(\phi) = \uparrow^i\phi$, where $i \in \mathbb{N}$;
- $\phi_1 \oplus_i \phi_2 = \overline{\phi_1}^{i} \wedge \downarrow^i\phi_2$, where $i \in \mathbb{N}$, and $\overline{\phi_1}^{i}$ is obtained from ϕ_1 by replacing every variable of the form x or $x^{\downarrow^i\downarrow^{j_1}\cdots\downarrow^{j_k}}$ 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 $[\![\phi]\!]\sigma$, the interpretation of the formula ϕ in the memory state σ. To appropriately adapt a standard semantics $[\![.]\!]$ to a hierarchy of memories, it suffices to extend the interpretation for the extended set of variables $\mathcal{N}^+$, as $[\![n]\!]\sigma = \sigma(n)$ 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 postconditions 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 1 (Verification Soundness). *Assume that $\{P\} \vdash G\ \{Q\}$ 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 $\sigma' \models Q$.*

3.3 Example Program

We illustrate the verification with an example. Consider a program, G_{Add}, that add two input arrays (A and B) producing on output array C. The code of the program is given by the following definitions:

$G_{\text{Add}} := \mathsf{Exec}_0(\mathsf{Forall}\ i = 0 : n-1\ \mathsf{do}\ \text{Add})$

$\text{Add} := \mathsf{Group}((\text{CopyAX} \parallel \text{CopyBY}); \text{AddP}; \text{CopyZC})$
$\text{CopyAX} := \mathsf{Copy}^{\downarrow}(A[i.S, (i+1)S], X[i.S, (i+1)S])$
$\text{CopyBY} := \mathsf{Copy}^{\downarrow}(B[i.S, (i+1)S], Y[i.S, (i+1)S])$
$\text{AddP} := \mathsf{Kernel}\langle Z[i.S, (i+1)S] = \text{VectAdd}(X[i.S, (i+1)S], Y[i.S, (i+1)S])\rangle$
$\text{CopyZC} := \mathsf{Copy}^{\uparrow}(Z[i.S, (i+1)S], C[i.S, (i+1)S])$

Assume that the arrays have size $n.S$, and note that the program is divided in n parallel subtasks, each operating on different array fragments, of size S. The best value for S may depend on the underlying architecture.

It is easy to see that this program is safe, since each subtask writes on a different fragment of the arrays.

We show, using the verification framework, how to derive the judgment $\{\mathsf{true}\} \vdash G_{\text{Add}}\ \{\text{Post}\}$, where $\text{Post} = \forall k,\ 0 \leq k < n.S \Rightarrow C[k] = A[k] + B[k]$. Using the rules **[A]**, **[G]** and **[SS]** we derive, for each $i \in [0 \ldots n-1]$, the following:

$$\{\mathsf{true}\} \vdash \text{Add}\ \{Q_i\}\,, \tag{1}$$

where $Q_i = \forall k,\ i.S \leq k < (i+1)S \Rightarrow C^{\uparrow}[k] = A^{\uparrow}[k] + B^{\uparrow}[k]$. Applying the rule **[F]** on (1) we obtain

$$\{\mathsf{true}\} \vdash \mathsf{Forall}\ i = 0 : n-1\ \mathsf{do}\ \text{Add}\ \left\{\textstyle\sum_{0 \leq j < n} Q_j\right\}\ . \tag{2}$$

Note that the postcondition of the i-th subtask only refers to variables that it modifies, therefore, it is not difficult to see that $\sum_{0 \leq j < n} Q_j \Rightarrow \bigwedge_{0 \leq j < n} Q_j$. Applying the subsumption rule to (2), we obtain

$$\{\mathsf{true}\} \vdash \mathsf{Forall}\ i = 0 : n-1\ \mathsf{do}\ \text{Add}\ \{Q\} \tag{3}$$

where $Q = \forall k,\ 0 \leq k < n.S \Rightarrow C^{\uparrow}[k] = A^{\uparrow}[k] + B^{\uparrow}[k]$. Finally, applying rule **[E]** to (3), we obtain the desired result, since $\text{Post} = \downarrow^{0} Q$.

