Type Inference Against Races (extended version) Williams College Technical Note 04-06

Cormac Flanagan[1] and Stephen N. Freund[2]

[1] Dept. of Computer Science, University of California at Santa Cruz, Santa Cruz, CA 95064*
[2] Dept. of Computer Science, Williams College, Williamstown, MA 01267*

Abstract. The race condition checker `rccjava` uses a formal type system to statically identify potential race conditions in concurrent Java programs, but it requires programmer-supplied type annotations. This paper describes a type inference algorithm for `rccjava`. Due to the interaction of parameterized classes and dependent types, this type inference problem is NP-complete. This complexity result motivates our new approach to type inference, which is via reduction to propositional satisfiability. This paper describes our type inference algorithm and its performance on programs of up to 30,000 lines of code.

This is an extended version of our paper appearing at the Static Analysis Symposium, 2004.

1 Introduction

A race condition occurs when two threads in a concurrent program manipulate a shared data structure simultaneously, without synchronization. Errors caused by race conditions are notoriously hard to catch using testing because they are scheduling dependent and difficult to reproduce. Typically, programmers attempt to avoid race conditions by adopting a programming discipline in which shared variables are protected by locks.

In a previous paper [10], we described a static analysis tool called `rccjava` that enforces this lock-based synchronization discipline. The analysis performed by `rccjava` is formalized as a type system, and it incorporates features such as *dependent types* (where the type of a field describes the lock protecting it) and *parameterized classes* (where fields in different instances of a class can be protected by different locks).

Our previous evaluation of `rccjava` indicates that it is effective for catching race conditions. However, `rccjava` relies on programmer-inserted type annotations that describe the locking discipline, such as which lock protects a particular field. The need for these type annotations limits `rccjava`'s applicability to large, legacy systems. Hence, to achieve practical static race detection for large programs, annotation inference techniques are necessary.

* This work was supported in part by the National Science Foundation under Grants CCR-0341179 and CCR-0341387.

In previous work along these lines, we developed Houdini/rcc [11], a type inference algorithm for `rccjava` that heuristically generates a large set of candidate type annotations and then iteratively removes all invalid annotations. However, this approach could not handle parameterized classes or methods, which limits its ability to handle many of the synchronization idioms of real programs.

In the presence of parameterized classes, the type inference problem for `rccjava` is NP-complete, meaning that any type inference algorithm will have an exponential worst-case behavior. This complexity result motivates our new approach to type inference, which is via reduction to propositional satisfiability. That is, given an unannotated (or partially-annotated) program, we translate this program into a propositional formula that is satisfiable if and only if the original program is typeable. Moreover, after computing a satisfying assignment for the generated formula, we translate this assignment into appropriate annotations for the program, yielding a valid, explicitly-typed program. This approach works well in practice, and we report on its performance on programs of up to 30,000 lines of code.

Producing a small number of meaningful error messages for erroneous or untypeable programs is often challenging. We tackle this aspect of type inference by generating a weighted MAX-SAT problem [4] and producing error messages for the unsatisfied clauses in the optimal solution. Our experience shows that the resulting warnings often correspond to errors in the original program, such as accessing a field without holding the appropriate lock.

We have implemented our algorithm in the *Rcc/Sat* tool for multithreaded Java programs. Experiments on benchmark programs demonstrate that it is effective at inferring valid type annotations for multithreaded code. The algorithm's precision is significantly improved by performing a number of standard analyses, such as control-flow and escape analysis, prior to type checking.

The key contributions of this paper include:

- demonstrating that the type inference problem for race-free type systems like `rccjava` is NP-complete;
- a type inference algorithm based on reduction to propositional satisfiability;
- a refinement of this approach to generate useful error messages via reduction to weighted MAX-SAT; and
- experimental results that validate the effectiveness of this approach.

The annotations constructed by Rcc/Sat also provide valuable documentation to the programmer; facilitate checking other properties such as atomicity [15, 14, 12]; and can help reduce state explosion in model checkers [23, 24, 13, 9].

2 Types Against Races

2.1 Type Checking

This section introduces RFJ2, an idealized multithreaded subset of Java with a type system that guarantees race freedom for well-typed programs. This type

$$
\begin{array}{rll}
P & ::= defn^*\ e & \text{(program)} \\
defn & ::= \texttt{class}\ cn\langle\texttt{ghost}\ x^*\rangle\ \{\ field^*\ meth^*\ \} & \text{(class declaration)} \\
field & ::= t\ fn\ \texttt{guarded\_by}\ l & \text{(field declaration)} \\
meth & ::= t\ mn\langle\texttt{ghost}\ x^*\rangle\texttt{(}arg^*\texttt{)}\ \texttt{requires}\ s\ \{\ e\ \} & \text{(method declaration)} \\
arg & ::= t\ x & \text{(argument declaration)} \\
c, t & ::= cn\langle l^*\rangle & \text{(type)} \\
l & ::= x \mid \alpha \mid l \cdot \theta & \text{(lock expression)} \\
s & ::= \emptyset \mid \{l\} \mid s \cup s \mid \beta \mid s \cdot \theta & \text{(lock set expression)} \\
\theta & ::= [x_1 := l_1, \ldots, x_n := l_n] & \text{(substitution)} \\
 & & \\
e, f & ::= x \mid \texttt{null} \mid \texttt{new}\ c\texttt{(}e^*\texttt{)} \mid e.fn \mid e.fn\ \texttt{=}\ e \mid e.mn\langle l^*\rangle\texttt{(}e^*\texttt{)} & \text{(expressions)} \\
 & \quad \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{synchronized}\ x\ e \mid e.\texttt{fork} &
\end{array}
$$

$$
\begin{array}{lll}
\alpha \in \mathit{LockVar} & x, y \in \mathit{Var} & fn \in \mathit{FieldName} \\
\beta \in \mathit{LockSetVar} & cn \in \mathit{ClassName} & mn \in \mathit{MethodName}
\end{array}
$$

Fig. 1. The idealized language RFJ2.

system extends our previous work on the `rccjava` type system [10], for example with parameterized methods. To clarify our presentation, RFJ2 also simplifies some aspects of `rccjava`. For example, it does not support inheritance. (Inheritance and other aspects of the full Java programming language are dealt with in our implementation, described in Section 4.)

An RFJ2 program (see Figure 1) is a sequence of class declarations together with an initial expression. Each class declaration associates a class name with a body that consists of a sequence of field and method declarations. The self-reference variable "`this`" is implicitly bound within the class body.

The RFJ2 language includes type annotations that specify the locking discipline. For example, the type annotation `guarded_by` x on a field declaration states that the lock denoted by the variable x must be held whenever that field is accessed (read or written). Similarly, the type annotation `requires` $x_1, \ldots, x_n$ on a method declaration states that these locks are held on method entry; the type system verifies that these locks are indeed held at each call-site of the method, and checks that the method body is race-free given this assumption.

The language provides *parameterized classes*, to allow the fields of a class to be protected by some lock external to the class. A parameterized class declaration

$$\texttt{class}\ cn\langle\texttt{ghost}\ x_1 \ldots x_n\rangle\ \{\ \ldots\ \}$$

introduces a binding for the *ghost* variables $x_1 \ldots x_n$, which can be referred to from type annotations within the class body. The type $cn\langle y_1 \ldots y_n\rangle$ refers to an *instantiated version* of cn, where each x_i in the body is replaced by y_i. As an example, the type $\texttt{Hashtable}\langle y_1, y_2\rangle$ may denote a hashtable that is protected by lock y_1, where each element of the hashtable is protected by lock y_2.

The RFJ2 language also supports *parameterized method* declarations, such as

$$t\ m\langle\texttt{ghost}\ x\rangle\texttt{(}cn\langle x\rangle\ y\texttt{)}\ \texttt{requires}\ x\ \{\ \ldots\ \}$$

(a) Example Program `Ref`

```
class Lock⟨⟩ { }
class Ref⟨ghost x⟩ {
   int y guarded_by α1
   boolean lessThan(Ref⟨α2⟩ o) requires β {
      this.y < o.y ;
   }
}
```

```
let lock = new Lock⟨⟩();
    r1 = new Ref⟨α3⟩(1);
    r2 = new Ref⟨α4⟩(2)
in synchronized (lock) {
    r1.lessThan(r2);
}
```

(b) Constraints

$\alpha_1 \in \{ \texttt{this}, \texttt{x} \}$		decl. of `y`
$\alpha_2 \in \{ \texttt{this}, \texttt{x} \}$		decl. of `lessThan`
$\beta \subseteq \{ \texttt{this}, \texttt{x}, \texttt{o} \}$		decl. of `lessThan`
$\alpha_3 \in \{ \texttt{lock} \}$		first `new` expr.
$\alpha_4 \in \{ \texttt{lock}, \texttt{r1} \}$		second `new` expr.
$\alpha_1 \in \beta$		access to `this.y`
$\alpha_1[\texttt{this} := \texttt{o}, \texttt{x} := \alpha_2] \in \beta$		access to `o.y`
$\beta[\texttt{this} := \texttt{r1}, \texttt{x} := \alpha_3, \texttt{o} := \texttt{r2}] \subseteq \{\texttt{lock}\}$		requires for call
$\alpha_2[\texttt{this} := \texttt{r1}, \texttt{x} := \alpha_3, \texttt{o} := \texttt{r2}] = \alpha_4$		arg. type for call

(c) Conditional Assignment

$Y(\alpha_1) = (b_1?\texttt{this} : \texttt{x})$	decl. of `y`
$Y(\alpha_2) = (b_2?\texttt{this} : \texttt{x})$	decl. of `lessThan`
$Y(\beta) = (b_4?\texttt{this} : \emptyset) \cup (b_5?\texttt{x} : \emptyset) \cup (b_6?\texttt{o} : \emptyset)$	decl. of `lessThan`
$Y(\alpha_3) = \texttt{lock}$	first `new` expr.
$Y(\alpha_4) = (b_3?\texttt{lock} : \texttt{r1})$	second `new` expr.

