Certificates and Separation Logic

Martin Nordio[1], Cristiano Calcagno[2], and Bertrand Meyer[1]

[1] ETH Zurich, Switzerland `firstname.lastname@inf.ethz.ch`
[2] ETH Zurich, Imperial College London and Monoidics Ltd `ccris@doc.ic.ac.uk`

Abstract. Modular and local reasoning about object-oriented programs has been widely studied for programing languages such as C# and Java. Once source programs have been proven, the next verification challenge is to ensure that the code produced by the compiler is correct. Since verifying a compiler can be extremely complex, this paper uses *proof-transforming compilation*, an alternative approach which automatically generates *certificates*, a bytecode proof, from proofs in the source language. The paper develops a bytecode logic using separation logic, and proof translation from proofs of object-oriented programs to bytecode. The translation also handles proofs for concurrent programs. The bytecode logic and the proof transformation are proven sound.
keywords: software verification, program proofs, separation logic, proof-carrying code

1 Introduction

Object-oriented programming has been increasingly attractive in the last decades, however, it has also introduced new verification challenges. Solutions have been proposed, for example, separation logic [20] has extended Hoare logics to reason about programs with mutable data structures; ownership [7] has introduced a technique to reason about the heap structure.

Once the object-oriented programs have been proven correct with respect to their specifications, the verification process should ensure that the code produced by the compiler is correct. Since verifying the compiler is complex [11], techniques such as *translation validation* [22] have been proposed. In *translation validation*, instead of proving that the compiler always generates a correct target code, each translation is validated showing that the target code correctly implements the source program. The *translation validation* approach compares the input and the output, using an analyzer, independently of how the compiler is implemented. Together with a source proof, this gives an indirect correctness proof for the bytecode program.

Expanding the ideas of Proof-Carrying Code [13], Barthe et al [4][3] and Nordio et al. [18] have proposed an alternative verification process based on *proof-transforming compilation (PTC)*. The PTC approach consists of translating proofs of object-oriented programs to bytecode proofs. The verification process is performed at the level of the source program taking advantage of already developed verification techniques. Then, a *proof-transforming compiler* translates automatically a program and its proof into

[3] Barthe et al. called this approach *preservation of proof obligations*.

bytecode representing both the program and the proof. The main advantage of PTC is that it addresses full functional correctness as expressed by the original specifications.

Previous work on proof-transforming compilation [1,3,12] has developed the basics of the technique, using either Hoare-style logics or verification condition generators. The main limitation of these works lies on the properties that can be proven in the source program. Those logics cannot prove programs with mutable data structures, for example the programs presented by Distefano et al. [8], which include a visitor pattern example. This restriction is produced by the techniques used to verify the source program.

This paper presents a bytecode logic using separation logic, and proof transformation from Java to bytecode. The translation takes a proof of object-oriented programs written using Parkinson and Bierman's logic [21], and produces a bytecode proof in separation logic style. The bytecode logic introduces dynamic and static specifications for bytecode methods, and framing for bytecode instructions. The use of separation logic allows us to handle proofs that previous works [1,3,12] could not. The definition of the bytecode logic using separation logic makes the translation feasible. In this paper, we also extend the proof transformation to handle proofs for concurrent programs.

2 Overview of Separation Logic

Separation logic [20] provides an elegant approach to reasoning about programs with mutable data structures. It extends Hoare logic with spatial connectives which allow assertions to define separation between parts of the heap. In this paper, we use Parkinson and Bierman's logic [21], which we briefly describe next.

2.1 The Core Language

The programming language used in this paper is a common subset of C# and Java extended with static and dynamic specifications. The syntax is:

$$
\begin{array}{lll}
L & ::= \mathbf{class}\ C\ [\mathbf{extends}\ C_1]\ \{\ \mathbf{public}\ \overline{D}\,\overline{f}\ ;\overline{A}\,\overline{M}\ \} & \textit{Class Definition} \\
A & ::= \mathbf{define}\ \alpha_C(\overline{x})\ \mathbf{as}\ P & \textit{Abstract Predicate Family} \\
M & ::= \mathbf{public\ virtual}\ C\ m(\overline{D}\,\overline{p})\ \mathit{DSspec}\ \overline{D}\,\overline{x};\ s; & \textit{Method Definition} \\
 & \ \mid\ \mathbf{public\ override}\ C\ m(\overline{D}\,\overline{p})\ \mathit{DSspec}\ \overline{D}\,\overline{x};\ s; & \\
\mathit{DSspec} & ::= \mathbf{dynamic}\ \mathit{Spec};\ \mathbf{static}\ \mathit{Spec} & \textit{Dynamic and Static Spec.} \\
\mathit{Spec} & ::= \{P\}\_\{Q\}\ \mid\ \mathit{Spec}\ \mathbf{also}\ \{P\}\_\{Q\} & \textit{Specification Combination} \\
s & ::= x = e\ \mid\ s;s\ \mid\ x = y.f\ \mid\ x.f = e & \textit{Statements} \\
 & \ \mid\ x = y.m(\overline{e})\ \mid\ x = y.C :: m(\overline{e})\ \mid\ x = \mathtt{new}\ C() &
\end{array}
$$

Programs are defined as a set of classes, where each class consists of a collection of methods and field definitions; a class can specify at most one superclass. The class definition also contains abstract predicates families (APF). A method declaration includes the method name, parameters with type and name, method specifications, as well as a method body. Method specifications include a static specification and a dynamic specification. Static specifications are used to verify the implementation of a method and direct method calls (in Java this would be with a **super** call); dynamic specifications are used for calls that are dynamically dispatched. The specifications consist of a sequence of pre- and postconditions separated by the keyword **also**: $\{P_1\}\_\{Q_1\}$ **also** $\{P_2\}\_\{Q_2\}$ is defined as $\{P_1 \wedge P_2\}\_\{Q_1 \wedge Q_2\}$. The return statement is not supported in the source language; the return value is assigned to a local variable result. The notation we use is the following: f ranges over field names, m ranges over method names, $\bar{x}$ over sequences of variables, $\bar{p}$ for sequences of method call parameters, C, C_1, D over class names; $\bar{e}$ denotes a sequence of expressions.

An *abstract predicate* is defined with a name, a definition, and a scope. The abstract predicate's name and its definition can be swapped within the scope; outside the scope, the abstract predicate is handled atomically, i.e. by its name. For example, in a class *Cell*, we define the abstract predicate $Val_{Cell}(x, y)$ as $x.val \mapsto y$. The scope of the predicate is inside of the class *Cell*; in the implementation of *Cell*, the predicate $Val_{Cell}(x, y)$ and its definition can be swapped; outside the class, the predicate is handled by its name.

To accommodate inheritance, Parkinson and Bierman [21] introduce *abstract predicates families*. Each class can define its own entry predicate for an APF; this definition allows weakening preconditions, and strengthening postconditions for method overriding. The relationship between the family and entry is given by $x : C \Rightarrow (\alpha(x, \bar{x}) \Leftrightarrow \alpha_c(x, \bar{x}))$ where α is an abstract predicate, and α_C is the definition of the predicate for the class C.

2.2 Separation Logic for the Source Language

Memory Model and Assertion Language. Program states are mappings from local variables and parameters to values, and from locations to values: $State \equiv Store \times Heap$, where $Store \equiv Var \rightharpoonup Value$, and $Heap \equiv Location \rightharpoonup Value$. The formulae of assertion language are given by the following grammar:

$$\begin{array}{lll} P, Q & := & true \mid false \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x.P \mid \exists x.P \mid P * Q \mid e = e \mid x.f \mapsto e' \mid \alpha(\bar{e}) \mid \alpha_c(\bar{e}) \\ e & := & x \mid \texttt{null} \mid e \; \texttt{op} \; e \end{array}$$

The semantics of formulae is defined as follows:

$$\sigma, h \models P * Q \stackrel{def}{=} \exists h_0, h_1.h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and } \sigma, h_0 \models P \text{ and } \sigma, h_1 \models Q$$

$$\sigma, h \models e = e' \stackrel{def}{=} \sigma(e) = \sigma(e') \qquad \sigma, h \models \alpha(\bar{x}) \stackrel{def}{=} h \in (\Lambda(\alpha)(\sigma(\bar{x})))$$

$$\sigma, h \models x.f \mapsto e' \stackrel{def}{=} h(\sigma(x)).f = \sigma(e')$$

For $\sigma \in Store$, $\sigma(e)$ denotes the evaluation of the expression e in the store σ. For $h \in Heap$, $h(e).f$ denotes the evaluation of the field f of the expression e. The connectives $(\wedge, \vee)$ and quantifiers $(\exists, \forall)$ are interpreted in the usual way, and omitted here. The formula $P * Q$ allows us to assert that two portions of the heap are disjoint in which P and Q hold respectively. The interpretation of abstract predicates is given by the function Λ, which maps predicate names to predicate definitions.

Method and Statement Specifications. Properties of methods are written as $\Delta; \Gamma \vdash C.m(\bar{x})$ **dynamic** $\{P_C\}\_\{Q_C\}$ **static** $\{R_C\}\_\{S_C\}$ where Δ is the environment containing the logical assumptions about APFs that are available in the scope of the method m, and Γ is the environment containing the dynamic and static method specifications. This specification informally means that the method m in class C can be verified to meet its specification. In particular, Γ is used to handle recursion; Γ is initialized at the beginning of the proof with all the static and dynamic specifications.

The environments are given by the following grammar

$$\Gamma ::= \epsilon \ \mid \ \{P\}C.m(\overline{p})\{Q\}, \Gamma \ \mid \ \{P\}C::m(\overline{p})\{Q\}, \Gamma$$
$$\Delta ::= \epsilon \ \mid \ \alpha_C \stackrel{def}{=} \lambda(x;\overline{x})P, \Delta$$

where dynamic specifications are denoted by $\{P\}\ C.m(\overline{p})\ \{Q\}$; static specification are denoted by $\{P\}\ C::m(\overline{p})\ \{Q\}$.