4 Certificate Translation

In this section, we focus on the interplay between program optimization and program verification. To maximize the performance of applications, the Sequoia compiler 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. More precisely, we provide an algorithm to transform a derivation for the original program into a derivation for the transformed program. The problem of transforming provably correct programs into provably correct programs is motivated by research in Proof Carrying Code (PCC) [15,14], and in particular by our earlier work on certificate translation [2,3].

We start by extending the analysis setting described in previous sections with a notion of certificates, to make it suitable for a PCC architecture. Then, we describe certificate translation in the presence of three optimizations: SPMD distribution, Exec Grouping and Copy grouping.

Certified setting. In a PCC setting, a program is distributed with a checkable certificate that the code complies with the specified policy. To extend the verification framework defined in Section 3.2 with a certificate infrastructure, we capture the notion of checkable proof with an abstract proof algebra.

Definition 6 (Certificate infrastructure). *A certificate infrastructure consists on a proof algebra $\mathcal{P}$ that assigns to every $\phi \in \mathsf{Prop}$ a set of certificates $\mathcal{P}(\vdash \phi)$. We assume that $\mathcal{P}$ 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 $c \in \mathcal{P}(\phi)$.*

We do not commit to an specific representation of certificates, since it is not relevant for this paper. To give an intuition, we can define them in terms of the Curry-Howard isomorphism by considering $\mathcal{P}(\phi) = \{e \in \mathcal{E} \mid \langle\rangle \vdash e : \phi\}$, where $\mathcal{E}$ is the set of expressions and $\vdash e : \phi$ a typing judgment in some λ-calculus.

In addition, we refine the notion of certified analysis judgment, to enable code consumers to check whether a judgment is a valid judgment. To this end, the definition of rule **[SS]** is extended to incorporate certificates attesting the validity of the (a priori undecidable) logical formulae required in rule **[SS]**.

Definition 7 (Certified Verification Judgment). *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*

$$\frac{\phi \Rightarrow \phi' \quad \{\phi'\}\vdash G\ \{\psi'\} \quad \psi' \Rightarrow \psi}{\{\phi\}\vdash G\ \{\psi\}}\textbf{[SS]}$$

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

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

A common characteristic of the optimizations considered in the following sections is that they are defined as a substitution of a subprogram g by another subprogram g' in a bigger program G. We denote with $G[\bullet]$ the fact that G is a program with a *hole*. Given a program g, we denote with $G[g]$ the program obtained by replacing the hole $\bullet$ with g. Then, optimizations are characterized by subprograms g and g', defining a transformation from a program $G[g]$ into a program $G[g']$. The following general result complements the results on certificate translators explained in the following sections.

Lemma 2. *Let $G[\bullet]$ be a program with a hole, g, g' programs and Φ, Ψ logic formulae. If the judgment $\{\Phi\}\vdash G[g]\ \{\Psi\}$ is certified, then the derivation of the latter contains a certificate for the judgment $\{\phi\} \vdash g\ \{\psi\}$, for some ϕ and ψ. If there is a certificate for the judgment $\{\phi\} \vdash g'\ \{\psi\}$, then we can construct a certificate for the judgment $\{\Phi\}\vdash G[g']\ \{\Psi\}$.*

4.1 SPMD Distribution

Consider a program that executes multiple times a single piece of code represented by a subprogram g. If every execution of g involves and independent portion of data, the tasks can be performed in any sequential order or in parallel. SPMD distribution is a common parallelization technique that exploits this condition distributing the tasks among the available processing units.