(d) Boolean Constraints

$(b_1?\texttt{this} : \texttt{x}) \in (b_4?\texttt{this} : \emptyset) \cup (b_5?\texttt{x} : \emptyset) \cup (b_6?\texttt{o} : \emptyset)$	access to `this.y`
$(b_1?\texttt{o} : (b_2?\texttt{this} : \texttt{x})) \in (b_4?\texttt{this} : \emptyset) \cup (b_5?\texttt{x} : \emptyset) \cup (b_6?\texttt{o} : \emptyset)$	access to `o.y`
$(b_4?\texttt{r1} : \emptyset) \cup (b_5?\texttt{lock} : \emptyset) \cup (b_6?\texttt{r2} : \emptyset) \subseteq \{\texttt{lock}\}$	requires for call
$(b_2?\texttt{r1} : \texttt{lock}) = (b_3?\texttt{lock} : \texttt{r1})$	arg. type for call

(e) Boolean Formula

$[(b_1 \wedge b_4) \vee (\neg b_1 \wedge b_5)]$	access to `this.y`
$\wedge\ [(b_1 \wedge b_6) \vee (\neg b_1 \wedge ((b_2 \wedge b_4) \vee (\neg b_2 \wedge b_5)))]$	access to `o.y`
$\wedge\ [\neg b_4 \wedge \neg b_6]$	requires for call
$\wedge\ [(b_2 \wedge \neg b_3) \vee (\neg b_2 \wedge b_3)]$	arg. type for call

Fig. 2. Example program and type inference constraints.

which defines a method m that is parameterized by lock x, and which takes an argument of type $cn\langle x\rangle$. A corresponding invocation $e.m\langle z\rangle(e')$ must supply a ghost argument z and an actual parameter e' of type $cn\langle z\rangle$.

Expressions include object allocation `new` $c(e^*)$, which initializes a new object's fields with its argument values; field read and update; method invocation; and variable binding and reference. The expression `synchronized` x e is evaluated in a manner similar to Java's `synchronized` statement: the lock for object x is acquired, the subexpression e is then evaluated, and finally the lock is released. The expression e.`fork` starts a new thread. Here, e should evaluate to an object that includes a nullary method `run`. The fork operation spawns a new thread that calls that `run` method.

The RFJ2 type system leverages parameterized methods to reason about thread-local data. (This approach replaces the escape by analysis embedded in our earlier type system [10].) Specifically, the `run` method of each forked thread takes a `ghost` parameter `tl_lock` denoting a *thread-local* lock that is always held by that thread:

$$t\ \texttt{run}\langle\texttt{ghost tl\_lock}\rangle\texttt{()}\ \texttt{requires tl\_lock}\ \{\ e\ \}$$

Intuitively, the underlying run-time system creates and acquires this thread-local lock when a new thread is created. This lock may be used to guard thread-local data and may be passed as a ghost parameter to other methods that access thread-local data. In a similar fashion, we also introduce an implicit, globally-visible lock called `main_lock`, which is held by the initial program thread and can be used to protect data exclusively accessed by that thread.

2.2 Type Inference

Our previous evaluation of the race-free type system `rccjava` indicates that it is effective for catching race conditions [10]. However, the need for programmer-inserted annotations limits its applicability to large, legacy systems, which motivates the development of type inference techniques for race-free type systems.

In this paper we describe a novel type inference system for RFJ2. We introduce *lock variables* α and *lockset variables* β, collectively referred to as *locking variables*. Locking variables may be mentioned in type annotations, as in `guarded_by` α, `requires` β, or $cn\langle\alpha_1, \alpha_2\rangle$. During type inference, each lock variable α is resolved to some specific program variable in scope, and each lock set variable β is resolved to some set of program variables in scope. As an example, Figure 2(a) presents a simple reference cell implementation, written in RFJ2 extended with primitive types and operations, that contains locking variables.

An RFJ2 program is *explicitly-typed* if it contains no locking variables. The *type inference* problem is, given a program with locking variables, to resolve these locking variables so that the resulting explicitly-typed program is well-typed.

Parameterized classes introduce substitutions that complicate the type inference problem. We use the notation $[x_1 := l_1, \ldots, x_n := l_n]$ to denote a substitution θ that replaces each program variable x_i with the lock expression l_i. To illustrate the need for these substitutions, consider the class declaration:

$$\texttt{class}\ cn\langle\texttt{ghost}\ x\rangle\ \{\ t\ fn\ \texttt{guarded\_by}\ l;\ \}$$

If a variable p has type $cn\langle y\rangle$, then the field $p.fn$ is protected by $\theta(l)$, where the substitution $\theta \equiv [x := y]$ replaces the formal ghost parameter x by the actual parameter y. The application of a substitution to most syntactic entities is straightforward; however, the application of a substitution θ to a lock expression l is delayed until any lock variables α in the lock expression are resolved. We use the syntax $l \cdot \theta$ to represent this *delayed substitution*. Similarly, if the *lock* *set expression* s denote the set of locks in a method's requires clause, then the application of a substitution θ to s yields the delayed substitution $s \cdot \theta$. The following examples illustrate substitutions on various syntactic entities. (Due to space limitations, we do not present an exhaustive definition.)

$$\begin{aligned} \theta(x) &= l \quad \text{if } \theta \equiv [\ldots, x := l, \ldots] \\ \theta(\texttt{synchronized}\ x\ e) &= \texttt{synchronized}\ \theta(x)\ \theta(e) \\ \theta(l) &= l \cdot \theta \\ \theta(s) &= s \cdot \theta \end{aligned}$$

Since the type rules reason about delayed substitutions, we include these delayed substitutions in the programming language syntax, but we require that substitutions do not appear in source programs.

The type rules for RFJ2 generate a collection of *constraints* that contain delayed substitutions. These constraints include equality constraints between lock expressions and containment constraints between lock set expressions:

$$C ::= s \subseteq s \mid l = l$$

The core of the type system is defined by the judgment:

$$P; E; s \vdash e : t \ \&\ \bar{C}$$

Here, the program P is included to provide access to class declarations; E is an environment providing types for the free variables of the expression e; the lock set s describes the locks held when executing e; t is the type inferred for e; and $\bar{C}$ is the generated set of constraints.

Most of the type rules are straightforward. The complete set of type judgments and rules is contained in Appendix A. Here we briefly explain two of the more crucial rules. The rule for `synchronized` x e checks e with an extended lock set that includes x, since the lock x is held when evaluating e. The rule for $e.fn$ checks that e is a well-typed expression of some class type $cn\langle l_{1..n}\rangle$ and that cn has a field fn of type t, guarded by lock l.

$$\frac{\begin{array}{c} P; E; s \vdash x : t' \ \&\ \bar{C} \\ P; E; s \cup \{x\} \vdash e : t \ \&\ \bar{C}' \end{array}}{P; E; s \vdash \texttt{synchronized}\ x\ e : t \ \&\ (\bar{C} \cup \bar{C}')} \qquad \frac{\begin{array}{c} P; E; s \vdash e : cn\langle l_{1..n}\rangle \ \&\ \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ldots t\ fn\ \texttt{guarded\_by}\ l \ldots\} \in P \\ \theta = [\texttt{this} := e, x_j := l_j\ ^{j \in 1..n}] \\ P; E \vdash \theta(t) \ \&\ \bar{C}' \end{array}}{P; E; s \vdash e.fn : \theta(t) \ \&\ (\bar{C} \cup \bar{C}' \cup \{\theta(l) \in s\})}$$

Since the protecting lock expression l (and type t) may refer to the ghost parameters $x_{1..n}$ and the implicitly-bound self-reference `this`, neither of which are in scope at the field access, we introduce the substitution θ which substitutes appropriate expressions for these variables. The constraint $\theta(l) \in s$, an abbreviation for $\{\theta(l)\} \subseteq s$, ensures that the substituted lock expression is in the current lock set. The type of the field dereference is computed by applying the substitution θ to the field type t, which must yield a well-formed type.

The type system defines the top-level judgment $P \vdash \bar{C}$, where $\bar{C}$ is the generated set of constraints for the program P. Applying these type rules to the example program `Ref` of Figure 2(a) yields the constraints shown in Figure 2(b). (We ignore `main_lock` in this example for simplicity).

We next address the question of when the generated constraints over the locking variables are satisfiable. An *assignment*

$$A : (LockVar \to Var) \cup (LockSetVar \to 2^{Var})$$

resolves lock and lock set variables to corresponding program variables and sets of program variables, respectively. We extend assignments to lock expressions, lock set expressions, and substitutions. In particular, since an assignment resolves all locking variables, any delayed substitutions can be immediately performed.

$$\begin{array}{rcl} A : l \to Var & & \\ A(x) & = & x \\ A(l \cdot \theta) & = & A(\theta)(A(l)) \end{array} \qquad \begin{array}{rcl} A : s \to 2^{Var} & & \\ A(\emptyset) & = & \emptyset \\ A(\{l\}) & = & \{A(l)\} \\ A(s_1 \cup s_2) & = & A(s_1) \cup A(s_2) \\ A(s \cdot \theta) & = & A(\theta)(A(s)) \end{array} \qquad \begin{array}{l} A : \theta \to \theta \\ A([x_1 := l_1, \ldots, x_n := l_n]) = \\ \quad [x_1 := A(l_1), \ldots, x_n := A(l_n)] \end{array}$$

We also extend assignments in a compatible manner to other syntactic units, such as constraints, expressions, programs, etc.

An assignment A *satisfies* a constraint C (written $A \models C$) as follows:

$$\begin{array}{ll} A \models s_1 \subseteq s_2 & \text{iff } A(s_1) \subseteq A(s_2) \\ A \models l_1 = l_2 & \text{iff } A(l_1) = A(l_2) \end{array}$$

If $A \models C$ for all $C \in \bar{C}$ then we say A is a *solution* for $\bar{C}$, written $A \models \bar{C}$. A set of constraints $\bar{C}$ is *valid*, written $\models \bar{C}$, if every assignment is a solution for $\bar{C}$. For example, the constraints of Figure 2(b) for the program `Ref` are satisfied by the assignment: $\alpha_1 = \alpha_2 = \mathtt{x}, \alpha_3 = \alpha_4 = \mathtt{lock}$, and $\beta = \{\mathtt{x}\}$.

We say P is *well-typed* if $P \vdash \bar{C}$ and the constraints $\bar{C}$ are satisfiable. If a solution A for the constraints $\bar{C}$ exists, the following theorem states that the explicitly-typed program $A(P)$ is well-typed.

Lemma 1. *Given an assignment A and constraints $\bar{C}$, $A \models \bar{C}$ iff $\models A(\bar{C})$.*

Proof: See Appendix B.

Lemma 2. *If $P \vdash \bar{C}$ then $A(P) \vdash A(\bar{C})$.*

Proof: See Appendix B.

Theorem 1. *If $P \vdash \bar{C}$ and $A \models \bar{C}$ then $A(P) \vdash A(\bar{C})$ and $\models A(\bar{C})$.*

Proof: Follows from Lemmas 1 and 2.

