Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 64 (2007) 140–165
Science of Computer Programming
www.elsevier.com/locate/scico

Type inference against races

Cormac Flanagan[a,*], Stephen N. Freund[b]

[a] *Computer Science Department, University of California at Santa Cruz, Santa Cruz, CA 95064, United States*
[b] *Computer Science Department, Williams College, Williamstown, MA 01267, United States*

Received 31 January 2005; received in revised form 15 October 2005; accepted 15 March 2006
Available online 29 September 2006

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.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Type systems; Type inference; Race conditions

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. If the underlying memory model is not sequentially consistent [1], then race conditions can cause surprising and counterintuitive behaviors. Typically, programmers attempt to avoid race conditions by adopting a programming discipline in which shared variables are protected by locks.

In a previous paper [2], 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. This type system incorporates *dependent types*, which allows the type of each field to specify a protecting lock for that field. The type system also allows class definitions to be parameterized by locks that protect the fields of the class, thus allowing different class instances to be protected by different locks. Parameterized classes are crucial for checking large programs with complex synchronization disciplines. Note that this notion of lock-parameterization is different from the generics supported in Java 5, which allow classes to be parameterized by types instead of 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

* Corresponding author.
E-mail address: cormac@cs.ucsc.edu (C. Flanagan).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.03.006

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.

In previous work along these lines, we developed Houdini/rcc [3], a type inference algorithm for `rccjava` that heuristically generates a large set of candidate type annotations and then iteratively removes all invalid annotations. If this process removes all the candidate protecting locks for a field, Houdini/rcc issues a warning that often reflects a race condition in the code. Unfortunately, Houdini/rcc cannot handle parameterized classes or methods, which limits its ability to check many synchronization idioms of real programs.

In the presence of parameterized classes (where different instances of a class are protected by different locks), the type inference problem for `rccjava` is much harder. Essentially, picking the right protecting lock for each field in the program is analogous to picking a Boolean value for each variable in a propositional satisfiability problem. This connection allows us to reduce any propositional satisfiability problem into a corresponding `rccjava` type inference problem, which implies that type inference for `rccjava` is NP-complete.

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 type 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 optimization 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; they facilitate checking other properties such as atomicity [5–7]; and they can help reduce state explosion in model checkers [8–11].

The presentation of our results proceeds as follows. The following section reviews our underlying type system, and shows how to reduce type inference to a constraint satisfaction problem. Section 3 solves these constraint systems by reduction to propositional satisfiability. Section 4 describes Rcc/Sat, our implementation of this algorithm, and Section 5 evaluates Rcc/Sat on a number of benchmark programs. Section 6 discusses related work, and we conclude with Section 7. The appendix contains proofs of various theorems and lemmas in the paper.

2. Types against races

2.1. The Language RFJ2

This section introduces RFJ2 (RaceFreeJava 2), an idealized multithreaded subset of Java with a type system that guarantees race freedom for well-typed programs. This type system extends our previous work on the `rccjava` type system [2], 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 Fig. 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.

$$
\begin{array}{rcll}
P & ::= & \mathit{defn}^*\ e & \text{(program)} \\
\mathit{defn} & ::= & \texttt{class}\ cn\langle\texttt{ghost}\ x^*\rangle\ \{\ \mathit{field}^*\ \mathit{meth}^*\ \} & \text{(class declaration)} \\
\mathit{field} & ::= & t\ fn\ \texttt{guarded\_by}\ l & \text{(field declaration)} \\
\mathit{meth} & ::= & t\ mn\langle\texttt{ghost}\ x^*\rangle(\mathit{arg}^*)\ \texttt{requires}\ s\ \{\ e\ \} & \text{(method declaration)} \\
\mathit{arg} & ::= & t\ x & \text{(argument declaration)} \\
t & ::= & cn\langle l^*\rangle & \text{(type)} \\
l & ::= & x & \text{(lock expression)} \\
s & ::= & \emptyset \mid \{l\} \mid s \cup s & \text{(lock set expression)} \\
 & & & \\
e, f & ::= & x \mid \texttt{null} \mid \texttt{new}_y\ t(e^*) \mid e.fn \mid e.fn\ \texttt{=}\ e & \text{(expressions)} \\
 & & \mid e.mn\langle l^*\rangle(e^*) \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e & \\
 & & \mid \texttt{synchronized}\ x\ e \mid e.\texttt{fork} &
\end{array}
$$

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

Fig. 1. The idealized language RFJ2.