Properties of statements are expressed by Hoare triples of the form $\Delta; \Gamma \vdash \ \{P\}\ s\ \{Q\}$. This triple defines the following refined partial correctness property [16]: if s's execution starts in a state satisfying P, then (1) s terminates normally in a state where Q holds, or (2) s aborts due to errors or actions than are beyond the semantics of the programming language, e.g., memory problem, or (3) s runs forever.

2.3 Proof Rules

The proof rules, taken from Parkinson and Bierman's work [21], for a subset of the source language is defined as follows:

Field Write
$$\frac{}{\Delta; \Gamma \vdash \{x.f \mapsto \_\}\ x.f := e\ \{x.f \mapsto e\}}$$

Dynamic Dispatch
$$\frac{C.m(\overline{p}):\ \ \{P\}\ \_\ \{Q\} \in \Gamma}{\Delta; \Gamma \vdash \{P[x, \overline{e}/\mathtt{this}, \overline{p}] \wedge \mathtt{this} \neq \mathtt{null}\}\ z = x.m(\overline{e})\ \{Q[z, x, \overline{e}/\mathsf{result}, \mathtt{this}, \overline{p}]\}}$$
where x has a static type C.

Direct Method Call
$$\frac{C::m(\overline{p}):\ \{R\}\ \_\ \{S\} \in \Gamma}{\Delta; \Gamma \vdash \{R[x, \overline{e}/\mathtt{this}, \overline{p}] \wedge \mathtt{this} \neq \mathtt{null}\}\ z = x.C::m(\overline{e})\ \{S[z, x, \overline{e}/\mathsf{result}, \mathtt{this}, \overline{p}]\}}$$

Method
$$\frac{\Delta; \Gamma \vdash \{R_C\}\ body\ \{S_C\}\ \ \text{(Body verification)} \qquad \Delta \vdash \{R_C\}\_\{S_C\}\ \Rightarrow\ \{P_C * this\!:\!C\}\_\{Q_C\}\ \ \text{(Dynamic dispatch)}}{\Delta; \Gamma \vdash \textbf{public virtual}\ C.m(\overline{x})\ \textbf{dynamic}\ \{P_C\}\_\{Q_C\}\ \textbf{static}\ \{R_C\}\_\{S_C\}\ \ body}$$

The rule for field write is standard. The rule for direct method call uses the static specification; $C::m(\overline{p}):\ \{R\}\ \_\ \{S\} \in \Gamma$ denotes that Γ contains the static specification $\{R\}\_\{S\}$, which is associated with the method m in class C. The rule for dynamic dispatch is similar to the direct method call but uses the dynamic specification; $C.m(\overline{p})$: $\{P\}\ \_\ \{Q\} \in \Gamma$ denotes that Γ contains the dynamic specification $\{P\}\_\{Q\}$ that is associated with the method m. The connection between the method body proofs and the method specifications is formalized with the *Method* rule. This rule has two proof obligations showing that (1) the method body satisfies its static specification; and (2) the use of the dynamic specification is valid for dynamic dispatch. The implication $\Delta \vdash \{R_C\}\_\{S_C\}\ \Rightarrow\ \{P_C * this\!:\!C\}\_\{Q_C\}$ means that the static precondition R_C implies the dynamic precondition $P_C * this\!:\!C$, and the dynamic postcondition Q_C implies the static postcondition S_C. Note that to handle recursion, the logic does not add any dynamic and static specifications to the environment Γ; Γ is initialized at the beginning with all these specifications. The logic also has a rule for overridden methods, which is similar to the *Method* rule and adds a proof obligation that shows the new dynamic specification is a valid behavioral subtype. This rule is omitted here.

To prove a class, the following *Class* rule is used:

$$\frac{\textit{for all } M_i \textit{ in } \overline{M} : \Delta; \Gamma \vdash M_i}{\Delta; \Gamma \vdash \textbf{class } C : D \; \{\textbf{public } \; \overline{T}\,\overline{f}; \overline{M}\; \}}$$

To be able to fold and unfold the definition of an abstract predicate, the logic has two axioms. These axioms allows folding and unfolding if and only if the abstract predicate is in scope. The axioms are:

$$\textit{Open: } (\alpha(\bar{x}) \overset{def}{=} P), \Lambda \models \alpha(\bar{e}) \Rightarrow P[\bar{e}/\bar{x}] \quad \textit{Close: } (\alpha(\bar{x}) \overset{def}{=} P), \Lambda \models P[\bar{e}/\bar{x}] \Rightarrow \alpha(\bar{e})$$

One of the most important rules for separation logic is the *Frame* rule. This rule is defined as follows:

$$\frac{\Delta; \Gamma \vdash \{P\} \;\; \bar{s} \;\; \{Q\}}{\Delta; \Gamma \vdash \{P * R\} \;\; \bar{s} \;\; \{Q * R\}} \quad \text{where } Mod(\bar{s}) \cap FV(R) = \emptyset$$

The expression $Mod(\bar{s}) \cap FV(R) = \emptyset$ expresses that $\bar{s}$ does not modify the free variables of R. The logic also has rules for weakening and elimination of abstract predicates. Space prevents us from presenting these rules, for a complete description of the logic see [21].

2.4 Example

Figure 1a shows an example from Parkinson and Bierman [21], which illustrates the use of static and dynamic specifications, and abstract predicates. The class *Cell* implements a single cell with an integer value; the class *Recell* extends the implementation of *Cell* storing the previous value of the cell. Each method has two specifications: a *dynamic specification*, that is used for dynamic method calls, and a *static specification*, that is used to verify the implementation and direct method calls. To define the dynamic specification of the method *set*, the abstract predicate family $Val(x, y)$ is used; the definition of this predicate for the class *Cell* is $Val_{Cell}(x, y) \overset{def}{=} x.val \mapsto y$. This predicate expresses that the field *val* of the object x points to the object y. In the class *Recell*, the method *set* is overridden. Its specification is extended, and the predicate *Val* takes an extra argument. The definition is $Val_{Recell}(x, y, z) \overset{def}{=} Val_{Cell}(x, y) * x.bak \mapsto z$. In this definition, the operator $*$ is used to express non-interference.

The proof of the source example consists of a proof for the classes *Cell* and *Recell*. The proof of the class *Recell* consist of the proof of the method *set*; these proofs are constructed applying the *Class* rule and the *Method* rule respectively. A sketch of the proof of the method *set* is presented in Figure 1b. It applies the rules *Direct Method Call* as well as the *Open and Close axioms*.

3 A Separation Logic for Bytecode

3.1 The Bytecode Language

The bytecode language consists of classes with methods and fields. Methods are implemented as method bodies consisting of a sequence of labeled bytecode instructions.

```
class Cell {
  public int val;
  public virtual void set(int x)
    dynamic {Val(this,_)} _ {Val(this,x)}
    static {this.val ↦ _}_{this.val ↦ x)}
    {  val = x;  }
  public virtual int get()
  dynamic
    {Val(this,v)} _ {Val(this,v) * result=v }
  static
    {this.val ↦ _}_{this.val ↦ x) * result=v}
    { ret := val;  } }
```

```
class Recell extends Cell {
  public int bak;
  public override void set(int x)
  dynamic
    {Val(this,v,_)}_{Val(this,x,v)}
  static
    {this.val ↦ v}_{this.val ↦ x * this.bak ↦ v}
    { bak = super.get(); super.set(x); } }
```

(a) Cell Example

$\{\ Val_{Recell}(this, v, \_)\ \}$ [*Open Axiom*]
$\{\ Val_{Cell}(this, v) * this.bak \mapsto \_\ \}$
this .*bak* = **super**.*get* () ;
$\{\ Val_{Cell}(this, v) * this.bak \mapsto v\ \}$ [*Direct Method Call*]
super. *set* (*x*) ;
$\{\ Val_{Cell}(this, x) * this.bak \mapsto v\ \}$ [*Direct Method Call*]
$\{\ Val_{Recell}(this, x, v)\ \}$ [*Close Axiom*]

(b) Sketch of the Proof for the Method *set*

Fig. 1. Example using Static and Dynamic Specifications.

Bytecode instructions operate on the operand stack, local variables (which also include parameters), and the heap. Each method body ends with a return instruction, which return the control flow to the caller; a method returns the value stored in a special special local variable result. This language is extended with dynamic and static specifications. We also introduced abstract predicates families to the bytecode language. This extension to the bytecode language makes the translation feasible. The syntax is:

$$L, A, M, DSspec, Spec ::= \textit{as defined in Section } 2.1$$
$$s ::= \overline{l:\ Inst}$$
$$Inst ::= \mathsf{pop}\ x \mid \mathsf{push}\ v \mid \mathsf{goto}\ l' \mid \mathsf{nop} \mid \mathsf{return} \mid \mathsf{brtrue}\ l' \mid \mathsf{putfld}\ f \mid \mathsf{newobj}\ C \mid \mathsf{invokespecial}\ C{::}m$$

This language is similar to Java bytecode. We treat local variables and method parameters using the same instructions. Instead of using an array of local variables like in Java Bytecode, we use the name of the source variable. To simplify the proof translation, we assume the bytecode language has a boolean type.

The semantics of the instructions is as follows: the instruction pop *x* removes the top element of the stack and assigns it to *x*; push *v* puts the value *v* on top of the stack; goto transfers control the program point l'; nop has no effect; return returns to caller; brtrue transfers control to the label l' if the top of the stack is *true* removing this value from the stack; the instruction putfld *f* updates the field *f*; newobj creates an object of type *C*. The instruction invokespecial is used to call private methods and super methods.

3.2 Memory Model

Bytecode program states are a triple consisting of an operand stack, a local store, and a heap: $State \equiv Stack \times Store \times Heap$, where $Stack \equiv Value^*$, $Store \equiv Var \rightharpoonup Value$, and $Heap \equiv Location \rightharpoonup Value$. The *Stack* type is defined as a list of values; *Store* is a mapping from local variables and parameters to values; *Heap* is a mapping from locations to values. In the following section, we present the axiomatic semantics; the operational semantics and the soundness proof are presented in our technical report [15].

