Rewrite Rules supporting Conditional Statements for Automatic Parallelization

Pascal Kuyten
pascal@kuyten.com

ABSTRACT

Hurlin recently proposed a technique for automatic parallelization based on separation logic [9]. This paper proposes an enhancement of the mechanism for situations where conditional statements are used in specifications of programs. With the proposed changes, specifications containing conditional statements are supported and programs can be automatically parallelized. Two approaches are identified and a preferable one is suggested. The proposed extension is illustrated by an exemplifying program and has been implemented.

Keywords

Proof Rewriting, Separation Logic, Automatic Parallelization and Optimization, Conditional Hoare Triples

1. INTRODUCTION

Current trend of multi-core systems and computational needs have led to the development of parallel systems. Parallel architectures are designed to run on multi-core systems as well as across network applications. Since manually writing parallel systems is much harder than writing sequential systems, automatic parallelization might provide effective and sound speedups [4, 1, 9].

Several techniques are proposed for automatic parallelization [7, 10, 6]. These techniques use pointer analysis to prove that if mutations on data done by a part of a program do not interfere with other parts of the program, these blocks can run in parallel. Once it is known which part of the data is accessed and which part is not, parallelization can be done in an automatic manner.

Hurlin proposes a new approach to automatic parallelization that uses separation logic to detect disjoint access to the heap [9].

This paper proposes an enhancement of the mechanism to support parallelization when conditional statements are used in specifications.

The following sections are outlined as follows: the mechanism of automatic parallelization using separation logic is presented in section 2, the limitations of this mechanism when using conditional statements is described in section 3, an extension to the mechanism is proposed in section 4 and a conclusion is drawn in section 5.

2. BACKGROUND INFORMATION

The behavior of a program can be specified by pre- and postconditions [8]. Pre- and postconditions are formulas describing the heap and store at method entry and exit. The heap and the store describe the data structure of a program. To check whether the pre- and postconditions are valid, they should be verified against the actual code.

Smallfoot is a framework used for automatic verification of a annotated programs written in Smallfoot's language [2, 3]. Smallfoot's verifier takes an annotated (i.e. pre- and postconditions) program and returns, if this program is correct, a proof. The annotations are written in separation logic, the proof is constructed using Hoare rules.

A correct program is described by a Hoare triple, written as:

$$\{X\}C\{\Theta\}$$

where X and Θ are formulas and C is a program. Such a triple states that if program C starts with a heap described by the precondition X, and if C terminates, the resulting heap is described by postcondition Θ [8].

The proof of a verified program can be transformed by rewrite rules. These rules induce local changes, which have no effect on specifications, but change the code. Because the proof of a verified program consists of Hoare triples, the proof include both annotations as well as actual commands. By rewriting parts of the proof the code of the program can be changed [9].

In other words a verified program can be rewritten in such a way that its pre- and postcondition remain valid, but the code itself has changed. In this way code can be parallelized.

2.1 Heaps

Heaps and stores describe the data structure of a program.

Heaps are partial functions from pointers (Loc) to finite collections (Fields) of non-addressable values (Val) and pointers. Stores are total functions from variables (Var) to non-addressable values and pointers (see figure 1) [3].

$$\begin{aligned} \mathsf{Heaps} &\stackrel{\text{def}}{=} \mathsf{Loc} \stackrel{\text{lim}}{\rightharpoonup} (\mathsf{Fields} \rightarrow \mathsf{Val} \cup \mathsf{Loc}) \\ \mathsf{Stores} &\stackrel{\text{def}}{=} \mathsf{Var} \rightarrow \mathsf{Val} \cup \mathsf{Loc} \end{aligned}$$

Figure 1: Definition of the heap

For example, consider a store s that maps variables x and y to address 42 and 43:

$$s \stackrel{\text{def}}{=} x \mapsto 42, y \mapsto 43$$

Then, consider a heap h that maps fields f and g of the cell at address 42 to 3 and 5 and maps fields f and g of the cell at address 43 to 1 and 2:

Figure 3 depicts heap h and store s graphically.

11th Twente Student Conference on IT, Enschede 29th June, 2009

$$h \stackrel{\text{def}}{=} 42 \mapsto \begin{pmatrix} f \mapsto 3 \\ g \mapsto 5 \end{pmatrix}, \quad 43 \mapsto \begin{pmatrix} f \mapsto 1 \\ g \mapsto 2 \end{pmatrix}$$

Figure 2: Heap h at address 42 and 43

(a) $h' : x \mapsto 42$ (b) $h'' : y \mapsto 43$

Figure 3: Heap $h = h' \cup h''$ and store s

2.2 Separation Logic

In separation logic, formulas describe the heap [12]. This relation between formulas and heaps is given by the semantics of formulas, i.e. by $\vDash$'s definition (see figure 4). The syntax of formulas is given in figure 7.

$$[\![x]\!]s \stackrel{\text{def}}{=} s(x) \quad [\![\texttt{nil}]\!]s \stackrel{\text{def}}{=} \texttt{nil}$$

$s, h \vDash \Pi_0 \wedge \Pi_1$	$\stackrel{\text{def}}{=}$	$s, h \vDash \Pi_0 \wedge s, h \vDash \Pi_1$
$s, h \vDash E_0 \mapsto [f_1 : E_1, \ldots, f_k : E_k]$	$\stackrel{\text{def}}{=}$	$h = [[\![E_0]\!]s \to r]$ where $r(f_i) = [\![E_i]\!]s$ for $i \in 1..k$
$s, h \vDash \texttt{emp}$	$\stackrel{\text{def}}{=}$	$h = \emptyset$
$s, h \vDash \Sigma_0 \star \Sigma_1$	$\stackrel{\text{def}}{=}$	$\exists h_0 h_1 . h = h_0 \star h_1$ and $s, h_0 \vDash \Sigma_0$ and $s, h_1 \vDash \Sigma_1$
$s, h \vDash \Pi \mathbin{¦} \Sigma$	$\stackrel{\text{def}}{=}$	$s, h \vDash \Pi$ and $s, h \vDash \Sigma$

Figure 4: Semantics of formulas

Top-level formulas Ξ and Θ are pairs of a pure formula Π and a spatial formula Σ. Π represents equality and inequality of expressions (e.g. $E \neq nil$). Multiple pure formulas are combined using the $\wedge$-operator.

