Automatic Parallelization and Optimization of Programs by Proof Rewriting

Clément Hurlin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

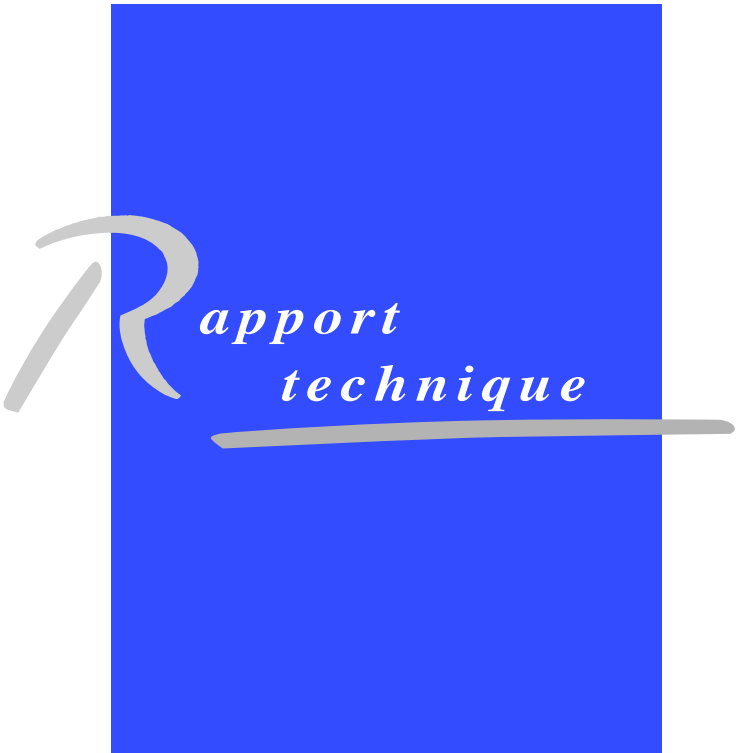
Automatic Parallelization and Optimization of Programs by Proof Rewriting

Clément Hurlin

N° 6806

Janvier 2009

Thème SYM



ISSN 0249-0803 ISRN INRIA/RT--6806--FR+ENG

Automatic Parallelization and Optimization of Programs by Proof Rewriting

Clément Hurlin*†

Thème SYM — Systèmes symboliques
Projet Everest et équipe Formal Methods & Tools de l'université de Twente

Rapport technique n° 6806 — Janvier 2009 — 30 pages

Abstract:

Proving a program in separation logic consists of tracking how parts of the heap are disjointly accessed by the different parts of the program. Parallelizing a program consists of finding parts of the program which access disjoint parts of the heap. Because proving a program in separation logic and parallelizing this program involve a similar reasoning, it follows that proving a program in separation logic considerably eases the task of automatically parallelizing this program.

In this paper 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. We present an implementation of our algorithms that uses smallfoot as the proof generator and tom as the proof transforming engine. Finally, we show how our technique will benefit of the recent advances in separation logic.

Key-words: Automatic parallelization, proof-preserving optimizations, separation logic

* Supported in part by IST-FET-2005-015905 Mobius project.
† Supported in part by ANR-06-SETIN-010 ParSec project.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Parallélisation et optimisation de programmes par réécriture de preuves

Résumé : Prouver un programme en logique de séparation consiste à trouver comment les différentes parties du tas sont utilisées par les différentes commandes du programme. Paralléliser un programme consiste à trouver les différentes commandes du programme qui accèdent des parties différentes du tas. Comme prouver un programme en logique de séparation et paralléliser un programme sont deux activités similaires, nous montrons que prouver un programme en logique de séparation est un prélude très utile à la parallélisation automatique de ce programme.

Dans cet article, nous montrons comment, à partir d'un programme et de sa preuve en logique de séparation, nous pouvons paralléliser et optimiser ce programme simultanément. Nous présentons une implémentation de nos algorithmes qui utilise *smallfoot* comme générateur de preuves et *tom* comme procédure de réécriture. Enfin, nous montrons comment notre technique peut bénéficier des avancées récentes en logique de séparation.

Mots-clés : Parallélisation automatique, logique de séparation

Contents

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. Techniques exist to infer parallelism within loops, however, to our knowledge, no technique extracts parallelism (soundly) from an arbitrary piece of code. That is because inferring parallelism is a major program transformation requiring multiple analyzes (data dependence, aliases etc.).

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 [28] to analyze (i.e., prove) programs before trying to parallelize them. In a sense, we push Knuths's advice

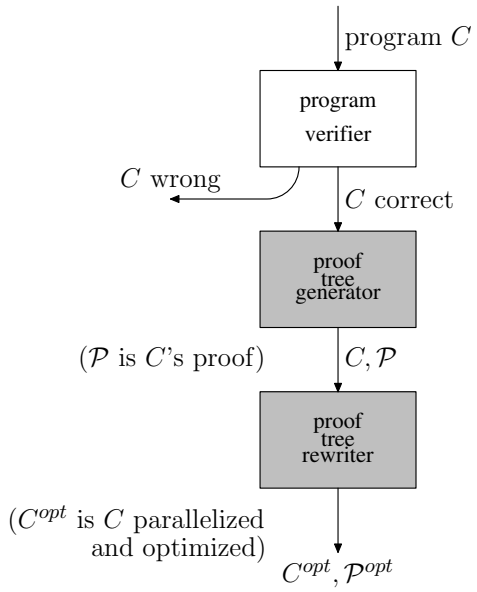


Figure 1: High level procedure with contributions in grey

"premature optimization is the root of all evil" to an extreme: we optimize programs once they have been proven correct.

Contrary to most previous works, our algorithms take as input proof trees representing derivations of Hoare triplets. Our procedure shows how to rewrite a valid proof tree of a program into a valid proof tree of a (parallelized and optimized version) of the program considered. The overall procedure is depicted in Fig. 1 where our contributions are indicated by grey blocks. $C, \mathcal{P}$ represents a program C together with its separation-logic proof $\mathcal{P}$. Fig. 1 shows how, from a program C which has been verified (the verifier outputs "C correct" and a proof tree of C), we obtain a parallelized and optimized version of C (C^{opt}) and its proof ($\mathcal{P}^{opt}$).

Our algorithm for rewriting proof trees focuses on two rules of separation logic: the (Frame) rule and the (Parallel) rule. First, the (Frame) rule [28] allows reasoning about a program in isolation from its environment, by focusing only on the part of the heap that this program accesses (where no variable occurring in Θ is modified by C):

$$\frac{\{\Xi\}C\{\Xi'\}}{\{\Xi \star \Theta\}C\{\Xi' \star \Theta\}} \text{ (Frame)}$$

Second, the (Parallel) rule [24] allows reasoning about parallel programs that access disjoint parts of the heap (where C does not modify any variables free in Ξ', C', Θ', and conversely):

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

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

$$\frac{\dfrac{\dfrac{\{\Xi\}C\{\Theta\}}{\{\Xi \star \Xi'\}C\{\Theta \star \Xi'\}}\text{(Frame)} \quad \dfrac{\{\Xi'\}C'\{\Theta'\}}{\{\Theta \star \Xi'\}C'\{\Theta \star \Theta'\}}\text{(Frame)}}{\{\Xi \star \Xi'\}C;\, C'\{\Theta \star \Theta'\}}\text{(Seq)}}{\downarrow\text{Parallelize}^{1}}$$

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

This diagram 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. The contributions of this paper are as follows:

- In Section 3, we present the careful design of proof rules adapted for our rewrite rules. These proof rules are derived from [6]'s proof rules.
- In Section 4, we present sound rewrite rules from proof trees to proof trees that yield optimized programs. Considered optimizations are parallelization, early disposal, late allocation, early lock releasing, late lock acquirement, and improvement of temporal locality.
- In Section 5, we present an implementation of our algorithms that uses smallfoot [5] as the proof tree generator and tom [3] as the rewrite engine. We provide evidence that an existing program verifier in separation logic (i.e., smallfoot) can easily be modified to generate proof trees corresponding to Section 3's proof rules.
- In Section 6, we show how our technique will benefit from the recent advances of separation logic. These advances include object-orientation [26], permission-accounting [9], fork/join parallelism [18, 17], and variables as resources [8].