3.3 Axiomatic Semantics

Assertion Language. Formulae for the assertion language of bytecode method specifications are the same as for the source language (described in Section 2.2). The formulae for the assertion language for preconditions of bytecode instructions are extended because the precondition can refer to the stack. Formulae are defined as $S \bullet P$ where S is a stack of values, and P is a formula defined as in the source language. The definition is $BytecodePre := S \bullet P$ where $S := e^*$, and P and e are defined as in Section 2.2. The formal semantics of formulae is defined as follows:

$$\begin{array}{rl} s, \sigma, h \models S \bullet P & \stackrel{def}{=} s, \sigma \models S \; and \; \sigma, h \models P \\ (v_1, ..., v_n), \sigma \models (e_1, ... e_m) & \stackrel{def}{=} n = m \; and \; \sigma(e_i) = v_i \\ \sigma, h \models P & \stackrel{def}{=} \text{as defined in Section 2.2} \end{array}$$

Following, we define the implication operator for bytecode preconditions:

Definition 1. *Given the stacks S_1 and S_2 and the expressions P and Q, then* $s, \sigma, h \models S_1 \bullet P \Rightarrow S_2 \bullet Q$ *iff* $s, \sigma, h \models S_1 \bullet P$ *implies* $s, \sigma, h \models S_2 \bullet Q$*. We write* $S_1 \bullet P \Rightarrow S_2 \bullet Q$ *to mean validity:* $\forall s, \sigma, h : \; s, \sigma, h \models S_1 \bullet P \Rightarrow S_2 \bullet Q$.

Proof Rules for Classes. A bytecode proof consists of a list of proofs for the bytecode classes. To prove the bytecode classes, the logic has the same *Class* rule and *Frame* rule as in the source language.

Proof Rules for Method Specifications. Properties of bytecode methods are defined as $\Delta; \Gamma \vdash C.m(\bar{x})$ **dynamic** $\{P_C\}\_\{Q_C\}$ **static** $\{R_C\}\_\{S_C\}$. This definition is the same as in the source language. In particular the treatment of recursion is the same as in the source logic: the environment Γ contains the static and dynamic specifications, and it is initialized at the beginning of the proof.

The logic has a similar *Method* rule and *Override* rule to the logic for the source language. The bytecode *Method* rule is defined as follows:

$$\frac{\begin{array}{ll} & \Delta \vdash \{R_C\}\_\{S_C\} \;\Rightarrow\; \{P_C * this : C\}\_\{Q_C\} \;\text{(Dynamic dispatch)} \\ R_C \Rightarrow E_1 & E_j \Rightarrow S_C \quad body = \{E_1\}\, 1 : I_1, \; ... \; \{E_j\}\, j : \mathsf{return} \quad \Psi = (l_1, E_1) \, ... \, (l_j, E_j) \\ & \forall i \in 1, ... j : \Delta; \Gamma; \Psi \vdash \{E_i\} \;\, i : I_i \;\text{(Bytecode body verification)} \end{array}}{\Delta; \Gamma \vdash \textbf{public}\; C.m(\bar{x}) \;\; \textbf{dynamic}\; \{P_C\}\_\{Q_C\}\; \textbf{static}\; \{R_C\}\_\{S_C\} \;\; body}$$

This rule, besides showing that the use of dynamic dispatch is valid, has three extra proof obligations: we need to verify that (1) the precondition of the method implies the precondition of the first bytecode instruction (E_1); (2) the postcondition of the last bytecode instruction (E_j) implies the method postcondition, and (3) all the instruction specifications of the method m hold. Note that the body of the method m, denoted by *body*, is a list of bytecode specifications of the form $\Delta; \Gamma; \Psi \vdash \{E_i\} \;\, i : I_i$.

Proof Rules for Instruction Specifications. The bytecode logic treats instructions individually since control can be transferred into the middle of a sequence. Each instruction at the label l has a precondition E_l. Bytecode specifications have the form $\Delta;\Gamma;\Psi \vdash \{E_l\}\ l : inst$ where Δ is the environment containing the APF, Γ is the environment containing the dynamic and static method specifications (as in the source logic), and Ψ is a mapping from labels to preconditions. We use the environment Ψ to make explicit the list of successor preconditions. This environment is used, in particular for the application of the *Frame* rule.

The semantics of $\Delta;\Gamma;\Psi \vdash \{E_l\}\ l\!:\!inst$ is: if the precondition E_l holds when the program counter is at the label l, then the preconditions of the successor instructions hold after successful execution of instruction *inst*.

Following, we present the rules for pop, push, and brtrue. For the complete definition see our technical report [15]:

$$\frac{S \bullet \exists x'.x = v[x'/x] \wedge P[x'/x] \ \Rightarrow\ E_{l+1}}{\Delta;\Gamma;\Psi,(l{+}1,E_{l+1}) \vdash \{(S,v)\bullet P\}\ l\!:\!\mathsf{pop}\ x} \qquad \frac{(S,v)\bullet P \ \Rightarrow\ E_{l+1}}{\Delta;\Gamma;\Psi,(l{+}1,E_{l+1}) \vdash \{S\bullet P\}\ l\!:\!\mathsf{push}\ v}$$

$$\frac{S \bullet P\ \wedge\ v = true \Rightarrow E_{l'} \qquad S \bullet P\ \wedge\ v = false \Rightarrow E_{l+1}}{\Delta;\Gamma;\Psi,(l',E'_l),(l+1,E_{l+1}) \vdash \{(S,v)\bullet P\}\ \ l : \mathsf{brtrue}\ l'}$$

In the rule of the instruction pop, the precondition assumes that the operand stack is not empty. The implication $S \bullet \exists x'.x = v[x'/x] \wedge P[x'/x] \ \Rightarrow\ E_{l+1}$ expresses that one has to show that the formula $S \bullet \exists x'.x = v[x'/x] \wedge P[x'/x]$ implies the precondition of the next instruction. In this formula, the operand stack is S since the value v has been popped and assigned to x. The replacements are similar to the assignment rule in the source language. The environment $\Psi,(l+1,E_{l+1})$ expresses that the precondition of the instruction at label $l+1$ is E_{l+1}. The rule for push adds a value v on top of the stack S, then one has to show that $(S,v)\bullet P$ implies the next instruction's precondition.

Below, we present the rule for invokespecial (the rule for invokevirtual is similar). Similar to the source logic, this rule uses the static specifications.

$$\frac{C\!::\!m(\overline{p}) : \{T\}\_\{R\} \in \Gamma \qquad (S,v)\bullet R[y/\mathtt{this},\overline{z}/\overline{p},v/result] \Rightarrow E_{l+1}}{\Delta;\Gamma;\Psi,(l+1,E_{l+1}) \vdash \{(S,y,\overline{z})\bullet T[y/\mathtt{this},\overline{z}/\overline{p}] \wedge y \neq \mathtt{null}\}\ l\!:\!\mathsf{invokespecial}\ C\!:\!m}$$

where v is a logical variable.

Frame Rule for Bytecode Instructions. The *Frame* rule of the logic of the source language can be applied to both method specifications and instructions. For example, the *Frame* rule could be applied to a triple where the instruction is an assignment. In our bytecode logic, we have developed a *Frame* rule for bytecode specifications. This rule is needed to translate the *Frame* rule from the source language. The rule is defined as follows:

$$\frac{\Delta;\Gamma;\Psi \vdash \{S\bullet P\}\,l : inst \quad \Psi' = Succ(l,\Psi) \quad \Psi = \Psi',\Psi''}{\Delta;\Gamma;(\Psi' * R),\Psi'' \vdash \{S \bullet P * R\}\ \ l : inst} \quad \text{where } Mod(inst) \cap FV(R) = \emptyset$$

Bytecode specifications can have several successors. For example, the bytecode branching instruction brtrue l has two successors: the next instruction and the instruction at label l. The standard *Frame* rule (in the source logic) strengthens both the precondition and the postcondition of the triple. Since bytecode specifications can have several successors, we need to strengthen all successor preconditions. The successor instructions are contained in the environment Ψ'. It is constructed using the function *Succ*, which yields the environment with the label l and its precondition, and l's successors. The environment $\Psi' * R$ is obtained from the successor instructions of l in Ψ', by adding $*R$ to each precondition. These separating conjunctions are only added to the preconditions of l and the successor instructions, so the environment Ψ'' is not modified.

Language-Independent Rules. The bytecode logic also has language-independent rules such as stack-disjointness. In this section, we present the most important language-independent rules; for a full description see our technical report [15]. The following rule is used in the proof translation to embed a local proof transformation in a wider context, for example to combine the results of applying the *Frame* rule to single instructions.

$$\textit{Env-weakening}\quad \frac{\Delta;\Gamma;\Psi \vdash \{P\}\quad l : \textit{inst}}{\Delta;\Gamma;\Psi,\Psi' \vdash \{P\}\quad l : \textit{inst}}$$

Another language-independent rule is the *stack-disjointness* rule, which allows reasoning about stacks that might have different values and sizes. For example, this rule allows reasoning about a program that might push either a value v_1 or a value v_2 into the stack, and therefore, the top of the stack is either v_1 or v_2. The rule is defined as:

$$\textit{stack-disjointness}\quad \frac{\Delta;\Gamma;\Psi \vdash \{(S,v_1)\bigvee(S,v2)\bullet P\}\quad l : \textit{inst}}{\Delta;\Gamma;\Psi \vdash \{(S,(v_1 \bigvee v_2))\bullet P\}\quad l : \textit{inst}}$$

The semantics of the formulae, denoted as $\models$, is extended to support stack disjointness: $S_1 \bigvee S_2 \bullet P$, and expression disjointness: $x = (v_1 \bigvee v_2)$[4]. The semantics is:

$$\begin{array}{lll}
s,\sigma & \models S_1 \bigvee S_2 & \stackrel{def}{=} (s,\sigma \models S_1 \; or \; s,\sigma \models S_2) \\
(s,e),\sigma & \models (S_1,(v1 \bigvee v_2)) & \stackrel{def}{=} (s,\sigma \models S_1 \; and \; (e = \sigma(v_1) \; or \; e = \sigma(v_2)\;) \\
s,h & \models x = (v_1 \bigvee v_2) & \stackrel{def}{=} s,\sigma \models (x = v_1) \vee (x = v_2)
\end{array}$$

3.4 Examples

This subsection presents two examples illustrating the application of the frame rule and disjointness rule for bytecode.