Spatial formulas Σ represent facts about the heap. A simple (S) spatial formula: $E \mapsto [\rho]$ represents a heap with one cell at address E and content ρ. This is formalized in the semantics by imposing $h = [E \mapsto \rho]$. Here ρ is a sequence of adjacent fields.

To exemplify the relation between heaps and formulas, consider figure 2 and the following formulas:

$$X = x \mapsto [f : 3, g : 5] = x \mapsto 3, 5$$

$$\Theta = y \mapsto [f : 1, g : 2] = y \mapsto 1, 2$$

Now the relation is given by the semantics (where h is $h' \cup h''$):

$$s, h' \vDash X \quad \text{and} \quad s, h'' \vDash \Theta$$

The logical operation $X \star \Theta$ asserts that X and Θ hold for disjoint portions of the addressable heap. Names for this operator are *separating conjunction*, *independent* or *spatial* conjunction. $X \star \Theta$ enforces X to be disjoint to Θ: no heap cells found in X are also in Θ [3].

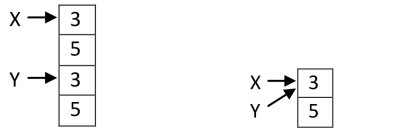
To illustrate the $\star$-operator, consider a heap h and store s: $s, h \vDash X$ and $s, h \vDash \Theta$ where $X = x \mapsto 3, 5$ and $\Theta = y \mapsto 3, 5$:

$$h \stackrel{\text{def}}{=} A \mapsto \begin{pmatrix} f \mapsto 3 \\ g \mapsto 5 \end{pmatrix}, \quad A' \mapsto \begin{pmatrix} f \mapsto 3 \\ g \mapsto 5 \end{pmatrix}$$

$$s \stackrel{\text{def}}{=} x \mapsto A, y \mapsto A'$$

In this example two cases are possible depending on the equality of A and A':

$s, h \vDash X \star \Theta$	:	A is **not** equal to A'	figure 5(a)
$s, h \vDash X \wedge s, h \vDash \Theta$	:	A is equal to A'	figure 5(b)



(a) $s, h \vDash X \star \Theta$ (b) $s, h \vDash X \wedge s, h \vDash \Theta$

Figure 5: A more global view of the heap

Simple spatial formulas S include the tree predicate $\texttt{tree}(E)$, which is defined by an equivalence (see figure 6). This equivalence states that an expression is a tree when it's either *nil* or it has two children as trees. Thus $E = \texttt{nil} \mathbin{¦} \texttt{emp}$ can also be written as $\texttt{tree}(E)$.

$$\texttt{tree}(\tau) \Leftrightarrow (\tau = \texttt{nil} \wedge \texttt{emp}) \vee$$
$$(\exists l, r . \tau \to l : l, r : r \star \texttt{tree}(l) \star \texttt{tree}(r))$$

Figure 6: Tree predicate and its equivalence

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

Figure 7: Syntax of formulas

2.3 Hoare Logic

Hoare logic describes the behavior of programs using triples and rules.

Hoare triples are written $\{X\}C\{\Theta\}$ where X and Θ are formulas and C is a program. Such a triple states that if program C starts with a heap described by the precondition X, and if C terminates, the resulting heap is described by postcondition Θ [8].

An example of a triple is: $\{x < N\}x := 2 \cdot x\{x < 2 \cdot N\}$.

Triples for sequential programs are built using the sequence rule (see figure 8), the lower premise can be deduced from the upper ones. This rule means that if two Hoare triples describe a pair of adjacent commands, such that the first triple's postcondition is equal to the second triple's precondition (Ξ'), then these triples can be combined in a single Hoare triple.

2.3.1 Frame Rule

$$\frac{\{\Xi\}C\{\Xi'\} \qquad \{\Xi'\}C'\{\Theta\}}{\{\Xi\}C; C'\{\Theta\}.} \text{ Sequence}$$

Figure 8: A rule used for building sequential programs

The frame rule (see figure 9) extends the local assertions about the heap to global specifications by using separating conjunction ($\star$). This rule means that the part of the heap described by Θ is untouched by C's execution [11].

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

Figure 9: Frame rule

As an example, consider a program C that alters the heap by changing the first cell (from 3 to 8) and adding a cell (2 after 5). C's precondition is described by Ξ, and C's postcondition is described by ζ:

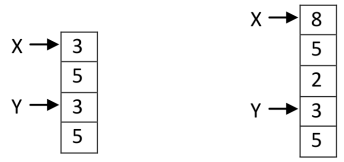
$$\Xi = x \mapsto 3, 5 \quad \zeta = x \mapsto 8, 5, 2$$

Using the frame rule C's pre- and postcondition (Ξ and ζ) can be extended to ($\Xi \star \Theta$ and $\zeta \star \Theta$):

$$\Theta = y \mapsto 3, 5$$

$$\frac{\{\Xi\}C\{\zeta\}}{\{\Xi \star \Theta\}C\{\zeta \star \Theta\}} \text{ Frame}$$

Θ is disjoint to Ξ and ζ, therefore the heap described by Θ is not accessed by C's execution (see figure 10).



(a) $\Xi \star \Theta$ (b) $\zeta \star \Theta$

Figure 10: C is framed by Θ

2.3.2 Parallel Rule

Figure 11 shows the Hoare rule for parallel statements. This rule shows that if two processes (C and C') access disjoint parts of the heap (Ξ and Ξ') and (Θ and Θ'), they can execute in parallel [12].

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

Figure 11: Parallel rule

As an example, figure 12 illustrates an update of two disjoint fields in parallel.