In Section 2 we present the formal language we use throughout the paper. We discuss related work in Section 7 and we conclude in Section 8.

2 Background

In this section, we recall [6]'s framework which we use in our work.

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

x, y, z	$\in$	Var	variables
E, F, G	$::=$	$\mathtt{nil} \mid x$	expressions
b	$::=$	$E = E \mid E \neq E$	boolean expressions
Π	$::=$	$\top \mid b \mid \Pi \wedge \Pi$	pure formulas
$f, g, f_i, \ldots$	$\in$	Fields	fields
ρ	$::=$	$f_1 : E_1, \ldots, f_n : E_n$	record expressions
S	$::=$	$E \mapsto [\rho] \mid \mathtt{ls}(E, F) \mid \mathtt{tree}(E)$	simple spatial formulas
Σ	$::=$	$\mathtt{emp} \mid S \mid \Sigma \star \Sigma$	spatial formulas
Ξ, Θ	$\in$	$\Pi \mathbin{¦} \Sigma$	formulas

The meaning of simple spatial formulas is as follows: $E \mapsto [\rho]$ represents a heap containing one cell at address E with contents 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. Formulas are pairs $\Pi \mathbin{¦} \Sigma$ where Π is a $\wedge$-separated sequence of pure formulas (indicating equalities between expressions) and Σ is a $\star$-separated list of spatial formulas (indicating facts about the heap). The semantics of formulas is omitted and can be found in [6].

Entailment between formulas is written $\Pi \mathbin{¦} \Sigma \vdash \Pi' \mathbin{¦} \Sigma'$. We lift $\vdash$ to pure formulas and $\star$ to formulas (note that $¦$ binds tighter than $\star$) as follows:

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

We range over substitutions of the form $x_0/y_0, \ldots, x_n/y_n$ with σ and 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:

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

Hoare triplets have the form $\{\Pi \mathbin{¦} \Sigma\} C \{\Pi' \mathbin{¦} \Sigma'\}$ where commands C are defined by the following grammar (where p range over procedure names):

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

The command $\mathtt{with}\ r\ \mathtt{do}\ C\ \mathtt{endwith}$ is *a conditional critical region* [24]. Operationally, it behaves as follows: first acquire resource (or lock) r, then execute C, then release r. Resources are declared in the (omitted) program's header. In procedure calls $p(\overline{E_1}; \overline{E_2})$, $\overline{E_1}$ are the parameters 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:

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

3 Proofs with Explicit Antiframes and Frames

In this section we show how to generate proof trees where antiframes [12] (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 later needed (see 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* [25] 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 [6] (where we have included the "rearrangement" step described in the work cited):

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

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.

Figure 2 shows proof trees (derived from [6]'s proof rules) for each atomic command where antiframes and frames are made explicit. In these proof trees, we subscript formulas representing *antiframes* by a and formulas representing frames by f. In addition, 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 proof trees, we detail the proof tree exhibiting the antiframe and frame at a heap mutation command (the third rule). The antiframe consists of *(1)* the pure part of the prestate which is needed to show $F = E$: it is Π_a and of *(2)* the spatial points-to predicate asserting that the cell at E exists: it is $F \mapsto [\rho]$. The frame is simply the complement of the antiframe: $\Pi_f \mid \Sigma_f$.

Theorem 1. *The rules in Fig. 2 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 reduce to [6]'s rules (which are sound).

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

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}x' \text{ fresh} \quad \mathsf{lookup}(\rho, f) = G \quad \Pi_a \vdash F = E \\ \Xi = \Pi_a[x'/x] \wedge x = G[x'/x] \mid (\Sigma_a \star F \mapsto [\rho])[x'/x]\end{array}}{\{\Pi_a \mid \Sigma_a \star F \mapsto [\rho]\}x := E \to f\{\Xi\}}\,(\text{Lookup}) \quad x \notin \Xi_f}{\{(\Pi_a \mid \Sigma_a \star F \mapsto [\rho]) \star \Xi_f\}x := E \to f\{\Xi \star \Xi_f\}}\,(\text{Frame } \Xi_f) \quad \{\Xi \star \Xi_f\}C\{\Xi'\}}{\{(\Pi_a \mid \Sigma_a \star F \mapsto [\rho]) \star \Xi_f\}x := E \to f;\, C\{\Xi'\}}\,(\text{Seq})$$

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

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

$$\dfrac{\dfrac{\dfrac{\Pi_a \vdash F = E}{\{\Pi_a \mid F \mapsto [\rho]\}\texttt{dispose}(E)\{\Pi_a \mid \texttt{emp}\}}\,(\text{Dispose})}{\{\Pi_a \mid F \mapsto [\rho] \star \Xi_f\}\texttt{dispose}(E)\{\Pi_a \mid \texttt{emp} \star \Xi_f\}}\,(\text{Frame } \Xi_f) \quad \{\Pi_a \mid \texttt{emp} \star \Xi_f\}C\{\Xi'\}}{\{\Pi_a \mid F \mapsto [\rho] \star \Xi_f\}\texttt{dispose}(E);\, C\{\Xi'\}}\,(\text{Seq})$$

Figure 2: Proof trees for atomic commands with explicit antiframes and frames

The rules shown in Fig. 2 are not actually used for program verification. However, because they are derived from [6]'s rules, we use the implementation of the work cited (i.e., smallfoot) to generate proof trees like the ones shown in Fig. 2: reading the rules bottom-up gives a recursive algorithm for generating proof trees with explicit antiframes and frames.

We have not discussed the proof rules for method calls. That is intentional: existing proof rules for method calls [28, 5] already compute frames and antiframes at procedure calls.

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 from [6]'s "Symbolic Execution Rules" and the (Frame) rule shown in the introduction. However, instead of using the work cited's algorithm for generating proof trees, we use the rules with explicit frames shown in the previous section. 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.

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

All our rewrite rules preserve specifications i.e., given a proof tree $\mathcal{P}$ whose root is $\{\Xi\}C\{\Xi'\}$, any tree returned by the rewrite system consisting in all our rewrite rules will have $\{\Xi\}C\{\Xi'\}$ as its root. This holds simply because all our rewrite rules leave the root of the proof tree given as input untouched.