Example Applying the Frame Rule. Assume the following valid bytecode proof:

$$\begin{array}{ll}
\Delta;\Gamma;(l_2,S_2\bullet P_2) & \vdash \{S_1\bullet P_1\}\; l_1 : \; \textsf{push}\; x \\
\Delta;\Gamma;(l_3,S_3\bullet P_3),(l_5,S_5\bullet P_5) & \vdash \{S_2\bullet P_2\}\; l_2 : \; \textsf{brtrue}\; l_5 \\
\Delta;\Gamma;(l_4,S_4\bullet P_4) & \vdash \{S_3\bullet P_3\}\; l_3 : \; \textsf{push}\; y \\
\Delta;\Gamma;(l_5,S_5\bullet P_5) & \vdash \{S_4\bullet P_4\}\; l_4 : \; \textsf{goto}\; l_6 \\
\Delta;\Gamma;(l_6,S_6\bullet P_6) & \vdash \{S_5\bullet P_5\}\; l_5 : \; \textsf{push}\; z \\
\Delta;\Gamma;\epsilon & \vdash \{S_6\bullet P_6\}\; l_6 : \; \textsf{return}
\end{array}$$

[4] The expression disjointness is used when the value $v_1 \bigvee v_2$ is popped from the stack and assigned to a variable x

where P_i is the precondition at label l_i. The application of the *Frame* rule to the instructions at labels $l_1 ... l_6$ adds $*R$ to each precondition. Given that each instruction specification contains a list of the successors, the rule also adds $*R$ to each precondition in the environment Ψ. After applying the *Frame* rule, we obtain the following proof:

$$\begin{array}{ll}
\Delta; \Gamma; (l_2, S_2 \bullet P_2 * R) & \vdash \{S_1 \bullet P_1 * R\}\ l_1\!: \mathsf{push}\ x \\
\Delta; \Gamma; (l_3, S_3 \bullet P_3 * R), (l_5, S_5 \bullet P_5 * R) & \vdash \{S_2 \bullet P_2 * R\}\ l_2\!: \mathsf{brtrue}\ l_5 \\
\Delta; \Gamma; (l_4, S_4 \bullet P_4 * R) & \vdash \{S_3 \bullet P_3 * R\}\ l_3\!: \mathsf{push}\ y \\
\Delta; \Gamma; (l_6, S_6 \bullet P_6 * R) & \vdash \{S_4 \bullet P_4 * R\}\ l_4\!: \mathsf{goto}\ l_6 \\
\Delta; \Gamma; (l_6, S_6 \bullet P_6 * R) & \vdash \{S_5 \bullet P_5 * R\}\ l_5\!: \mathsf{push}\ z \\
\Delta; \Gamma; \epsilon & \vdash \{S_6 \bullet P_6 * R\}\ l_6\!: \mathsf{return}
\end{array}$$

Note that the instruction l_2 has two successors: l_3 and l_5. Thus, the application of the *frame* rule changes the environment $(l_3, P_3), (l_5, P_5)$ to $(l_3, P_3 * R), (l_5, P_5 * R)$. Applying the *Env-weakening* rule, we obtain the following proof:

$$\begin{array}{l}
\Delta; \Gamma; \Psi \vdash \{S_1 \bullet P_1 * R\}\ l_1 : \ \mathsf{push}\ x \\
\Delta; \Gamma; \Psi \vdash \{S_2 \bullet P_2 * R\}\ l_2 : \ \mathsf{brtrue}\ l_5 \\
\Delta; \Gamma; \Psi \vdash \{S_3 \bullet P_3 * R\}\ l_3 : \ \mathsf{push}\ y \\
\Delta; \Gamma; \Psi \vdash \{S_4 \bullet P_4 * R\}\ l_4 : \ \mathsf{goto}\ l_6 \\
\Delta; \Gamma; \Psi \vdash \{S_5 \bullet P_5 * R\}\ l_5 : \ \mathsf{push}\ z \\
\Delta; \Gamma; \Psi \vdash \{S_6 \bullet P_6 * R\}\ l_6 : \ \mathsf{return} \\
\mathit{where}\ \Psi \stackrel{def}{=} (l_1, P_1 * R) \ldots (l_6, P_6 * R)
\end{array}$$

Example Applying the Disjointness Rule. Assume we want to prove the following program:

$$\begin{array}{l}
l_1 : \ \mathsf{push}\ b \\
l_2 : \ \mathsf{brtrue}\ l_5 \\
l_3 : \ \mathsf{push}\ 0 \\
l_4 : \ \mathsf{goto}\ l_6 \\
l_5 : \ \mathsf{push}\ 1 \\
l_6 : \ \mathsf{pop}\ x \\
l_7 : \ \mathsf{nop}
\end{array}$$

where at the instruction l_7 the expression $x = 0 \ \vee x = 1$ holds. To simplify the proof, the omit the details of the environments $\Delta; \Gamma; \Psi$ and we write $\Delta; \Gamma; \Psi$ without defining the successor instructions in Ψ. The preconditions for these instructions are as follows (assuming the stack is S before the execution of this code):

$$\begin{array}{ll}
\Delta; \Gamma; \Psi \vdash \{S \bullet \mathit{True}\} & l_1 : \ \mathsf{push}\ b \\
\Delta; \Gamma; \Psi \vdash \{(S, b) \bullet \mathit{True}\} & l_2 : \ \mathsf{brtrue}\ l_5 \\
\Delta; \Gamma; \Psi \vdash \{S \bullet \mathit{True}\} & l_3 : \ \mathsf{push}\ 0 \\
\Delta; \Gamma; \Psi \vdash \{(S, 0) \bullet \mathit{True}\} & l_4 : \ \mathsf{goto}\ l_6 \\
\Delta; \Gamma; \Psi \vdash \{S \bullet \mathit{True}\} & l_5 : \ \mathsf{push}\ 1 \\
\Delta; \Gamma; \Psi \vdash \{(S, (0 \bigvee 1)) \bullet \mathit{True}\} & l_6 : \ \mathsf{pop}\ x \\
\Delta; \Gamma; \Psi \vdash \{S \bullet x = 0 \vee x = 1\} & l_7 : \ \mathsf{nop}
\end{array}$$

The preconditions at labels l_1 to l_5 hold by applying the push, brtrue, push, and goto rules. The interesting part of the proof is at labels l_6 and l_7. Applying the *stack disjointness* rule we can prove:

$$\textit{stack-disjointness}\quad \frac{\Delta;\Gamma;\Psi \vdash \{(S,0)\bigvee(S,1)\bullet \mathit{True}\}\quad l : \mathit{inst}}{\Delta;\Gamma;\Psi \vdash \{(S,(0\bigvee 1))\bullet \mathit{True}\}\quad l : \mathit{inst}}$$

Now, we need to prove that the instructions at labels l_4 and l_5 satisfy the precondition $(S,0)\bigvee(S,1)\bullet \mathit{True}$. By definition of $(S,0)\bigvee(S,1)\bullet \mathit{True}$, the precondition $\{(S,0)\bullet \mathit{True}\}$ implies $(S,0)\bigvee(S,1)\bullet \mathit{True}$, and the precondition $\{(S,1)\bullet \mathit{True}\}$ implies $(S,0)\bigvee(S,1)\bullet$ *True*. Then, applying the goto and pop rules, the instructions at labels l_4 and l_5 hold.

To prove the instruction of line l_7, we apply the pop rule, obtaining:

$$\frac{S\bullet x = (0\bigvee 1)\wedge \mathit{True} \;\Rightarrow\; S\bullet x = 0 \vee x = 1}{\Delta;\Gamma;\Psi \vdash \{(S,(0\bigvee 1))\bullet \mathit{True}\} l_6 : \ \mathsf{pop}\ x}$$

The implication holds by definition of $x = 0\bigvee 1$ which is defined as $x = 0 \vee x = 1$. Therefore, the proof is a valid proof.

4 Proof Transformation for Separation Logic

The proof translation takes a proof in the source language (Section 2), and produces a proof in the bytecode logic (Section 3). The proof translation is developed using the translation functions ∇_C, ∇_M ∇_S, and ∇_E, which translate classes, methods, instructions, and expressions respectively. The signature of these functions are as follows:

$$\begin{array}{ll} \nabla_C : \mathit{ProofTree} \rightarrow \mathit{BytecodeProofTree} & \nabla_M : \mathit{ProofTree} \rightarrow \mathit{BytecodeProofTree} \\ \nabla_S : \mathit{ProofTree} \rightarrow \mathit{List}[\mathit{BytecodeSpec}] & \nabla_E : \mathit{Pre} \times \mathit{Exp} \times \mathit{Post} \rightarrow \mathit{List}[\mathit{BytecodeSpec}] \end{array}$$

A *ProofTree* is a derivation in the logic of the source language. A *BytecodeProofTree* is a derivation in the bytecode logic; the function ∇_S produces the proof for the body of a bytecode method; it consists of a list of bytecode specifications. The postcondition in the function ∇_E is used to prove soundness of the translation. In the following sections, we present the translation for method specifications, the *Frame* rule, and statements.

