Automatic Parallelization and Optimization of Programs by Proof Rewriting

Clément Hurlin

INRIA Sophia Antipolis – Méditerranée and University of Twente

Abstract. We show how, given a program and its separation logic proof, one can parallelize and optimize this program and transform its proof simultaneously to obtain a proven parallelized and optimized program. To achieve this goal, we present new proof rules for generating proof trees and a rewrite system on proof trees.

1 Introduction

As the trend towards multi-core processors is growing, software developers must write parallel code. Because writing parallel software is notoriously harder than writing sequential software, inferring parallelism automatically is a possible solution to the challenges faced by software developers. A well-known technique for inferring parallelism is to detect pieces of programs that access disjoint parts of the heap. Previously [19,21,24], various pointer analysis have been used to achieve this goal for programs manipulating simple data structures and arrays.

In this paper, we describe a new technique to infer parallelism from proven programs. Instead of designing ad-hoc analysis techniques, we use separation logic [34] to analyze programs before parallelizing them. We use separation logic's $\star$ operator – which expresses disjointness of parts of the heap – to detect potential parallelism. Compared to [19,21,24], using the $\star$ operator avoids relying on reachabality properties. This permits to discover disjointness of arbitrary data structures, paving the way to parallelize and optimize object-oriented programs proven with separation logic [32].

Contrary to most previous works that manipulate proofs [5,29,33], our algorithms manipulate proof trees representing derivations of Hoare triplets. The overall procedure is as follows: we generate a proof tree $\mathcal{P}$ of a program C, then we rewrite $\mathcal{P}, C$ into $\mathcal{P}', C'$ such that $\mathcal{P}'$ is a proof of C' and C' is a parallelized and optimized version of C. The generation of proof trees is done with a modified version of smallfoot [8] and the rewrite system is implemented in tom [4].

Our algorithm for rewriting proof trees focuses on two rules of separation logic: the (Frame) and the (Parallel) rules. First, the (Frame) rule [34] allows reasoning about a program in isolation from its environment, by focusing only on the part of the heap that this program accesses. Second, the (Parallel) rule [30] allows reasoning about parallel programs that access disjoint parts of the heap.

$$\frac{\{\Xi\}C\{\Xi'\}}{\{\Xi \star \Theta\}C\{\Xi' \star \Theta\}}\ \text{(Frame)} \qquad \frac{\{\Xi\}C\{\Theta\} \quad \{\Xi'\}C'\{\Theta'\}}{\{\Xi \star \Xi'\}C\|C'\{\Theta \star \Theta'\}}\ \text{(Parallel)}$$

J. Palsberg and Z. Su (Eds.): SAS 2009, LNCS 5673, pp. 52–68, 2009.

The basic idea of our reasoning is depicted by the following rewrite rule:

$$\dfrac{\dfrac{\{\Xi\}C\{\Theta\}}{\{\Xi \star \Xi'\}C\{\Theta \star \Xi'\}}\,(\text{Frame}) \quad \dfrac{\{\Xi'\}C'\{\Theta'\}}{\{\Theta \star \Xi'\}C'\{\Theta \star \Theta'\}}\,(\text{Frame})}{\{\Xi \star \Xi'\}C;\, C'\{\Theta \star \Theta'\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{Parallelize}}^{1}$$

$$\dfrac{\{\Xi\}C\{\Theta\} \qquad \{\Xi'\}C'\{\Theta'\}}{\{\Xi \star \Xi'\}C \| C'\{\Theta \star \Theta'\}}\,(\text{Parallel})$$

The diagram above should be read as follows: Given a proof of the sequential program $C; C'$ we rewrite this proof into a proof of the parallel program $C \| C'$. If the initial proof tree is valid, this rewriting yields a valid proof tree because the leaves of the rewritten proof tree are included in the leaves of the initial proof tree.

Our procedure differs from recent and concurrent work [33] on three main points: *(1)* instead of attaching labels to heaps, we use the (Frame) rule to statically detect independent parts of the program, leading to a technically simpler procedure; *(2)* we express optimizations by rewrite rules on proof trees, allowing us to feature other optimizations than parallelization and to use different optimization strategies; and *(3)* we have an implementation. Having said these differences, both our work and Raza et al.'s work [33] build upon the insight that separation logic proofs are a convenient tool to detect parallelism.

Contributions. We present the careful design of proof rules adapted for our rewrite rules (Section 3). These proof rules are derived from [8]'s proof rules. We present sound rewrite rules from proof trees to proof trees that yield optimized programs (Section 4). Considered optimizations are parallelization, early disposal, late allocation, early lock releasing, and late lock acquirement. We present an implementation of our algorithms that uses smallfoot [7] as the proof tree generator and tom [4] as the rewrite engine (Section 5). We illustrate our algorithms by two examples (Section 6).

Outline. The rest of the paper is organized as follows: we present the formal language we use throughout the paper in Section 2, we show how our technique will benefit from the recent advances of separation logic in Section 7, we discuss related work in Section 8, and we conclude in Section 9.

2 Background

This section recalls the relevant parts of [8]'s framework that we use in our work.

Our assertion language distinguishes between pure (heap independent) and spatial (heap dependent) assertions:

[1] To disambiguate between Hoare rules (enclosed in parentheses) and rewrite rules, we underline rewrite rules.

$$
\begin{array}{rcll}
x, y, z & \in & \mathsf{Var} & \text{variables} \\
E, F, G & ::= & \mathtt{nil} \mid x & \text{expressions} \\
b & ::= & E = E \mid E \neq E & \text{boolean expressions} \\
\Pi & ::= & b \mid \Pi \wedge \Pi & \text{pure formulas} \\
f, g, f_i, l, r, \ldots & \in & \mathsf{Fields} & \text{fields} \\
\rho & ::= & f_1 : E_1, \ldots, f_n : E_n & \text{record expressions} \\
S & ::= & E \mapsto [\rho] \mid \mathtt{ls}(E, F) \mid \mathtt{tree}(E) & \text{simple spatial formulas} \\
\Sigma & ::= & \mathtt{emp} \mid S \mid \Sigma \star \Sigma & \text{spatial formulas} \\
\Xi, \Theta & \in & \Pi \,¦\, \Sigma & \text{formulas}
\end{array}
$$

The meaning of simple spatial formulas is as follows: $E \mapsto [\rho]$ represents a heap containing one cell at address E with content ρ, $\mathtt{ls}(E, F)$ represents a heap containing a linked list segment from address E to address F, and $\mathtt{tree}(E)$ represents a heap containing a tree whose root is at address E and whose left and right subtrees can be dereferenced with fields l and r. The formula $E \mapsto [\rho]$ can mention any number of fields in ρ: the values of omitted fields are implicitly existentially quantified. Top-level formulas are pairs $\Pi \,¦\, \Sigma$ where Π is a $\wedge$-separated sequence of pure formulas (indicating equalities/inequalities between expressions) and Σ is a $\star$-separated sequence of spatial formulas (indicating facts about the heap). The semantics of formulas is omitted and can be found in [8].