4.1 A First Try at Parallelization

The left-hand side of the rewrite rule for parallelization Parallelize shown in the introduction does not match most of the proof trees generated by the proof rules with explicit frames. To exemplify this statement, we need to define some abbreviations:

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

Note that this abbreviation enjoys the following equivalence (where we lift $\Leftrightarrow$ to spatial formulas in the straightforward way):

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

Now, to see why Parallelize's left-hand side does not match the shape of the proof trees generated by the proof rules with explicit frames, consider the proof tree shown in Fig. 3 (where pure formulas have been omittedand where (Frame) is abbreviated by (Fr)). This proof tree has been obtained by applying the proof rules with explicit frames on the following Hoare triplet (where _ denotes existentially quantified values):

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

Figure 3: A proof tree obtained by applying Fig. 2's rules

$$\{\Lambda^{-,-,-}_{x,y,z}\}x\to f := E;\ y\to f := F;\ z\to f := G\{\Lambda^{E,F,G}_{x,y,z}\}$$

The rewrite rule Parallelize cannot fire on Fig. 3's proof tree because this proof tree contains applications of (Frame) at each atomic command. Generally, given a program $A_0; A_1; \ldots$, the proof rules with explicit frames generate a proof tree with the following shape:

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

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. 3, the formula Λ^{E}_{x} is framed twice: once in the center (Frame) and once in the right (Frame).

4.2 Avoiding 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.

Fig. 4 shows the rewrite rule FactorizeFrames which we use to remove redundancy in applications of (Frame). Intuitively, this rule 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).

Before explaining FactorizeFrames in greater details, we invite the reader to refer to Fig. 5 for an example of applying FactorizeFrames once to the proof tree shown in Fig. 3: Redundancies in the center and the left (Frame)s have been removed. Note, however, that

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

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

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

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

Figure 4: Rewrite rule to factorize applications of (Frame)

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

Figure 5: Fig. 3's proof tree after applying FactorizeFrames once

$\Lambda_{\overline{z}}$ still appears in the left and center (Frame)s. The reason is that Fig. 3's proof tree is "parenthesized to the right": the symbolic execution procedure first proceeds with $x \to f := E$ and then continues with $y \to f := F$; $z \to f := G$. Because $\Lambda_{\overline{z}}$ is framed on both sides of the proof tree, FactorizeFrames does not fire.

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

Proof. Suppose FactorizeFrames's left-hand side (abbreviated by lhs below) is valid. The goal is to show that FactorizeFrames's right-hand side (abbreviated by rhs below) is valid.

For the application of (Frame Ξ_c) to be valid, we must show the two following equivalences:

$$\Xi_a \star \Xi_f \Leftrightarrow \Xi_a \star \Xi_{f_0} \star \Xi_c \tag{1}$$
$$\Theta_p \star \Theta_f \Leftrightarrow \Theta_p \star \Theta_{f_0} \star \Xi_c \tag{2}$$

But (1) and (2) 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:

$$\Xi_p \star \Xi_f \Leftrightarrow \Xi_p \star \Xi_{f_0} \star \Xi_c \tag{3}$$

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:

$$\Xi_p \star \Xi_f \Leftrightarrow \Theta_a \star \Theta_{f_0} \star \Xi_c \tag{4}$$

By simplifying Ξ_c on the right hand sides of (3) and (4), we obtain the desired goal.

Now FactorizeFrames's validity is deduced as follows: *(1)* each inference in FactorizeFrames's lhs 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). □

Both FactorizeFrames's lhs and FactorizeFrames's rhs 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 our rewrite rules use this "possible continuation" trick. For clarity reasons, however, in the rest of the paper, we describe rewrite rules without such continuations and refer the interested reader to Appendix A.

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. 5 yields a proof of the following Hoare triplet:

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

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).

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

Proof. Suppose Parallelize's lhs is valid. The goal is to show that Parallelize's rhs is valid.

This holds for the two following reasons: *(1)* Parallelize's rhs is a valid application of (Parallel) (because the guard is exactly (Parallel)'s side condition) and *(2)* the leaves of Parallelize's rhs are identical to the leaves of Parallelize's lhs (which are valid by hypothesis). □

Example Fig. 6 shows [5]'s procedure for disposing a tree. This procedure walks a tree, recursively disposing left and right subtrees and then the root pointer. This procedure has $\texttt{tree}(t)$ as a precondition which represents a heap where t points to a tree whose left and right subtrees can be dereferenced with fields l and r.

```
requires tree(t); ensures emp;
disp_tree(t;) {
 local i, j;
 if(t = nil){} else {
  i := t→l; j := t→r;
  disp_tree(i;); disp_tree(j;);
  dispose(t);
 }
}
```

Figure 6: Procedure for disposing a tree

Procedure `disp_tree` is correct and verifiable with smallfoot. Applying FactorizeFrames and Parallelize to the proof generated with the proof rules with explicit frames yields a proof where $\texttt{disp\_tree}(i;)\texttt{;}\ \texttt{disp\_tree}(j;)$ has been parallelized to $\texttt{disp\_tree}(i;) \parallel \texttt{disp\_tree}(j;)$.

4.4 Early Disposal and Late Allocation

Many verification techniques assume unbounded heaps i.e., verification techniques do not prevent out of memory errors. However, out of memory errors are an important issue to consider. In this paragraph, we present two optimizations yielding programs that need smaller heaps (i.e., programs that are less likely to raise out of memory errors).

The first optimization we consider is *early disposal* which consists of disposing heap cells as soon as possible. Intuitively, given a program C; $\texttt{dispose}(x)$ such that C "does not access the cell pointed to by x", x can be disposed before C: This leaves C with more allocatable cells. Because our framework considers unbounded heaps, we cannot speak about the size of the unallocated memory. However, an optimization for early disposal can be written with a simple rewrite rule because the (Frame) rule naturally expresses that "some cells are not accessed during a command". Fig. 7 presents our rewrite rule for early disposal.

Theorem 4. *The rewrite rule EarlyDisposal is sound.*

Proof. Observe that the leaves of EarlyDisposal's rhs are identical to the leaves of EarlyDisposal's lhs (which are valid by hypothesis). □

Example Applying the rewrite rule EarlyDisposal to the program shown in Fig. 6 yields a proof where $\texttt{dispose}(t)$ is moved before $\texttt{disp\_tree}(i;) \parallel \texttt{disp\_tree}(j;)$. Note that

$$\dfrac{\dfrac{\{\Xi_f\}C\{\Xi_{f'}\}}{\{\Xi_f \star \Xi_a\}C\{\Xi_{f'} \star \Xi_a\}}\,(\text{Frame } \Xi_a) \quad \dfrac{\dfrac{}{\{\Xi_a\}\texttt{dispose}(E)\{\Xi_{a'}\}}\,(\text{Dispose})}{\{\Xi_{f'} \star \Xi_a\}\texttt{dispose}(E)\{\Xi_{f'} \star \Xi_{a'}\}}\,(\text{Frame } \Xi_{f'})}{\{\Xi_f \star \Xi_a\}C;\ \texttt{dispose}(E)\{\Xi_{f'} \star \Xi_{a'}\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{EarlyDisposal}}$$

$$\dfrac{\dfrac{\dfrac{}{\{\Xi_a\}\texttt{dispose}(E)\{\Xi_{a'}\}}\,(\text{Dispose})}{\{\Xi_f \star \Xi_a\}\texttt{dispose}(E)\{\Xi_f \star \Xi_{a'}\}}\,(\text{Frame } \Xi_f) \quad \dfrac{\{\Xi_f\}C\{\Xi_{f'}\}}{\{\Xi_f \star \Xi_{a'}\}C\{\Xi_{f'} \star \Xi_{a'}\}}\,(\text{Frame } \Xi_{a'})}{\{\Xi_f \star \Xi_a\}\texttt{dispose}(E);\ C\{\Xi_{f'} \star \Xi_{a'}\}}\,(\text{Seq})$$

Figure 7: Rewrite rule for early disposal

this rewrite step is compatible with the rewrite step performed by Parallelize. Applying FactorizeFrames, Parallelize and EarlyDisposal successively yields the program visible in Fig. 8 (which is verifiable by smallfoot).

```
requires tree(t); ensures emp;
disp_tree(t;) {
 local i, j;
 if(t = nil){} else {
  i := t→l; j := t→r;
  dispose(t);
  disp_tree(i;) || disp_tree(j;);
 }
}
```

Figure 8: Optimized procedure for disposing a tree

The second optimization we consider is *late allocation* which consists of allocating heap cells as late as possible. This optimization is dual to the early disposal optimization. Intuitively, given a program $x := \texttt{new}(); C$ such that C does not access the cell pointed to by x, x can be allocated after C: This leaves C with more allocatable cells. Fig. 9 presents our rewrite rule for late allocation.