For explicitly-typed programs, since the generated constraints $\bar{C}$ do not contain locking variables, checking the satisfiability of $\bar{C}$ is straightforward. In the more general case where P is not explicitly-typed, the type inference problem involves *searching* for a solution A for the generated constraints $\bar{C}$. Due to the interaction between parameterized classes and dependent types, the type inference problem for RFJ2 (and similarly for `rccjava`) is NP-complete. (The proof is via a reduction from propositional satisfiability.)

Theorem 2. *For an arbitrary program P, the problem of finding an assignment A such that $A(P)$ is explicitly-typed and $A(P) \vdash \bar{C}$ and $\models \bar{C}$ is NP-complete.*

Proof: By a reduction of the propositional satisfiability problem into the type inference problem, as shown in Appendix C.

Despite this worst-case complexity result, we demonstrate a technique in the next section that has proven effective in practice.

3 Solving Constraint Systems

3.1 Generating Boolean Constraints

For each lock variable α mentioned in the program, the type rules introduce a *scope constraint* $\alpha \in \{x_1, \ldots, x_n\}$ that constrains α to be one of the variables $x_1, \ldots, x_n$ in scope. A similar constraint $\beta \subseteq \{x_1, \ldots, x_n\}$ is introduced for each lock set variable β. These scope constraints specify the possible choices for each locking variable, and enable us to translate each constraint C over locking variables into a *Boolean constraint* D that uses Boolean variables to encode the possible choices for each locking variable. The notation $b\,?\,X : Y$ denotes X if the Boolean variable b is true, and denotes Y otherwise.

$$
\begin{array}{rcll}
D & ::= & S \subseteq S \mid L = L & \text{(Boolean constraints)} \\
L & ::= & x \mid b\,?\,L : L & \text{(conditional lock expressions)} \\
S & ::= & \emptyset \mid \{L\} \mid b\,?\,S : S \mid S \cup S & \text{(conditional lock set expressions)} \\
b & \in & \mathit{BoolVar} & \text{(Boolean variables)}
\end{array}
$$

From the scope constraints, we generate a *conditional assignment*

$$Y : (\mathit{LockVar} \rightarrow L) \cup (\mathit{LockSetVar} \rightarrow S)$$

that encodes the possible choices for each locking variable. For example, the scope constraints $\alpha \in \{x_1, \ldots, x_n\}$ and $\beta \subseteq \{y_1, \ldots, y_m\}$ yield:

$$
\begin{array}{l}
Y(\alpha) = b_1?x_1 : (b_2?x_2 : (\ldots b_{n-1}?x_{n-1} : x_n)\ldots) \\
Y(\beta) = (b'_1?\{y_1\} : \emptyset) \cup \cdots \cup (b'_m?\{y_m\} : \emptyset)
\end{array}
$$

where each Boolean variable b_i and b'_i is fresh.[3]

We extend the conditional assignment to translate each constraint C to a Boolean constraint $D = Y(C)$, and to translate lock expressions, lock set expressions, and substitutions, as follows. Since the conditional assignment (conditionally) resolves locking variables, as part of this translation we immediately

[3] We could encode the choice for the first constraint as a decision tree with only $\log n$ Boolean variables.

apply any delayed substitutions, to yield a substitution-free Boolean constraint:

$$\begin{array}{rcl} Y : C & \to & D \\ Y(s_1 \subseteq s_2) & = & Y(s_1) \subseteq Y(s_2) \\ Y(l_1 = l_2) & = & Y(l_1) = Y(l_2) \end{array} \qquad \begin{array}{rcl} Y : s & \to & S \\ Y(\emptyset) & = & \emptyset \\ Y(\{l\}) & = & \{Y(l)\} \\ Y(s_1 \cup s_2) & = & Y(s_1) \cup Y(s_2) \\ Y(s \cdot \theta) & = & Y(\theta)(Y(s)) \end{array}$$

$$\begin{array}{rcl} Y : l & \to & L \\ Y(x) & = & x \\ Y(l \cdot \theta) & = & Y(\theta)(Y(l)) \end{array} \qquad \begin{array}{l} Y([x_1 := l_1, \ldots, x_n := l_n]) \;= \\ \qquad [x_1 := Y(l_1), \ldots, x_n := Y(l_n)] \end{array}$$

Figure 2(c) and (d) show the conditional assignment and Boolean constraints for the example program `Ref`.

A *truth assignment* $B : \mathit{BoolVar} \rightharpoonup \mathit{Boolean}$ assigns truth values to Boolean variables. We extend truth assignments to L and S in a straightforward manner:

$$\begin{array}{rcl} B : L & \to & \mathit{Var} \\ B(x) & = & x \\ B(b \,?\, L_1 : L_2) & = & \begin{cases} B(L_1) \text{ if } B(b) \\ B(L_2) \text{ if } \neg B(b) \end{cases} \end{array} \qquad \begin{array}{rcl} B : S & \to & 2^{\mathit{Var}} \\ B(\emptyset) & = & \emptyset \\ B(\{L\}) & = & \{B(L)\} \\ B(b \,?\, S_1 : S_2) & = & \begin{cases} B(S_1) \text{ if } B(b) \\ B(S_2) \text{ if } \neg B(b) \end{cases} \\ B(S_1 \cup S_2) & = & B(S_1) \cup B(S_2) \end{array}$$

A truth assignment B satisfies a set of Boolean constraints $\bar{D}$ if $B \models D$ for each $D \in \bar{D}$, where:

$$\begin{array}{ll} B \models S_1 \subseteq S_2 & \text{iff } B(S_1) \subseteq B(S_2) \\ B \models L_1 = L_2 & \text{iff } B(L_1) = B(L_2) \end{array}$$

For example, the Boolean constraints of Figure 2(d) are satisfied by the following truth assignment: $b_1 = b_2 = b_4 = b_6 = \textbf{false}$ and $b_3 = b_5 = \textbf{true}$.

The application of a truth assignment B to a conditional assignment Y yields the (unconditional) assignment $B(Y)$, defined as $B(Y)(x) = B(Y(x))$.

The translation from constraints to Boolean constraints is semantics-preserving, in the sense that if the generated Boolean constraints are satisfiable, then the original constraints are also satisfiable.

Theorem 3. *Suppose $\bar{D} = Y(\bar{C})$ and let B be a truth assignment. Then $B(Y) \models \bar{C}$ if and only if $B \models \bar{D}$.*

Proof: See Appendix B.

3.2 Solving Boolean Constraints

The final step is to find a truth assignment B satisfying the generated Boolean constraints $\bar{D}$. We accomplish this step by translating $\bar{D}$ into a Boolean formula F, which can then be solved by a standard propositional satisfiability solver such as Chaff [20]. The Boolean formula syntax and this translation are as follows:

$$F \;::=\; \textbf{true} \mid \textbf{false} \mid b \mid F \vee F \mid F \wedge F \mid \neg F$$

$$[\![\cdot]\!] : \bar{D} \to F$$
$$[\![\bar{D}]\!] = \wedge_{D \in \bar{D}} [\![D]\!]$$

$$[\![\cdot]\!] : D \to F$$

$$\begin{aligned}
[\![x = x]\!] &= \mathbf{true} \\
[\![x = y]\!] &= \mathbf{false} \ \text{ if } x \not\equiv y \\
[\![L = (b?L_1 : L_2)]\!] &= (b \wedge [\![L = L_1]\!]) \\
&\quad \vee(\neg b \wedge [\![L = L_2]\!]) \\
[\![(b?L_1 : L_2) = L]\!] &= [\![L = (b?L_1 : L_2)]\!] \\
[\![\emptyset \subseteq S]\!] &= \mathbf{true} \\
[\![(S_1 \cup S_2) \subseteq S]\!] &= [\![S_1 \subseteq S]\!] \\
&\quad \wedge [\![S_2 \subseteq S]\!]
\end{aligned}
\qquad
\begin{aligned}
[\![(b?S_1 : S_2) \subseteq S]\!] &= (b \wedge [\![S_1 \subseteq S]\!]) \\
&\quad \vee(\neg b \wedge [\![S_2 \subseteq S]\!]) \\
[\![\{L\} \subseteq \emptyset]\!] &= \mathbf{false} \\
[\![\{L\} \subseteq (b?S_1 : S_2)]\!] &= (b \wedge [\![\{L\} \subseteq S_1]\!]) \\
&\quad \vee(\neg b \wedge [\![\{L\} \subseteq S_2]\!]) \\
[\![\{L\} \subseteq (S_1 \cup S_2)]\!] &= [\![\{L\} \subseteq S_1]\!] \\
&\quad \vee [\![\{L\} \subseteq S_2]\!] \\
[\![\{L_1\} \subseteq \{L_2\}]\!] &= [\![L_1 = L_2]\!]
\end{aligned}$$

Figure 2(e) presents the formulas for the four constraints from our example program. This translation is semantics preserving with respect to the standard notion of satisfiability $B \models F$ for Boolean formulas.

Theorem 4. *If $F = [\![\bar{D}]\!]$ then for all B, $B \models F$ if and only if $B \models \bar{D}$.*

Proof: By structural induction on each $D \in \bar{D}$.

In summary, our type inference algorithm proceeds as follows: Given a program P with locking variables, we generate from P a collection of constraints $\bar{C}$ over the locking variables; we extract a conditional assignment Y from $\bar{C}$ and generate Boolean constraints $\bar{D} = Y(\bar{C})$; and we generate a corresponding Boolean formula $F = [\![\bar{D}]\!]$. We use a propositional satisfiability solver to determine a truth assignment B for F, in which case we also have that $B \models \bar{D}$ by Theorem 4 and $(B(Y)) \models \bar{C}$ by Theorem 3, and therefore the explicitly-typed program $(B(Y))(P)$ is well-typed. Conversely, if the generated formula F is unsatisfiable, then there is no assignment A such that $A(P)$ is well-typed.

4 Implementation

We have implemented our inference algorithm in the Rcc/Sat checker, which supports the full Java programming language (although it does not currently detect race conditions on array accesses). Rcc/Sat takes as input an unannotated or partially-annotated program, where any typing annotations are provided in comments starting with "`#`", as in `/*# guarded_by y */`.