2.4 Smallfoot

Smallfoot is a framework used for automatic verification of a specified program [2, 3]. Smallfoot consists of three parts: 1) a while language, shown in section 2.4.1; 2) a set of formulas that is used to annotate programs written in this language, shown in section 2.2; and 3) a set of Hoare rules to verify that annotated programs are correct. Some of these rules are shown in section 2.3.

$$\begin{array}{c} [x \mapsto 3 \star y \mapsto 3] \\ [x \mapsto 3] \qquad [y \mapsto 3] \\ C \qquad \| \qquad C' \\ [x \mapsto 4] \qquad [y \mapsto 5] \\ [x \mapsto 4 \star y \mapsto 5] \end{array}$$

Figure 12: An update of two fields in parallel

$$p(\overline{E_1}; \overline{E_2})[\Xi]C[\Theta]$$

Figure 13: Smallfoot's program specification

A Smallfoot program is specified as shown in figure 13; where p is the program name; Ξ and Θ are formulas describing the heap; Ξ is the precondition; Θ is the postcondition; $\overline{E_1}$ and $\overline{E_2}$ are a comma-separated group of expressions (the former is passed by reference, the latter by value); and C is the actual program (written in the Smallfoot language).

2.4.1 Smallfoot language

Smallfoot's language specification is shown in figure 14.

$$\begin{array}{rcl} r & \in & \text{Resources} \\ x, y, z & \in & \text{Var} \\ E, F, G & ::= & \texttt{nil} \mid x \\ b & ::= & E = E \mid E \neq E \\ f, g, f_i, l, r, \ldots & \in & \text{Fields} \\ A & ::= & x := E \mid x := E \rightarrow f \mid E \rightarrow f := F \\ & \mid & x := \texttt{new}() \mid \texttt{dispose}(E) \\ C & ::= & A \mid \texttt{empty} \mid \texttt{if } b \texttt{ then } C \texttt{ else } C' \\ & \mid & \texttt{while}(b)\{C\} \mid \texttt{lock}(r) \mid \texttt{unlock}(r) \\ & \mid & p(\overline{E_1}; \overline{E_2}) \mid C;\ C \mid C\|C' \end{array}$$

Figure 14: Smallfoot's language specification

The language specifies several atomic commands A: $x := E$ makes x an alias of E; $x := E \rightarrow f$ assigns the content of field f at address E to x; $E \rightarrow f := F$ mutates the content of field f at address E to F; $x :=$ `new`() assigns the address of a new cell to x; and `dispose`(E) disposes the cell at address E.

Other commands C are: conditionals `if` b `then` C `else` C'; procedure calls $p(\overline{E_1}; \overline{E_2})$; sequential execution C; C; concurrent execution $C\|C'$; loops `while`(b)$\{C\}$; and resources acquirement and releasing `lock`(r) | `unlock`(r).

Smallfoot's resources sharing mechanism goes beyond the scope of this paper, see Berdine et al. for further references [3].

2.4.2 Verification mechanism

The Smallfoot verifier has information about how the heap is changed by each atomic commands, and it tries to compose a proof using Hoare rules. The atomic commands are described by the leaves of a proof tree and the top level annotation is described by the root.

Intuitively, Smallfoot's verifier has a Hoare rule for each atomic and non-atomic command (see figure 14). Hoare triples for commands are then combined by using the sequence rule (see figure 8) [2, 3].

If a program can be verified, a proof tree is generated, as the next section exemplifies.

2.4.3 Shallow

Shallow is a small program written in the Smallfoot language and shall be used for illustrative purposes (see figure 15). The program takes three trees (tree(t), tree(p) and tree(q)) and an expression (d) as input. This is formalized by Shallow's pre and postcondition (enclosed in square brackets) [1].

Shallow updates tree(p) and tree(q) using the method update (see figure 16). The recursive method mirror (see figure 17) disposes nodes in tree tree(t) lower than depth d and mirrors the resulting subtree (see figure 18).

The correctness of Shallow is described in figure 26 (see appendix B, upper tree). The tree shows applications of Hoare rules: frame and sequence (the or-rule will be explained in section 3). The root node of the proof tree is equal to the specification (see figure 15), showing that Shallow's specification is correct:

$$\{\texttt{tree}(t) \star \texttt{tree}(p) \star \texttt{tree}(q)\}$$
$$\texttt{update}(p;);\texttt{mirror}(t,d;);\texttt{update}(q;)$$
$$\{\texttt{tree}(p) \star \texttt{tree}(q) \star \text{if } d == 0 \text{ then emp else } \texttt{tree}(t)\}$$

```
shallow(t,p,q,d;)[tree(t) * tree(p) * tree(q)]{
    update(p;);
    mirror(t,d;);
    update(q;);
    } [ tree(p) * tree(q) *
        (if(d==0) then emp else tree(t))]
```

Figure 15: Shallow, executes on three disjoint trees.

```
update(t;)[tree(t)]{
    ..
}[tree(t)]
```

Figure 16: Update, increases all values in the tree by 1.

```
mirror(t,d;)[tree(t)]{
..
    if(d==0){
     tree_deallocate(t);
    } else {
        t->l = tl; t->r = tr;
        mirror(tl, d-1;);
        mirror(tr, d-1;);
        t->l = tr; t->r = tl;
    }
..
}[if(d==0) then emp else tree(t)]
```

Figure 17: Mirror, mirrors a subtree (with depth d) of a tree.

2.5 Proof Rewriting

The concept of proof rewriting is based upon the idea that Smallfoot proofs contain information about how the heap is accessed (because of the $\star$-operator). By combining and shifting code in an intelligent and automatic manner, verified programs can be optimized and parallelized [9].

Proof trees are rewritten using a set of local updates that are defined by rewrite rules (see figure 19). A rewrite rule consist of two proof trees, the upper tree describes a pattern to be matched (of the input), the lower describes the output to be generated when the pattern matches.

[1] Recall that the definition of a tree predicate was defined in figure 6.

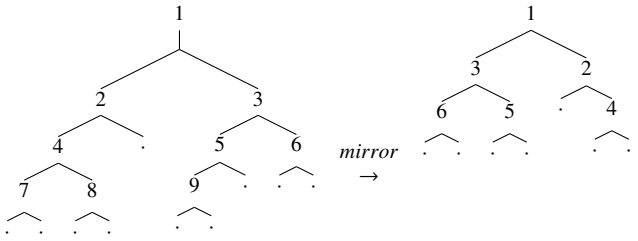


Figure 18: Mirror, executed on some tree with $d = 3$.

The proof of a verified program can be transformed by rewrite rules. These rules induce local changes, but have no effect on the code's specification. Because the proof of a verified program consists of Hoare triples, the proof includes both annotations as well as actual commands. By rewriting parts of the proof the program's code can be changed.