Theorem 5. *The rewrite rule LateAllocation is sound.*

Proof. Suppose LateAllocation's lhs is valid. The goal is to show that LateAllocation's rhs is valid.

The crux is to show that the second premise of (Seq) in LateAllocation's rhs is valid. The problem lies in the (Frame) on $\Xi_{f'}$. For the second premise of (Seq) to be an instance

$$\dfrac{\dfrac{\dfrac{}{\{\Xi_a\}x := \texttt{new}()\{\Xi_{a'}\}}\,(\text{New})\quad x \notin \Xi_f}{\{\Xi_a \star \Xi_f\}x := \texttt{new}()\{\Xi_{a'} \star \Xi_f\}}\,(\text{Frame } \Xi_f)\quad \dfrac{}{\{\Xi_f\}C\{\Xi_{f'}\}}\,(\text{Frame } \Xi_{a'})}{\{\Xi_a \star \Xi_f\}x := \texttt{new}(); C\{\Xi_{a'} \star \Xi_{f'}\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{LateAllocation}}$$

$$\dfrac{\dfrac{}{\{\Xi_f\}C\{\Xi_{f'}\}}}{\{\Xi_a \star \Xi_f\}C\{\Xi_a \star \Xi_{f'}\}}\,(\text{Frame } \Xi_a)\quad \dfrac{\dfrac{}{\{\Xi_a\}x := \texttt{new}()\{\Xi_{a'}\}}\,(\text{New})\quad x \notin \Xi_{f'}}{\{\Xi_a \star \Xi_{f'}\}x := \texttt{new}()\{\Xi_{a'} \star \Xi_{f'}\}}\,(\text{Frame } \Xi_{f'})}{\{\Xi_a \star \Xi_f\}C;\ x := \texttt{new}()\{\Xi_{a'} \star \Xi_{f'}\}}\,(\text{Seq})$$

Figure 9: Rewrite rule for late allocation

of the rule for (New) with explicit frames (i.e., Fig. 2's fourth rule), we need to show that $x \notin \Xi_{f'}$. However, because $\Xi_{f'}$ is the postcondition of the Hoare triplet $\{\Xi_f\}C\{\Xi_{f'}\}$ the free variables of $\Xi_{f'}$, consist in the free variables of Ξ_f plus some fresh variables. From this statement and $x \notin \Xi_f$ (this follows from a leaf of LateAllocation's lhs), we deduce $x \notin \Xi_{f'}$. The proof is concluded by observing that the leaves of LateAllocation's rhs are identical to the leaves of LateAllocation's lhs (which are valid by hypothesis).

□

4.5 Early Lock Releasing and Late Lock Acquirement

Locks are a very common primitive of parallel languages. However, they are difficult to use for various reasons: deadlock may happen, processes may starve for a lock, processes can retain locks for unnecessary long times etc. In this paragraph we show how we can optimize usage of locks by proof rewriting. Because O'Hearn [24] first proposed rules for critical regions (which are less general than locks), we will focus on critical regions in the rest of this paragraph.

The first optimization we consider is *early lock releasing* which consists of releasing locks as soon as possible. Early lock releasing reduces starvation of processes that all try to acquire a single lock, hence it reduces execution times.

To verify programs containing critical regions, O'Hearn proposed the following rule[2]:

$$\dfrac{\{\Xi \star \Theta\}C\{\Xi' \star \Theta\}}{\{\Xi\}\texttt{with}\ r_\Theta\ \texttt{do}\ C\ \texttt{endwith}\{\Xi'\}}\,(\text{Region})$$

Intuitively this rule can be understood as follows: *(1) resource* (or lock) r keeps the part of the heap denoted by Θ and *(2)* when a process acquires r (i.e., it enters the `with` block),

[2]Compared to O'Hearn's presentation of the rule, for simplicity, we omit the `when` clause of the `with` construct and the side condition about free variables in Ξ and Ξ'. In addition, we subscript resources by their invariant.

Θ is transferred to this process until this process releases r (i.e., it leaves the `with` block) and transfers Θ back to the lock r.

Expressing early lock releasing with O'Hearn's rule for critical regions can be done (once again) with the (Frame) rule. Intuitively, if a program that has acquired a resource r (with associated resource invariant Θ) frames Θ before releasing r, this means that r could be released earlier. Fig. 10 formally presents our rewrite rule for early lock releasing.

$$\dfrac{\dfrac{\{\Xi\star\Theta\}C\{\Xi''\star\Theta\}\quad\dfrac{\{\Xi''\}C'\{\Xi'\}}{\{\Xi''\star\Theta\}C'\{\Xi'\star\Theta\}}\,(\text{Frame }\Theta)}{\{\Xi\star\Theta\}C;\,C'\{\Xi'\star\Theta\}}\,(\text{Seq})}{\{\Xi\}\texttt{with}\ r_\Theta\ \texttt{do}\ C;\,C'\ \texttt{endwith}\{\Xi'\}}\,(\text{Region})$$

$$\downarrow \underline{\text{EarlyLockReleasing}}$$

$$\dfrac{\dfrac{\{\Xi\star\Theta\}C\{\Xi''\star\Theta\}}{\{\Xi\}\texttt{with}\ r_\Theta\ \texttt{do}\ C\ \texttt{endwith}\{\Xi''\}}\,(\text{Region})\quad\{\Xi''\}C'\{\Xi'\}}{\{\Xi\}\texttt{with}\ r_\Theta\ \texttt{do}\ C\ \texttt{endwith};\,C'\{\Xi'\}}\,(\text{Seq})$$

Figure 10: Rewrite rule for early lock releasing

The second optimization we consider is *late lock acquirement* which consists of acquiring locks as late as possible. This optimization is dual to the early lock releasing optimization. Fig. 11 presents our rewrite rule for late acquirement.

$$\dfrac{\dfrac{\dfrac{\{\Xi\}C\{\Xi''\}}{\{\Xi\star\Theta\}C\{\Xi''\star\Theta\}}\,(\text{Frame }\Theta)\quad\{\Xi''\star\Theta\}C'\{\Xi'\star\Theta\}}{\{\Xi\star\Theta\}C;\,C'\{\Xi'\star\Theta\}}\,(\text{Seq})}{\{\Xi\}\texttt{with}\ r_\Theta\ \texttt{do}\ C;\,C'\ \texttt{endwith}\{\Xi'\}}\,(\text{Region})$$

$$\downarrow \underline{\text{LateLockAcquirement}}$$

$$\dfrac{\{\Xi\}C\{\Xi''\}\quad\dfrac{\{\Xi''\star\Theta\}C'\{\Xi'\star\Theta\}}{\{\Xi''\}\texttt{with}\ r_\Theta\ \texttt{do}\ C'\ \texttt{endwith}\{\Xi'\}}\,(\text{Region})}{\{\Xi\}C;\,\texttt{with}\ r_\Theta\ \texttt{do}\ C'\ \texttt{endwith}\{\Xi'\}}\,(\text{Seq})$$