Entailment between formulas is written $\Xi \vdash \Theta$. We lift $\vdash$ to pure formulas and $\star$ to formulas (note that ¦ binds tighter than $\star$) as follows:

$$\Pi \vdash \Pi' \;\triangleq\; \Pi \,¦\, \mathtt{emp} \vdash \Pi' \,¦\, \mathtt{emp} \qquad \Pi \,¦\, \Sigma \star \Pi' \,¦\, \Sigma' \;\triangleq\; (\Pi \wedge \Pi') \,¦\, (\Sigma \star \Sigma')$$

We use σ to range over substitutions of the form $x_0/y_0, \ldots, x_n/y_n$. Below we abusively write $\Pi \vdash x_0/y_0, \ldots, x_n/y_n$ to denote $\Pi \vdash x_0 = y_0 \wedge \cdots \wedge x_n = y_n$. We define a syntactical equivalence relation between formulas as follows:

$$\Pi \,¦\, \Sigma \Leftrightarrow \Pi' \,¦\, \Sigma' \text{ iff } \begin{cases} \Pi \text{ is a permutation of } \Pi' \\ \exists \sigma, \Pi \vdash \sigma \text{ and } \Sigma[\sigma] \text{ is a permutation of } \Sigma' \end{cases}$$

Hoare triplets have the form $\{\Xi\} C \{\Theta\}$ where C is a command. Atomic commands A and commands C are defined by the following grammar (where p ranges over procedure names):

$$
\begin{array}{rcl}
A & ::= & x := E \mid x := E \rightarrow f \mid E \rightarrow f := F \mid x := \mathtt{new}() \mid \mathtt{dispose}(E) \\
C & ::= & A \mid \mathtt{empty} \mid \mathtt{if}\ b\ \mathtt{then}\ C\ \mathtt{else}\ C \mid \mathtt{while}(b)\{C\} \\
 & \mid & \mathtt{lock}(r) \mid \mathtt{unlock}(r) \mid p(\overline{E_1}; \overline{E_2}) \mid C; C \mid C \| C'
\end{array}
$$

Atomic command $x := E \rightarrow f$ looks up the content of field f of cell at address E, while $E \rightarrow f := F$ mutates the content of field f of cell at address E. In $\mathtt{lock}(r)$ and $\mathtt{unlock}(r)$[2], r is a lock or a *resource* [30]. Resources are declared

[2] To smallfoot's experts: smallfoot uses conditional critical regions `with do endwith` instead of `lock`/`unlock` commands. However, because smallfoot generates *verification conditions* [8], conditional critical regions are treated like `lock`/`unlock` commands in smallfoot's implementation. That is why we use `lock`/`unlock` commands.

in the (omitted) program's header and come with a *resource invariant*, i.e., a formula describing the part of the heap guarded by the resource. Intuitively, when a resource is locked by a process, the resource's invariant is transferred to the process; while when a resource is unlocked, the resource's invariant is transferred from the process back to the resource. In procedure calls $p(\overline{E_1}; \overline{E_2})$, $\overline{E_1}$ are the parameters that are unchanged in p's body, while $\overline{E_2}$ are the parameters that are assigned in p's body.

To mutate and lookup the content of records, we use the following notations:

$$\mathsf{mutate}(\rho, f, F) = \begin{cases} f : F, \rho' & \text{if } \rho = f : E, \rho' \\ f : F, \rho & \text{if } f \notin \rho \end{cases} \quad \mathsf{lkp}(\rho, f) = \begin{cases} E & \text{if } \rho = f : E, \rho' \\ x \text{ fresh} & \text{if } f \notin \rho \end{cases}$$

3 Derived Rules with Explicit Antiframes and Frames

In this section we show how to generate proof trees where *antiframes* [13] (portions of the state needed to execute a command) and frames (portions of the state useless to execute a command) are made explicit. Making antiframes and frames explicit will be needed in Section 4 for our rewrite rules to work.

The (Frame) rule is one of the central ingredients of separation logic's success. It allows reasoning with *small axioms* [31] about atomic commands. In practice, however, the small axioms are not used and frames are not computed at each atomic command. Consider, for example, the rule for field lookup used in [8][3]:

$$\frac{\Pi \vdash F = E \qquad x' \text{ fresh} \qquad \mathsf{lkp}(\rho, f) = G}{\{\Pi \mid \Sigma \star F \mapsto [\rho]\} x := E \to f \{x = G[x'/x] \wedge \Pi[x'/x] \mid (\Sigma \star F \mapsto [\rho])[x'/x]\}}$$

This rule does not frame the precondition: the whole pure part of the precondition (Π) is used to show $F = E$ and the substitution x'/x is applied to the whole precondition ($\Pi \mid \Sigma \star F \mapsto [\rho]$). In other words, this rule does not exhibit the part of the precondition that is framed i.e., *(1)* the pure part of the precondition that is useless to show $F = E$ and *(2)* the part of the precondition that is left unaffected by the substitution x'/x.

Fig. 1 shows rules (derived from [8]) for each atomic command where antiframes and frames are made explicit. In these rules, we subscript formulas representing antiframes by a and formulas representing frames by f. We indicate on the right-hand side of applications of (Frame) the formula being framed. Finally, extra side conditions of (Frame) are indicated as additional premises.

To help the reader understand these rules, we detail the rule exhibiting the antiframe and frame at a field lookup command (the second rule). The antiframe consists of *(1)* the pure part of the precondition which is necessary to show $F = E$: it is Π_a and of *(2)* the spatial part of the precondition asserting that the cell at E exists ($F \mapsto [\rho]$) and the spatial part of the precondition affected by the substitution x'/x (Σ_a). The frame is the antiframe's complement (Ξ_f).