Proof Translation for Method Specifications. A source proof for a class C consists of a list of method names with a dynamic and static specification, and proofs for the method bodies. The source logic uses the *Class rule* to prove the method bodies. Since the source and the bytecode logic treat the heap in the same way, use the same abstract predicate definitions, and have the same method specifications, these environments are not modified by the translation. To translate classes, the translation applies the *Class* rule in the bytecode. The translation is defined as follows:

$$\nabla_C\left(\frac{\mathit{for\ all}\ M_i\ \mathit{in}\ \overline{M} : \Delta;\Gamma \vdash M_i}{\Delta;\Gamma \vdash \textbf{class}\ C{:}D\{\textbf{public}\ \overline{T}\,\overline{f};\overline{M}\}}\right) = \frac{\mathit{for\ all}\ M_i\ \mathit{in}\ \overline{M} : \nabla_M(\Delta;\Gamma \vdash M_i)}{\Delta;\Gamma \vdash \textbf{class}\ C{:}D\{\textbf{public}\ \overline{T}\,\overline{f};\overline{M}\ \}}$$

The function ∇_M maps proofs of methods in Java to proofs of methods in bytecode. Given that the signature of the methods in Java and bytecode are the same (both use dynamic and static specifications), the translation does not modify the signature of the methods. The resulting bytecode proof uses the *Method* rule in bytecode where the body of the method is produced by the translation ∇_S. The translation is defined as follows:

$$\nabla_M\left(\frac{\begin{array}{c}\Delta;\Gamma\vdash\{R_C\}\ body\ \{S_C\}\ \ \text{(Body verification)}\\ \Delta\vdash\{R_C\}\_\{S_C\}\ \Rightarrow\ \{P_C * this\!:\!C\}\_\{Q_C\}\ \ \text{(Dynamic dispatch)}\end{array}}{\begin{array}{l}\Delta;\Gamma\vdash\textbf{public virtual}\ C.m(\overline{x})\\ \qquad\textbf{dynamic}\ \{P_C\}\_\{Q_C\}\ \textbf{static}\ \{R_C\}\_\{S_C\}\ \ body\end{array}}\right)=$$

$$\frac{\begin{array}{c}\Delta\vdash\{R_C\}\_\{S_C\}\ \Rightarrow\ \{P_C * this\!:\!C\}\_\{Q_C\}\ \ \text{(Dynamic dispatch)}\\ R_C\Rightarrow E_1\qquad E_j\Rightarrow S_C\qquad body\_bytecode=\nabla_S(body)\ \ \text{(Bytecode body verification)}\end{array}}{\Delta;\Gamma\vdash\textbf{public}\ C.m(\overline{x})\ \ \textbf{dynamic}\ \{P_C\}\_\{Q_C\}\ \textbf{static}\ \{R_C\}\_\{S_C\}\ \ body\_bytecode}$$

Proof Translation of the Frame Rule. To translate the *Frame* rule applied to statements, first we apply the translation ∇_S to the triple $\Delta;\Gamma\vdash\{P\}\ s\ \{Q\}$ producing the bytecode derivations

$$\Delta;\Gamma;\Psi_1\vdash\{S_1\bullet P_1\}\ \ l_1 : i_1\ \ ...\ \ \Delta;\Gamma;\Psi_n\vdash\{S_n\bullet P_n\}\ \ l_n : i_n$$
where Ψ_k only contains the labels and preconditions relevant to instruction i_k

Then, we apply the frame rule for bytecode instructions (page 8) to add the predicate $*R$ to the conjunction to the precondition of each derivation, and to the environment Ψ_i. Finally, we use the *Env-weakening* rule to unify the environments resulting from the application of the *Frame* rule into a single environment for the whole block of instructions. The translation produces the following proof:

$$\Delta;\Gamma;\Psi\vdash\{S_1\bullet P_1 * R\}\ \ l_1 : i_1\ \ ...\ \ \Delta;\Gamma;\Psi\vdash\{S_n\bullet P_n * R\}\ \ l_n : i_n$$
where $\Psi\overset{def}{=}\Psi_1 * R,\Psi_2 * R,...,\Psi_k * R$

Proof Translation of Statements. In this section, we present the translation functions for compound and direct method call; for a complete definition see our technical report [15]. The translation of a compound is defined as:

$$\nabla_S\left(\frac{\{P\}s_1\{Q\}\quad\{Q\}s_2\{R\}}{\{P\}s_1;s_2\{R\}}\right)=\nabla_S(\{P\}s_1\{Q\})+\nabla_S(\{Q\}s_2\{R\})$$

The direct method call translation is as follows:

$$\nabla_S\left(\frac{C.m(\overline{p}):\ \ \{P\}\_\{Q\}\in\Gamma}{\Delta;\Gamma\vdash\{P'\}\ z=x.C\!::\!m(\overline{e})\ \ \{Q[z,x,\overline{e}/\text{result},\texttt{this},\overline{p}]\}}\right)=$$

$$\begin{array}{ll}\Delta;\Gamma;\Psi_1\vdash\{\epsilon\bullet P'\} & L_A:\ \text{push}\ x\\ & \nabla_E(\ x\bullet P'\ ,\ \overline{e},\ \ (x,\overline{e})\bullet P'\ )\\ \Delta;\Gamma;\Psi_2\vdash\{\ (x,\overline{e})\bullet P'\} & L_B:\ \text{invokespecial}\ C\!::\!m\\ \Delta;\Gamma;\Psi_3\vdash\{\ \text{result}\bullet Q[x,\overline{e}/\texttt{this},\overline{p}]\ \} & L_C:\ \text{pop}\ z\end{array}$$

where P' is defined as $P[x,\overline{e}/\texttt{this},\overline{p}]\wedge\texttt{this}\neq\texttt{null}$, and Ψ_1,Ψ_2,Ψ_3 only contain the labels relevant to the instructions at labels L_A, L_B, and L_C respectively.

5 Proof Transformation for Concurrent Programs

This section extends the PTC approach to handle concurrent programs. We first present the source logic, the bytecode logic and its proof transformation for disjoint concurrency. Then, we expand the approach to critical regions.

5.1 Basic Concurrency

In Java, concurrency is implemented using the *Thread* class. This class contains methods such as *start*: to execute a thread, and *join*: to wait for the termination of a thread. To handle critical regions, the instruction synchronized is used. To simplify the semantics, we assume an instruction $s_1 \;||\; s_2$ in the source language, which runs the instructions s_1 and s_2 concurrently. This instruction is equivalent to execute $s_1.start();s_2.start();s_1.join();s_2.join()$. For the bytecode language, we also assume the threads are first run and then joined; thus, we assume an instruction invokeStartJoin.

Concurrency for the Source Logic. In this paper, we use the axiomatic semantics of the instruction $s_1 \;||\; s_2$ defined by O'Hearn [19]. The rule, called the *Disjoint Concurrency* rule, is defined as follows:

$$\frac{\Delta;\Gamma \vdash \{P_1\}\; s_1\; \{Q_1\} \quad \Delta;\Gamma \vdash \{P_2\}\; s_2\; \{Q_2\}}{\Delta;\Gamma \vdash \{P_1 * P_2\}\; s_1 \;||\; s_2\; \{Q_1 * Q_2\}} \quad \text{where } s_1 \text{ does not modify any variables free in } P_2, s_2, Q_2\text{, and conversely.}$$

Concurrency for the Bytecode Logic. Let $C_1{:}run$ and $C_2{:}run$ be bytecode methods. The instruction invokeStartJoin $C_1{:}run$ $C_2{:}run$ executes the *run* methods concurrently and waits for the termination of both. To simplify the semantics, we assume these methods are procedures. The rule for invokeStartJoin extends the rule for invokespecial (Section 3.3) to concurrency.

Let P'_1, P'_2, Q'_1 and Q'_2 be: $P'_1 \stackrel{def}{=} P_1[y_1/\texttt{this}] \wedge y_1 \neq \texttt{null}$, $Q'_1 \stackrel{def}{=} Q_1[y_1/\texttt{this}]$, $P'_2 \stackrel{def}{=} P_2[y_2/\texttt{this}] \wedge y_2 \neq \texttt{null}$, and $Q'_2 \stackrel{def}{=} Q_2[y_2/\texttt{this}]$. The rule is defined as follows:

$$\frac{\begin{array}{c} C_1{::}\,run : \{P_1\}\_\{Q_1\} \in \Gamma \qquad C_2{::}\,run : \{P_2\}\_\{Q_2\} \in \Gamma \\ S \bullet Q'_1 * Q'_2 \Rightarrow E_{l+1} \end{array}}{\Delta;\Gamma;\Psi,(l+1,E_{l+1}) \vdash \{(S,y_1,y_2) \bullet P'_1 * P'_2\}\;\; l : \text{invokeStartJoin } C_1{:}run\; C_2{:}run}$$

where $C_1{:}run$ does not modify any variables free in P_2, $C_2{:}run$, Q_2, and conversely.