Programs of the form $\mathsf{Forall}\ j = 0 : k.n - 1\ \mathsf{do}\ g$ are candidates for SPMD distribution, since $k.n$ instances of the single subprogram g are executed in parallel along the range of the iteration variable j. Furthermore, for each value of the iteration value j, the subprogram g operates over an independent partition of the data, as assumed for every program subject to verification.

G' is transformed from G by applying SPMD distribution if G' is the result of substituting every subprogram $\mathsf{Exec}_i(\mathsf{Forall}\ j = 0 : k.n - 1\ \mathsf{do}\ g)$ by the equivalent subprogram $\mathsf{Group}(G_1 \parallel \ldots \parallel G_k)$, with G_i defined as the program $\mathsf{Exec}_i(\mathsf{Forall}\ j = i.n : (i+1)n - 1\ \mathsf{do}\ g)$ for all $i \in [0, k-1]$.

Normally, a real compiler will also consider whether it is convenient to span the computation of g over other child nodes. However, since orthogonal to the transformation of the verification judgment, we do not consider this issue.

Lemma 2 in combination with the following lemma that states that the local substitutions defining SPMD distribution preserve certified judgments, implies the feasibility of certificate translation.

Lemma 3. *Given a program* $G = \mathsf{Exec}_i(\mathsf{Forall}\ j = 0 : k.n - 1\ \mathsf{do}\ g)$*, and a certified judgment* $\{\Phi\} \vdash G\ \{\Psi\}$*, it is possible to generate a certified judgment* $\{\Phi\} \vdash \mathsf{Group}(G_1 \parallel \ldots \parallel G_k)\ \{\Psi\}$*, where* G_i *is defined as* $\mathsf{Exec}_i(\mathsf{Forall}\ j = i.n : (i+1)n - 1\ \mathsf{do}\ g)$ *for any* $i \in [0, k-1]$*.*

Example: Consider again the program G_{Add} of Section 3.3. Assume that at the level of the memory hierarchy at which G_{Add} is executed there are k available child processing units, and that $n = k.m$ for some m. Then, we are interested in distributing the independent computations along the iteration range $[0, n-1]$ splitting them in k subsets of independent computations in ranges of length m. We obtain then, after applying SPMD distribution to program G_{Add}, the following transformed program:

$$\begin{aligned} G'_{\mathrm{Add}} := {} & \mathsf{Exec}_0(\mathsf{Forall}\ i = 0 : m - 1\ \mathsf{do}\ \mathrm{Add}) \\ & \parallel \mathsf{Exec}_1(\mathsf{Forall}\ i = m : 2m - 1\ \mathsf{do}\ \mathrm{Add}) \\ & \quad \cdots \\ & \parallel \mathsf{Exec}_{k-1}(\mathsf{Forall}\ i = (k-1)m : k.m - 1\ \mathsf{do}\ \mathrm{Add}) \end{aligned}$$

Applying the result stated above, we can transform the derivation of the judgment $\{\mathsf{true}\} \vdash G_{\mathrm{Add}}\ \{\mathrm{Post}\}$ into a derivation of $\{\mathsf{true}\} \vdash G'_{\mathrm{Add}}\ \{\mathrm{Post}\}$, proving that the verification judgment is preserved. Recall that we can derive the judgment

$$\{\mathsf{true}\} \vdash \mathsf{Exec}_r(\mathsf{Forall}\ i = r.m : (r+1)m - 1\ \mathsf{do}\ \mathrm{Add})\ \left\{ \uparrow^r \Big(\sum_{r.m \le j < (r+1)m} Q_j \Big) \right\}$$

for every $0 \leq r < k$. One more application of rule [**G**] allows us to derive the judgment $\{\mathsf{true}\} \vdash G'_{\mathrm{Add}} \left\{ \sum_{0 \leq r < k} \uparrow^r (\sum_{r.m \leq j < (r+1)m} Q_j) \right\}$. Finally, requiring a certificate of the distributivity of $\uparrow^r$ over the operator $+$, and a certificate for $\sum_{0 \leq r < k} \sum_{r.m \leq j < (r+1)m} Q_j \Rightarrow \sum_{0 \leq j < k.m} Q_j$ we get by rule [**SS**]

$$\{\mathsf{true}\} \vdash G'_{\mathrm{Add}} \left\{ (\sum_{0 \leq j < k.m} \uparrow^r Q_j) \right\} \ .$$