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).[1] 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*, which 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 $x_1 \ldots x_n$ to which type annotations within the class body can refer. 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.

Since parameterized classes contain variables, our type system supports a (restricted) form of dependent types. We avoid the decidability limitations associated with dependent type systems by approximating semantic equality of variables by syntactic equality. This approximation has proven sufficient on all our benchmarks.

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

$$t\ m\langle\texttt{ghost}\ x_1, x_2\rangle(\texttt{Hashtable}\langle x_1, x_2\rangle\ y)\ \texttt{requires}\ x_1, x_2\ \{\ \ldots\ \}$$

which defines a method m that is parameterized by two locks x_1 and x_2, and which takes an argument of type $\texttt{Hashtable}\langle x_1, x_2\rangle$. A corresponding invocation $e.m\langle y_1, y_2\rangle(h)$ must supply two locks y_1 and y_2 that are currently held and an actual parameter h of type $\texttt{Hashtable}\langle y_1, y_2\rangle$.

Expressions include object allocation, field read and update, method invocation, and variable binding and reference. The object allocation expression $\texttt{new}_y\ t(e^*)$ includes a sequence e^* of expressions used to initialize the object fields. For technical reasons, the `new` keyword is subscripted by y, which is a ghost variable bound to the object being created while evaluating the field initialization expressions. This enables the types of the initialization expressions to refer to the new object. We omit this binding from examples when it is not needed.

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 method `run`. The fork operation spawns a new thread that calls that `run` method. The RFJ2 type system leverages parameterized methods

[1] For simplicity, RFJ2 does not include `final` and `volatile` fields, but this extension is straightforward.

to reason about thread-local data. (This approach replaces the escape analysis embedded in our earlier type system [2] and provides an alternative to the ownership types of [12].) Specifically, the `run` method of each forked thread takes a `ghost` parameter `thread_lock` denoting a lock that is always held by that thread:

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

Intuitively, the underlying run-time system creates and acquires this thread 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 [2]. However, the need for programmer-inserted annotations limits its applicability to large, legacy systems, thus motivating the development of type inference techniques for race-free type systems.

We next describe our type inference system for RFJ2. We introduce *lock variables* α and *lock set variables* β, collectively referred to as *locking variables*. 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. Locking variables may be mentioned in type annotations, as in `guarded_by` α, `requires` β, or $cn\langle\alpha_1, \alpha_2\rangle$. As an example, Fig. 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.

This type inference problem for RFJ2 is NP-complete, due in part to the complexities of dealing with parameterized classes. To illustrate this difficulty, 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 denotes the set of locks in a method's requires clause, then the application of a substitution θ to s yields the delayed substitution $s \cdot \theta$.

Since the type rules reason about delayed substitutions, we include these delayed substitutions in the language by extending the syntax of lock and lock set expressions, but we require that substitutions do not appear in source programs.

$$\begin{array}{lll} l & ::= & x \mid \alpha \mid l \cdot \theta & \text{(lock expression)} \\ s & ::= & \emptyset \mid \{l\} \mid s \cup s \mid \beta \mid s \cdot \theta & \text{(lock set expression)} \\ \theta & ::= & [x_1 := l_1, \ldots, x_n := l_n] & \text{(substitution)} \end{array}$$

The following examples illustrate substitutions on various syntactic entities. (We do not present an exhaustive definition, since the remaining cases are straightforward.)

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

(a) Example Program `Ref`

```
class Lock⟨⟩ { }
class Ref⟨ghost x⟩ {
  int y guarded_by α1
  boolean less(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.less(r2);
}
```

(b) Constraints

$$
\begin{array}{rcll}
\alpha_1 & \in & \{\,\texttt{this}, \texttt{x}\,\} & \text{declaration of } \texttt{y} \\
\alpha_2 & \in & \{\,\texttt{this}, \texttt{x}\,\} & \text{declaration of } \texttt{less} \\
\beta & \subseteq & \{\,\texttt{this}, \texttt{x}, \texttt{o}\,\} & \text{declaration of } \texttt{less} \\
\alpha_3 & \in & \{\,\texttt{lock}\,\} & \text{first } \texttt{new} \text{ expression} \\
\alpha_4 & \in & \{\,\texttt{lock}, \texttt{r1}\,\} & \text{second } \texttt{new} \text{ expression} \\
 & & & \\
\alpha_1 & \in & \beta & \text{access to } \texttt{this.y} \\
\alpha_1[\texttt{this} := \texttt{o}, \texttt{x} := \alpha_2] & \in & \beta & \text{access to } \texttt{o.y} \\
\beta[\texttt{this} := \texttt{r1}, \texttt{x} := \alpha_3, \texttt{o} := \texttt{r2}] & \subseteq & \{\texttt{lock}\} & \text{requires clause for call} \\
\alpha_2[\texttt{this} := \texttt{r1}, \texttt{x} := \alpha_3, \texttt{o} := \texttt{r2}] & = & \alpha_4 & \text{parameter type eq call}
\end{array}
$$