Figure 11: Rewrite rule for late lock acquirement

Theorem 6. *The rewrite rules EarlyLockReleasing and LateLockAcquirement are sound.*

Proof. Observe that both for EarlyLockReleasing and LateLockAcquirement, the rhs's leaves are identical to the lhs's leaves (which are valid by hypothesis). □

4.6 Improvement of Temporal Locality

Temporal locality [30] measures the time between successive accesses to a heap cell. The lower this time is, the faster a program executes because the underlying processors are less likely to access the main memory. Ideally, if a heap cell is accessed by two instructions successively, this allows the executing processor to keep the content of this heap cell in the cache, hence reducing execution time (no load/store to the main memory).

Because our framework does not express properties at the level of memory, we cannot formally speak about temporal locality. However, a rewrite rule that optimizes temporal locality can be written at our level of abstraction: heaps denoted by separation logic formulas.

Intuitively, a program $C; C'; C''$ such that C and C'' access a part of the heap disjoint from the part of the heap accessed by C' can be transformed into $C; C''; C'$ to improve temporal locality. In the optimized program, C and C'' executes consecutively: because C and C'' access the same part of the heap, the temporal locality of the optimized program has improved. Fig. 12 presents our rewrite rule for improving temporal locality.

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

$$\downarrow \underline{\text{TemporalLocality}}$$

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

Figure 12: Rewrite rule to improve temporal locality

Theorem 7. *The rewrite rule TemporalLocality is sound.*

Proof. Observe that the leaves of TemporalLocality's rhs are identical to the leaves of TemporalLocality'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 [5]. The proof tree generator generates proof trees like the ones depicted in Fig. 2. Our extension does not interfere with the algorithms already present in smallfoot: It only computes antiframes and frames at each atomic command (by using smallfoot's built-in algorithms). Our extension is 320 lines long, representing approximately 8% of extra code. This indicates that our technique of optimizing programs by proof rewriting can be decoupled from the program verifier.

A proof tree rewriter which implements the various rewrite rules shown in this paper. The proof tree rewriter is written in tom [3], 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. The proof tree rewriter is 2508 lines long and the longest implementation of a rewrite rule is 45 lines long.

All the examples of this paper have been generated with éterlou. We have tested éterlou against example programs provided in smallfoot's distribution and pointer programs of our own. Even though our experiments are restricted to approximately 30 methods, they give promising results: In half of the methods an optimization (without counting FactorizeFrames) fires.

Our experiments revealed that to obtain the best optimizations possible, the rewrite rules must sometimes be applied in a given order and/or with specific strategies. Generally, we found that FactorizeFrames must be applied before any other rewrite rules. Another observation is that TemporalLocality must be applied before Parallelize to maximize the chances of Parallelize 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 bigger (compared to other rewriting strategies such as bottom to top).

6 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.

In this section we are sometimes sloppy on definitions because we use features from other papers. We appeal to the reader's intuition to understand the notations used.

$$\frac{\{\texttt{tree}(t,\frac{1}{2})\}\texttt{contains}(t,n)\{\texttt{tree}(t,\frac{1}{2})\} \qquad \{\texttt{tree}(t,\frac{1}{2})\}\texttt{contains}(t,m)\{\texttt{tree}(t,\frac{1}{2})\}}{\{\texttt{tree}(t,\frac{1}{2})\star\texttt{tree}(t,\frac{1}{2})\}\texttt{contains}(t,n)\|\texttt{contains}(t,m)\{\texttt{tree}(t,\frac{1}{2})\star\texttt{tree}(t,\frac{1}{2})\}}\ \text{(Parallel)}$$

Figure 13: Proof tree of a program readonly accessing a tree in parallel

$$\frac{\{\texttt{tree}(t,1)\}\texttt{contains}(t,n)\{\texttt{tree}(t,1)\} \qquad \{\texttt{tree}(t,1)\}\texttt{contains}(t,n)\{\texttt{tree}(t,1)\}}{\{\texttt{tree}(t,1)\}\texttt{contains}(t,n);\ \texttt{contains}(t,m)\{\texttt{tree}(t,1)\}}\ \text{(Seq)}$$

Figure 14: Proof tree of a parallelizable program

6.1 Object-Orientation

[26] 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 very-fine grained parallelism which allows to prove such a Hoare triplet:

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

In our framework, integrating Parkinson's semantics of $\star$ in the proof tree generator would allow Parallelize to fire strictly more often.

6.2 Permission Accounting

[8][3] 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:

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

In the flavor of separation logic with permissions, one can define (as in [8]) a predicate `tree(`t`,`π`)` representing a tree t with permission π to access t. The following property holds:

$$\texttt{tree}(x,\pi)\Leftrightarrow\texttt{tree}(x,\frac{\pi}{2})\star\texttt{tree}(x,\frac{\pi}{2}) \qquad \text{(split)}$$

Now, consider a procedure `contains(`t`,`n`)` that looks up if n is in t's values with the following specification:

[3] In this paragraph, we consider only fractional permissions [11] but our remarks apply for the general model.

```
<π> requires tree(t,π); ensures tree(t,π);
contains(t,n){ ... }
```

As indicated by <π>, `contains`'s specification is parametrized by permission π (this has been shown sound by [18]). This means that `contains`'s precondition `tree(`t`,`π`)` can be instantiated by any π. This technique allows to verify programs where different processes simultaneously read access a tree as the proof tree in Fig. 13 shows.

Now consider the proof tree in Fig. 14. It is very simple and depicts the typical reasoning performed by a program verifier in separation logic. However, the rewrite rule Parallelize shown in this paper does not fire on this proof tree. To obtain Fig. 13's proof tree by rewriting Fig. 14's proof tree, we first need to apply the (split) equivalence from left to right (then applying FactorizeFrames and Parallelize yields Fig. 14's proof tree). This requires observing that *(1)* `tree`$(t, 1)$ can be split and that *(2)* `contains`'s specifications is parametrized. We have not found an elegant solution to this issue yet but we observe that permissions-accounting would allow Parallelize to fire strictly more often.

6.3 Fork/Join Parallelism

[18, 17] lifted the (Parallel) rule to Java's fork/join style of parallelism. Calling `fork`(p) starts a new process p that executes in parallel with the rest of the program. Calling `join`(p) stops the calling process until process p finishes: When p finishes the calling process is resumed.