Rcc/Sat first adds a predetermined number of `ghost` parameters to all classes and methods lacking user-specified parameters. Next, for each unguarded field, Rcc/Sat adds the annotation `guarded_by` α, where α is fresh. Rcc/Sat also uses fresh locking variables to add any missing `requires` annotations and class and method instantiation parameters. Rcc/Sat then performs our type inference algorithm. If the generated constraints are satisfiable, then the satisfying assignment is used to generate an explicitly-typed version of the program. Section 4.2 outlines how we generate meaningful error messages when they are not.

4.1 Java Features

We handle additional features of the Java programming language as follows.

Scope constraints. Rcc/Sat permits lock expressions to be any *final* object references, including: (1) `this`; (2) ghost parameters; (3) `final` variables, static fields, and parameters; and (4) well-typed expressions of the form $e.f$, where e is a constant expression and f is a `final` field. This set may be infinite, and we heuristically limit it to expressions with at most two field accesses.

Inheritance, subtyping, and interfaces. Given the declaration

$$\texttt{class C}\langle\texttt{ghost a}_1\texttt{,...,a}_n\rangle\ \texttt{extends D}\langle\texttt{ghost b}_1\texttt{,...,b}_k\rangle\ \ \{\ \dots\ \}$$

we consider the type instantiation $\texttt{C}\langle l_{1..n}\rangle$ to be an immediate subtype of $\texttt{D}\langle m_{1..k}\rangle$ provided $m_i \equiv \texttt{b}_i[\texttt{a}_j := l_j\ ^{j\in 1..n}]$ for all $i \in 1..k$. The subtyping relation is the reflexive and transitive closure of this rule. The signature of an overriding method must match that of the overridden form, after applying the type parameter substitutions induced by the inheritance hierarchy. Interfaces are handled similarly.

Inner classes. Non-static inner classes may access the type parameters from the enclosing class and may declare their own parameters. Thus, the complete type for such a class is $\texttt{Outer}\langle l_{1..n}\rangle\texttt{.Inner}\langle m_{1..k}\rangle$.

Static fields, methods, and inner classes. Static members may not refer to the enclosing class' type parameters since static members are not associated with a specific instantiation of the class.

Thread objects. To allow `Thread` objects to store thread-local data in their fields, Rcc/Sat adds an implicit `final` field `tl_lock` to each `Thread` class. This field is analogous to (and replaces) the `ghost` parameter on the `run` method in RFJ2. It may guard other fields and is assumed to be held when `run` is invoked.

Escape mechanisms. We provide escapes from the RFJ2 type system through a "`no_warn`" annotation that suppresses the generation of constraints for a line of code. Also, since ghost parameters are erased at run time, the ghost parameters in typecasts of the form $(\texttt{C}\langle\texttt{a}\rangle)\texttt{x}$ are not checked dynamically.

4.2 Reporting Errors

We introduce two important improvements that enable the tool to pinpoint likely errors in the program when the generated constraints are unsatisfiable.

First, we change the algorithm to check each field declaration in a program separately, thereby enabling us to distinguish fields with potential races from those that are race-free. To check a single field, we generate the constraints as before, except that we only add field access constraints for accesses to the field of interest. The analysis is compositional in this manner because the presence or absence of races on one field is independence of races on other fields.

There is a possibility that the same locking variable will be assigned different values when checking different fields. If this occurs, we can compose the results of the separate checks together by introducing additional type parameters and renaming locking variables as necessary. For example, if a type instantiation $\texttt{C}\langle\alpha\rangle$ of class $\texttt{C}\langle\texttt{ghost x}\rangle$ becomes $\texttt{C}\langle\texttt{l1}\rangle$ when checking one field of `C` and $\texttt{C}\langle\texttt{l2}\rangle$ when checking another, we can change the class declaration to C⟨ghost x1, x2⟩, and instantiate it as C⟨l1,l2⟩ at the conflicting location.

Second, when there are race conditions on a field, it is often desirable to infer the most likely lock protecting it and then generate errors for locations where that lock is not held. For example, the following program is not well-typed:

```
1:  class C⟨ghost y⟩ {
2:    int c guarded_by α;
3:    void f1() requires y { c = 1; }
4:    void f2() requires y { c = 2; }
5:    void f3() requires this { c = 3; }
6:  }
```

Our tool produces the following diagnostic message at the likely error site:

```
C.java:5: Lock 'y' not held on access to 'c'.  Locks held: { this }.
```

To pinpoint likely error locations in this way, we express type inference as an optimization problem instead of a satisfiability problem. First, we add weights to some of the generated constraints, as follows. A constraint C with weight w is written as the weighted constraint $W = C|_w$.

$\alpha \in \{\mathtt{y}, \mathtt{this}, \mathtt{no\_lock}\}$	Scope constraint for c
$\alpha \in \{\mathtt{y}, \mathtt{this}\}\ |_2$	Requirement that c is guarded by a valid lock
$\alpha \in \{\mathtt{y}, \mathtt{no\_lock}\}\ |_1$	Access constraint for c from f1
$\alpha \in \{\mathtt{y}, \mathtt{no\_lock}\}\ |_1$	Access constraint for c from f2
$\alpha \in \{\mathtt{this}, \mathtt{no\_lock}\}\ |_1$	Access constraint for c from f3

These five constraints refer to no_lock, a lock name used in the checker to indicate that no reasonable guarding lock can be found for a field. Given constraints $\bar{C}$ and weighted constraints $\bar{W}$, we compute the optimal assignment A such that:

1. $A \models C$ for all $C \in \bar{C}$, and
2. the sum $\sum \{w \mid C|_w \in \bar{W} \ \wedge\ A \models C\}$ is maximized.

Note that we do not require all constraints in $\bar{W}$ be satisfied by A. For the constraints above, A is the assignment $\alpha = \mathtt{y}$, with a value of 4. We then generate error messages for all constraints in $\bar{W}$ that are not satisfied by A. The constraint $\alpha \in \{\mathtt{this}, \mathtt{no\_lock}\}\ |_1$ is not satisfied by the optimal assignment A, yielding the above error message. Conversely, if the optimal assignment A did not satisfy the constraint $\alpha \in \{\mathtt{y}, \mathtt{this}\}\ |_2$, then we would generate the error message:

```
C.java:2: No consistent guarding lock for field 'c'.
```

We have found that the heuristic of weighting declaration constraints 2–4 times more than field access constraints works well in practice.

We solve the constraint optimization problem for $\bar{W}$ and $\bar{C}$ by translating the constraints into a weighted MAX-SAT problem and solving it with the PBS tool [4]. The translation is similar to the case without weights. PBS and similar tools can find optimal assignments for formulas including up to 50–100 weighted clauses. Optimizing over a larger number of weighted clauses is currently computationally intractable. Thus, we still check one field at a time and only optimize over constraints generated by field accesses, placing all constraints for `requires` clauses and type equality in $\bar{C}$. If $\bar{C}$ is not satisfiable, we forego the optimization step and instead generate error messages for constraints in the smallest unsatisfiable core of $\bar{C}$, which we find with Chaff [20].

4.3 Improving Precision

Rcc/Sat implements a somewhat more expressive type system than that described in Section 2 to handle the synchronization patterns of large programs more effectively. In particular:

- Unreachable code is not type checked.
- Read-shared fields do not need guarding locks. A *read-shared* field is a field that is initialized while local to its creating thread, and subsequently shared in read-only mode among multiple threads.
- A field's protecting lock need not be held for accesses occurring when only a single thread exists or when the object has not escaped its creating thread.

Programs typically relax the core lock-based synchronization discipline along these lines. The checker currently uses quite basic implementations of rapid type analysis [5], escape analysis [6], and control-flow analysis for this step. Using more precise analyses would further improve our type inference algorithm.

5 Evaluation

We applied Rcc/Sat to benchmark programs including `elevator`, a discrete event simulator [27]; `tsp`, a Traveling Salesman Problem solver [27]; `sor`, a scientific computing program [27]; the `mtrt` ray-tracing program and `jbb` business objects simulator benchmarks [22]; and the `moldyn`, `montecarlo`, and `raytracer` benchmarks [19]. We ran these experiments on a 3.06GHz Pentium 4 processor with 2GB of memory, with Rcc/Sat configured to insert one `ghost` parameter on classes, interfaces, and instance methods and two parameters on static methods.

Table 1 shows, for each benchmark, the size in lines of code, the overall time for type inference, and the average type inference time per field. It also shows the size of the constraint problem generated, in number of constraints and the number of variables and clauses in the resulting Boolean formula, after conversion to CNF. The preliminary analyses described in Section 4.3 typically consumed less than 2% of the run time on the larger benchmarks.

The "Manual Annotations" column reflects the number of annotations manually inserted to guide the analysis. We added these few annotations to suppress warnings only in situations where immediately identifiable local properties ensured correctness. The manual annotations were inserted, for example, to delineate single-threaded parts of the program after joining all spawned threads; to explicitly instantiate classes in two places where the scope constraint generation

Program	Size (LOC)	Time (s)	Time/ Field (s)	Number of Constraints	Formula Size		Manual Annot.	Fields			
					vars	clauses		Total	read-shared	race-free	no guard
elevator	529	5.0	0.22	215	1,449	3,831	0	23	17	6	0
tsp	723	6.9	0.19	233	2,090	7,151	3	37	21	16	3
sor	687	4.5	0.15	130	562	1,205	1	29	22	7	0
raytracer	1,982	21.0	0.27	801	9,436	29,841	2	77	45	28	4
moldyn	1,408	12.6	0.12	904	4,011	10,036	3	107	57	44	6
montecarlo	3,674	20.7	0.19	1,097	9,003	25,974	1	110	68	42	0
mtrt	11,315	138.8	1.5	5,636	38,025	123,046	6	181	112	69	4
jbb	30,519	2,773.5	3.52	11,698	146,390	549,667	40	787	472	295	20

Table 1. Summary of test program performance.

heuristics did not consider the appropriate locks; and to identify thread-local object references not found by our escape analysis. In `jbb`, we also added annotations to suppress spurious race-condition warnings on roughly 25 fields with benign races. These fields were designed to be *write-protected* [12], meaning that a lock guarded write accesses, but read accesses were not synchronized. This idiom is unsafe if misused but permits synchronization-free accessor methods.

The last four columns show the total number of fields in the program, as well as their breakdown into read-shared fields, race-free fields, and fields for which no guarding lock was inferred. The analyses described in Section 4.3 reduced the number of fields without valid guards by 20%–75%, a significant percentage.