[3] Where, for clarity, we do the following modifications to [8]'s presentation: we include the "rearrangement" step and we omit the continuation.

$$\frac{\dfrac{x' \text{ fresh}}{\{\Xi_a\}x := E\{\Xi_a[x'/x]\}}\ (\text{Assign}) \quad x \notin \Xi_f}{\{\Xi_a \star \Xi_f\}x := E\{\Xi_a[x'/x] \star \Xi_f\}}\ (\text{Frame } \Xi_f)$$

$$\frac{\dfrac{\Pi_a \vdash F = E \quad x' \text{ fresh} \quad \mathsf{lkp}(\rho, f) = G \quad \Xi = \Pi_a[x'/x] \wedge x = G[x'/x] \restriction (\Sigma_a \star F \mapsto [\rho])[x'/x]}{\{\Pi_a \restriction \Sigma_a \star F \mapsto [\rho]\}x := E \to f\{\Xi\}}\ (\text{Lookup}) \quad x \notin \Xi_f}{\{(\Pi_a \restriction \Sigma_a \star F \mapsto [\rho]) \star \Xi_f\}x := E \to f\{\Xi \star \Xi_f\}}\ (\text{Frame } \Xi_f)$$

$$\frac{\dfrac{\Pi_a \vdash F = E \quad \mathsf{mutate}(\rho, f, G) = \rho'}{\{\Pi_a \restriction F \mapsto [\rho]\}E \to f := G\{\Pi_a \restriction F \mapsto [\rho']\}}\ (\text{Mutate})}{\{\Pi_a \restriction F \mapsto [\rho] \star \Xi_f\}E \to f := G\{\Pi_a \restriction F \mapsto [\rho'] \star \Xi_f\}}\ (\text{Frame } \Xi_f)$$

$$\frac{\dfrac{x' \text{ fresh}}{\{\Xi_a\}x := \texttt{new}()\{\Xi_a[x'/x] \star x \mapsto []\}}\ (\text{New}) \quad x \notin \Xi_f}{\{\Xi_a \star \Xi_f\}x := \texttt{new}()\{\Xi_a[x'/x] \star x \mapsto [] \star \Xi_f\}}\ (\text{Frame } \Xi_f)$$

$$\frac{\dfrac{\Pi_a \vdash F = E}{\{\Pi_a \restriction F \mapsto [\rho]\}\texttt{dispose}(E)\{\Pi_a \restriction \texttt{emp}\}}\ (\text{Dispose})}{\{\Pi_a \restriction F \mapsto [\rho] \star \Xi_f\}\texttt{dispose}(E)\{\Pi_a \restriction \texttt{emp} \star \Xi_f\}}\ (\text{Frame } \Xi_f)$$

Fig. 1. Derived rules for atomic commands with explicit antiframes and frames

Theorem 1. *The rules in Fig. 1 are sound.*

Sketch of the proof. Observe that the restrictions imposed on explicit frames ($x \notin \Xi_f$) make explicit frames immune to substitutions x'/x (cases (Assign), (Lookup), and (New)). Then, further observe that these rules derive from [8]'s rules (which are sound). □

We have not discussed the rule for method calls. That is intentional: existing proof rules for method calls [34,7] already compute frames and antiframes at procedure calls. Similarly, we use standard rules for loops and conditionals. There is a caveat though: because smallfoot generates verification conditions [7], proofs for while loops are "separated" from the enclosing method. This forbids to move code from within a loop outside of the loop (and conversely).

4 Automatic Optimizations by Proof Rewriting

In this section, we show rewrite rules for proof trees (ranged over by the meta-variable $\mathcal{P}$). The proof trees we consider are built using [8]'s framework but we use Fig. 1's rules for atomic commands. This is crucial because all our rewrite rules mention the (Frame) rule on their left hand side i.e., they cannot fire if frames are not explicit.

$$\dfrac{\dfrac{\dfrac{}{\{\Lambda^{-}_{x}\}x \to f := E\{\Lambda^{E}_{x}\}}\,(\text{Mutate})}{\{\Lambda^{-,-,-}_{x,y,z}\}x \to f := E\{\Lambda^{E,-,-}_{x,y,z}\}}\,(\text{Fr}\ \Lambda^{-,-}_{y,z}) \quad \dfrac{\dfrac{\dfrac{}{\{\Lambda^{-}_{y}\}y \to f := F\{\Lambda^{F}_{y}\}}\,(\text{Mutate})}{\{\Lambda^{E,-,-}_{x,y,z}\}y \to f := F\{\Lambda^{E,F,-}_{x,y,z}\}}\,(\text{Fr}\ \Lambda^{E,-}_{x,z}) \quad \dfrac{\dfrac{}{\{\Lambda^{-}_{z}\}z \to f := G\{\Lambda^{G}_{z}\}}\,(\text{Mutate})}{\{\Lambda^{E,F,-}_{x,y,z}\}z \to f := G\{\Lambda^{E,F,G}_{x,y,z}\}}\,(\text{Fr}\ \Lambda^{E,F}_{x,y})}{\{\Lambda^{E,-,-}_{x,y,z}\}y \to f := F;\ z \to f := G\{\Lambda^{E,F,G}_{x,y,z}\}}\,(\text{Seq})}{\{\Lambda^{-,-,-}_{x,y,z}\}x \to f := E;\ y \to f := F;\ z \to f := G\{\Lambda^{E,F,G}_{x,y,z}\}}\,(\text{Seq})$$

Fig. 2. A proof tree obtained by applying Fig. 1's rules

A proof tree is *valid* if each inference is an instance of the proof rules. A rewrite rule $\mathcal{P} \to \mathcal{P}'$ takes an input proof tree $\mathcal{P}$ and yields an output proof tree $\mathcal{P}'$.

Definition 1. *A rewrite rule* $\to$ *is* sound *iff for all valid proof trees* $\mathcal{P}$ *such that* $\mathcal{P} \to \mathcal{P}'$, $\mathcal{P}'$ *is valid.*

The rewrite rules we present in the paper satisfy the following properties: *(1)* the rewrite rules are sound and *(2)* the rewrite rules preserve specifications i.e., given a proof tree whose root is $\{\Xi\}\_\{\Theta\}$, any tree returned by the rewrite system will have $\{\Xi\}\_\{\Theta\}$ as its root. This holds simply because all our rewrite rules leave the pre/postcondition of the root of the input proof tree untouched.