By the same reasoning in Section 3.3 we have $\sum_{0 \leq j < k.m} \uparrow^r Q_j \Rightarrow \bigwedge_{0 \leq j < n} \uparrow^r Q_j$, and finally by subsumption rule we get $\{\mathsf{true}\} \vdash G'_{\mathrm{Add}} \{\mathrm{Post}\}$. Notice that judgment reconstruction entails the application of the [**SS**] rule and, thus, requires discharging extra proof obligations. These obligations include, for instance, proving commutativity of $+$, associativity of $+$ and distributivity of $\uparrow^r$ over $+$.

4.2 Exec Grouping

An Exec operation pushes the execution of a piece of code down to one of the memory subtrees. Since the cost of transferring code and data between different levels of the hierarchy is not negligible, there is an unnecessary overhead when several Exec operations contain code with short execution time. Hence, there is a motivation to reduce the cost of invoking code in child nodes, by grouping the computation defined inside a set of Exec operations into a single Exec operation.

We say that a program G' is the result of applying Exec grouping, if it is the result of replacing a set of Exec operations targeting the same child node, by an single and semantically equivalent Exec operation. More precisely, every subprogram $\mathsf{Group}(\{\mathsf{Exec}_i(G_1), \ldots, \mathsf{Exec}_i(G_k)\} \cup H)$ such that $(\mathsf{Exec}_i(G_j))_{j=1}^{k}$ are maximal in the dependence graph and mutually independent, is substituted by the equivalent subprogram $\mathsf{Group}(\{\mathsf{Exec}_i(\mathsf{Group}(\{G_1, \ldots, G_k\}))\} \cup H)$. In addition, the dependence relation that defines the graph $\{\mathsf{Exec}_i(\mathsf{Group}(\{G_1, \ldots, G_k\}))\} \cup H$ must be accordingly updated. More precisely, if the subprogram $g \in H$ originally depends on G_i for some $i \in [1, k]$ then g depends on $\mathsf{Exec}_i(\mathsf{Group}(\{G_1, \ldots, G_k\}))$ in the modified dependence graph.

The following result expresses that a certified judgment corresponding to set of independent Exec operations can be translated to a certified judgment for the result of merging the Exec operations into a single one. This result, together with Lemma 2, implies the existence of a certificate translator for Exec grouping.

Lemma 4. *Consider a set of mutually independent tasks $G_1, \ldots, G_k$ and a dependence graph $\{\mathsf{Exec}_i(G_1), \ldots, \mathsf{Exec}_i(G_k)\} \cup H$ s.t. $(\mathsf{Exec}_i(G_j))_{1 \leq j \leq k}$ are maximal elements. Assume that $\{\Phi\} \vdash \mathsf{Group}(\{\mathsf{Exec}_i(G_1), \ldots, \mathsf{Exec}_i(G_k)\} \cup H)\ \{\Psi\}$ is a certified judgment. Then, it is possible to generate a certified judgment $\{\Phi\} \vdash \mathsf{Group}(\{\mathsf{Exec}_i(\mathsf{Group}(\{G_1, \ldots, G_k\}))\} \cup H)\ \{\Psi\}$.*

4.3 Copy Grouping

Commonly, for the execution environments targeted by Sequoia programs, transferring several fragments of data to a different level of the memory hierarchy in

a single copy operation is more efficient that transferring each fragment of data in a separate operation. For this reason, and since array copy operations are frequent in programs targeting data intensive applications, it is of interest to cluster a set of copy operations involving small and independent regions of the memory into a single transfer operation.