Rcc/Sat identified three fields in the `tsp` benchmark on which there are intentional races. On `raytracer`, Rcc/Sat identified a previously known race on a checksum field and reported spurious warnings on three fields. It also identified a known race on a counter in `mtrt`. The remaining warnings were spurious and could be eliminated by additional annotations or, in some cases, by improving the precision of the additional analyses of Section 4.3.

Overall, these results are quite promising. Manually inserting a small number of annotations enables Rcc/Sat to verify that the vast majority (92%–100%) of fields are race-free. These results show a substantial improvement over previous type inference algorithms for race-free type systems, such as Houdini/rcc.

6 Related Work

Boyapati and Rinard have defined a race-free type system with a notion of object ownership [7]. They include special owners to indicate thread-local data, thereby allowing a single class declaration to be used for both thread-local instances and shared instances, which motivated some of our refinements in RFJ2. They present an *intra*procedural algorithm to infer ownership parameters for class instantiations within a method. This simpler intraprocedural context yields equality constraints over lock variables, which can be efficiently solved using union-find. We believe it may be possible to extend our interprocedural type inference algorithm to accommodate ownership types. Grossman has developed a race-free type system for Cyclone, a statically safe variant of C [17]. Cyclone has a number of additional features, such as existential quantification and singleton types, and it remains to be seen how our techniques would apply in this setting.

The `requires` annotations used in our type system essentially constrain the *effects* that the method may produce. Thus, we are performing a form of effect reconstruction [26, 25], but our dependent types are not amenable to traditional effect reconstruction techniques. Similarly, the constraints of our type system do not exhibit the monotonicity properties that facilitate the polynomial time solvers used in other constraint-based analyses (see, for example, Aiken's survey [2]). Cardelli [8] was among the first to explore type checking for dependent types. Our dependent types are comparatively limited in expressive power, but the resulting type checking and type inference problems are decidable.

Eraser [21] is a tool for detecting race conditions in unannotated programs dynamically (though it may fail to detect certain errors because of insufficient test coverage). Agarwal and Stoller [1] present a dynamic type inference technique for the type system of Boyapati and Rinard. Their technique extracts locking information from a program trace and then performs a static analysis involving unique pointer analysis [3] and intraprocedural ownership inference [7] to construct annotations. These dynamic analyses complement our static approach, and it may be possible to leverage their results to facilitate type inference.

A common and significant problem with many type-inference techniques is the inability to construct meaningful error messages when inference fails (see, for example, [28, 29, 18]). An interesting contribution of our approach is that we view type inference as an optimization problem over a set of constraints that attempts to produce the most reasonable error messages for a program.

7 Conclusions

This paper contributes a new type inference algorithm for race-free type systems, which is based on reduction to propositional satisfiability. Our experimental results demonstrate that this approach works well in practice on benchmarks of up to 30,000 lines of code. Extending and evaluating this approach on significantly larger benchmarks remains an issue for future work. We also demonstrate extensions to facilitate reliable error reporting. We believe the resulting annotations and race-free guarantee provided by our type inference system have a wide range of applications in the analysis, validation, and verification of multithreaded programs. In particular, they provide valuable documentation to the programmer, they facilitate checking other program properties such as atomicity, and they can help reduce state explosion in model checkers.

Acknowledgments: We thank Peter Applegate for implementing parts of Rcc/Sat.

References

1. R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation*, pages 149–160, 2004.

2. A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2):79–111, 1999.
3. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 311–330, 2002.
4. F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. PBS: A backtrack-search pseudo-boolean solver and optimizer. In *Proceedings of the Symposium on the Theory and Applications of Satisfiability Testing*, pages 346–353, 2002.
5. D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 324–341, 1996.
6. J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 35–46, 1999.
7. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 56–69, 2001.
8. L. Cardelli. Typechecking dependent types and subtypes. In *Lecture notes in computer science on Foundations of logic and functional programming*, pages 45–57, 1988.
9. M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting Object Escape and Locking Information in Partial Order Reduction for Concurrent Object-Oriented Programs. *Formal Methods in System Design*, 2003.
10. C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 219–232, 2000.
11. C. Flanagan and S. N. Freund. Detecting Race Conditions in Large Programs. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 90–96, 2001.
12. C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 256–267, 2004.
13. C. Flanagan and S. Qadeer. Transactions for Software Model Checking. In *Proceedings of the Workshop on Software Model Checking*, pages 338–349, June 2003.
14. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 338–349, 2003.
15. C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 1–12, 2003.
16. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 171–183, 1998.
17. D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 13–25, 2003.
18. C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *European Symposium on Programming*, pages 284–301, 2003.
19. Java Grande Forum. Java Grande benchmark suite. Available from `http://www.javagrande.org/`, 2003.
20. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, 2001.

21. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
22. Standard Performance Evaluation Corporation. SPEC benchmarks. Available from `http://www.spec.org/`, 2003.
23. S. D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, Oct. 2002.
24. S. D. Stoller and E. Cohen. Optimistic Synchronization-Based State-Space Reduction. In H. Garavel and J. Hatcliff, editors, *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 489–504. Springer-Verlag, 2003.
25. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
26. M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 188–201, 1994.
27. C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 115–128, 2003.
28. M. Wand. Finding the source of type errors. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 38–43, 1986.
29. J. Yang, G. Michaelson, P. Trinder, and J. B. Wells. Improved type error reporting. In *Proceedings of the International Workshop on Implementation of Functional Languages*, pages 71–86, 2000.

A Type System

This appendix provides a complete definition of RFJ2. We first informally define a number of predicates. (See [16] for their precise definition.)

Predicate	Meaning
$ClassOnce(P)$	no class is declared twice in P
$FieldsOnce(P)$	no class contains two fields with the same name
$MethodsOncePerClass(P)$	no method name appears more than once per class

A typing environment is defined as: $E ::= \emptyset \mid E, t\ x \mid E, \texttt{ghost}\ x$

$$\boxed{P \vdash \bar{C}}$$

$$\frac{\begin{array}{c} ClassOnce(P) \quad FieldsOnce(P) \\ MethodsOncePerClass(P) \\ P = defn_{1..n}\ e \\ P \vdash defn_i\ \&\ \bar{C}_i \qquad \forall i \in 1..n \\ P; \texttt{ghost main\_lock}; \{\texttt{main\_lock}\} \vdash e : t\ \&\ \bar{C} \end{array}}{P \vdash \bar{C}_{1..n} \cup \bar{C}}$$

$$\boxed{P \vdash defn\ \&\ \bar{C}}$$

$$\frac{\begin{array}{c} garg_i = \texttt{ghost}\ x_i \qquad \forall i \in 1..n \\ E = garg_{1..n}, cn\langle x_{1..n}\rangle\ \texttt{this} \\ P; E \vdash field_i\ \&\ \bar{C}_i \qquad \forall i \in 1..j \\ P; E \vdash meth_i\ \&\ \bar{C}'_i \qquad \forall i \in 1..k \\ \bar{C} = \bar{C}_{1..j} \cup \bar{C}'_{1..k} \end{array}}{\begin{array}{l} P \vdash \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle \\ \qquad \{\ field_{1..j}\ \ meth_{1..k}\}\ \&\ \bar{C} \end{array}}$$

$$\boxed{P; E \vdash wf\ \&\ \bar{C}}$$

$$\frac{}{P; \emptyset \vdash wf\ \&\ \emptyset} \qquad \frac{\begin{array}{c} P; E \vdash t\ \&\ \bar{C} \\ x \notin dom(E) \end{array}}{P; E, t\ x \vdash wf\ \&\ \bar{C}} \qquad \frac{\begin{array}{c} P; E \vdash wf\ \&\ \bar{C} \\ x \notin dom(E) \end{array}}{P; E, \texttt{ghost}\ x \vdash wf\ \&\ \bar{C}}$$

$$\boxed{P; E \vdash t\ \&\ \bar{C}}$$

$$\frac{\begin{array}{c} P; E \vdash wf\ \&\ \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_i^{\ i\in 1..n}\rangle\ \ldots \in P \\ \bar{C}' = \bar{C} \cup \{l_i \in dom(E)\ ^{i\in 1..n}\} \end{array}}{P; E \vdash cn\langle l_{1..n}\rangle\ \&\ \bar{C}'}$$

$$\boxed{P; E \vdash field\ \&\ \bar{C}}$$

$$\frac{\begin{array}{c} P; E \vdash t\ \&\ \bar{C} \\ \bar{C}' = \bar{C} \cup \{l \in dom(E)\} \end{array}}{P; E \vdash t\ fn\ \texttt{guarded\_by}\ l\ \&\ \bar{C}'}$$

$$\boxed{P; E \vdash meth\ \&\ \bar{C}}$$

$$\frac{\begin{array}{c} garg_i = \texttt{ghost}\ x_i \qquad \forall i \in 1..n \\ E' = E, garg_{1..n}, arg_{1..d} \\ P; E'; s \vdash e : t\ \&\ \bar{C} \\ \bar{C}' = \bar{C} \cup \{s \subseteq dom(E')\} \\ s \text{ is either } \{y_1, \ldots, y_k\} \text{ or } \beta \end{array}}{P; E \vdash t\ mn\langle \texttt{ghost}\ x_{1..n}\rangle(arg_{1..d})\ \texttt{requires}\ s\ \{\ e\ \}\ \&\ \bar{C}'}$$

$$\boxed{P; E; s \vdash e : t\ \&\ \bar{C}}$$

$$\frac{P; E \vdash c\ \&\ \bar{C}}{P; E; s \vdash \texttt{null} : c\ \&\ \bar{C}} \qquad \frac{\begin{array}{c} P; E \vdash wf\ \&\ \bar{C} \\ E = E_1, t\ x, E_2 \end{array}}{P; E; s \vdash x : t\ \&\ \bar{C}} \qquad \frac{\begin{array}{c} P; E; s \vdash e : cn\langle l_{1..n}\rangle\ \&\ \bar{C} \\ P; E \vdash cn\langle l'_{1..n}\rangle\ \&\ \bar{C}' \\ \bar{C}'' = \bar{C} \cup \bar{C}' \cup \{l_i = l'_i\ ^{i\in 1..n}\} \end{array}}{P; E; s \vdash e : cn\langle l'_{1..n}\rangle\ \&\ \bar{C}''}$$