We conjecture that our rewrite system actually provides a stronger guarantee than preserving specifications. Plausibly, the set of final states of an input program and the set of final states of the corresponding optimized program are related. We leave this study, however, as future work.

4.1 Generated Proof Trees Have a Particular Shape

Most proof trees generated by Fig. 1's rules do not match the left-hand side of the rewrite rule Parallelize shown in the introduction. To exemplify this statement, we define the following abbreviation:

$$\Lambda^{E_0,\ldots,E_m}_{x_0,\ldots,x_m} \triangleq x_0 \mapsto [f : E_0] \star \cdots \star x_m \mapsto [f : E_m]$$

Note that this abbreviation enjoys the following equivalence:

$$\Lambda^{E_0,\ldots,E_m,E_{m+1},\ldots,E_{m+k}}_{x_0,\ldots,x_m,x_{m+1},\ldots,x_{m+k}} \Leftrightarrow \Lambda^{E_0,\ldots,E_m}_{x_0,\ldots,x_m} \star \Lambda^{E_{m+1},\ldots,E_{m+k}}_{x_{m+1},\ldots,x_{m+k}}$$

Now, to see why proof trees generated by Fig. 1's rules do not match the left-hand side of the rewrite rule Parallelize, consider the proof tree shown in Fig. 2 (where pure formulas are omitted, (Fr) abbreviates (Frame), and _ denotes existentially quantified values). The rewrite rule Parallelize cannot fire on Fig. 2's proof tree because this proof tree contains applications of (Frame) at each atomic command. Generally, given a program A_0; A_1; ..., the proof rules with explicit frames generate a proof tree with the following shape:

$$\dfrac{\dfrac{\cdots}{\{\ldots\}A_0\{\ldots\}}\,(\text{Frame}) \quad \dfrac{\dfrac{\cdots}{\{\ldots\}A_1\{\ldots\}}\,(\text{Frame}) \quad \cdots}{\{\ldots\}A_1;\ldots\{\ldots\}}\,(\text{Seq})}{\{\ldots\}A_0;\ A_1;\ldots\{\ldots\}}\,(\text{Seq})$$

Proof trees with the shape above are inappropriate for the rewrite rule Parallelize. Intuitively, the problem lies in the successive applications of (Frame) being redundant: the same formula is framed multiple times. For example, in the proof tree shown in Fig. 2, the formula Λ_x^E is framed twice: once in the center (Frame) and once in the right (Frame).

More generally, the presence of redundant frames means that applications of (Frame) are on *short* commands. However, as the left-hand side of the rewrite rule for parallelization described in the introduction shows, to parallelize *long* commands, applications of (Frame) have to be on long commands. Hence, removing redundant frames is a mandatory step before parallelizing. The next section shows how to remove redundancy by inferring applications of (Frame) on long commands from application of (Frame) on short commands.

4.2 Removing Redundancy in Frames

The redundancy in applications of (Frame) originally comes from the symbolic execution algorithm. Because symbolic execution mimics an operational update of the state at each atomic command, it cannot reason about a succession of commands: each atomic command is treated independently. To fix this issue, two solutions are available. The first solution is to build a new program verifier that infers frames for non-atomic commands. We think this solution is inadequate because it requires to design a program verifier with proof rewriting in mind (breaking separation of concerns). The second solution, chosen in this paper, is to minimize the modifications of the program verifier and to do as much work as possible on the proof rewriting side.

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\,(\text{Frame } \Xi_f) \quad \dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\,(\text{Frame } \Theta_f) \quad \{\Theta_p \star \Theta_f\}C''\{\Xi'\}}{\{\Theta_a \star \Theta_f\}C';\, C''\{\Xi'\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_f\}C;\, C';\, C''\{\Xi'\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{FactorizeFrames}}$$

$$\dfrac{\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_{f_0}\}C\{\Xi_p \star \Xi_{f_0}\}}\,(\text{Frame } \Xi_{f_0}) \quad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_{f_0}\}C'\{\Theta_p \star \Theta_{f_0}\}}\,(\text{Frame } \Theta_{f_0})}{\{\Xi_a \star \Xi_{f_0}\}C;\, C'\{\Theta_p \star \Theta_{f_0}\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_f\}C;\, C'\{\Theta_p \star \Theta_f\}}\,(\text{Frame } \Xi_c) \quad \{\Theta_p \star \Theta_f\}C''\{\Xi'\}}{\{\Xi_a \star \Xi_f\}C;\, C';\, C''\{\Xi'\}}\,(\text{Seq})$$

Guard: $\Xi_f \Leftrightarrow \Xi_{f_0} \star \Xi_c$ and $\Theta_f \Leftrightarrow \Theta_{f_0} \star \Xi_c$

Fig. 3. Rewrite rule to factorize applications of (Frame)

Fig. 3 shows the rewrite rule FactorizeFrames that removes redundancy in applications of (Frame). FactorizeFrames fires if C and C' are two consecutive commands that both frame a part of the state (Ξ_f and Θ_f respectively) such that the two parts of the state share a common part (Ξ_c as imposed by the guard). In FactorizeFrames's right-hand side, the common part of the state is framed *once*, below the application of (Seq).

Both the left-hand side of FactorizeFrames (abbreviated by lhs below) and the right-hand side of FactorizeFrames (abbreviated by rhs below) include the proof tree of the triplet $\{\Theta_p \star \Theta_f\} C'' \{\Xi'\}$. We need to include such a proof tree to match two possible cases: C'' can be a dummy "continuation" (represented by the `empty` command) or a "normal" continuation. In the implementation, all rewrite rules use this "possible continuation" trick.