Proof Transformation. The proof translator takes a proof using the *Disjoint Concurrency* rule, and generates a bytecode proof. To translate it, we first extend the definition of the translation function ∇_C. This function applies the translation function ∇_M to all the methods M_i in a class C, and also uses a new function ∇_P. The function ∇_P produces classes C_1 and C_2 with a method *run* for each use of the instruction $s_1 \;||\; s_2$. The function ∇_C is defined as follows:

$$\nabla_C\left(\frac{\textit{forall } M_i : \Delta;\Gamma \vdash M_i}{\Delta;\Gamma \vdash \{P_1\}\; \textbf{class } C{:}D\; \{\textbf{public } \overline{M}\}}\right) = \frac{\textit{forall } M_i\; \nabla_M(\Delta;\Gamma \vdash M_i); \nabla_P(\Delta;\Gamma \vdash M_i)}{\Delta;\Gamma \vdash \{P_1\}\; \textbf{class } C{:}D\; \{\textbf{public } \overline{M}\}}$$

The function ∇_P generates method proofs only when the *Disjoint Concurrency* rule is used. For other rules, this function is applied recursively. The definition of ∇_P for the case of the *Disjoint Concurrency* rule is as follows:

$$\nabla_P\left(\frac{\Delta;\Gamma \vdash \{P_1\}\; s_1\; \{Q_1\} \quad \Delta;\Gamma \vdash \{P_2\}\; s_2\; \{Q_2\}}{\Delta;\Gamma \vdash \{P_1 * P_2\}\; s_1 \;||\; s_2\; \{Q_1 * Q_2\}}\right) =$$

$$\frac{b = \nabla_S(\Delta;\Gamma \vdash \{P_1\}\; s_1\; \{Q_1\}) \text{ (Bytecode body verification)}}{\Delta;\Gamma \vdash \textbf{public } C_1.run(\overline{p_1})\;\; \textbf{dynamic } \{P_1\}\_\{Q_1\}\; \textbf{static } \{P_1\}\_\{Q_1\}\;\; b}$$

$$\frac{b = \nabla_S(\Delta;\Gamma \vdash \{P_2\}\; s_2\; \{Q_2\}) \text{ (Bytecode body verification)}}{\Delta;\Gamma \vdash \textbf{public } C_2.run(\overline{p_2})\;\; \textbf{dynamic } \{P_2\}\_\{Q_2\}\; \textbf{static } \{P_2\}\_\{Q_2\}\;\; b}$$

The translation function ∇_S is extended to handle concurrency; the translation first creates two objects of type C_1 and C_2, and then applies the invokeStartJoin rule. The translation is:

$$\nabla_S\left(\frac{\Delta;\Gamma \vdash \{P_1\}\ s_1\ \{Q_1\} \quad \Delta;\Gamma \vdash \{P_2\}\ s_2\ \{Q_2\}}{\Delta;\Gamma \vdash \{P_1 * P_2\}\ s_1 \mid\mid s_2\ \{Q_1 * Q_2\}}\right) =$$

$$\Delta;\Gamma;\Psi_1 \vdash \{\epsilon \bullet P_1 * P_2\}\ \ L_A :\ \mathsf{newobj}\ C_1$$
$$\Delta;\Gamma;\Psi_2 \vdash \{y_1 \bullet P_1 * P_2\}\ L_B :\ \mathsf{newobj}\ C_2$$

$$\frac{\begin{array}{c} C_1{::}\,run : \{P_1\}\_\{Q_1\} \in \Gamma \qquad C_2{::}\,run : \{P_2\}\_\{Q_2\} \in \Gamma \\ (y_1, y_2) \bullet Q_1' * Q_2' \Rightarrow E_{L_C+1} \end{array}}{\Delta;\Gamma;\Psi_3 \vdash \left\{(y_1, y_2) \bullet P_1' * P_2'\right\}\ \ L_C : \mathsf{invokeStartJoin}\ C_1{:}run\ C_2{:}run}$$

where $P_1' \stackrel{def}{=} P_1[y_1/\mathtt{this}] \wedge y_1 \neq \mathtt{null}$ $\quad P_2' \stackrel{def}{=} P_2[y_2/\mathtt{this}] \wedge y_2 \neq \mathtt{null}$
$Q_1' \stackrel{def}{=} Q_1[y_1/\mathtt{this}]$ $\quad Q_2' \stackrel{def}{=} Q_2[y_2/\mathtt{this}]$
y_1 and y_2 are fresh objects of type C_1 and C_2 resp.,
and Ψ_1, Ψ_2, Ψ_3 only contain the labels relevant to the instructions at L_A, L_B, L_C resp.

5.2 Critical Regions

Critical Regions in the Source Logic. To access a resource in a critical region, O'Hearn's work [19] uses a statement `with` r `do` s. This statement can be implemented in Java using synchronized statements. O'Hearn's rule, adapted to Java, is defined as follows:

$$\frac{\Delta;\Gamma \vdash \{P * RI_r\}\ s_1\ \{Q * RI_r\}}{\Delta;\Gamma \vdash \{P\}\ \mathsf{synchronized}\ (r)\ s_1\ \{Q\}} \quad \text{where no other process modifies variables free in } P \text{ or } Q.$$

In this rule, the code in the critical region can see the state RI_r associated with the resource r. However, outside this region, reasoning proceeds without this knowledge. The state RI_r is called *resource invariant*; it is fixed for each resource r.

Critical Regions for the Bytecode Logic. To model critical regions, Java Bytecode provides two instructions: monitorenter and monitorexit to entering and leaving a critical region. To simplify the semantics and the proof transformation, we assume these instructions take a given resource r as argument (in Java Bytecode, these resources are pushed onto the stack). The rules for these instructions are defined as follows:

$$\frac{S \bullet P * RI_r \Rightarrow\ E_{l+1}}{\Delta;\Gamma;\Psi,(l{+}1,E_{l+1}) \vdash \{S \bullet P\}\ \ l : \mathsf{monitorenter}\ r}$$

$$\frac{S \bullet Q \Rightarrow\ E_{l+1}}{\Delta;\Gamma;\Psi,(l{+}1,E_{l+1}) \vdash \{S \bullet Q * RI_r\}\ \ l : \mathsf{monitorexit}\ r}$$

The first rule adds the *resource invariant* RI_r to the precondition P; the second rule removes this *resource invariant* from the precondition $S \bullet Q * RI_r$.

Proof Transformation. The translation of critical regions uses the bytecode instructions monitorenter and monitorexit. The translation is:

$$\nabla_S \left(\frac{\Delta; \Gamma \vdash \{P * RI_r\}\ s_1\ \{Q * RI_r\}}{\Delta; \Gamma \vdash \{P\}\ \textsf{synchronized}\ (r)\ s_1\ \{Q\}} \right) =$$

$$\begin{array}{ll} \Delta; \Gamma; \Psi_1 \vdash \{\epsilon \bullet P\} & L_A : \textsf{monitorenter}\ r \\ & \nabla_S(\ \Delta; \Gamma \vdash \{P * RI_r\}\ s_1\ \{Q * RI_r\}\ ) \\ \Delta; \Gamma; \Psi_2 \vdash \{\epsilon \bullet Q * RI_r\} & L_B : \textsf{monitorexit}\ r \end{array}$$

where Ψ_1 and Ψ_2 only contain the labels relevant to the instructions at labels L_A and L_B respectively.

To check the validity of the translation, we need to show the validity of each generated instruction. Since the precondition of the first instruction of s_1 is $P * RI_r$, then the instruction monitorenter is valid because $P * RI_r \Rightarrow P * RI_r$. The postcondition of s_1 is $Q * RI_r$, which is the precondition of monitorexit. By the definition of monitorexit, we need to show Q implies the postcondition of s_1, which is Q. Therefore, the translation is valid.

6 Example

This section presents an example of the application of the proof transformation.

Our proof translation takes the proof of the cell example (Figure 1), and produces a bytecode proof. The source proof consist of the proof for the classes *Cell* and *Recell* where each proof contains the proof of their methods. The proof translation is performed in two steps. In the first step, the rules for classes and method specifications are translated using the functions ∇_C and ∇_M. In the second step, the method bodies are translated using the function ∇_S. This function takes the proof of Figure 1b, and produces the bytecode proof of Figure 2 .

The static and dynamic specifications, highlighted in Figure 2, express the same properties as in the source program. The body of the method consists of a sequence of precondition, label, and instruction. Bytecode preconditions are pairs $S \bullet P$ where S is a list of expressions representing the stack, and P is a formula in separation logic. For example, the precondition at label 03 expresses that the object `this` is on the top of the stack and that the property $Val(this, v) * this.bak \mapsto \_$ holds. The stack grows to the right, e.g. in $(this, x)$ the top element is x; we denote the empty stack with ϵ. The translation function ∇_S first applies the *Open axiom* generating the bytecode proof at label 01. Then, the triple for the assignment $bak = super.get()$ is translated, producing the proof at labels 02–05. Then, the triple for the method invocation $super.set(x)$; is translated producing the proof at labels 05–07. Finally, the *Close axiom* is translated producing the proof at label 09. The last instruction of the proof is the return instruction.

7 Soundness of the Proof-Transforming Compiler

In this section, we present the soundness theorems for the proof-transforming compiler. Soundness informally means that the translation produces valid bytecode proofs.

public override void *set* (**int** *x*)

dynamic $\{Val(\mathbf{this}, v, \_)\}\ \_\ \{Val(\mathbf{this}, x, v)\}$	
static $\{\mathbf{this}.val \mapsto v\}\ \_\ \{\mathbf{this}.val \mapsto x * \mathbf{this}.bak \mapsto v\}$	
$\{\ \epsilon \bullet Val_{Recell}(this, v, \_)\}$	01 : **nop**
$\{\ \epsilon \bullet Val_{Cell}(this, v) * this.bak \mapsto \_\ \}$	02 : **push** *this*
$\{\ this \bullet Val_{Cell}(this, v) * this.bak \mapsto \_\ \}$	03 : **invokespecial** *Cell:get*
$\{\ ret' \bullet Val_{Cell}(this, v) * this.bak \mapsto \_ *\ ret' = v\}$	04 : **push** *this*
$\{\ (ret', this) \bullet Val_{Cell}(this, v) * this.bak \mapsto \_ *\ ret' = v\}$	05 : **putfld** *bak*
$\{\ \epsilon \bullet Val_{Cell}(this, v) * this.bak \mapsto v\ \}$	06 : **push** *this*
$\{\ this \bullet Val_{Cell}(this, v) * this.bak \mapsto v\ \}$	07 : **push** *x*
$\{\ (this, x) \bullet Val_{Cell}(this, v) * this.bak \mapsto v\ \}$	08 : **invokespecial** *Cell* : *set*
$\{\ \epsilon \bullet Val_{Cell}(this, x) * this.bak \mapsto v\ \}$	09 : **nop**
$\{\ \epsilon \bullet Val_{Recell}(this, x, v)\}$	10 : **ret**

Fig. 2. Example of the Application of Proof-Transforming Compilation.

Soundness is defined with three theorems for the translation of classes, methods, and instructions. The proofs can be found in our technical report [15].

The following theorem expresses that if the *class* rule in the source logic is a valid derivation, then the translation produces a valid derivation in the bytecode logic.