$$\frac{\begin{array}{c} y \text{ is fresh} \\ \theta = [x_j := l_j\ ^{j\in 1..n}, \texttt{this} := y] \\ P; E, cn\langle l_{1..n}\rangle\ y; s \vdash e_i : \theta(t_i)\ \&\ \bar{C}_i \qquad \forall i \in 1..k \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ field_{1..k}\ \ meth_{1..m}\ \} \in P \\ field_i = t_i\ fn_i\ \texttt{guarded\_by}\ l'_i \qquad \forall i \in 1..k \\ P; E \vdash cn\langle l_{1..n}\rangle\ \&\ \bar{C}' \\ \bar{C}'' = \bar{C}_{1..k} \cup \bar{C}' \cup \{l_i \in dom(E)\ ^{i\in 1..n}\} \end{array}}{P; E; s \vdash \texttt{new}\ cn\langle l_{1..n}\rangle(e_{1..k}) : cn\langle l_{1..n}\rangle\ \&\ \bar{C}''}$$

$$\frac{\begin{array}{c} P; E; s \vdash e : cn\langle l_{1..n}\rangle\ \&\ \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle \\ \qquad \{\ldots t\ fn\ \texttt{guarded\_by}\ l \ldots\} \in P \\ \theta = [\texttt{this} := e, x_j := l_j\ ^{j\in 1..n}] \\ P; E \vdash \theta(t)\ \&\ \bar{C}' \end{array}}{P; E; s \vdash e.fn : \theta(t)\ \&\ (\bar{C} \cup \bar{C}' \cup \{\theta(l) \in s\})}$$

$$\frac{\begin{array}{c} P; E; s \vdash e : cn\langle l_{1..n}\rangle\ \&\ \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle \\ \qquad \{\ldots t\ fn\ \texttt{guarded\_by}\ l \ldots\} \in P \\ \theta = [\texttt{this} := e, x_j := l_j\ ^{j\in 1..n}] \\ P; E \vdash e' : \theta(t)\ \&\ \bar{C}' \end{array}}{P; E; s \vdash e.fn = e' : \theta(t)\ \&\ (\bar{C} \cup \bar{C}' \cup \{\theta(l) \in s\})}$$

$$\frac{\begin{array}{c} P; E; s \vdash e_1 : t_1\ \&\ \bar{C}_1 \\ P; E, t\ x; s \vdash e_2 : t_2\ \&\ \bar{C}_2 \\ \theta = [x := e_1] \\ P; E \vdash \theta(t_2)\ \&\ \bar{C}_3 \\ C = (\bar{C}_1 \cup \bar{C}_2 \cup \bar{C}_3) \end{array}}{P; E; s \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \theta(t_2)\ \&\ C}$$

$$\frac{\begin{array}{c} P; E; s \vdash e : cn\langle l_{1..n} \rangle \ \& \ \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n} \rangle \ \{ \ldots t\ mn\langle \texttt{ghost}\ y_{1..k} \rangle \texttt{(} t_j\ z_j{}^{j \in 1..d} \texttt{)}\ \texttt{requires}\ s'\ \{\ e'\ \} \ldots \} \in P \\ \theta = [\texttt{this} := e, x_i := l_i{}^{i \in 1..n}, y_i := l'_i{}^{i \in 1..k}, z_i := e_i{}^{i \in 1..d}] \\ P; E; s \vdash e_j : \theta(t_j) \ \& \ \bar{C}_j \qquad \forall j \in 1..d \\ P; E \vdash \theta(t) \ \& \ \bar{C}' \\ \bar{C''} = \bar{C} \cup \bar{C}_{1..d} \cup \bar{C}' \cup \{\theta(s') \subseteq s\} \end{array}}{P; E; s \vdash e.mn\langle l'_{1..k} \rangle \texttt{(} e_{1..d} \texttt{)} : \theta(t) \ \& \ \bar{C''}}$$

$$\frac{\begin{array}{c} P; E; s \vdash x : t' \ \& \ \bar{C} \\ P; E; s \cup \{x\} \vdash e : t \ \& \ \bar{C}' \end{array}}{P; E; s \vdash \texttt{synchronized}\ x\ e : t \ \& \ (\bar{C} \cup \bar{C}')} \qquad \frac{\begin{array}{c} P; E; s \vdash e : cn\langle l_{1..n} \rangle \ \& \ \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n} \rangle \ \{ \ldots meth \ldots \} \in P \\ meth = t'\ \texttt{run}\langle \texttt{ghost tl\_lock} \rangle \texttt{()}\ \texttt{requires tl\_lock}\ \{\ e'\ \} \end{array}}{P; E; s \vdash e.\texttt{fork} : t \ \& \ \bar{C}}$$

B Proofs

This section presents the proofs for the lemmas and theorems presented in the paper (except for Theorem 2, whose proof is presented in Appendix C). We start with a technical lemma stating that a lock expression l is insensitive to assignment application if it does not contain any locking variables:

Lemma 3. *Given assignments A and A' and lock expression l, $A'(A(l)) = A(l)$.*

Proof By case analysis on the structure of l. □

We can extend this lemma to all other syntactic forms as well, and it allows us to prove the following lemma.

Restatement of Lemma 1 *Given an assignment A and constraints $\bar{C}$, $A \models \bar{C}$ iff $\models A(\bar{C})$.*

Proof Let A' be an arbitrary assignment. We prove the lemma by case analysis by each $C \in \bar{C}$:

- Suppose $C = (l_1 = l_2)$. Then

$$\begin{array}{ll} & A \models C \\ \text{iff} & A(l_1) = A(l_2) \\ \text{iff} & A'(A(l_1)) = A'(A(l_2)) \quad \text{by Lemma 3} \\ \text{iff} & A' \models (A(l_1) = A(l_2)) \\ \text{iff} & A' \models A(C) \end{array}$$

- The case where $C = (s_1 \subseteq s_2)$ follows by similar reasoning.

□

We next show that application of an assignment distributes over substitution.

Lemma 4. *For any assignment A and substitution θ,*

1. *$A(\theta(l)) = (A(\theta))(A(l))$ for all l*

2. $A(\theta(s)) = (A(\theta))(A(s))$ *for all* s
3. $A(\theta(t)) = (A(\theta))(A(t))$ *for all* t

Proof The first two cases follow from the definition of the application of an assignment to a lock expressions and lock set expressions, respectively. The last case holds since $t = cn\langle l_1, \ldots, l_n \rangle$. $\square$

The constraints generated for a program are related to the constraints generated for the same program after an assignment has been applied, as follows.

Restatement of Lemma 2 *If* $P \vdash \bar{C}$ *then* $A(P) \vdash A(\bar{C})$.

Proof We proceed by simultaneous induction over the following seven statements:

1. If $P \vdash \bar{C}$ then $A(P) \vdash A(\bar{C})$.
2. If $P \vdash \mathit{defn}\ \&\ \bar{C}$ then $A(P) \vdash \mathit{defn}\ \&\ A(\bar{C})$.
3. If $P; E \vdash t\ \&\ \bar{C}$ then $A(P); A(E) \vdash A(t)\ \&\ A(\bar{C})$.
4. If $P; E \vdash \mathit{wf}\ \&\ \bar{C}$ then $A(P); A(E) \vdash \mathit{wf}\ \&\ A(\bar{C})$.
5. If $P; E \vdash \mathit{field}\ \&\ \bar{C}$ then $A(P); A(E) \vdash A(\mathit{field})\ \&\ A(\bar{C})$.
6. If $P; E \vdash \mathit{meth}\ \&\ \bar{C}$ then $A(P); A(E) \vdash A(\mathit{meth})\ \&\ A(\bar{C})$.
7. If $P; E; s \vdash e : t\ \&\ \bar{C}$ then $A(P); A(E); A(s) \vdash A(e) : A(t)\ \&\ A(\bar{C})$.

Each part is shown by case analysis on the rules to derive that judgment form. We show a few representative cases.

Consider statement (1). The only way to derive $P \vdash \bar{C}$ is by rule [PROG]. Thus, it must be that:

$$
\begin{array}{l}
\mathit{ClassOnce}(P) \\
\mathit{FieldsOnce}(P) \\
\mathit{MethodsOncePerClass}(P) \\
P = \mathit{defn}_{1..n}\ e \\
P \vdash \mathit{defn}_i\ \&\ \bar{C}_i \qquad \forall i \in 1..n \\
P; \texttt{ghost main\_lock}; \{\texttt{main\_lock}\} \vdash e : t\ \&\ \bar{C}'
\end{array}
$$

where $\bar{C} = \bar{C}_{1..n} \cup \bar{C}'$. Since all field, class, and method names remain the same when constructing $A(P) = A(\mathit{defn}_{1..n})\ A(e)$, it is clear that

$$
\begin{array}{l}
\mathit{ClassOnce}(A(P)) \\
\mathit{FieldsOnce}(A(P)) \\
\mathit{MethodsOncePerClass}(A(P))
\end{array}
$$

By the inductive hypotheses:

$$
\begin{array}{l}
A(P) \vdash A(\mathit{defn}_i)\ \&\ A(\bar{C}_i) \qquad \forall i \in 1..n \\
A(P); A(\texttt{ghost main\_lock}); A(\{\texttt{main\_lock}\}) \vdash A(e) : A(t)\ \&\ A(\bar{C}')
\end{array}
$$

and we may conclude $A(P) \vdash A(\bar{C})$ by rule [PROG].

Consider statement (5). The only way to derive $P; E \vdash t\ fn$ `guarded_by` $l\ \&\ \bar{C}$ is by rule [FIELD]. Thus, it must be that $P; E \vdash t\ \&\ \bar{C}'$ and $\bar{C} = \bar{C}' \cup \{l \in dom(E)\}$. By the inductive hypothesis, $A(P); A(E) \vdash A(t)\ \&\ A(\bar{C})$. We may construct the set

$$A(\bar{C}) = A(\bar{C})' \cup \{A(l) \in A(dom(E))\}.$$

From this, [FIELD] allows us to conclude that $A(P); A(E) \vdash A(t\ fn\ \texttt{guarded\_by}\ l)\ \&\ A(\bar{C})$.