Fig. 4 exemplifies an application of FactorizeFrames to Fig. 2's proof tree: the redundancy of Λ_x^E in the center and the right (Frame)s is factorized in a single (Frame).

$$\dfrac{\dfrac{\dfrac{}{\{\Lambda_x^E\} x \to f := E \{\Lambda_x^E\}}\,(\text{Mutate})}{\{\Lambda_{x,y,z}^{E,-,-}\} x \to f := E \{\Lambda_{x,y,z}^{E,-,-}\}}\,(\text{Fr } \Lambda_{y,z}^{-,-}) \quad \dfrac{\dfrac{\dfrac{}{\{\Lambda_y^F\} y \to f := F \{\Lambda_y^F\}}\,(\text{Mutate})}{\{\Lambda_{y,z}^{F,-}\} y \to f := F \{\Lambda_{y,z}^{F,-}\}}\,(\text{Fr } \Lambda_z^-) \quad \dfrac{\dfrac{}{\{\Lambda_z^G\} z \to f := G \{\Lambda_z^G\}}\,(\text{Mutate})}{\{\Lambda_{y,z}^{F,G}\} z \to f := G \{\Lambda_{y,z}^{F,G}\}}\,(\text{Fr } \Lambda_y^F)}{\{\Lambda_{y,z}^{F,G}\} y \to f := F;\ z \to f := G \{\Lambda_{y,z}^{F,G}\}}\,(\text{Seq})}{\{\Lambda_{x,y,z}^{E,F,G}\} y \to f := F;\ z \to f := G \{\Lambda_{x,y,z}^{E,F,G}\}}\,(\text{Fr } \Lambda_x^E)$$

$$\{\Lambda_{x,y,z}^{-,-,-}\} x \to f := E;\ y \to f := F;\ z \to f := G \{\Lambda_{x,y,z}^{E,F,G}\}\ (\text{Seq})$$

Fig. 4. Fig. 2's proof tree after applying FactorizeFrames once

Theorem 2. *The rewrite rule FactorizeFrames is sound.*

Proof. Suppose the left-hand side of FactorizeFrames is valid. The goal is to show that the right-hand side of FactorizeFrames rhs is valid.

For the application of (Frame Ξ_c) to be valid, we must show the two following equivalences: $\Xi_a \star \Xi_f \Leftrightarrow \Xi_a \star \Xi_{f_0} \star \Xi_c$ and $\Theta_p \star \Theta_f \Leftrightarrow \Theta_p \star \Theta_{f_0} \star \Xi_c$. But these two equivalences follow directly from FactorizeFrames's guard.

For the application of (Frame Θ_{f_0}) to be valid, we must show the following equivalence:

$$\Xi_p \star \Xi_{f_0} \Leftrightarrow \Theta_a \star \Theta_{f_0} \tag{goal}$$

From FactorizeFrames's first guard, we obtain:

$$\Xi_p \star \Xi_f \Leftrightarrow \Xi_p \star \Xi_{f_0} \star \Xi_c \tag{1}$$

From the validity of the application of (Seq) in FactorizeFrames's lhs, we obtain: $\Xi_p \star \Xi_f \Leftrightarrow \Theta_a \star \Theta_f$. Then, from FactorizeFrames's second guard, we obtain:

$$\Xi_p \star \Xi_f \Leftrightarrow \Theta_a \star \Theta_{f_0} \star \Xi_c \tag{2}$$

By simplifying Ξ_c on the right hand sides of (1) and (2), we obtain the desired goal. Now FactorizeFrames's validity is deduced as follows: *(1)* each inference in FactorizeFrames's rhs is a valid instance of the proof rules and *(2)* the leaves of FactorizeFrames's rhs are identical to the leaves of FactorizeFrames's lhs (which are valid by hypothesis). □

Because FactorizeFrames's guard uses the syntactical equivalence $\Leftrightarrow$, it might miss some semantical equivalences. Using an entailment relation $\vdash$ would be more

powerful. However, we leave open the problem of finding common frames with a semantical equivalence for the following reason: finding a common frame (i.e., given Ξ and Θ; find Ξ_c, Ξ_r, and Θ_r such that $\Xi \vdash \Xi_r \star \Xi_c$ and $\Theta \vdash \Theta_r \star \Xi_c$) cannot be expressed efficiently in terms of known problems; such as a frame problem [8] (given Ξ and Θ, find Ξ_f such that $\Xi \vdash \Xi_f \star \Theta$), or a bi-abduction problem [13] (given Ξ and Θ, find Ξ_a and Θ_f such that $\Xi \star \Xi_a \vdash \Theta \star \Theta_f$).

4.3 Parallelization

In practice, factorizing frames is a mandatory step before applying the Parallelize rewrite rule shown in the introduction. For example, applying Parallelize to the proof tree shown in Fig. 4 yields a proof of the following Hoare triplet:

$$\{\Lambda_{x,y,z}^{-,-,-}\} x \rightarrow f := E \parallel (y \rightarrow f := F \parallel z \rightarrow f := G) \{\Lambda_{x,y,z}^{E,F,G}\}$$

For the Parallelize rewrite rule to be sound, we add the guard that C does not modify variables in Ξ', C', and Θ' (and conversely). Note that, for clarity of presentation, the Parallelize rule shown in the introduction does not include the "possible continuation" trick mentioned above. We refer the interested reader to our companion report [25] for the complete rule.

4.4 Generic Optimizations

In this subsection, we present an optimization that changes the program's execution order. This optimization has four concrete applications: *(1)* dispose memory as soon as possible to avoid out of memory errors, *(2)* allocate memory as late as possible to leave more allocatable memory, *(3)* release locks as soon as possible to increase parallelism, and *(4)* acquire locks as late as possible to increase parallelism. Fig. 5 shows the rewrite rule for changing the program's execution order.

GenericOptimization fires if the program has the shape $C; C'; C''$ such that C' frames the postcondition of C (as imposed by the guard). Then, the program's order is changed so that C' executes before C. It should be noted that this rule imposes that C frames the precondition of C' by the following reasoning: for the first application of (Seq) to be valid in GenericOptimizations's lhs, we have $\Xi_p \star \Xi_f \Leftrightarrow \Theta_a \star \Theta_f$. From the guard, it follows that $\Xi_p \star \Xi_f \Leftrightarrow \Theta_a \star \Xi_p \star \Xi_r$. By simplifying Ξ_p on both sides, we obtain: $\Xi_f \Leftrightarrow \Theta_a \star \Xi_r$. We can conclude that C frames the precondition of C' (Θ_a).