Theorem 1 (Soundness of the Class Translator)

$$\frac{\textit{for all } M_i \textit{ in } \overline{M} : \Delta; \Gamma \vdash M_i}{\Delta; \Gamma \vdash \textbf{\textit{class}}\ C{:}D\ \{\textbf{\textit{public}}\ \ \overline{T}\,\overline{f}; \overline{M}\ \}} \quad \Rightarrow \quad \nabla_C \left(\frac{\textit{for all } M_i \textit{ in } \overline{M} : \Delta; \Gamma \vdash M_i}{\Delta; \Gamma \vdash \textbf{\textit{class}}\ C{:}D\ \{\textbf{\textit{public}}\ \ \overline{T}\,\overline{f}; \overline{M}\ \}} \right)$$

The soundness theorem for the method translator expresses that if the proof of the method *m* is a valid derivation, then the proof translation produces a valid bytecode proof. It is defined as follows:

Theorem 2 (Soundness of the Method Translator) *Let* $Tree_1$ *be the derivation tree of the Method rule. Then,*

$$\frac{Tree_1}{\Delta; \Gamma \vdash \begin{array}{l}\textbf{\textit{public virtual}}\ C.m(\overline{x}) \\ \textbf{\textit{dynamic}}\ \{P_C\}\_\{Q_C\}\ \textbf{\textit{static}}\ \{R_C\}\_\{S_C\}\ \ body\end{array}} \quad \Rightarrow$$

$$\nabla_M \left(\frac{Tree_1}{\Delta; \Gamma \vdash \begin{array}{l}\textbf{\textit{public virtual}}\ C.m(\overline{x}) \\ \textbf{\textit{dynamic}}\ \{P_C\}\_\{Q_C\}\ \textbf{\textit{static}}\ \{R_C\}\_\{S_C\}\ \ body\end{array}} \right)$$

The following theorem, for instruction translation, states that if (1) we have a valid source proof for the instruction *s*, and (2) we have a proof translation from the source proof that produces the instructions $I_{l_{start}} \ldots I_{l_{end}}$, and their respective preconditions $E_{l_{start}} \ldots E_{l_{end}}$, and (3) the postcondition in the source logic implies the next precondition of the last generated instruction (if the last generated instruction is the last instruction of the method, we use the postcondition in the source logic), then every bytecode specification holds:$\Delta; \Gamma; \Psi \vdash \{E_l\}\ \ l : I_l$. The theorem is the following:

Theorem 3 (Soundness of the Instruction Translator) *Let $Tree_1$ be the derivation tree used to prove the instruction s. Then,*

$$
\begin{array}{l}
\dfrac{Tree_1}{\Delta; \Gamma \vdash \{P\}\ s\ \{Q\}} \quad \wedge \\
(\ (E_{l_{start}}, I_{l_{start}}) ... (E_{l_{end}}, I_{l_{end}})\ ) = \nabla_S \left(\dfrac{Tree_1}{\Delta; \Gamma \vdash \{P\}\ s\ \{Q\}} \right) \wedge \\
\left(Q \ \Rightarrow \ E_{l_{end+1}} \right) \\
\Rightarrow \\
\forall\ l \ \in \ l_{start} \ ... \ l_{end} : \ \Delta; \Gamma; \Psi \vdash \{E_l\}\ l : I_l
\end{array}
$$

The proof runs by induction on the structure of the derivation tree of

$$\frac{Tree_1}{\Delta; \Gamma \vdash \{P\}\ s\ \{Q\}}$$

8 Related Work

Bytecode Analysis. Several logics for bytecode have been developed. Stata and Abadi [24] first introduced a type system for Java bytecode. To verify bytecode with frame properties, Benton [5] has developed compositional logic for a stack-based abstract machine. The logic is a separation style logic and uses shifting operations to reindex stack assertions. Chin et al. [6] also present a heap model for a bytecode language to support separation logic. Dong et al. [9] develop a modular reasoning technique for low-level intermediate programs. However, those works do not support object-oriented features. Bannwart and Müller [1] present a Hoare-style logic for a bytecode language with object-oriented features similar to the JVM language. Dynamic and static specifications are treated in their logic, however, their inter-relationship is not considered.

Proof-Transforming Compilation. There has been several works on proof-transforming compilation [1,3,12,18,23]. The closest related work to our proof-transforming compiler are the works by Barthe *et al.* [4,3] on proof preserving compilation. They prove the preservation of proof obligations from Java programs to JVM programs; thus, they show that if the certificate proves the verification condition in the source, then this certificate can be used to prove the verification condition in the bytecode. Our bytecode logic and proof transformation can handle more complex examples that those works cannot; for example, programs using mutable data structures such as the programs proven by Distefano et al. [8], which include the factory, observer, and visitor patterns. The limitation on those works is given by the techniques used to verify the source program. Our work introduces a bycode logic using separation logic and its proof transformation, which makes possible to translate the proofs of programs using mutable data structures.

Kunz [10] presents proof preserving compilation for concurrent programs using an Owicki/Gries-like proof system. Our work handles non-interference and concurrent programs using separation logic.

Compared to our earlier effort on proof transformation [18,12,17], this work has a cleaner treatment of the stack, develops a more powerful bytecode logic, uses a different and more powerful source code proof system, and supports concurrency. Barthe [2] et al. implemented an infrastructure for Proof Carrying Code (PCC). Our current implementation [14] of the PCC infrastructure consist of proof transforming compiler for a Hoare-style logic, and a proof checker formalized in Isabelle. As future work, we plan to extend this implementation to handle separation logic.

9 Conclusions

We have developed a separation logic for bytecode; the logic adapts Parkinson and Bierman's work on abstract predicates [21] for bytecode. We also present proof transforming compilation from a separation logic for object-oriented programs to our bytecode logic. The bytecode logic and the proof transformation are sound. To prove soundness of the proof translation, we show that the translation of a valid source proof yields a valid bytecode proof. The proofs can be found in our technical report [15]. The use of a separation logic for bytecode allows us to translate more complex source proofs that previous works cannot handle, for example, programs using mutable data structures. The results show that the proof transformation can be extended to handle proofs of concurrent programs.

Acknowledgements We thank Stephan van Staden, Sebastian Nanz, Scott West, and Mei Tang for their insightful comments on drafts of this paper. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389.

References

1. F. Y. Bannwart and P. Müller. A Program Logic for Bytecode. In F. Spoto, editor, *BYTECODE*, volume 141(1) of *ENTCS*, pages 255–273. Elsevier, 2005.
2. G. Barthe, P. Crégut, B. Grégoire, T. Jensen, and D. Pichardie. Formal methods for components and objects. chapter The MOBIUS Proof Carrying Code Infrastructure, pages 1–24. Springer-Verlag, Berlin, Heidelberg, 2008.
3. G. Barthe, B. Grégoire, and M. Pavlova. Preservation of Proof Obligations from Java to the Java Virtual Machine. In *IJCAR*, pages 83–99. Springer, 2008.
4. G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In *Third International Workshop on Formal Aspects in Security and Trust, Newcastle, UK*, pages 112–126, 2005.
5. N. Benton. A typed, compositional logic for a stack-based abstract machine. In *APLAS '05*, volume 3780 of *LNCS*, 2005.
6. W. Chin, C. David, H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In *POPL '08*, ACM, pages 87–99, 2008.
7. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02*, volume 37. ACM Press, 2002.
8. D. Distefano and M. J. Parkinson. jStar: Towards Practical Verification for Java. In *OOPSLA '08*, pages 213–226, 2008.
9. Y. Dong, S. Wang, L. Zhang, and P. Yang. Modular certification of low-level intermediate representation programs. In *ICSAC*, pages 563–570. IEEE Computer Society, 2009.

10. C. Kunz. Certificate translation for the verification of concurrent programs. In *Proceedings of TGC'10*, pages 237–252. Springer-Verlag, 2010.
11. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06*, volume 41, pages 42–54. ACM, 2006.
12. P. Müller and M. Nordio. Proof-Transforming Compilation of Programs with Abrupt Termination. In *SAVCBS '07*, pages 39–46, 2007.
13. G. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1998.
14. M. Nordio. *Proofs and Proof Transformations for Object-Oriented Programs*. PhD thesis, ETH Zurich, 2009.
15. M. Nordio, C. Calcagno, and B. Meyer. Certificates and separation logic. Technical report, ETH Zurich, 2013.
16. M. Nordio, C. Calcagno, P. Müller, and B. Meyer. A Sound and Complete Program Logic for Eiffel. In M. Oriol, editor, *TOOLS-EUROPE 2009*, volume 33 of *LNBIP*, pages 195–214, 2009.
17. M. Nordio, P. Müller, and B. Meyer. Formalizing Proof-Transforming Compilation of Eiffel programs. Technical Report 587, ETH Zurich, 2008.
18. M. Nordio, P. Müller, and B. Meyer. Proof-Transforming Compilation of Eiffel Programs. In *TOOLS-EUROPE*, LNBIP, pages 316–335. Springer, 2008.
19. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375:271–307, 2007.
20. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04*, ACM, pages 268–280, 2004.
21. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08*, pages 75–86. ACM, 2008.
22. A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *TACAS '98*, pages 151–166. Springer-Verlag, 1998.
23. A. Saabas and T. Uustalu. Program and proof optimizations with type systems. *Journal of Logic and Algebraic Programming*, 77(1–2):131–154, 2008.
24. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *POPL '98*, pages 149–160. ACM, 1998.

A Soundness of the Bytecode Logic

In this section, we present the soundness theorems of the bytecode logic. First, we define the operational semantic of the bytecode language, and then we present the theorems. The soundness proofs can be found in our technical report [15].

A.1 Operational Semantics

The transitions of the operational semantics have the form

$$\langle p;\ s, \sigma, h, l\rangle \rightarrow \langle s', \sigma', h', l'\rangle \ \mid\ \mathit{fault}$$

where s, s' are stacks, σ, σ' are stores, and h, h' are heaps. The transition $\langle p;\ s, \sigma, h, l\rangle \rightarrow \langle s', \sigma', h', l'\rangle$ expresses that, executing a instruction I_l of the program body p at the location l with the stack s, the store σ, and the heap h produces the configuration $\langle s', \sigma', h', l'\rangle$. For a given method body p, the multistep relation $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$.