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

$$\frac{\Xi \text{ is } p\text{'s precondition}}{\{\Xi\}\texttt{fork}(p)\{\texttt{emp}\}} \text{ (fork)}$$

Dually, when a process joins another process, the former "takes back" a part of the latter's state. To formalize this behavior, [18]'s assertion language contains a new predicate `Join`(p, π) which asserts that the process in which it appears can take back part π of process p's state (like in Bornat's work, π is a fractional permission in $(0, 1]$). In addition, the assertion language allows to multiply formulas (with restrictions irrelevant to this paper) by a permission. Multiplication is written $\pi \cdot \Xi$ and 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 process t (when joining a process p) can take back a part of p's state:

$$\frac{\Xi \text{ is } p\text{'s postcondition}}{\{\texttt{Join}(p, \pi)\}\texttt{join}(p)\{\pi \cdot \Xi\}} \text{ (join)}$$

Integrating (fork) and (join) in our framework would lead to two new optimizations: EarlyForking and LateJoining. EarlyForking would rewrite proofs so that new processes are forked as soon as possible (increasing parallelism) and LateJoining would rewrite proofs so that processes join other processes as late as possible (increasing parallelism and reducing joining time).

6.4 Variable as Resources

[8] showed how to treat variables like resources (heap cells in our terminology). This allows to get rid of the side condition in the (Parallel) and (Region) rules 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 full permissions while reading a variable requires some permissions (in analogy with the permission-accounting model). This rules out programs with races on shared variables:

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

Above, ? cannot be filled with a predicate asserting ownership of x (needed for verifying the parallel statement's rhs) because $\texttt{Own}_1(x)$ is already needed to verify the parallel statement's lhs (and $\texttt{Own}_1(x)$ cannot be $\star$-combined with $\texttt{Own}_\pi(x)$ 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: $\texttt{Own}_\pi(\_)$ 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. 2's proof trees do). Finally, the variable as resource technique would allow a more uniform treatment of the Parallelize rewrite rule because the guard corresponding to (Parallel)'s side condition would be deleted.

7 Related Work

Separation logic was discovered by [28]. [24] extended separation logic to deal with disjoint concurrency and lock based concurrency. Parkinson [26] adapted separation logic to object-oriented programs. Program verifiers in separation logic include smallfoot [5] and two tools for object-oriented programs [15, 13].

Formal approaches for optimizing programs include certified compilers [31, 22] and certifying compilers [4, 23]. 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 it would be harder to consider low-level optimizations considered in [4, 23] such as constant propagation. The Rhodium tool [21] is a framework for declaring and proving

correct program optimizations. Rhodium generates proof obligations to show that program optimizations are correct.

The closest related work is [29] which uses a type system to provide information to transform proof trees representing derivation of Hoare triplets. This work focuses on partial redundancy elimination and uses classical Hoare logic whereas we provide high-level optimizations such as parallelism and use separation logic.

Practical approaches for parallelizing programs include parallelizing compilers [7, 2]. 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 [20, 1, 32].

Early disposal has been studied in the context of machine register [16] where the goal is to use as few registers as possible. Works on atomicity [14, 10] include early unlocking inference. Improving temporal locality has been studied for a particular type of programs [19].

8 Conclusion and Future Work

We showed a new technique for optimizing programs which requires programs to be proven correct. Optimizations are done by rewriting proofs represented as derivation of Hoare triplets. The core of the procedure uses separation logic's (Frame) rule to 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. The approach is entirely automatic.

The procedure has been implemented in the éterlou tool. Éterlou consists of a proof tree generator (a modified version of the smallfoot program verifier) and a proof tree rewriter written in tom. Preliminary 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 [27] 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 [15, 13].

On a more practical side, future work includes study of the different rewriting strategies and their impact on the efficiency of optimizations. For this, we will have to define metrics for measuring efficiency of optimizations and to evaluate the runtime improvements of optimized programs. Finally, future work includes study of how the program verifier and the proof tree generator can be decoupled. We believe our technique would benefit from modular program verifiers (that provide interfaces for frame inference, formula entailment, etc.).

References

[1] A. Aiken, A. Nicolau. Optimal loop parallelization. *ACM SIGPLAN Notices*, 23(7), 1988.

[2] P. Artigas, M. Gupta, S. Midkiff, J. Moreira. J.: Automatic loop transformations and parallelization for Java. In *International Conference on Supercomputing*, 2000.

[3] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, A. Reilles. Tom: Piggybacking rewriting on Java. In *Rewriting Techniques and Applications*, Paris, France, 2007.

[4] G. Barthe, B. Grégoire, C. Kunz, T. Rezk. Certificate translation for optimizing compilers. In K. Yi, ed., *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science. Springer-Verlag, 2006.

[5] J. Berdine, C. Calcagno, P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, 2005.

[6] J. Berdine, C. Calcagno, P. W. O'Hearn. Symbolic execution with separation logic. In *Asian Programming Languages and Systems Symposium*, vol. 3780 of *Lecture Notes in Computer Science*, 2005.

[7] A. Bik, D. Gannon. Automatically exploiting implicit parallelism in Java. In *Concurrency: Practice and Experience*, vol. 9, 1997.

[8] R. Bornat, C. Calcagno, H. Yang. Variables as resource in separation logic. In *Mathematical Foundations of Programming Semantics*, vol. 155 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.

[9] R. Bornat, P. W. O'Hearn, C. Calcagno, M. Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages*, New York, NY, USA, 2005. ACM Press.

[10] C. Boyapati, R. Lee, M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2002.

[11] J. Boyland. Checking interference with fractional permissions. In R. Cousot, ed., *Static Analysis Symposium*, vol. 2694 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[12] C. Calcagno, D. Distefano, P. O'Hearn, H. Yang. Compositional shape analysis. In *Principles of Programming Languages*, Savannah, Georgia, USA, 2009. ACM Press. To appear.

[13] W. Chin, C. David, H. Nguyen, S. Qin. Enhancing modular OO verification with separation logic. In *POPL '08: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2008.

[14] D. Cunningham, K. Gudka, S. Eisenbach. Keep off the grass: Locking the right path for atomicity. In *International Conference on Compiler Construction*, Lecture Notes in Computer Science, 2008.

[15] D. DiStefano, M. Parkinson. jStar: Towards practical verification for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2008. To appear.

[16] O. Ergin, D. Balkan, D. Ponomarev, K. Ghose. Early register deallocation mechanisms using checkpointed register files. In *IEEE Computer*, vol. 55, 2006.

[17] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, M. Sagiv. Local reasoning for storable locks and threads. In *Asian Programming Languages and Systems Symposium*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[18] C. Haack, C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In J. Meseguer, G. Rosu, eds., *Algebraic Methodology and Software Technology*, number 5140 in Lecture Notes in Computer Science, Urbana, Illinois, USA, 2008. Springer-Verlag.

[19] G. Jin, J. Mellor-Crummey, R. Fowler. Increasing temporal locality with skewing and recursive blocking. In *International Conference on Supercomputing*, New York, NY, USA, 2001. ACM Press.

[20] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2), 1974.

[21] S. Lerner, T. Millstein, E. Rice, C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2005. ACM.

[22] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett, S. L. P. Jones, eds., *Principles of Programming Languages*. ACM Press, 2006.

[23] G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5), 2000.

[24] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3), 2007.

[25] P. W. O'Hearn, J. Reynolds, H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, number 2142 in Lecture Notes in Computer Science, Paris, 2001. Springer-Verlag. Invited paper.

[26] M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[27] M. Parkinson, G. Bierman. Separation logic and abstraction. In *Principles of Programming Languages*, 2005.

[28] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, Copenhagen, Denmark, 2002. IEEE Press.

[29] A. Saabas, T. Uustalu. Proof optimization for partial redundancy elimination. In *ACM Workshop on Partial Evaluation and Semantics-based Program Manipulation*. ACM Press, 2008.