(c) Conditional Assignment

$$
\begin{array}{rcll}
Y(\alpha_1) & = & (b_1?\texttt{this} : \texttt{x}) & \text{declaration of } \texttt{y} \\
Y(\alpha_2) & = & (b_2?\texttt{this} : \texttt{x}) & \text{declaration of } \texttt{less} \\
Y(\beta) & = & (b_4?\texttt{this} : \emptyset) \cup (b_5?\texttt{x} : \emptyset) \cup (b_6?\texttt{o} : \emptyset) & \text{declaration of } \texttt{less} \\
Y(\alpha_3) & = & \texttt{lock} & \text{first } \texttt{new} \text{ expression} \\
Y(\alpha_4) & = & (b_3?\texttt{lock} : \texttt{r1}) & \text{second } \texttt{new} \text{ expression}
\end{array}
$$

(d) Boolean Constraints

$$
\begin{array}{rcll}
(b_1?\texttt{this} : \texttt{x}) & \in & (b_4?\texttt{this} : \emptyset) \cup (b_5?\texttt{x} : \emptyset) \cup (b_6?\texttt{o} : \emptyset) & \text{access to } \texttt{this.y} \\
(b_1?\texttt{o} : (b_2?\texttt{this} : \texttt{x})) & \in & (b_4?\texttt{this} : \emptyset) \cup (b_5?\texttt{x} : \emptyset) \cup (b_6?\texttt{o} : \emptyset) & \text{access to } \texttt{o.y} \\
(b_4?\texttt{r1} : \emptyset) \cup (b_5?\texttt{lock} : \emptyset) \cup (b_6?\texttt{r2} : \emptyset) & \subseteq & \{\texttt{lock}\} & \text{requires clause for call} \\
(b_2?\texttt{r1} : \texttt{lock}) & = & (b_3?\texttt{lock} : \texttt{r1}) & \text{parameter type eq call}
\end{array}
$$

(e) Boolean Formula

$$
\begin{array}{rll}
 & [(b_1 \wedge b_4) \vee (\neg b_1 \wedge b_5)] & \text{access to } \texttt{this.y} \\
\wedge & [(b_1 \wedge b_6) \vee (\neg b_1 \wedge ((b_2 \wedge b_4) \vee (\neg b_2 \wedge b_5)))] & \text{access to } \texttt{o.y} \\
\wedge & [\neg b_4 \wedge \neg b_6] & \text{requires clause for call} \\
\wedge & [(b_2 \wedge \neg b_3) \vee (\neg b_2 \wedge b_3)] & \text{parameter type eq call}
\end{array}
$$

Fig. 2. Example program and type inference constraints.

2.3. Type rules

The core of the RFJ2 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.

A typing environment can include bindings for both regular and ghost variables:

$$E ::= \emptyset \mid E, t\ x \mid E, \texttt{ghost}\ x$$

Our constraint language includes equality constraints between lock expressions and containment constraints between lock set expressions:

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

The complete set of type rule for expressions appears in Fig. 3, and auxiliary judgments and rules are presented in Fig. 4. Most of the type rules are straightforward, and we briefly describe the more interesting aspects of each rule.

The rule [EXP NULL] assigns any well-formed type to the term `null`. The auxiliary judgment $P; E \vdash t \;\&\; \bar{C}$ checks that the type t is well-formed and that it only mentions valid lock names. For example, if t is $c\langle l \rangle$, then the generated constraint set $\bar{C}$ would include the requirement that $l \in dom(E)$, which is an abbreviation for the constraint $\{l\} \subseteq dom(E)$.

The rule [EXP VAR] extracts the type of a variable from a well-formed environment. An environment is well-formed, written $P \vdash E \;\&\; \bar{C}$, if all types appearing in E are well-formed. The rule [EXP SYNC] for $\texttt{synchronized}\ x\ e$ type checks e with an extended lock set that includes x, since the lock x is held when evaluating e.