Figure 3 shows the semantics for all the instructions except method invocation. The rule for the instruction pop x removes the top element of the stack and assigns it to x; push v puts the value v on top of the stack; goto transfers control the program point l'; nop has no effect; brtrue transfers control to the label l' if the top of the stack is *true* removing this value from the stack; if the value is *false*, it is removed and control continues in the next instruction; The instruction putfld f updates the field f. If the instructions pop, brtrue, and putfld are applied with an empty stack, the transition yields the state *fault*. If putfld is applied with a stack with one element, the transition also yields the state *fault*.

$$\begin{array}{l}
\langle l : \mathsf{pop}\ x; (s, v), \sigma, h, l\rangle \ \rightarrow\ \langle s, \sigma[x := v], h, l+1\rangle \\
\langle l : \mathsf{push}\ v; s, \sigma, h, l\rangle \ \rightarrow\ \langle (s, v), \sigma, h, l+1\rangle \\
\langle l : \mathsf{goto}\ l'; s, \sigma, h, l\rangle \ \rightarrow\ \langle s, \sigma, h, l'\rangle \\
\langle l : \mathsf{nop}; s, \sigma, h, l\rangle \ \rightarrow\ \langle s, \sigma, h, l+1\rangle \\
\langle l : \mathsf{brtrue}\ l'; (s, \mathit{true}), \sigma, h, l\rangle \ \rightarrow\ \langle s, \sigma, h, l'\rangle \\
\langle l : \mathsf{brtrue}\ l'; (s, \mathit{false}), \sigma, h, l\rangle \ \rightarrow\ \langle s, \sigma, h, l+1\rangle \\
\langle l : \mathsf{putfld}\, f; (s, x, v), \sigma, h, l\rangle \ \rightarrow\ \langle s, \sigma, h[h(\sigma(x)).f := v], l+1\rangle \text{ when } \sigma(x).f \in \mathit{dom}(h) \\
\langle l : \mathsf{newobj}\ C; s, \sigma, h, l\rangle \ \rightarrow\ \langle ((s, y), \sigma, h[y/\mathit{this}], l+1\rangle \\
\langle l : s; \epsilon, \sigma, h, l\rangle \ \rightarrow\ \mathit{fault} \text{ when } s = \mathsf{pop}, \mathsf{brtrue}\ l',\ \mathit{or}\ \mathsf{putfld}\, f \\
\langle l : s; v, \sigma, h, l\rangle \ \rightarrow\ \mathit{fault} \text{ when } s = \mathsf{putfld}\, f
\end{array}$$

Fig. 3. Operational Semantics for the Basic Bytecode Instructions.

The Java Bytecode instruction invokespecial is used to call (1) private methods, and (2) super methods (invocations using `super` in Java). The rule is defined as follows:

$$\frac{y \neq \texttt{null} \qquad \langle \mathit{body};\ s, \sigma[\texttt{this} := y, \overline{p} := \overline{z}], h, l_1\rangle \ \rightarrow^*\ \langle s', \sigma', h', l'\rangle}{\langle l : \mathsf{invokespecial}\ C{::}m;\ (s, y, \overline{z}), \sigma, h, l\rangle \rightarrow\ \langle (s, \sigma'(\mathit{ret})), \sigma', h', l+1\rangle}$$

This rule assumes that the target object and the arguments are already on the stack. First, the arguments and the current object are updated, and then the body of the method is executed producing the configuration $\langle s', \sigma', h', l' \rangle$. The configuration of the method invocation is updated with the result of the method m, and the program counter is increased. If the instruction invokespecial is invoked with a stack that does not contain the target object and the arguments, the operational semantics produces *fault*.

A.2 Soundness Theorems

In this section, we define soundness of the bytecode logic. First, we introduce the semantics for Hoare triples in bytecode, and the semantics for instruction specifications. Then, we define soundness for bytecode instructions and soundness for method specifications in bytecode.

The following definition, taken from Parkinson and Bierman [21], gives semantics of abstract predicates. The step index n is used to deal with mutual recursion in method definitions.

Definition 2 (Abstract Predicates).

$$\begin{array}{l}
\textit{for all } \Lambda : \textit{Preds} \rightarrow (\textit{Vals}^* \rightarrow P(\Sigma)) \ , \ \ \Lambda \models_n \ \{P\} \ p \ \{Q\} \textit{ iff} \\
\forall m \leq n : \\
\textit{for all } s, \sigma, h \models \{P\} \ : \langle p; s, \sigma, h, l \rangle \not\rightarrow^* \textit{fault}, \textit{ and} \\
\langle p; s, \sigma, h, l \rangle \rightarrow^m \langle s', \sigma', h', l' \rangle \textit{ then} \\
s', \sigma', h' \models Q
\end{array}$$

Following, we define the semantics of Hoare triples. This definition expresses that for all interpretations satisfying the abstract predicate definition in Δ, and assuming all the methods executed for at most n steps meet their specifications in Γ, then $\{P\} \ p \ \ \{Q\}$ is satisfied for at least $n+1$ steps.

Definition 3 (Hoare Triples $\models$).

$$\begin{array}{l}
\Delta; \Gamma \models \{P\} \ \ p \ \ \{Q\} \textit{ iff} : \\
\textit{for all } \Lambda \textit{ and } n, \textit{ if } \Lambda \models \Delta \textit{ and } \Lambda \models_n \Gamma, \textit{ then} \\
\Lambda \models_{n+1} \ \{P\} \ p \ \ \{Q\}.
\end{array}$$

The semantics for the instruction specification $\Delta; \Gamma; \Psi \models \{S \bullet P\} \ \ l\!:\!\textit{inst}$ is defined as follows:

Definition 4 (Instructions Specifications $\models$).

$$\begin{array}{l}
\Delta; \Gamma; \Psi \models \{S \bullet P\} \ \ l : \textit{inst iff} : \\
\textit{for all } s, \sigma, h \models (S \bullet P), \textit{and } \langle l : \textit{inst}; s, \sigma, h, l \rangle \not\rightarrow \textit{fault}, \textit{and} \\
\langle l : \textit{inst}; s, \sigma, h, l \rangle \rightarrow \langle s', \sigma', h', l' \rangle \textit{ then} \\
s', \sigma', h' \models \Psi(l')
\end{array}$$

The semantics for the method $C.m$ with the dynamic specification $\{P_C\}\_\{Q_C\}$, the static specification $\{R_C\}\_\{S_C\}$, and body b is defined as follows:

Definition 5 (Methods $\models$).

$$\begin{array}{l}\Delta;\Gamma \models \textbf{\textit{public}}\ C.m(\bar{x})\ \ \textbf{\textit{dynamic}}\ \{P_C\}\_\{Q_C\}\ \textbf{\textit{static}}\ \{R_C\}\_\{S_C\}\ b\ \textit{iff}\ :\\ \Delta;\Gamma \models \{R_C\}\_\{S_C\}\ \ \textit{implies}\ \Delta;\Gamma \models \{P_c * \textit{this} : C\}\_\{Q_c\}\\ \textit{and for all}\ s,\sigma,h \models R_C\ \textit{implies}\ E_{l_1}\\ \textit{and for all}\ s,\sigma,h \models E_{l_n}\ \textit{implies}\ S_C,\ \textit{then}\\ \textit{for all inst in}\ b :\ \ \Delta;\Gamma;\Psi \models \{S \bullet P\}\ \ \ l : \textit{inst}\end{array}$$

Definition 6 (Methods $\models$).

$$\begin{array}{l}\Delta;\Gamma \models \textbf{\textit{public}}\ C.m(\bar{x})\ \ \textbf{\textit{dynamic}}\ \{P_C\}\_\{Q_C\}\ \textbf{\textit{static}}\ \{R_C\}\_\{S_C\}\ b\ \textit{iff}\ :\\ \Delta;\Gamma \models \{R_C\}\_\{S_C\}\ \ \textit{implies}\ \Delta;\Gamma \models \{P_c * \textit{this} : C\}\_\{Q_c\}\\ \textit{and for all}\ s,\sigma,h \models R_C : \langle b; s,\sigma,h,l\rangle \not\rightarrow^* \textit{fault},\ \textit{and}\\ \langle b; s,\sigma,h,l\rangle \rightarrow^* \langle s',\sigma',h',l'\rangle\ \textit{then}\\ s',\sigma',h' \models S_C\end{array}$$

The following theorem defines soundness for bytecode instructions:

Theorem 4 (Soundness for Instructions)

$$\Delta;\Gamma;\Psi \vdash \{S \bullet P\}\ \ l : \textit{inst}\ \ \textit{implies}\ \ \Delta;\Gamma;\Psi \models \{S \bullet P\}\ \ \ l : \textit{inst}$$

Proof. The proof of this theorem runs by induction on the structure of the derivation tree of $\Delta;\Gamma;\Psi \vdash \{S \bullet P\}\ l : \textit{inst}$. The complete proof is presented in our technical report [15].

Finally, we define the soundness theorem for method specifications:

Theorem 5 (Soundness for Methods)

$$\begin{array}{l}\Delta;\Gamma \vdash \textbf{\textit{public}}\ C.m(\bar{x})\ \ \textbf{\textit{dynamic}}\ \{P_C\}\_\{Q_C\}\ \textbf{\textit{static}}\ \{R_C\}\_\{S_C\}\ b\\ \textit{implies}\\ \Delta;\Gamma \models \textbf{\textit{public}}\ C.m(\bar{x})\ \ \textbf{\textit{dynamic}}\ \{P_C\}\_\{Q_C\}\ \textbf{\textit{static}}\ \{R_C\}\_\{S_C\}\ b\end{array}$$

Proof. By induction on the structure of de derivation tree of $C.m(\bar{x})$, and the application of Theorem 4. The complete proof is presented in our technical report [15].