The whole procedure is as follows: given a program C, C is proven using Smallfoot, a proof tree $\mathcal{P}$ is generated if C can be verified. Then, $\mathcal{P}, C$ is rewritten into $\mathcal{P}', C'$, such that $\mathcal{P}'$ is a proof of C' and C' is a parallelized and optimized version of C. This procedure has been implemented in a tool called éterlou [9].

A rewrite rule will be explained in section 2.5.2.

2.5.1 Frames and Anti-frames

Proof trees of verified programs contain information about 1) parts of the heap which are accessed by commands, called anti-frames; and 2) parts of the heap which are not touched, called frames [5].

The proof tree of the Shallow program (see appendix B, figure 26, upper tree) contains an explicit description of frames and anti-frames. The anti-frames can be found in the pre- and postconditions of the leaves (e.g. tree(p)), the frames are indicated in applications of the frame rule (e.g. Frame tree(t) $\star$ tree(q)).

2.5.2 Parallelization

The rewrite rule for parallelization (see figure 19) can be understood as follows: Given a proof of the sequential program C; C' this proof is rewritten into a proof of the parallel program $C \| C'$. The newly obtained proof stays valid because it is a valid instance of the Hoare rules and its leaves are included in the leaves of the input tree [9].

The rewrite rule for parallelization uses the information of frames and anti-frames, for example the postcondition of C is matched against the frame of C' (Θ).

Application of this rule on Shallow's proof tree (see figure 15) is shown in figure 26 (appendix B, third and fourth tree). The upper postcondition of the left frame's update(p;) of Seq (tree(p)) is equal to the frame in the right node of Seq. Also the upper precondition of the right frame's mirror(t, d;)$\|$update(q;) of Seq (tree(t) $\star$ tree(q)) is equal to the frame in the left node of Seq. This implies that the left and right node of Seq can be parallelized. The rewritten tree is depicted in the last tree of (see appendix B, figure 26).

2.5.3 Common frames

Every leaf in a proof tree is preceded by an application of frame (see section 2.5.1). When multiple commands are present, redun-

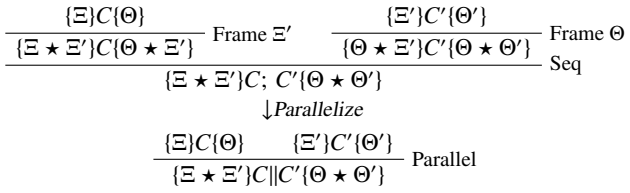


Figure 19: Rewrite rule for parallelization

dancy in frames can appear. This redundancy should be removed for the rewrite rules to work [9].

For example, consider the rewrite rule for parallelization (see figure 19): C's postcondition is matched against C''s frame (Θ). When C''s frame contains more elements (i.e. $\Theta \star \zeta$) the matching fails. Even though C and C' can be parallelized.

A solution is to move the redundant frame ζ to a separate frame-node by applying a separate frame rule, this solution is implemented by the rewrite rule illustrated in figure 20. This rule means that when ζ is present in both frames of the input tree, ζ is moved to a separate application of the frame rule at the root in the output tree [9].

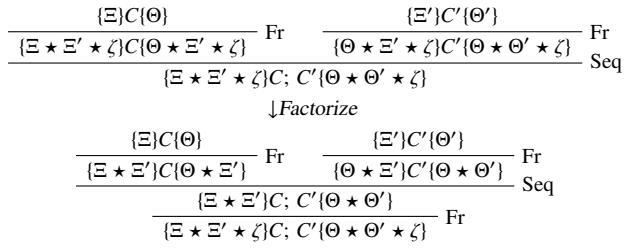


Figure 20: Rewrite rule for removing redundancy

An example of this redundancy can be found in the proof tree of the Shallow program (see appendix B, figure 26, first tree). This tree contains the frames $\texttt{tree}(q) \star \texttt{tree}(p)$ (in the center) and $\texttt{tree}(p) \star \texttt{tree}(t)$ (on the right), these frames share $\texttt{tree}(p)$ [2]. Consequently $\texttt{tree}(p)$ is moved to a separate frame-node by applying the rewrite rule for factorization (see figure 26, appendix B, second tree).

3. FINDINGS

The rewrite rules for factorization and parallelization, described by Hurlin, have several limitations [9]. This paper proposes an extension to support conditional annotation of programs. Shallow (see section 2.4.3) is an example where the proposed solution solves this problem.

Smallfoot's annotation includes conditional statements described as:

$$\text{if } b \text{ then } \Phi_a \text{ else } \Phi_b$$

The approach based on Hurlin's work, does not handle such formulas.

As an example, consider a program C, where Ξ describes C's precondition; and if b then Φ_a else Φ_b describes C's postcondition. Then, Φ_a describes C's postcondition when b is true; and Φ_b describes C's postcondition when b is false.

[2]Note that $\texttt{tree}(p)$ is also the anti-frame of $\texttt{update}(p;)$ (see section 2.5.1).

Conditional statements in postconditions are handled by Smallfoot's verifier using a case split:

$$\frac{\mathcal{P} \qquad \mathcal{P}'}{\{\Xi\}C\{\Theta\}} \text{ Or}$$

This rule means that $\{\Xi\}C\{\Theta\}$ can be proven either by $\mathcal{P}$ or by $\mathcal{P}'$.

The relation between conditional statements in postconditions and frames is shown in figure 25 (appendix B). C's postcondition (Φ) induces a split case on C''s frame. C's frame is either Φ_a (in the center) or Φ_b (on the right).

The proof tree (see appendix B, figure 26, first tree, on the right) of the Shallow program, contains such a split case on the frame of $\texttt{update}(q;)$. This split case is induced by $\texttt{mirror}(t, d;)$'s conditional postcondition:

$$\text{if } d == 0 \text{ then } \texttt{emp} \text{ else } \texttt{tree}(t)$$

The generated proof describes that $\texttt{update}(q)$ is framed conditionally by either ($\texttt{tree}(p)$) or ($\texttt{tree}(p) \star \texttt{tree}(t)$). Note that $\texttt{mirror}(t, d;)$'s and $\texttt{update}(q;)$'s common frame is $\texttt{tree}(p)$. This common frame should be factorized before other rewrite rules can be used.