The rule [EXP REF] for a field reference $e.fn$ checks that e has some type $cn\langle l_{1..n} \rangle$ and that cn has a field fn of type t, guarded by lock l. 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`, none 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 $\theta(t)$, which must be well-formed. The rule [EXP ASSIGN] is the analogous rule for field update.

The rule [EXP INVOKE] for a method invocation expression $e.mn\langle l'_{1..k} \rangle(e_{1..d})$ is similar to field access, but slightly more complex due to method parameters and ghost parameters. The rule checks that e has some type $cn\langle l_{1..n} \rangle$ and that cn includes a matching method declaration

$$t\ mn\langle \texttt{ghost}\ y_{1..k} \rangle (t_j\ z_j^{\ j \in 1..d})\ \texttt{requires}\ s'\ \{\ e'\ \}$$

The rule then constructs the substitution

$$\theta = [\texttt{this} := e, x_i := l_i^{\ i \in 1..n}, y_i := l'^{\ i \in 1..k}_i, z_i := e_i^{\ i \in 1..d}]$$

which substitutes

- the receiver's name e for `this`;
- the lock expressions $l_{1..n}$ in the receiver's type for the class' ghost parameters $x_{1..n}$;
- the actual lock expression arguments $l'_{1..k}$ for the method's ghost parameters $y_{1..k}$; and
- the actual method arguments $e_{1..d}$ for the method's formal parameters $z_{1..d}$.

Each method argument e_j must have type $\theta(t_j)$, and the return type $\theta(t)$ must be well-formed. In addition, the constraint $\theta(s') \subseteq s$ ensures that all locks specified in the `requires` set s' are held at this call site.

The rule [EXP NEW] for an object creation expression $\texttt{new}_y\ cn\langle l_{1..n} \rangle(e_{1..k})$ first retrieves the corresponding class declaration

$$\texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n} \rangle\ \{ field_{1..k}\ meth_{1..m}\ \}$$

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

[EXP NULL]

$$\frac{P; E \vdash t \,\&\, \bar{C}}{P; E; s \vdash \texttt{null} : t \,\&\, \bar{C}}$$

[EXP VAR]

$$\frac{P \vdash E \,\&\, \bar{C} \qquad E = E_1, t\ x, E_2}{P; E; s \vdash x : t \,\&\, \bar{C}}$$

[EXP SYNC]

$$\frac{P; E; s \vdash x : t' \,\&\, \bar{C} \qquad P; E; s \cup \{x\} \vdash e : t \,\&\, \bar{C}'}{P; E; s \vdash \texttt{synchronized}\ x\ e : t \,\&\, (\bar{C} \cup \bar{C}')}$$

[EXP REF]

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

[EXP ASSIGN]

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

[EXP INVOKE]

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

[EXP NEW]

$$\frac{\begin{array}{c} \theta = [x_j := l_j\ ^{j\in 1..n}, \texttt{this} := y] \\ P; E, \texttt{ghost}\ y; s \vdash e_i : \theta(t_i) \,\&\, \bar{C}_i \qquad \forall i \in 1..k \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ field_{1..k}\ meth_{1..m}\ \} \in P \\ field_i = t_i\ fn_i\ \texttt{guarded\_by}\ l'_i \qquad \forall i \in 1..k \\ P; E \vdash cn\langle l_{1..n}\rangle \,\&\, \bar{C}' \\ \bar{C}'' = \bar{C}_{1..k} \cup \bar{C}' \cup \{l_1 \in dom(E), \ldots, l_n \in dom(E)\} \end{array}}{P; E; s \vdash \texttt{new}_y\ cn\langle l_{1..n}\rangle (e_{1..k}) : cn\langle l_{1..n}\rangle \,\&\, \bar{C}''}$$

[EXP LET]

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

[EXP FORK]

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

[EXP TYPE EQ]

$$\frac{\begin{array}{c} P; E; s \vdash e : cn\langle l_{1..n}\rangle \,\&\, \bar{C} \\ P; E \vdash cn\langle l'_{1..n}\rangle \,\&\, \bar{C}' \\ \bar{C}'' = \bar{C} \cup \bar{C}' \cup \{l_1 = l'_1, \ldots, l_n = l'_n\} \end{array}}{P; E; s \vdash e : cn\langle l'_{1..n}\rangle \,\&\, \bar{C}''}$$

Fig. 3. Type rules, part I.