We now detail GenericOptimization's four concrete applications. *(1)* If C' is a `dispose` command, because C frames the precondition of C', it means that C does not access the state disposed by C': better execute C' first to dispose memory as soon as possible. *(2)* If C is a `new` command, because C' frames the postcondition of C, it means that C' does not access the state allocated by C: better execute C after C' to leave C' more allocatable memory. *(3)* If C' is an `unlock` command, because C frames the precondition of C' (i.e. the lock's resource invariant), it means that C does not access the part of the heap represented by the lock's resource invariant: better execute C' to release the lock

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\,(\text{Fr } \Xi_f) \quad \dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\,(\text{Fr } \Theta_f) \quad \{\Theta_p \star \Theta_f\}C''\{\Xi'\}}{\{\Theta_a \star \Theta_f\}C';\, C''\{\Xi'\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_f\}C;\, C';\, C''\{\Xi'\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{GenericOptimization}}$$

$$\dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Xi_a \star \Theta_a \star \Xi_r\}C'\{\Xi_a \star \Theta_p \star \Xi_r\}}\,(\text{Fr } \Xi_r \star \Xi_a) \quad \dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Theta_p \star \Xi_r\}C\{\Xi_p \star \Theta_p \star \Xi_r\}}\,(\text{Fr } \Theta_p \star \Xi_r) \quad \{\Xi_p \star \Theta_p \star \Xi_r\}C''\{\Xi'\}}{\{\Xi_a \star \Theta_p \star \Xi_r\}C;\, C''\{\Xi'\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_f\}C';\, C;\, C''\{\Xi'\}}\,(\text{Seq})$$

Guard: $\Theta_f \Leftrightarrow \Xi_p \star \Xi_r$

Fig. 5. Rewrite rule to change the program's execution order

first. *(4)* If C is a `lock` command, because C' frames the postcondition of C (i.e. the lock's resource invariant), it means that C' does not access the part of the heap represented by the lock's resource invariant: better execute C after C' to acquire the lock as late as possible.

Theorem 3. *The rewrite rule GenericOptimization is sound.*

Proof. Apply the guard's equivalence in the right places and observe that *(1)* each inference in GenericOptimization's rhs is a valid instance of the proof rules and *(2)* the leaves of GenericOptimization's rhs are identical to the leaves of GenericOptimization's lhs (which are valid by hypothesis). □

5 Implementation

The techniques described in the previous sections have been implemented in a tool called *éterlou*. Éterlou consists of two distinct modules:

A proof tree generator which is an extended version of smallfoot [7]. The proof tree generator generates proof trees using Fig. 1's rules. Our extension does not interfere with the algorithms already present in smallfoot: it only computes antiframes and frames at each atomic command (by using both smallfoot's built-in algorithms and dedicated algorithms).

Because the (Frame) rule is the central ingredient of our procedure, it is crucial that the implementation of Fig 1's rules computes the *biggest frames* (formulas Ξ_f) possible. As an example, our implementation of the rule for field lookup (Fig 1's second rule) computes the smallest antiframe Π_a that suffices to show $F = E$. By computing the smallest antiframes, our implementation also computes the biggest frames.

A proof tree rewriter which implements the various rewrite rules shown in this paper. The proof tree rewriter is written in tom [4], an extension of Java that adds constructs for pattern matching. We make extensive use of tom's mapping facility to pattern match against user-defined Java objects. Another crucial feature is the possibility to define rewriting strategies.

All the examples of this paper have been generated with ésterlou. We have tested ésterlou against several example programs provided in smallfoot's distribution and pointer programs of our own. Our experiments revealed that to obtain the best optimizations possible, the rewrite rules must usually be applied in a given order and/or with specific strategies. For example, FactorizeFrames must be applied before Parallelize for the latter rewrite rule to fire. In addition, applying rewrite rules from top to bottom (i.e., rewriting at the root before trying to rewrite in subtrees) generally yields programs where parallelized commands are longer (compared to other strategies such as bottom to top).

6 Examples

Fig. 6 shows procedure *rotate_tree* (borrowed from [33]) that takes a tree at x and rotates it by recursively swapping its left and right subtrees. Applying FactorizeFrames and Parallelize to *rotate_tree* yields a program where the field assignments and the recursive calls are executed in parallel. We achieve better parallelism than [33] by parallelizing the field assignments and the recursive calls.

```
requires tree(x);
ensures  tree(x);
rotate_tree(x;){
 local x1, x2;
 if(x = nil){}
 else{
   x1 := x→l;
   x2 := x→r;
   x→l := x2;
   x→r := x1;
   rotate_tree(x1;);
   rotate_tree(x2;); }}
```

→

```
requires tree(x);
ensures  tree(x);
rotate_tree(x;){
 local x1, x2;
 if(x = nil){}
 else{
   x1 := x→l;
   x2 := x→r;
   (x→l := x2; x→r := x1) ||
   rotate_tree(x1;) ‖ rotate_tree(x2;); }}
```

Fig. 6. Parallelization of a recursive procedure

```
requires x ↦ [val : _];
ensures  emp;
copy_and_dispose(x;){
 local v;
 lock(r_{c↦[val:_]});
   v := x→val;
   c→val := v;
   dispose(x);
 unlock(r_{c↦[val:_]}); }
```

→

```
requires x ↦ [val : _];
ensures  emp;
copy_and_dispose(x;){
 local v;
 v := x→val;
 dispose(x);
 lock(r_{c↦[val:_]});
   c→val := v;
 unlock(r_{c↦[val:_]}); }
```

Fig. 7. Optimization of a critical region

Fig. 7 shows procedure *copy_and_dispose* (where resource r is subscripted by its invariant) that copies the content of field *val* of cell x to field *val* of cell c (r's resource invariant). Applying GenericOptimization optimizes *copy_and_dispose* in two ways: the critical region is shortened and cell x is disposed earlier.

The proof trees corresponding to Fig. 6 and Fig. 7's transformations as well as éterlou's implementation are available [1].

7 Benefits from Separation Logic's Advances

In this section, we review advances of separation logic that have not been implemented in smallfoot and we describe how our technique would benefit from these advances. As we use features from other papers, we are sometimes sloppy on definitions and appeal to the reader's intuition to understand the notations.