Current rewrite rules do not match proof trees containing conditional statements and split cases. The shape of current rules (e.g. figure 20) describe proof trees containing only applications of sequence and frame rules, thus excluding proof trees having split cases. Éterlou's rewrite engine stops when split cases are found.

4. SOLUTION

There are two rewrite rules responsible for automatic parallelization: Factorize, which prepares the proof tree by removing common frames and Parallelize, the actual parallelization rewrite rule [9]. Both rewrite rules need to be extended for the procedure to work with conditional statements (see section 3).

4.1 Factorization

There are two possible locations where an split case (see appendix B, figure 25) can be factorized.

1. at the root of the split case's application, this shall remove the common frame ζ_1 of Φ_a and Φ_b, followed by a second factorization at the root of the sequence-node, removing the common frame of Ξ_f and ζ_2 (see appendix C, figure 27).
2. at the root of the sequential's application, this shall remove the common frame ζ of Ξ_f, Φ_a and Φ_b in a single pass (see figure 21).

The first type of factorization involves a smaller part of the proof tree than the second one. When choosing for simplicity type one is preferable. Note that ζ_1 and Ξ_f might not share a common frame ζ_2. In that case a second factorization cannot be applied, which results in an unnecessary frame-node ζ_1.

The second type will factorize the three frames (Ξ_f, Φ_a and Φ_b) at once. In the case the three trees share a common frame (ζ), the resulting proof tree will have four frames (against five frames in the case of successful application of type one). More importantly, type two does include split cases containing a possible continuation (sequences will be explained in section 4.2).

Because factorization of type two is more general this rewrite rule is preferable. Besides, other rewrite rules will become smaller

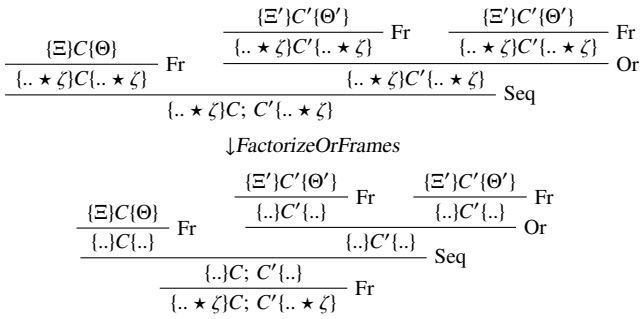


Figure 21: Rewrite rule for removing redundancy including split cases, type two

when factorization of type two is used, which is further explained in section 4.3.

The simplified rewrite rule for factorization of type two is given in figure 21 (see appendix C). A more complete rewrite rule, including a possible continuation, is given in figure 23 (see appendix B).

As an example Shallow's proof tree (see appendix B, figure 26, first tree) is factorized by applying FactorizeOrFrames (see appendix A. The resulting tree (the second tree) after applying FactorizeOrFrames shows that $\texttt{tree}(p)$ is the common frame:

$$\Xi_f \Leftrightarrow \texttt{tree}(q) \star \texttt{tree}(p)$$
$$\Phi_a \Leftrightarrow \texttt{tree}(p) \quad \Phi_b \Leftrightarrow \texttt{tree}(p) \star \texttt{tree}(t)$$
$$\zeta \Leftrightarrow \texttt{tree}(p)$$

4.2 Sequences

The rewrite rule shown in figure 21 is too restrictive, because it does not consider a possible continuation after C'. The rule shown in figure 23 (see appendix A) does consider a possible continuation after C' (C'').

FactorizeOrFrames's input tree includes a split case for C''s frame (Θ_{fa} or Θ_{fb}) and C'''s precondition ($\Theta_p \star \Theta_{fa}$ or $\Theta_p \star \Theta_{fb}$). FactorizeOrFrames's output tree includes two split cases: one for C''s frame (Θ_{fa0} or Θ_{fb0}); and one for C'''s precondition ($\Theta_p \star \Theta_{fa}$ or $\Theta_p \star \Theta_{fb}$).

FactorizeOrFrames's output tree includes an application of the frame rule for C and C''s common frame; and a separate application of the split case for C'''s precondition; allowing continuation as C'''s frame is left unchanged.

See Hurling for further references about continuation [9].

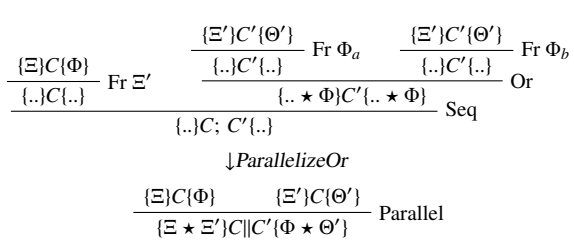
4.3 Parallelization

In Hoare logic, the parallel rule describes that when two commands (C and C') do not interfere then they can be executed concurrently (see section 2.3.2). Consider that when C's postcondition is described by a conditional statement [3], C and C' can only be parallelized, when C's access to the heap (Ξ, Φ_a and Φ_b) are disjoint to C''s access to the heap (Ξ' and Θ').

The simplified rewrite rule for parallelization is given in figure 22. The fact that C and C' access disjoint parts of the heap is formalized by the tree applications of frame (Ξ', Φ_a and Φ_b). A complete rewrite rule, including a possible continuation, is given in appendix B, figure 24.

[3] $\texttt{if}\ b\ \texttt{then}\ \Phi_a\ \texttt{else}\ \Phi_b$

ParallelizeOr's guard ensures that the two processes (C and C') access disjoint parts of the heap (see section 2.3.2).