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

[TYPE C]

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

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

[ENV EMPTY]

$$\frac{}{P \vdash \emptyset \;\&\; \emptyset}$$

[ENV X]

$$\frac{P; E \vdash t \;\&\; \bar{C} \qquad x \notin dom(E)}{P \vdash E, t\ x \;\&\; \bar{C}}$$

[ENV GHOST]

$$\frac{P \vdash E \;\&\; \bar{C} \qquad x \notin dom(E)}{P \vdash E, \texttt{ghost}\ x \;\&\; \bar{C}}$$

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

[METHOD]

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

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

[FIELD]

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

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

[CLASS]

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

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

[PROG]

$$\frac{\begin{array}{c} \text{no class is declared twice in } P \\ \text{no field name appears more than once per class} \\ \text{no method name appears more than once per class} \\ P = defn_{1..n}\ e \\ P \vdash defn_i \;\&\; \bar{C}_i \qquad \forall i \in 1..n \\ P; \texttt{ghost}\ \texttt{main\_lock}; \{\texttt{main\_lock}\} \vdash e : t \;\&\; \bar{C} \end{array}}{P \vdash \bar{C}_{1..n} \cup \bar{C}}$$

Fig. 4. Type rules, part II.

The substitution $\theta = [x_j := l_j^{\ j\in 1..n}, \texttt{this} := y]$ then replaces the ghost parameters to cn with the actual arguments for the given instantiation, and it replaces occurrences of the self-reference `this` by y. In effect, this rule uses the name y as a placeholder for the object that is about to be constructed. The rule checks that each field initialization expression e_i has the appropriate type $\theta(t_i)$ and that each lock $l_{1..n}$ is in scope.

The rule [EXP LET] for `let` x `=` e_1 `in` e_2 checks the body e_2 in an environment extended with a binding for x. Note that t_2, the type of e_2, may refer to x. Thus, the rule assigns to the whole `let` expression the type $t_2[x := e_1]$ in order to prevent x from escaping its scope. The rule [EXP FORK] for e.`fork` ensures that e contains a `run` method with the signature:

$$t'\ \texttt{run}\langle \texttt{ghost}\ \texttt{thread\_lock}\rangle\texttt{()}\ \texttt{requires}\ \texttt{thread\_lock}\ \{\ e'\ \}$$

This signature contains a special ghost parameter named `thread_lock`. This lock is held throughout the entire lifetime of the new thread, and so may be used to protect data local to this thread.

Fig. 4 includes judgments and rules to determine whether types, methods, fields, environments, and classes are well-formed. The rule [METHOD] states that a method is well-formed if its body has the declared return type when checked in an environment extended with the method's ghost and normal parameters. The `requires` clause must also be either a set of valid lock names $\{y_1, \ldots, y_k\}$ or a lock set variable β. The rule [FIELD] states that a field is

well-formed if its type is well-formed and its protecting lock is a valid lock name. The rule [CLASS] ensures that all fields and methods of a class definition are well-formed.

The rule [PROG] 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 Fig. 2(a) yields the constraints shown in Fig. 2(b). (We ignore `main_lock` in this example for simplicity.)

In previous work, we proved the soundness of a somewhat more complex type system for preventing race conditions [13]. The soundness of RFJ2 could be established via a similar argument, but is omitted here for brevity.

2.4. Constraint satisfiability

We next address the question of how to solve the generated constraints $\bar{C}$ over the locking variables. An *assignment*

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

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

$$\begin{array}{rcl} A : l & \to & \mathit{Var} \\ A(x) & = & x \\ A(l \cdot \theta) & = & A(\theta)(A(l)) \end{array} \qquad \begin{array}{rcl} A : s & \to & 2^{\mathit{Var}} \\ A(\emptyset) & = & \emptyset \\ A(\{l\}) & = & \{A(l)\} \\ A(s_1 \cup s_2) & = & A(s_1) \cup A(s_2) \\ A(s \cdot \theta) & = & A(\theta)(A(s)) \end{array}$$

$$\begin{array}{rcl} & & A : \theta \to \theta \\ A([x_1 := l_1, \ldots, x_n := l_n]) & = & [x_1 := A(l_1), \ldots, x_n := A(l_n)] \end{array}$$

We 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:

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

If $A \models C$ for all $C \in \bar{C}$ then 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 Fig. 2(b) for the program `Ref` are satisfied by the assignment:

$$\begin{array}{l} \alpha_1 = \alpha_2 = \mathtt{x} \\ \alpha_3 = \alpha_4 = \mathtt{lock} \\ \beta = \{\mathtt{x}\} \end{array}$$

A program 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 lemmas show that the explicitly typed program $A(P)$ yields the constraint set $A(\bar{C})$, and furthermore, that this constraint set $A(\bar{C})$ is valid. Hence, the explicitly typed program $A(P)$ is well-typed.