Consider statement (7). We proceed by case analysis on the rules to conclude $P; E; s \vdash e : t\ \&\ \bar{C}$, and show a representative case:

– [EXP REF]: In this case $e = e'.fn$, $t = \theta(t')$, and $\bar{C} = \bar{C}'' \cup \bar{C}' \cup \{\theta(l) \in s\}$ such that

$$\begin{array}{l} P; E; s \vdash e' : cn\langle l_{1..n}\rangle\ \&\ \bar{C}'' \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ldots t'\ fn\ \texttt{guarded\_by}\ l \ldots\} \in P \\ \theta = [\texttt{this} := e', x_j := l_j\ {}^{j \in 1..n}] \\ P; E \vdash \theta(t')\ \&\ \bar{C}' \end{array}$$

By the inductive hypotheses:

$$\begin{array}{l} A(P); A(E); A(s) \vdash A(e') : cn\langle A(l_{1..n})\rangle\ \&\ A(\bar{C}'') \\ A(P); A(E) \vdash A(\theta(t'))\ \&\ A(\bar{C}') \end{array}$$

Also, we know that

$$\begin{array}{l} \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ldots A(t')\ fn\ \texttt{guarded\_by}\ A(l) \ldots\} \in A(P) \\ A(\theta) = [\texttt{this} := A(e'), x_j := A(l_j)\ {}^{j \in 1..n}] \end{array}$$

Note that $A(\theta(t')) = (A(\theta))(A(t'))$ and $(A(\theta(l)) = (A(\theta))(A(l))$ from Lemma 4. From these, we have

$$\begin{array}{l} A(P); A(E) \vdash (A(\theta))(A(t'))\ \&\ A(\bar{C}') \\ A(\bar{C}) = A(\bar{C}'') \cup A(\bar{C}') \cup \{(A(\theta))(A(l)) \in A(s)\} \end{array}$$

and we can conclude that

$$A(P); A(E); A(s) \vdash A(e').fn : (A(\theta))(A(t'))\ \&\ A(\bar{C}'') \cup A(\bar{C}') \cup \{(A(\theta))(A(l)) \in A(s)\}$$

which is equivalent to

$$A(P); A(E); A(s) \vdash A(e').fn : A(\theta(t'))\ \&\ A(\bar{C})$$

□

Let $\varphi = [x_1 := L_1, \ldots, x_n := L_n]$ be a conditional substitution, that is, a substitution that contains conditional expressions.

Lemma 5. *For all boolean assignments B, conditional substitutions φ, conditional lock expressions L, and conditional lock set expressions S:*

1. $B(\varphi(L)) = B(\varphi(B(L)))$, *and*
2. $B(\varphi(S)) = B(\varphi(B(S)))$.

Proof

1. By structural induction and case analysis on L.
   - If $L = x$, then
$$\begin{aligned} & B(\varphi(B(L))) \\ = & B(\varphi(B(x))) \\ = & B(\varphi(x)) \\ = & B(\varphi(L)) \end{aligned}$$
   - If $L = b \,?\, L_1 : L_2$, then, assuming $B(b)$, have
$$\begin{aligned} & B(\varphi(B(L))) & \\ = & B(\varphi(B(b \,?\, L_1 : L_2))) & \\ = & B(\varphi(B(L_1))) & \\ = & B(\varphi(L_1)) & \text{by induction} \\ = & B(b \,?\, \varphi(L_1) : \varphi(L_2)) & \\ = & B(\varphi(L)) & \end{aligned}$$
     A similar argument holds if $\neg B(b)$.
2. The case for S is similar.

□

Lemma 6. *For all boolean assignments B, conditional assignments Y, lock expressions l, and lock set expressions s:*

1. $(B(Y))(l) = B(Y(l))$, *and*
2. $(B(Y))(s) = B(Y(s))$.

Proof

1. By structural induction and case analysis on l.
   - Suppose $l = x$. Then
$$(B(Y))(x) = x = B(Y(x))$$
   - Suppose $l = \alpha$. Then
$$(B(Y))(\alpha) = B(Y(\alpha))$$
     by the definition of $B(Y)$.

- Suppose $l = l' \cdot \theta$. Then

$$\begin{array}{lll} & (B(Y))(l) & \\ = & (B(Y))(l' \cdot \theta) & \\ = & (B(Y)(\theta))(B(Y)(l')) & \\ = & (B(Y)(\theta))(B(Y(l'))) & \text{by induction} \\ = & (B(Y)(\theta))(x) & \text{where } x = B(Y(l')) \\ = & (B(Y))(\theta(x)) & \text{by definition of the application of} \\ & & \text{an assignment } B(Y) \text{ to substitution } \theta \\ = & B(Y(\theta(x))) & \text{by induction} \end{array}$$

Similarly,

$$\begin{array}{lll} & B(Y(l)) & \\ = & B(Y(l' \cdot \theta)) & \\ = & B(Y(\theta)(Y(l'))) & \\ = & B(Y(\theta)(B(Y(l')))) & \text{by Lemma 5} \\ & & \text{replacing } \varphi \text{ and } L \text{ with } Y(\theta) \text{ and } Y(l') \\ = & B(Y(\theta)(x)) & \text{by definition of } x \text{ above} \\ = & B(Y(\theta(x))) & \text{by definition of the application of} \\ & & \text{a conditional assignment } Y \text{ to substitution } \theta \end{array}$$

Hence $(B(Y))(l' \cdot \theta) = B(Y(l' \cdot \theta))$.

2. The proof for s is similar.

□

Restatement of Theorem 3 *Suppose $\bar{D} = Y(\bar{C})$ and let B be a truth assignment. Then $B(Y) \models \bar{C}$ if and only if $B \models \bar{D}$.*

Proof By case analysis of each $C \in \bar{C}$.

- Suppose $C = (l_1 = l_2)$. Then

$$\begin{array}{lll} & (B(Y)) \models C & \\ \text{iff} & (B(Y)) \models l_1 = l_2 & \\ \text{iff} & (B(Y))(l_1) = (B(Y))(l_2) & \\ \text{iff} & B(Y(l_1)) = B(Y(l_2)) & \text{by Lemma 6} \\ \text{iff} & B \models Y(l_1) = Y(l_2) & \\ \text{iff} & B \models Y(l_1 = l_2) & \\ \text{iff} & B \models Y(C) & \end{array}$$

- The case where $C = (s_1 \subseteq s_2)$ follows by similar reasoning.

□

C Proof of Theorem 2

Restatement of Theorem 2 *For an arbitrary program P, the problem of finding an assignment A such that $A(P)$ is explicitly-typed and $A(P) \vdash \bar{C}$ and $\models \bar{C}$ is NP-complete.*

Proof This follows from the following Lemmas 7 and 8, which show that propositional satisfiability is reducible to type inference, and the fact that the type rules describe a syntax-directed type checker that can verify that an explicitly-annotated program is well-typed in polynomial time. $\square$

We first show how to translate a propositional satisfiability problem into a RFJ2 type inference problem, and then prove the correctness of this translation.

C.1 Translation

We start with a 3-SAT instance T with u variables $v_1, \ldots, v_u$ and n clauses:

$$\begin{array}{c} d_{1,1} \ \vee \ d_{1,2} \ \vee \ d_{1,3} \\ d_{2,1} \ \vee \ d_{2,2} \ \vee \ d_{2,3} \\ \vdots \\ d_{n,1} \ \vee \ d_{n,2} \ \vee \ d_{n,3} \end{array}$$

where each $d_{j,k}$ is either v_i or $\neg v_i$ for some i. We assume no clause refers to any variable v_i more than once.

Given such a 3-SAT instance T we generate a corresponding partially-annotated program P such that P is typeable if and only if T is satisfiable. Moreover, given an annotated version P' of P, we can generate a corresponding satisfying assignment for T.

The program P contains n classes, one for each clause, plus three additional classes `Object`, `F`, and `S`. Classes `Object` and `F` are trivial:

```
class Object { }
class F⟨ghost X⟩ { }
```

In addition, for each clause j, we define a class `C`j in P. We use the lock variables γ_j as place-holders for unknown locks.

```
class Cj⟨ghost a1, a2, a3⟩ {
    Object y guarded_by γi;
    Object z1 guarded_by a1;
    Object z2 guarded_by a2;
    Object z3 guarded_by a3;
    F⟨a1⟩ w1 guarded_by this;
    F⟨a2⟩ w2 guarded_by this;
    F⟨a3⟩ w3 guarded_by this;
}
```

Finally, P also contains the following class `S`. In this definition, we use $p_{j,k}$ to refer to `p`i if $d_{j,k} = v_i$, and use $p_{j,k}$ to refer to `n`i if $d_{j,k} = \neg v_i$. In addition, we use the notation

$$\texttt{synchronized(}e_1\texttt{, }e_2\texttt{)}\ \{\ e\ \}$$

as an abbreviation for

$$\texttt{synchronized(}e_1\texttt{)}\ \{\ \texttt{synchronized(}e_2\texttt{)}\ \{\ e\ \}\ \}$$

```
class S  {
  void m() requires ∅ {
    let pi = new Object()             // for i ∈ 1..u
        ni = new Object()

        cj = new Cj⟨α_{j,1}, α_{j,2}, α_{j,3}⟩ // for j ∈ 1..n
    in {
        // (a)
        synchronized(pi, ni)          // for j ∈ 1..n, k ∈ 1..3, i ∈ 1..u
             { cj.zk = null; }        // such that d_{j,k} ∈ {v_i, ¬v_i}

        // (b)
        synchronized(cj, cj')         // for j, j' ∈ 1..n, k, k' ∈ 1..3
             { cj.wk = cj'.wk'; }     // such that j ≠ j' and d_{j,k} and d_{j',k'}
                                      // refer to the same variable

        // (c)
        synchronized(p_{j,1},p_{j,2},p_{j,3})  // for j ∈ 1..n
             { cj.y = null; }
    }
  }
}
```

In essence, the code in part (a) enforces that the lock $\alpha_{j,k}$ corresponding to term $d_{j,k}$ is set to either `p`i or `n`i. It will be `p`i if v_i is true in the truth assignment we construct and `n`i if v_i is false. Part (b) ensures that all type variables corresponding to terms referring to the same variable are equal, i.e. the type variables for terms referring to v_i are either all `p`i or all `n`i. Part (c) ensures that at least one term in each clause of T is satisfied.