Guard: $\Phi \Leftrightarrow \texttt{if}\ \beta\ \texttt{then}\ \Phi_a\ \texttt{else}\ \Phi_b$

Figure 22: Rewrite rule for parallelization including split cases

As an example Shallow's proof tree (see section 2.4.3) is parallelized by applying ParallelizeOr (see appendix A, figure 24) on Shallow's factorized proof tree (see appendix B, figure 26, second tree). The resulting tree shows that $\texttt{mirror}(t, d;)$ and $\texttt{update}(q;)$ are parallelized. In this case the guards are instantiated as follows:

$\Theta' \Leftrightarrow \texttt{t}(q)$ The anti-frame of $\texttt{update}(q;)$ is equal to the frame of $\texttt{mirror}(t, d;)$

$\Phi_a \Leftrightarrow \texttt{emp}$ The frame of the left leaf of the split case is equal to the first part of the conditional anti-frame (Φ) of $\texttt{mirror}(t, d;)$

$\Phi_b \Leftrightarrow \texttt{t}(t)$ The frame of the right leaf of the split case (Φ_b) is equal to the second part of the conditional anti-frame (Φ) of $\texttt{mirror}(t, d;)$

One can note that ParallelizeOr is not able to parallelize $\texttt{update}(p;)$ and $\texttt{mirror}(t, d;)$, this is done in the last rewrite step using the Parallelize rule (see figure 19).

In practise should the proposed rewrite rule for parallelization be applied after the rule for factorization of type two. Using the rule for factorization of type one is cumbersome, because it requires to write a dedicated parallelization rule (to match frame ζ_2). Hence, factorization of type two is preferable.

5. CONCLUSION

The mechanism proposed by Hurlin where separation logic is used for automatic parallelization has several limitations. This paper addressed one of these limitations: we allow the usage of conditional statements in annotations.

With the proposed changes a specification's proof tree containing conditional statements is supported and the program can be automatically rewritten to parallelize it. Two approaches for factorization of conditional statements have been described, and the factorization of type two is identified as the preferable one.

Shallow, an example program, has been used to illustrate this paper findings and proposed solution. This solution has been implemented in the éterlou tool and tested for soundness. There are other rules than factorization and parallelization and they might have to be changed to allow conditional statements. This is left open for further research.

6. ACKNOWLEDGMENTS

I would like to thank Clément Hurlin for his guidance during the research and writing of this paper, Michael Weber, for his

patience while listening and giving sharp feedback, as well as Hasan Sozer, Bas van Gijzel, Charl de Leur, Sander Bockting and Wouter IJgosse for reviewing my work.

REFERENCES

[1] B. Armstrong and R. Eigenmann. Application of Automatic Parallelization to Modern Challenges of Scientific Computing Industries. In *ICPP*, pages 279–286, 2008.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO*, pages 115–137, 2005.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*, pages 52–68, 2005.

[4] U. Bondhugula, M. M. Baskaran, A. Hartono, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Towards effective automatic parallelization for multicore systems. In *IPDPS*, pages 1–5, 2008.

[5] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

[6] R. Gupta, S. Pande, K. Psarris, and V. Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25(13-14):1741–1783, 1999.

[7] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):35–47, 1990.

[8] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969.

[9] C. Hurlin. Automatic parallelization and optimization of programs by proof rewriting. In *SAS*, Lecture Notes in Computer Science. Springer-Verlag, August 2009.

[10] K. Koskimies, editor. *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1383 of *Lecture Notes in Computer Science*. Springer, 1998.

[11] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL*, pages 1–19, 2001.

[12] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74, 2002.

APPENDIX A: REWRITE RULES

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\,\text{Frame}\ \Xi_f \qquad \dfrac{\dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_{fa}\}C'\{\Theta_p \star \Theta_{fa}\}}\,\text{Frame}\ \Theta_{fa} \qquad \{\Theta_p \star \Theta_{fa}\}C''\{\Xi'\}}{\{\Theta_a \star \Theta_{fa}\}C';\ C''\{\Xi'\}}\,\text{Seq} \qquad \dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_{fb}\}C'\{\Theta_p \star \Theta_{fb}\}}\,\text{Frame}\ \Theta_{fb} \qquad \{\Theta_p \star \Theta_{fb}\}C''\{\Xi'\}}{\{\Theta_a \star \Theta_{fb}\}C';\ C''\{\Xi'\}}\,\text{Seq}}{\{\Theta_a \star \Theta_f\}C';\ C''\{\Xi'\}}\,\text{Or}}{\{\Xi_a \star \Xi_f\}C;\ C';\ C''\{\Xi'\}}\,\text{Seq}$$

$$\downarrow \textit{FactorizeOrFrames}$$

$$\dfrac{\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_{f_0}\}C\{\Xi_p \star \Xi_{f_0}\}}\,\text{Frame}\ \Xi_{f_0} \qquad \dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_{fa0}\}C'\{\Theta_p \star \Theta_{fa0}\}}\,\text{Frame}\ \Theta_{fa0} \qquad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_{fb0}\}C'\{\Theta_p \star \Theta_{fb0}\}}\,\text{Frame}\ \Theta_{fb0}}{\{\Theta_a \star \Theta_{f_0}\}C'\{\Theta_p \star \Theta_{f_0}\}}\,\text{Or}}{\{\Xi_a \star \Xi_{f_0}\}C;\ C'\{\Theta_p \star \Theta_{f_0}\}}\,\text{Seq} \qquad \dfrac{\{\Theta_p \star \Theta_{fa}\}C''\{\Xi'\} \qquad \{\Theta_p \star \Theta_{fb}\}C''\{\Xi'\}}{\{\Theta_p \star \Theta_f\}C''\{\Xi'\}}\,\text{Or}}{\{\Xi_a \star \Xi_f\}C;\ C';\ C''\{\Xi'\}}\,\text{Seq}$$

Guard: $\Xi_f \Leftrightarrow \Xi_{f0} \star \Xi_c$ and $\Theta_{fa} \Leftrightarrow \Xi_{fa0} \star \Xi_c$ and $\Theta_{fb} \Leftrightarrow \Xi_{fb0} \star \Xi_c$