[30] M. Snir, J. Yu. On the theory of spatial and temporal locality. Technical Report UIUCDCS-R-2005-2611, University of Illinois at Urbana-Champaign, 2005.

[31] M. Strecker. Formal Verification of a Java Compiler in Isabelle. In A. Voronkov, ed., *Conference on Automated Deduction*, vol. 2392 of *Lecture Notes in Computer Science*, London, UK, 2002. Springer-Verlag.

[32] C. Xue, Z. Shao, E.-M. Sha. Maximize parallelism minimize overhead for nested loops via loop striping. *Journal of VLSI Signal Processing Systems*, 47(2), 2007.

A Implementation of the Rewrite Rules

For the rewrite rules to work in practice, the implementation of the rewrite rules is more complex than the rules shown in the body of the paper. The caveat is the following: *(1)* the implementation expects that there is a continuation unaffected by the optimizations (except in the optimizations EarlyLockReleasing and LateLockAcquirement whose proof trees have a special shape, dictated by smallfoot's algorithm), *(2)* the implementation preserves the invariant that the first premise of an application of (Seq) is always an application of (Frame) and, *(3)* the implementation preserves the invariant that there are no double (Frame)s i.e., two applications of (Frame) on top of each other.

Rule *(1)* ensures that, given C and C' that can be parallelized, the Parallelize rule will fire on the program $C; C'; C''$ (and similarly for other optimizations). For rule *(1)* to allow optimizations to fire when there is *no* continuation, the rewrite rules insert dummy `empty` continuation in appropriate places. Rule *(2)* and *(3)* ensure that rewrite rules can assume proof trees have a given shape (without the need for investigating different cases). Rule *(2)*'s invariant that the first premise of an application of (Seq) is always a (Frame) is ensured by inserting "dummy" (Frame)s $\top \restriction \texttt{emp}$ in appropriate places. Rule *(3)*'s invariant that there are no double (Frame)s is ensured by $\star$-merging double (Frame)s when needed.

In the implementation of the rewrite rules, for space reasons, we write ϵ for `empty` and we sometimes abbreviate (Frame) by (Fr). FactorizeFrames's implementation is shown in Fig. 15, Parallelize in Fig. 16, EarlyDisposal in Fig. 19, LateAllocation in Fig. 20, EarlyLockReleasing in Fig. 19, LateLockAcquirement in Fig. 20.

The implementation features two variations of TemporalLocality. Fig. 21 shows a version meant to be applied before FactorizeFrames while Fig. 22 shows a version meant to be applied after FactorizeFrames. That is because FactorizeFrames takes as input a proof tree balanced to the right and outputs a proof tree balanced to the left. To be applied before FactorizeFrames, the first version of TemporalLocality (Fig. 21) takes as input a proof tree balanced to the right, while the second version of TemporalLocality (Fig. 22) takes as input a proof tree balanced to the left. In practice, both versions of TemporalLocality are useful.

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

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

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

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

Figure 15: FactorizeFrames's implementation

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

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

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

Figure 16: Parallelize's implementation

$$\dfrac{\dfrac{\{\Xi_f\}C\{\Xi_{f'}\}}{\{\Xi_f \star \Xi_a\}C\{\Xi_{f'} \star \Xi_a\}}\,(\text{Fr } \Xi_a) \quad \dfrac{\dfrac{}{\{\Xi_a\}\texttt{dispose}(E)\{\Xi_{a'}\}}\,(\text{Dispose})}{\{\Xi_{f'} \star \Xi_a\}\texttt{dispose}(E)\{\Xi_{f'} \star \Xi_{a'}\}}\,(\text{Frame } \Xi_{f'}) \quad \{\Xi_{f'} \star \Xi_{a'}\}C'\{\Xi''\}}{\{\Xi_{f'} \star \Xi_a\}\texttt{dispose}(E);\, C'\{\Xi''\}}\,(\text{Seq})}{\{\Xi_f \star \Xi_a\}C;\, \texttt{dispose}(E);\, C'\{\Xi''\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{EarlyDisposal}}$$

$$\dfrac{\dfrac{}{\{\Xi_a\}\texttt{dispose}(E)\{\Xi_{a'}\}}\,(\text{Dispose})}{\{\Xi_f \star \Xi_a\}\texttt{dispose}(E)\{\Xi_f \star \Xi_{a'}\}}\,(\text{Fr } \Xi_f) \quad \dfrac{\dfrac{\{\Xi_f\}C\{\Xi_{f'}\}}{\{\Xi_f \star \Xi_{a'}\}C\{\Xi_{f'} \star \Xi_{a'}\}}\,(\text{Frame } \Xi_{a'}) \quad \{\Xi_{f'} \star \Xi_{a'}\}C'\{\Xi''\}}{\{\Xi_f \star \Xi_{a'}\}C;\, C'\{\Xi''\}}\,(\text{Seq})}{\{\Xi_f \star \Xi_a\}\texttt{dispose}(E);\, C;\, C'\{\Xi''\}}\,(\text{Seq})$$

Figure 17: EarlyDisposal's implementation

$$\dfrac{\dfrac{\dfrac{}{\{\Xi_a\}x := \texttt{new}()\{\Xi_{a'}\}}\,(\text{New}) \quad x \notin \Xi_f}{\{\Xi_a \star \Xi_f\}x := \texttt{new}()\{\Xi_{a'} \star \Xi_f\}}\,(\text{Fr } \Xi_f) \quad \dfrac{\dfrac{\{\Xi_f\}C\{\Xi_{f'}\}}{\{\Xi_{a'} \star \Xi_f\}C\{\Xi_{a'} \star \Xi_{f'}\}ff}\,(\text{Fr } \Xi_{a'}) \quad \{\Xi_{a'} \star \Xi_{f'}\}C'\{\Xi''\}}{\{\Xi_{a'} \star \Xi_f\}C;\, C'\{\Xi''\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_f\}x := \texttt{new}();\, C;\, C'\{\Xi''\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{LateAllocation}}$$

$$\dfrac{\dfrac{\{\Xi_f\}C\{\Xi_{f'}\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_a \star \Xi_{f'}\}}\,(\text{Fr } \Xi_a) \quad \dfrac{\dfrac{}{\{\Xi_a\}x := \texttt{new}()\{\Xi_{a'}\}}\,(\text{New}) \quad x \notin \Xi_{f'}}{\{\Xi_a \star \Xi_{f'}\}x := \texttt{new}()\{\Xi_{a'} \star \Xi_{f'}\}}\,(\text{Fr } \Xi_{f'}) \quad \{\Xi_{a'} \star \Xi_{f'}\}C'\{\Xi''\}}{\{\Xi_a \star \Xi_{f'}\}x := \texttt{new}();\, C'\{\Xi''\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_f\}C;\, x := \texttt{new}();\, C'\{\Xi''\}}\,(\text{Seq})$$