Naturally, this transformation may require an analysis to detect whether two copy operations referring to regions of the same array are indeed independent. However, for simplicity, we consider the case in which the original set of small copy operations are performed over different array variables.

Consider a subprogram $g = \mathsf{Group}(H \cup \{g_1, g_2\})$, where $g_1 = \mathsf{Copy}(A_1, B_1)$ and $g_2 = \mathsf{Copy}(A_2, B_2)$ are mutually independent and maximal in H. Copy propagation consists on substituting g by the equivalent program g' defined as $\mathsf{Group}(H \cup \{g_{1,2}\})$, where $g_{1,2}$ is a copy operation that merges atomic programs g_1 and g_2 into a single transfer operation. In addition, the dependence relation on $\mathsf{Group}(H \cup \{g_{1,2}\})$ must be updated accordingly, such that $g'' \in H$ depends on $g_{1,2}$ iff g'' depended on g_1 or g_2 in the original dependence graph.

Lemma 5. *Consider the programs g and g' as defined above. Then, from a certified judgment $\{\Phi\} \vdash g \ \{\Psi\}$ we can construct a certified derivation for the judgment $\{\Phi\} \vdash g' \ \{\Psi\}$.*

The program G' is the result of applying copy grouping to program G, if every subprogram of the form g in G is replaced by g', where g and g' are as characterized above. The existence of certificate translators follows from Lemma 2.

Example: Consider again the program G_{Add} of Section 3.3, that adds two arrays. From the definition of the subprogram Add, we can see that it is a candidate for a copy grouping transformation, since it can be replaced by the equivalent subprogram Add$'$ defined as $\mathsf{Group}(\mathrm{CopyAXBY}; \mathrm{AddP}; \mathrm{CopyZC})$, where CopyAXBY is defined as $\mathsf{Copy}^{\downarrow}(A[i.S, (i+1)S], B[i.S, (i+1)S], X[i.S, (i+1)S], Y[i.S, (i+1)S])$. Assume that judgment of for the example in Section 3.3 is certified. To translate this result after applying the transformation above, we must certify the judgment $\{\mathsf{true}\} \vdash \mathrm{CopyAXBY}\ \{Q_{AX} + Q_{BY}\}$. To this end, we reuse the certified judgments $\{\mathsf{true}\} \vdash \mathrm{CopyAX}\ \{Q_{AX}\}$ and $\{\mathsf{true}\} \vdash \mathrm{CopyBY}\ \{Q_{BY}\}$ that are included in the certificate for the judgment $\{\mathsf{true}\} \vdash G_{\mathrm{Add}}\ \{\mathrm{Post}\}$, where Q_{AX} is defined as $\big(\forall k,\ 0 \leq k < S \Rightarrow X[k] = A^{\uparrow}[k + i.S]\big)$ and Q_{BY} as $\big(\forall k,\ 0 \leq k < S \Rightarrow Y[k] = B^{\uparrow}[k + i.S]\big)$.

The fact that makes the translation through, is the validity of the formula $\mathsf{wp}_{\mathrm{CopyAX}}(\phi) \wedge \mathsf{wp}_{\mathrm{CopyBY}}(\psi) \Rightarrow \mathsf{wp}_{\mathrm{CopyAXBY}}(\phi + \psi)$.

5 Related Work

The present work follows the motivations on the framework introduced by some of the authors for certificate translation [2,3]. Certificate translation as originally proposed [2] targets low level unstructured code and a weakest precondition based verification, in contrast to the Hoare-like environment that we have defined

to verify Sequoia programs. Another difference is that in previous work programs transformations consist on standard compiler optimizations on sequential code, whereas in this paper we deal with transformations that take advantage of the concurrent programming model of Sequoia. Extending certificate translation for Sequoia programs with transformations that optimize sequential components in isolation is feasible.