Figure 23: Rewrite rule to factorize applications of frame and split case

$$\dfrac{\dfrac{\{\Xi\}C\{\Theta\}}{\{\Xi \star \Xi'\}C\{\Theta \star \Xi'\}}\,\text{Frame}\ \Xi' \qquad \dfrac{\dfrac{\dfrac{\{\Xi'\}C'\{\Theta'\}}{\{\Theta_a \star \Xi'\}C'\{\Theta_a \star \Theta'\}}\,\text{Frame}\ \Theta_a \qquad \{\Theta_a \star \Theta'\}C''\{\Xi''\}}{\{\Theta_a \star \Xi'\}C';\ C''\{\Xi''\}}\,\text{Seq} \qquad \dfrac{\dfrac{\{\Xi'\}C'\{\Theta'\}}{\{\Theta_b \star \Xi'\}C'\{\Theta_b \star \Theta'\}}\,\text{Frame}\ \Theta_b \qquad \{\Theta_b \star \Theta'\}C''\{\Xi''\}}{\{\Theta_b \star \Xi'\}C';\ C''\{\Xi''\}}\,\text{Seq}}{\{\Theta \star \Xi'\}C';\ C''\{\Xi''\}}\,\text{Or}}{\{\Xi \star \Xi'\}C;\ C';\ C''\{\Xi''\}}\,\text{Seq}$$

$$\downarrow \textit{ParallelizeOr}$$

$$\dfrac{\dfrac{\{\Xi\}C\{\Theta\} \qquad \{\Xi'\}C'\{\Theta'\}}{\{\Xi \star \Xi'\}C \| C'\{\Theta \star \Theta'\}}\,\text{Parallel} \qquad \dfrac{\{\Theta_a \star \Theta'\}C''\{\Xi''\} \qquad \{\Theta_b \star \Theta'\}C''\{\Xi''\}}{\{\Theta \star \Theta'\}C''\{\Xi''\}}\,\text{Or}}{\{\Xi \star \Xi'\}C \| C';\ C''\{\Xi''\}}\,\text{Seq}$$

Guard: $\Theta \Leftrightarrow \texttt{if}\ \beta\ \texttt{then}\ \Phi_a\ \texttt{else}\ \Phi_b$ and $\Theta_a \Leftrightarrow \beta \upharpoonright \Phi_a$ and $\Theta_b \Leftrightarrow !\beta \upharpoonright \Phi_b$

Figure 24: Rewrite rule to parallelize applications of frame and split case

APPENDIX B: PROOF TREES

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Phi\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\ \text{Frame } \Xi_f \qquad \dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Phi_a \star \Theta_a\}C'\{\Phi_a \star \Theta_p\}}\ \text{Frame } \Phi_a \qquad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Phi_b \star \Theta_a\}C'\{\Phi_b \star \Theta_p\}}\ \text{Frame } \Phi_b}{\{\Phi \star \Theta_a\}C'\{\Phi \star \Theta_p\}}\ \text{Or}}{\{\Xi_a\}C;\ C'\{\Phi \star \Theta_p\}}\ \text{Seq}$$

$\Phi \Leftrightarrow \texttt{if}\ \beta\ \texttt{then}\ \Phi_a\ \texttt{else}\ \Phi_b$

Figure 25: Shape of an split case

$$\dfrac{\dfrac{\{\mathtt{t}(p)\}f_1\{\mathtt{t}(p)\}}{\{\mathtt{t}(t) \star \mathtt{t}(p) \star \mathtt{t}(q)\}f_1\{\mathtt{t}(t) \star \mathtt{t}(q) \star \mathtt{t}(p)\}}\ \text{Fr } \mathtt{t}(t) \star \mathtt{t}(q) \qquad \dfrac{\dfrac{\{\mathtt{t}(t)\}f_2\{\Phi\}}{\{\mathtt{t}(t) \star \mathtt{t}(q) \star \mathtt{t}(p)\}f_2\{\mathtt{t}(q) \star \mathtt{t}(p) \star \Phi\}}\ \text{Fr } \mathtt{t}(q) \star \mathtt{t}(p) \qquad \dfrac{\dfrac{\dfrac{\{\mathtt{t}(q)\}f_3\{\mathtt{t}(q)\}}{\{\mathtt{t}(q) \star \mathtt{t}(p)\}f_3\{\mathtt{t}(p) \star \mathtt{t}(q)\}}\ \text{Fr } \mathtt{t}(p) \qquad \dfrac{\{\mathtt{t}(q)\}f_3\{\mathtt{t}(q)\}}{\{\mathtt{t}(q) \star \mathtt{t}(p) \star \mathtt{t}(t)\}f_3\{\mathtt{t}(p) \star \mathtt{t}(t) \star \mathtt{t}(q)\}}\ \text{Fr } \mathtt{t}(p) \star \mathtt{t}(t)}{\{\mathtt{t}(q) \star \mathtt{t}(p) \star \Phi\}f_3\{\mathtt{t}(p) \star \mathtt{t}(q) \star \Phi\}}\ \text{Or}}{\{\mathtt{t}(t) \star \mathtt{t}(q) \star \mathtt{t}(p)\}f_2; f_3\{\mathtt{t}(p) \star \mathtt{t}(q) \star \Phi\}}\ \text{Seq}}{\{\mathtt{t}(t) \star \mathtt{t}(p) \star \mathtt{t}(q)\}f_1; f_2; f_3\{\mathtt{t}(p) \star \mathtt{t}(q) \star \Phi\}}\ \text{Seq}$$

↓ *FactorizeOrFrames*

$$\dfrac{\dfrac{\{\mathtt{t}(p)\}f_1\{\mathtt{t}(p)\}}{\{\mathtt{t}(t) \star \mathtt{t}(p) \star \mathtt{t}(q)\}f_1\{\mathtt{t}(t) \star \mathtt{t}(q) \star \mathtt{t}(p)\}}\ \text{Fr } \mathtt{t}(t) \star \mathtt{t}(q) \qquad \dfrac{\dfrac{\dfrac{\{\mathtt{t}(t)\}f_2\{\Phi\}}{\{\mathtt{t}(t) \star \mathtt{t}(q)\}f_2\{\mathtt{t}(q) \star \Phi\}}\ \text{Fr } \mathtt{t}(q) \qquad \dfrac{\{\mathtt{t}(q)\}f_3\{\mathtt{t}(q)\}}{\{\mathtt{t}(q)\}f_3\{\mathtt{t}(q)\}}\ \text{Fr emp} \qquad \dfrac{\{\mathtt{t}(q)\}f_3\{\mathtt{t}(q)\}}{\{\mathtt{t}(q) \star \mathtt{t}(t)\}f_3\{\mathtt{t}(t) \star \mathtt{t}(q)\}}\ \text{Fr } \mathtt{t}(t)}{\{\mathtt{t}(q) \star \Phi\}f_3\{\mathtt{t}(q) \star \Phi\}}\ \text{Or}}{\{\mathtt{t}(t) \star \mathtt{t}(q)\}f_2; f_3\{\mathtt{t}(q) \star \Phi\}}\ \text{Seq} \qquad \dfrac{\{\mathtt{t}(t) \star \mathtt{t}(q) \star \mathtt{t}(p)\}f_2; f_3\{\mathtt{t}(p) \star \mathtt{t}(q) \star \Phi\}}{\{\mathtt{t}(t) \star \mathtt{t}(q) \star \mathtt{t}(p)\}f_2; f_3\{\mathtt{t}(p) \star \mathtt{t}(q) \star \Phi\}}\ \text{Fr } \mathtt{t}(p)}{\{\mathtt{t}(t) \star \mathtt{t}(p) \star \mathtt{t}(q)\}f_1; f_2; f_3\{\mathtt{t}(p) \star \mathtt{t}(q) \star \Phi\}}\ \text{Seq}$$