7.1 Object-Orientation

[32] applied separation logic to object-oriented programs. In Parkinson's work, separation logic's $\star$ splits objects per field. With our notations, this means that $p \mapsto [x : \_] \star p \mapsto [y : \_]$ represents a point with two fields x and y (and omitted fields are *not* existentially quantified). Splitting on a per-field basis provides fine-grained parallelism which allows to build such a proof:

$$\begin{array}{c} \{p \mapsto [x : \_] \star p \mapsto [y : \_]\} \\ p \rightarrow x := E \parallel p \rightarrow y := F \\ \{p \mapsto [x : E] \star p \mapsto [y : F]\} \end{array}$$

Integrating Parkinson's semantics of $\star$ in the proof tree generator would allow Parallelize to fire more often. For example, in *rotate_tree*, $x \rightarrow l := x_2; x \rightarrow r := x_1$ would be parallelized to $x \rightarrow l := x_2 \parallel x \rightarrow r := x_1$.

In addition, we highlight that lifting our technique to object-oriented programs is straightforward since our procedure's key mechanism is the (Frame) rule which is supported by object-oriented separation logic [32,23,22].

7.2 Permission Accounting

[10][4] gave an alternative reading of the points-to predicate $\mapsto$ by adding an extra parameter (called a *permission* π) to it. Permissions are fractions in $(0, 1]$. Now, the points-to predicate $x \overset{\pi}{\mapsto} [\rho]$ has the following meaning: *(1)* it asserts that x points to the record ρ and *(2)* if $\pi = 1$, it asserts write and read permission to the record pointed to by x; if $\pi < 1$, it asserts readonly permission to the record pointed to by x. The following property holds:

$$x \overset{\pi}{\mapsto} [\rho] \Leftrightarrow x \overset{\frac{\pi}{2}}{\mapsto} [\rho] \star x \overset{\frac{\pi}{2}}{\mapsto} [\rho]$$

Integrating permission accounting in the proof tree generator would allow Parallelize to fire more often.

[4] In this paragraph, we consider only fractional permissions [12] but our remarks also apply for the counting model and the combined model.

7.3 Fork/Join Parallelism

[23,20] lifted the (Parallel) rule to Java's fork/join style of parallelism. Calling $\texttt{fork}(t)$ starts a new thread t that executes in parallel with the rest of the program. Calling $\texttt{join}(t)$ stops the calling thread until thread t finishes: when t finishes the calling thread is resumed.

When a parent thread forks a new thread, a part of the parent's state is transferred to the new thread. This is formalized by the following rule:

$$\frac{\Xi \text{ is } t\text{'s precondition}}{\{\Xi\}\texttt{fork}(t)\{\texttt{emp}\}} \text{ (Fork)}$$

Dually, when a thread joins another thread, the former "takes back" a part of the latter's state. To formalize this behavior, [23]'s assertion language contains a new predicate $\texttt{Join}(t, \pi)$ which asserts that the thread in which it appears can take back part π of thread t's state (like in Bornat's work, π is a permission). In addition, the assertion language allows to multiply formulas by a permission, written $\pi \cdot \Xi$. To give the reader an intuition of the meaning of multiplication, we note that integrating multiplication in our framework would make the following property true:

$$\pi \cdot (\Pi \upharpoonright x \overset{1}{\mapsto} [\rho]) \Leftrightarrow \Pi \upharpoonright x \overset{\pi}{\mapsto} [\rho]$$

With the `Join` predicate and formula multiplication, one can formalize `join`'s behavior. The rule below expresses that a thread joining another thread t can take back a part of t's state:

$$\frac{\Xi \text{ is } t\text{'s postcondition}}{\{\texttt{Join}(t, \pi)\}\texttt{join}(t)\{\pi \cdot \Xi\}} \text{ (Join)}$$

Integrating (Fork) and (Join) in our framework would add two concrete applications to the rewrite rule GenericOptimization. *(1)* If C' is a `fork` command, GenericOptimization would rewrite proofs so that new threads are forked as soon as possible (increasing parallelism). *(2)* If C is a `join` command, GenericOptimization would rewrite proofs so that threads join other processes as late as possible (increasing parallelism and reducing joining time).

7.4 Variable as Resources

[10] showed how to treat variables like resources (heap cells in our terminology). This allows to get rid of the side condition in the (Parallel) rule resulting in a more uniform proof system. The assertion language contains a new predicate $\texttt{Own}_\pi(x)$ that asserts ownership π of variable x.

Roughly, writing a variable requires permission 1 while reading a variable requires some permission π (in analogy with the permission-accounting model). This rules out programs with races on shared variables:

$$\{\mathtt{Own}_1(x) \star ?\}$$
$$x = y \parallel x = z$$

Above, ? cannot be filled with a predicate asserting ownership of x (needed for verifying the parallel statement's rhs) because $\mathtt{Own}_1(x)$ is already needed to verify the parallel statement's lhs (and $\mathtt{Own}_1(x)$ cannot be $\star$-combined with $\mathtt{Own}_\pi(x)$ for any π).

The variable as resources technique would fit perfectly in our framework because variables that are not accessed by commands would be made explicit: $\mathtt{Own}_\pi(\_)$ predicates would appear in frames. In other words, program verifiers implementing the variable as resources technique would compute explicit frames and antiframes for atomic commands like Fig. 1's rules do.

8 Related Work

Separation logic was discovered by Reynolds [34]. O'Hearn [30] extended separation logic to deal with disjoint and lock-based concurrency. Parkinson [32] adapted separation logic to object-oriented programs. Program verifiers in separation logic include smallfoot [7], a tool for C [26], and tools for object-oriented programs [17,14].

The closest (and concurrent) related work is [33] which uses separation logic to parallelize programs. Our work differs in four ways: *(1)* [33] attaches labels to heaps and uses disjointness of labels to detect possible parallelism, while we use the (Frame) rule to statically detect possible parallelism, leading to a technically simpler procedure; *(2)* we express optimizations by rewrite rules on proof trees, allowing us to feature other optimizations than parallelization and to use different optimization strategies; *(3)* [33] is applied after a shape analysis [16,6], while our analysis is applied after verification with a program verifier; and *(4)* contrary to [33], we have an implementation.

Practical approaches for parallelizing programs include parallelizing compilers [9,3]. Parallelizing compilers focuses on loop parallelization and do not consider arbitrary pieces of code. Parallelizing compilers can yield code that executes an order of magnitude faster than classical compilers. Loop parallelization has been actively studied [27,2,36].