Recent work on concurrent separation logic [16] can be used to reason about Sequoia programs as well. Concurrent separation logic extends the standard Owicki-Gries logic [17] for concurrent programs to enable one to reason about heap operations. One key feature of separation logic is that it allows to express whether a concurrent component owns a specific resource. From the application of the logic rules, it is required that each resource is shared by at most one process (although ownership may change along program execution), and that a sequential component may only affect the resources it owns. Therefore, it is possible to show independence of parallel subtasks using the logic rules.

However, since interleaving, and hence interaction, in a Sequoia program occur only in specific program points, the disjointness conditions are simpler to check. Additionally, it is convenient that checking the disjointness conditions is performed systematically by compilers, in which case it is reasonable to rely on such information for proving properties of programs.

6 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].

There are several directions for future work. First, it would be of interest to investigate whether our results hold for a relaxed semantics of Sequoia programs, allowing 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]. Second, we intend to investigate certificate translation in the context of parallelizing compilers. 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 [18] or permission-accounting separation logic [4].

References

1. Alpern, B., Carter, L., Ferrante, J.: Modeling parallel computers as memory hierarchies. In: Proc. Programming Models for Massively Parallel Computers (1993)
2. Barthe, G., Grégoire, B., Kunz, C., Rezk, T.: Certificate translation for optimizing compilers. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 301–317. Springer, Heidelberg (2006)

3. Barthe, G., Kunz, C.: Certificate translation in abstract interpretation. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 368–382. Springer, Heidelberg (2008)
4. Bornat, R., O'Hearn, P.W., Calcagno, C., Parkinson, M.: Permission accounting in separation logic. In: Principles of Programming Languages, pp. 259–270. ACM Press, New York (2005)
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, pp. 238–252 (1977)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Principles of Programming Languages, pp. 269–282 (1979)
7. Dally, W.J., Labonte, F., Das, A., Hanrahan, P., Ho Ahn, J., Gummaraju, J., Erez, M., Jayasena, N., Buck, I., Knight, T.J., Kapasi, U.J.: Merrimac: Supercomputing with streams. In: Conference on Supercomputing, p. 35. ACM, New York (2003)
8. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: Conference on Supercomputing, p. 83. ACM Press, New York (2006)
9. Hermenegildo, M.V., Rossi, F.: Strict and nonstrict independent and-parallelism in logic programs: Correctness, efficiency, and compile-time conditions. J. Log. Program. 22(1), 1–45 (1995)
10. Houston, M., Young Park, J., Ren, M., Knight, T., Fatahalian, K., Aiken, A., Dally, W.J., Hanrahan, P.: A Portable Runtime Interface For Multi-Level Memory Hierarchies. In: Scott, M.L. (ed.) PPOPP, ACM, New York (2008)
11. Kapasi, U.J., Rixner, S., Dally, W.J., Khailany, B., Ho Ahn, J., Mattson, P.R., Owens, J.D.: Programmable stream processors. IEEE Computer 36(8), 54–62 (2003)
12. Knight, T.J., Young Park, J., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W.J., Hanrahan, P.: Compilation for explicitly managed memory hierarchies. In: Yelick, K.A., Mellor-Crummey, J.M. (eds.) PPOPP, pp. 226–236. ACM, New York (2007)
13. Narayanasamy, S., Wang, Z., Tigani, J., Edwards, A., Calder, B.: Automatically classifying benign and harmful data races using replay analysis. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 22–31. ACM, New York (2007)
14. Necula, G.C.: Proof-carrying code. In: Principles of Programming Languages, New York, NY, USA, pp. 106–119. ACM Press, New York (1997)
15. Necula, G.C., Lee, P.: Safe kernel extensions without run-time checking. In: Operating Systems Design and Implementation, Seattle, WA, October 1996, pp. 229–243. USENIX Assoc. (1996)
16. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375(1-3), 271–307 (2007)
17. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Informatica Journal 6, 319–340 (1975)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, Copenhagen, Denmark, July 2002. IEEE Computer Society, Los Alamitos (2002)