↓ *ParallelizeOr*

$$\dfrac{\dfrac{\{\mathtt{t}(p)\}f_1\{\mathtt{t}(p)\}}{\{\mathtt{t}(t) \star \mathtt{t}(p) \star \mathtt{t}(q)\}f_1\{\mathtt{t}(t) \star \mathtt{t}(q) \star \mathtt{t}(p)\}}\ \text{Fr } \mathtt{t}(t) \star \mathtt{t}(q) \qquad \dfrac{\dfrac{\{\mathtt{t}(t)\}f_2\{\Phi\} \qquad \{\mathtt{t}(q)\}f_3\{\mathtt{t}(q)\}}{\{\mathtt{t}(t) \star \mathtt{t}(q)\}f_2; f_3\{\mathtt{t}(q) \star \Phi\}}\ \text{Parallel}}{\{\mathtt{t}(t) \star \mathtt{t}(q) \star \mathtt{t}(p)\}f_2 \| f_3\{\mathtt{t}(p) \star \mathtt{t}(q) \star \Phi\}}\ \text{Fr } \mathtt{t}(p)}{\{\mathtt{t}(t) \star \mathtt{t}(p) \star \mathtt{t}(q)\}f_1; f_2 \| f_3\{\mathtt{t}(p) \star \mathtt{t}(q) \star \Phi\}}\ \text{Seq}$$

$\Phi \Leftrightarrow \text{if } d == 0 \text{ then } \texttt{emp} \text{ else } \mathtt{t}(t)$

↓ *Parallelize*

$$\dfrac{\{\mathtt{t}(p)\}f_1\{\mathtt{t}(p)\} \qquad \dfrac{\{\mathtt{t}(t)\}f_2\{\Phi\} \qquad \{\mathtt{t}(q)\}f_3\{\mathtt{t}(q)\}}{\{\mathtt{t}(t) \star \mathtt{t}(q)\}f_2 \| f_3\{\mathtt{t}(q) \star \Phi\}}\ \text{Parallel}}{\{\mathtt{t}(t) \star \mathtt{t}(p) \star \mathtt{t}(q)\}f_1 \| (f_2 \| f_3)\{\mathtt{t}(p) \star \mathtt{t}(q) \star \Phi\}}\ \text{Parallel}$$

$\mathtt{t}(\tau) \Leftrightarrow \mathtt{tree}(\tau)$
$\text{Fr } \Xi \Leftrightarrow \text{Frame } \Xi$
$f_1 \Leftrightarrow \mathtt{update}(p;)$
$f_2 \Leftrightarrow \mathtt{mirror}(t, d;)$
$f_3 \Leftrightarrow \mathtt{update}(q;)$

Figure 26: Proofs of the Shallow program.

APPENDIX C: AN ALTERNATIVE FOR THE FACTORIZATION REWRITE RULE

$$\dfrac{\dfrac{\{\Xi'\}C'\{\Theta'\}}{\{..\star\zeta_1\}C'\{..\star\zeta_1\}}\,\mathrm{Fr} \qquad \dfrac{\{\Xi'\}C'\{\Theta'\}}{\{..\star\zeta_1\}C'\{..\star\zeta_1\}}\,\mathrm{Fr}}{\{..\star\zeta_1\}C'\{..\star\zeta_1\}}\,\mathrm{Or}$$

$$\downarrow \mathit{FactorizeOrFrames}$$

$$\dfrac{\dfrac{\dfrac{\{\Xi'\}C'\{\Theta'\}}{\{..\}C'\{..\}}\,\mathrm{Fr} \qquad \dfrac{\{\Xi'\}C'\{\Theta'\}}{\{..\}C'\{..\}}\,\mathrm{Fr}}{\{..\}C'\{..\}}\,\mathrm{Or}}{\{..\star\zeta_1\}C'\{..\star\zeta_1\}}\,\mathrm{Fr}$$

(a) Factorizing ζ_1

$$\dfrac{\dfrac{\{\Xi\}C\{\Theta\}}{\{..\star\zeta_2\}C\{..\star\zeta_2\}}\,\mathrm{Fr} \qquad \dfrac{\dfrac{..}{\{..\}C'\{..\}}\,\mathrm{Or}}{\{..\star\zeta_2\}C'\{..\star\zeta_2\}}\,\mathrm{Fr}}{\{..\star\zeta_2\}C;\, C'\{..\star\zeta_2\}}\,\mathrm{Seq}$$

$$\downarrow \mathit{FactorizeFrames}$$

$$\dfrac{\dfrac{\dfrac{\{\Xi\}C\{\Theta\}}{\{..\}C\{..\}}\,\mathrm{Fr} \qquad \dfrac{\dfrac{..}{\{..\}C'\{..\}}\,\mathrm{Or}}{\{..\}C'\{..\}}\,\mathrm{Fr}}{\{..\}C;\, C'\{..\}}\,\mathrm{Seq}}{\{..\star\zeta_2\}C;\, C'\{..\star\zeta_2\}}\,\mathrm{Fr}$$

(b) Factorizing ζ_2

Figure 27: Rewrite rule for removing redundancy including split cases, type one