Figure 18: LateAllocation's implementation

$$\dfrac{\dfrac{\{\Xi\star\Theta\}C\{\Xi''\star\Theta\}\quad\dfrac{\dfrac{\{\Xi''\}C'\{\Xi'\}}{\{\Xi''\star\Theta\}C'\{\Xi'\star\Theta\}}\,(\text{Frame }\Theta)\quad\dfrac{\Xi'\star\Theta\vdash\Xi'\star\Theta}{\{\Xi'\star\Theta\}\epsilon\{\Xi'\star\Theta\}}\,(\text{Empty})}{\{\Xi''\star\Theta\}C';\,\epsilon\{\Xi'\star\Theta\}}\,(\text{Seq})}{\{\Xi\star\Theta\}C;\,C';\,\epsilon\{\Xi'\star\Theta\}}\,(\text{Seq})}{\{\Xi\}\texttt{with}\ r_\Theta\ \texttt{do}\ C;\,C';\,\epsilon\ \texttt{endwith}\{\Xi'\}}\,(\text{Region})$$

$$\downarrow \underline{\text{EarlyLockReleasing}}$$

$$\dfrac{\dfrac{\dfrac{\{\Xi\star\Theta\}C\{\Xi''\star\Theta\}}{\{\Xi\}\texttt{with}\ r_\Theta\ \texttt{do}\ C\ \texttt{endwith}\{\Xi''\}}\,(\text{Region})}{\{\Xi\}\texttt{with}\ r_\Theta\ \texttt{do}\ C\ \texttt{endwith}\{\Xi''\}}\,(\text{Frame }\top\restriction\texttt{emp})\quad\dfrac{\dfrac{\{\Xi''\}C'\{\Xi'\}}{\{\Xi''\}C'\{\Xi'\}}\,(\text{Frame }\top\restriction\texttt{emp})\quad\dfrac{\Xi'\vdash\Xi'}{\{\Xi'\}\epsilon\{\Xi'\}}\,(\text{Empty})}{\{\Xi''\}C';\,\epsilon\{\Xi'\}}\,(\text{Seq})}{\{\Xi\}\texttt{with}\ r_\Theta\ \texttt{do}\ C\ \texttt{endwith};\,C';\,\epsilon\{\Xi'\}}\,(\text{Seq})$$

Figure 19: EarlyLockReleasing's implementation

$$\dfrac{\dfrac{\dfrac{\{\Xi\}C\{\Xi''\}}{\{\Xi\star\Theta\}C\{\Xi''\star\Theta\}}\,(\text{Frame }\Theta)\quad\{\Xi''\star\Theta\}C'\{\Xi'\star\Theta\}}{\{\Xi\star\Theta\}C;\,C'\{\Xi'\star\Theta\}}\,(\text{Seq})}{\{\Xi\}\texttt{with}\ r_\Theta\ \texttt{do}\ C;\,C'\ \texttt{endwith}\{\Xi'\}}\,(\text{Region})$$

$$\downarrow \underline{\text{LateLockAcquirement}}$$

$$\dfrac{\dfrac{\{\Xi\}C\{\Xi''\}}{\{\Xi\}C\{\Xi''\}}\,(\text{Frame }\top\restriction\texttt{emp})\quad\dfrac{\{\Xi''\star\Theta\}C'\{\Xi'\star\Theta\}}{\{\Xi''\}\texttt{with}\ r_\Theta\ \texttt{do}\ C'\ \texttt{endwith}\{\Xi'\}}\,(\text{Region})}{\{\Xi\}C;\,\texttt{with}\ r_\Theta\ \texttt{do}\ C'\ \texttt{endwith}\{\Xi'\}}\,(\text{Seq})$$

Figure 20: LateLockAcquirement's implementation

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

$$\downarrow \underline{\text{TemporalLocality}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\Xi\}C\{\Xi'\}\quad\dfrac{\{\Xi'\}C''\{\Xi''\}\quad\dfrac{\Xi''\vdash\Xi''}{\{\Xi''\}\epsilon\{\Xi''\}}\,(\text{Empty})}{\{\Xi'\}C'';\,\epsilon\{\Xi''\}}\,(\text{Seq})}{\{\Xi\}C;\,C'';\,\epsilon\{\Xi''\}}\,(\text{Seq})}{\{\Xi\star\Theta\}C;\,C'';\,\epsilon\{\Xi''\star\Theta\}}\,(\text{Fr }\Theta)\quad\dfrac{\dfrac{\{\Theta\}C'\{\Theta'\}}{\{\Xi''\star\Theta\}C'\{\Xi''\star\Theta'\}}\,(\text{Fr }\Xi'')\quad\{\Xi''\star\Theta'\}C'''\{\Xi_p\}}{\{\Xi''\star\Theta\}C';\,C'''\{\Xi_p\}}\,(\text{Seq})}{\{\Xi\star\Theta\}C;\,C'';\,\epsilon;\,C';\,C'''\{\Xi_p\}}\,(\text{Seq})$$

Figure 21: TemporalLocality's implementation to be applied before FactorizeFrames

$$\dfrac{\dfrac{\{\Xi\}C\{\Xi'\}}{\{\Xi\star\Theta\star\Theta''\}C\{\Xi'\star\Theta\star\Theta''\}}\,(\text{Fr }\Theta\star\Theta'')\quad\dfrac{\dfrac{\{\Theta\}C'\{\Theta'\}}{\{\Xi'\star\Theta\}C'\{\Xi'\star\Theta'\}}\,(\text{Fr }\Xi')\quad\dfrac{\dfrac{\{\Xi'\}C''\{\Xi''\}}{\{\Xi'\star\Theta'\}C''\{\Xi''\star\Theta'\}}\,(\text{Fr }\Theta')\quad\dfrac{\Xi''\star\Theta'\vdash\Xi''\star\Theta'}{\{\Xi''\star\Theta'\}\epsilon\{\Xi''\star\Theta'\}}\,(\text{Empty})}{\{\Xi'\star\Theta'\}C'';\,\epsilon\{\Xi''\star\Theta'\}}\,(\text{Seq})}{\{\Xi'\star\Theta'\}C';\,C'';\,\epsilon\{\Xi''\star\Theta'\}}\,(\text{Seq})\quad\dfrac{\{\Xi''\star\Theta\star\Theta''\}C'''\{\Xi_p\}}{\{\Xi'\star\Theta\star\Theta''\}C';\,C'';\,\epsilon;\,C'''\{\Xi_p\}}\,(\text{Seq})}{\{\Xi\star\Theta\star\Theta''\}C;\,C';\,C'';\,\epsilon;\,C'''\{\Xi_p\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{TemporalLocality}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\Xi\}C\{\Xi'\}\quad\dfrac{\{\Xi'\}C''\{\Xi''\}\quad\dfrac{\Xi''\vdash\Xi''}{\{\Xi''\}\epsilon\{\Xi''\}}\,(\text{Empty})}{\{\Xi'\}C'';\,\epsilon\{\Xi''\}}\,(\text{Seq})}{\{\Xi\}C;\,C'';\,\epsilon\{\Xi''\}}\,(\text{Seq})}{\{\Xi\star\Theta\star\Theta''\}C;\,C'';\,\epsilon\{\Xi''\star\Theta\star\Theta''\}}\,(\text{Fr }\Theta\star\Theta'')\quad\dfrac{\dfrac{\{\Theta\}C'\{\Theta'\}}{\{\Xi''\star\Theta\star\Theta''\}C'\{\Xi''\star\Theta'\star\Theta''\}}\,(\text{Fr }\Xi''\star\Theta'')\quad\{\Xi''\star\Theta'\star\Theta''\}C'''\{\Xi_p\}}{\{\Xi''\star\Theta\star\Theta''\}C';\,C'''\{\Xi_p\}}\,(\text{Seq})}{\{\Xi\star\Theta\star\Theta''\}C;\,C'';\,\epsilon;\,C';\,C'''\{\Xi_p\}}\,(\text{Seq})$$

Figure 22: TemporalLocality's implementation to be applied after FactorizeFrames

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803