C.2 Correctness of the Translation

Suppose we can infer valid locks for the resulting program. We construct the satisfying assignment for the variables in the 3-SAT problem as follows. For all $i \in 1..u$,

$$v_i = \begin{cases} \textbf{true} & \text{if there exist } j,k \text{ such that } d_{j,k} \text{ is } v_i \text{ and } \gamma_j \text{ is } \texttt{a}k \\ \textbf{false} & \text{otherwise} \end{cases}$$

Lemma 7. *This truth assignment satisfies T.*

Proof Consider any clause j. It must be that $\gamma_j \in \{\texttt{a1}, \texttt{a2}, \texttt{a3}\}$ because only locks external to class `C`j are held when `c`j`.y` is accessed in the code for clause j in part (c). Suppose that $\gamma_j = \texttt{a}k$. There are two cases to consider:

1. Suppose $d_{j,k}$ is v_i. Then $v_i = \textbf{true}$ and clause j is satisfied.
2. Suppose $d_{j,k}$ is $\neg v_i$.
   (a) If $v_i = \textbf{false}$, then clause j is satisfied.
   (b) Suppose $v_i = \textbf{true}$. Then there exists j', k' such that

$$\begin{aligned} \gamma_j &= \texttt{a}k \ \wedge \ d_{j,k} = \neg v_i \\ \gamma_j' &= \texttt{a}k' \ \wedge \ d_{j',k'} = v_i \end{aligned}$$

These two lines imply that the following two statements appear in `S`:

```
synchronized(pi, ni) { cj.zk = null; }
synchronized(pi, ni) { cj'.zk = null; }
```

Since no other locks besides `p`i and `n`i are held when `c`j`.z`k and `c`j'`.z`k' are accessed, we have that

$$\begin{aligned} \alpha_{j,k} &\in \{\texttt{p}i, \texttt{n}i\} \\ \alpha_{j',k'} &\in \{\texttt{p}i, \texttt{n}i\} \end{aligned}$$

Also, the type of `c`j`.z`k is $\texttt{F}\langle\alpha_{j,k}\rangle$, and the the type of `c`j'`.z`k' is $\texttt{F}\langle\alpha_{j',k'}\rangle$. Since $d_{j,k}$ and $d_{j',k'}$ refer to the same variable, we have the following code

```
synchronized(cj, cj') { cj.wk = cj'.wk'; }
```

From this, it must be that $\texttt{F}\langle\alpha_{j,k}\rangle$ is the same type as $\texttt{F}\langle\alpha_{j',k'}\rangle$. Therefore, $\alpha_{j,k} = \alpha_{j',k'}$.
In addition, the following line must exist in `m` for clause j:

```
synchronized(p_{j,1},p_{j,2},p_{j,3}) { cj.y = null; }
```

Since the field `y` in class `C`j is guarded by γ_j and $\gamma_j = \texttt{a}k$, the lock $\alpha_{j,k}$ must be held when `c`j`.y` is accessed in the statement above. Since $d_{j,k} = \neg v_i$, then `n`i is in the lock set $\{p_{j,1}, p_{j,2}, p_{j,3}\}$ However, `p`i is not in this set since we assume no SAT clause contains both a variable and its negation. Therefore, $\alpha_{j,k} = \texttt{n}i$.
Similarly, the following line of code appears in `m` for clause j':

```
synchronized(p_{j',1},p_{j',2},p_{j',3}) { cj'.y = null; }
```

$$\texttt{synchronized(}p_{j',1}\texttt{,}p_{j',2}\texttt{,}p_{j',3}\texttt{) \{ c}j'\texttt{.y = null; \}}$$

The lock $\alpha_{j',k'}$ must be held when $\texttt{c}j'\texttt{.y}$ is accessed in this statement. Since $d_{j',k'} = v_i$, then $\texttt{p}i$ is in the set $\{ p_{j,1}, p_{j,2}, p_{j,3}\}$. However, $\texttt{n}i$ is not in this set since we assume no clause contains both a variable and its negation. Therefore, $\alpha_{j',k'} = \texttt{p}i$.
But, $\alpha_{j,k} = \alpha_{j',k'}$, and we have a contradiction. Thus, the case where $d_{j,k}$ is $\neg v_i$ and $v_i = \textbf{true}$ cannot happen.

□

Consider any satisfying truth assignment for T. We may construct an assignment to locking variables from this truth assignment that shows P is typeable as follows. For all $j \in 1..n$,

$$\gamma_i = \texttt{a}k \text{ such that } d_{j,k} \text{ is } \textbf{true} \text{ under the satisfying truth assignment.}$$

Such a k must exist for each j. For all j, k,

$$\alpha_{j,k} = \begin{cases} \texttt{p}i \text{ if } d_{j,k} \text{ refers to } v_i \text{ and } v_i = \textbf{true} \\ \texttt{n}i \text{ if } d_{j,k} \text{ refers to } v_i \text{ and } v_i = \textbf{false} \end{cases}$$

Lemma 8. *This assignment shows that P is typeable.*

Proof We consider the typing requirements for the method `m` in `S` and demonstrate that all `guarded_by` and type equality requirements are satisfied.

First, consider each statement of the form

$$\texttt{synchronized(p}i\texttt{, n}i\texttt{) \{ c}j\texttt{.z}k\texttt{ = null; \}}$$

where $d_{j,k} \in \{v_i, \neg v_i\}$. The lock $\alpha_{j,k}$ guards the field $\texttt{c}j\texttt{.z}k$ and must acquired before accessing $\texttt{c}j\texttt{.z}k$. If $v_i = \textbf{true}$, then $\alpha_{j,k} = \texttt{p}i$, and the field access is race free since the code synchronizes on $\texttt{p}i$. Similarly, if $v_i = \textbf{false}$, then $\alpha_{j,k} = \texttt{n}i$, and the field access is race free.

Next, consider each statement

$$\texttt{synchronized(c}j\texttt{, c}j'\texttt{) \{ c}j\texttt{.w}k\texttt{ = c}j'\texttt{.w}k'\texttt{; \}}$$

where $d_{j,k}$ and $d_{j',k'}$ refer to the same variable v_i. In this case, the typing requirements are that the locks guarding the fields $\texttt{c}j\texttt{.w}k$ and $\texttt{c}j'\texttt{.w}k'$ are held, and that the types of $\texttt{c}j\texttt{.w}k$ and $\texttt{c}j'\texttt{.w}k'$ are equal. The locks are clearly held since the code synchronizes on them. The type of $\texttt{c}j\texttt{.w}k$ is $\texttt{F}\langle\alpha_{j,k}\rangle$ and the type of $\texttt{c}j'\texttt{.w}k'$ is $\texttt{F}\langle\alpha_{j',k'}\rangle$. Given the construction of the substitution above, $\alpha_{j,k}$ and $\alpha_{j',k'}$ will either both be $\texttt{p}i$ or both be $\texttt{n}i$.

Finally, consider each statement

$$\texttt{synchronized(}p_{j,1}\texttt{,}p_{j,2}\texttt{,}p_{j,3}\texttt{) \{ c}j\texttt{.y = null; \}}$$

The lock γ_j guarding `cj.y` is `a`k, for some k such that $d_{j,k}$ is true under the truth assignment. Since `c`j is an instantiation for `C`j, with lock argument $\alpha_{j,k}$ for `a`k, the lock $\alpha_{j,k}$ must be held when `cj.y` is accessed. There are two cases:

1. $\alpha_{j,k} = \mathtt{p}i$: Then $d_{j,k}$ refers to v_i and $v_i = \textbf{true}$. Since $d_{j,k}$ must be true, $d_{j,k} = v_i$. Thus, $p_{j,k} = \mathtt{p}i$, and $\alpha_{j,k}$ is held inside the synchronized statement.
2. $\alpha_{j,k} = \mathtt{n}i$: Then $d_{j,k}$ refers to v_i and $v_i = \textbf{false}$. Since $d_{j,k}$ must be true, $d_{j,k} = \neg v_i$. Thus, $p_{j,k} = \mathtt{n}i$, and $\alpha_{j,k}$ is held inside the synchronized statement.

□

C.3 Example

Consider the SAT problem:

$$\begin{array}{l} v_1 \vee \neg v_2 \vee v_3 \\ \neg v_1 \vee v_2 \vee v_3 \end{array}$$

This SAT problem reduces to the type and effect inference problem for the program in Figure 3.

The following substitution for lock variables converts the above partially-typed program into a well-typed program. (Other substitutions also exist.)

$$\begin{array}{lll} \gamma_1 = \mathtt{a1} & \alpha_{1,1} = \mathtt{p1} & \alpha_{2,1} = \mathtt{p1} \\ \gamma_2 = \mathtt{a3} & \alpha_{1,2} = \mathtt{n2} & \alpha_{2,2} = \mathtt{n2} \\ & \alpha_{1,3} = \mathtt{p3} & \alpha_{2,3} = \mathtt{p3} \end{array}$$

From this substitution we generate the truth assignment where $v_1 = \textbf{true}$, $v_2 = \textbf{false}$, and $v_3 = \textbf{true}$. This truth assignment satisfies the original 3-SAT problem.

It is also straight-forward to construct substitutions for the type variables that make the program well-typed for any satisfying assignment of the 3-SAT problem.

```
class Object { }
class F⟨ghost X⟩ { }

class C1⟨ghost a1, a2, a3⟩ {
  Object y guarded_by γ1;
  Object z1 guarded_by a1;
  Object z2 guarded_by a2;
  Object z3 guarded_by a3;
  F⟨a1⟩ w1 guarded_by this;
  F⟨a2⟩ w2 guarded_by this;
  F⟨a3⟩ w3 guarded_by this;
}

class C2⟨ghost a1, a2, a3⟩ {
  Object y guarded_by γ2;
  Object z1 guarded_by a1;
  Object z2 guarded_by a2;
  Object z3 guarded_by a3;
  F⟨a1⟩ w1 guarded_by this;
  F⟨a2⟩ w2 guarded_by this;
  F⟨a3⟩ w3 guarded_by this;
}
```

```
class S {
  void m() requires ∅ {
    let p1 = new Object()
      n1 = new Object()
      p2 = new Object()
      n2 = new Object()
      p3 = new Object()
      n3 = new Object()
      c1 = new Cj⟨α1,1, α1,2, α1,3⟩
      c2 = new Cj⟨α2,1, α2,2, α2,3⟩
    in {
      synchronized(p1, n1)
         { c1.z1 = null; }
      synchronized(p2, n2)
         { c1.z2 = null; }
      synchronized(p3, n3)
         { c1.z3 = null; }
      synchronized(p1, n1)
         { c2.z1 = null; }
      synchronized(p2, n2)
         { c2.z2 = null; }
      synchronized(p3, n3)
         { c2.z3 = null; }

      synchronized(c1, c2)
         { c1.w1 = c2.w1; }
      synchronized(c1, c2)
         { c1.w2 = c2.w2; }
      synchronized(c1, c2)
         { c1.w3 = c2.w3; }

      synchronized(p1, n2, p3)
         { c1.y = null; }
      synchronized(n1, p2, p3)
         { c2.y = null; }
    }
  }
}
```

Fig. 3. Program corresponding to the example SAT instance.