Formal approaches for optimizing programs include certified compilers [35,28] and certifying compilers [5,29]. Certified compilers include optimizations that we do not consider and provide fully machine-checked proofs. Certifying compilers manipulate formulas representing proof obligations whereas we manipulate proof trees representing derivation of Hoare triplets. For this reason, we can consider high-level optimizations such as parallelization whereas we cannot consider the low-level optimizations described in [5,29].

Techniques to dispose memory as soon as possible have been studied for machine registers [18] where the goal is to use as few registers as possible. Works on atomicity [15,11] include techniques to release locks as soon as possible.

9 Conclusion and Future Work

We show a new technique to optimize programs proven correct in separation logic. Optimizations are done by rewriting proofs represented as derivation of Hoare triplets. The core of the procedure uses separation logic's (Frame) rule to statically detect parts of the state which are useless for a command to execute. Considered optimizations are parallelization, early disposal, late allocation, early lock releasing, and late lock acquirement. Optimizations are expressed as rewrite rules between proof trees and are performed automatically.

The procedure has been implemented in the éterlou tool. Éterlou consists of a proof tree generator (a modified version of the smallfoot program verifier [7]) and a proof tree rewriter written in tom [4]. Small-scale experiments show that the approach is practical.

Future work includes extension to permission-accounting separation logic and object-oriented programs. The extension to permission-accounting is expected to increase the efficiency of the Parallelize rewrite rule. The extension to object-oriented programs will allow us to do larger scale experiments and to study how abstraction [32] behaves w.r.t. to our technique. For this, we plan to use recent implementations of program verifiers for object-oriented programs annotated with separation logic [17,14]. On a practical side, future work includes study of the different rewriting strategies and their impact on the efficiency of optimizations.

Acknowledgments. I thank Gilles Barthe, Radu Grigore, Christian Haack, Marieke Huisman, and Tamara Rezk for their very useful comments that helped crafting this paper. I have been supported in part by IST-FET-2005-015905 Mobius project and in part by ANR-06-SETIN-010 ParSec project.

References

1. Proof trees of the examples, additional examples, and open source implementation, `http://www-sop.inria.fr/everest/Clement.Hurlin/eterlou/eterlou.shtml`
2. Aiken, A., Nicolau, A.: Optimal loop parallelization. ACM SIGPLAN Notices 23(7) (1988)
3. Artigas, P., Gupta, M., Midkiff, S., Moreira, J.: Automatic loop transformations and parallelization for Java. In: International Conference on Supercomputing. ACM Press, New York (2000)
4. Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., Reilles, A.: Tom: Piggybacking rewriting on Java. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007)
5. 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)
6. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)

7. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111. Springer, Heidelberg (2006)
8. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
9. Bik, A., Gannon, D.: Automatically exploiting implicit parallelism in Java. Concurrency: Practice and Experience 9 (1997)
10. Bornat, R., Calcagno, C., Yang, H.: Variables as resource in separation logic. In: Mathematical Foundations of Programming Semantics. Electronic Notes in Theoretical Computer Science, vol. 155. Elsevier, Amsterdam (2005)
11. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. ACM Press, New York (2002)
12. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694. Springer, Heidelberg (2003)
13. Calcagno, C., Distefano, D., OHearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Shao, Z., Pierce, B.C. (eds.) Principles of Programming Languages. ACM Press, New York (to appear, 2009)
14. Chin, W., David, C., Nguyen, H., Qin, S.: Enhancing modular OO verification with separation logic. In: Necula, G.C., Wadler, P. (eds.) Principles of Programming Languages. ACM Press, New York (2008)
15. Cunningham, D., Gudka, K., Eisenbach, S.: Keep off the grass: Locking the right path for atomicity. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 276–290. Springer, Heidelberg (2008)
16. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
17. DiStefano, D., Parkinson, M.: jStar: Towards practical verification for Java. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, vol. 43. ACM Press, New York (2008)
18. Ergin, O., Balkan, D., Ponomarev, D., Ghose, K.: Early register deallocation mechanisms using checkpointed register files. IEEE Computer 55 (2006)
19. Ghiya, R., Hendren, L.J., Zhu, Y.: Detecting parallelism in c programs with recursive data structures. In: Koskimies, K. (ed.) CC 1998, vol. 1383. Springer, Heidelberg (1998)
20. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 19–37. Springer, Heidelberg (2007)
21. Gupta, R., Pande, S., Psarris, K., Sarkar, V.: Compilation techniques for parallel systems. Parallel Computing 25(13) (1999)
22. Haack, C., Huisman, M., Hurlin, C.: Reasoning about Java's reentrant locks. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 171–187. Springer, Heidelberg (2008)
23. Haack, C., Hurlin, C.: Separation logic contracts for a Java-like language with fork/join. In: Meseguer, J., Rosu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 199–215. Springer, Heidelberg (2008)
24. Hendren, L.J., Nicolau, A.: Parallelizing programs with recursive data structures. IEEE Transactions on Parallel and Distributed Systems 1 (1990)

25. Hurlin, C.: Automatic parallelization and optimization of programs by proof rewriting. Technical Report 6806, INRIA. Initial version: January 2009, revised version: April 2009
26. Jacobs, B., Piessens, F.: The verifast program verifier. Technical Report CW520, Katholieke Universiteit Leuven (2008)
27. Lamport, L.: The parallel execution of do loops. Communications of the ACM 17(2) (1974)
28. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) Principles of Programming Languages. ACM Press, New York (2006)
29. Necula, G.C.: Translation validation for an optimizing compiler. ACM SIGPLAN Notices 35(5) (2000)
30. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375(1-3) (2007)
31. O'Hearn, P.W., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 1. Springer, Heidelberg (2001) (invited paper)
32. Parkinson, M.: Local Reasoning for Java. Ph.D thesis, University of Cambridge (2005)
33. Raza, M., Calcagno, C., Gardner, P.: Automatic parallelization with separation logic. In: European Symposium on Programming (to appear, 2009)
34. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science. IEEE Press, Los Alamitos (2002)
35. Strecker, M.: Formal Verification of a Java Compiler in Isabelle. In: Voronkov, A. (ed.) CADE 2002. LNCS, vol. 2392, p. 63. Springer, Heidelberg (2002)
36. Xue, C., Shao, Z., Sha, E.-M.: Maximize parallelism minimize overhead for nested loops via loop striping. Journal of VLSI Signal Processing Systems 47(2) (2007)