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

Proof. See Appendix A. □

Lemma 2. *Given an assignment A and constraints $\bar{C}$, $A \models \bar{C}$ iff $\models A(\bar{C})$.*

Proof. See Appendix A. □

Theorem 3. *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, the generated constraints $\bar{C}$ do not contain locking variables, and so 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.

Theorem 4. *For an arbitrary* RFJ2 *program P, the problem of finding an assignment A such that $A(P)$ is explicitly typed and well-typed is NP-complete.*

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

Despite this worst-case complexity result, the next section presents a type inference algorithm for RFJ2 that has proven effective in practice.

3. Solving constraint systems

3.1. Generating Boolean constraints

The type rules generate constraints over the various locking variables mentioned in the program. For each lock variable α, these constraints include a *scope constraint* $\alpha \in \{x_1, \ldots, x_n\}$ that constrains α to be one of the variables in scope where α is mentioned. A similar constraint $\beta \subseteq \{x_1, \ldots, x_n\}$ is introduced for each lock set variable β. These scope constraints allow us to use Boolean variables to encode the possible choices for each locking variable. A *conditional lock expression* L is a either a variable x or a conditional expression $b?\,L_1 : L_2$, which denotes L_1 if the Boolean variable b is true, and denotes L_2 otherwise. A *conditional lock set expression* S is either the empty set, a set union, a singleton set, or a conditional expression $b?\,S_1 : S_2$. A *Boolean constraint* D is either an equality constraint between conditional lock expressions, or a subset constraint between conditional lock set expressions.

$$
\begin{array}{lll}
b & \in & \mathit{BoolVar} \qquad \text{(Boolean variables)} \\
L & ::= & x \mid b?\,L_1 : L_2 \qquad \text{(conditional lock expressions)} \\
S & ::= & \emptyset \mid \{L\} \mid S \cup S \mid b?\,S_1 : S_2 \qquad \text{(conditional lock set expressions)} \\
D & ::= & S \subseteq S \mid L = L \qquad \text{(Boolean constraints)}
\end{array}
$$

We now describe how to translate the constraints $\bar{C}$ into an equivalent set of Boolean constraints $\bar{D}$. From the scope constraints in $\bar{C}$, we generate a *conditional assignment*

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

that encodes the possible choices for each locking variable using fresh Boolean variables. For example, the scope constraints $\alpha \in \{x_1, \ldots, x_n\}$ and $\beta \subseteq \{y_1, \ldots, y_m\}$ yield[2]:

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

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

[2] We could encode the choice for the first constraint as a decision tree with only $\log n$ Boolean variables. Our implementation uses this alternative to minimize the number of variables introduced during translation.

$$
\begin{array}{rcl}
Y : l \to L & & Y : s \to S \\
Y(x) & = & x \qquad\qquad Y(\emptyset) = \emptyset \\
Y(l \cdot \theta) & = & Y(\theta)(Y(l)) \qquad Y(\{l\}) = \{Y(l)\} \\
 & & Y(s_1 \cup s_2) = Y(s_1) \cup Y(s_2) \\
Y : C \to D & & Y(s \cdot \theta) = Y(\theta)(Y(s)) \\
Y(s_1 \subseteq s_2) & = & Y(s_1) \subseteq Y(s_2) \\
Y(l_1 = l_2) & = & Y(l_1) = Y(l_2) \qquad Y([x_1 := l_1, \ldots, x_n := l_n]) = \\
 & & [x_1 := Y(l_1), \ldots, x_n := Y(l_n)]
\end{array}
$$

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

3.2. Satisfiability of Boolean constraints

A *truth assignment*

$$B : \mathit{BoolVar} \rightharpoonup \mathit{Boolean}$$

assigns truth values to Boolean variables. We extend truth assignments from Boolean variables to conditional lock expressions (L) and conditional lock set expressions (S) as follows.

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

A truth assignment B satisfies a Boolean constraint D as follows:

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

Similarly, B satisfies a set of Boolean constraints $\bar{D}$ if $B \models D$ for each $D \in \bar{D}$. For example, the Boolean constraints of Fig. 2(d) are satisfied by the truth assignment:

$$
\begin{array}{l}
B(b_1) = B(b_2) = B(b_4) = B(b_6) = \textbf{false} \\
B(b_3) = B(b_5) = \textbf{true}
\end{array}
$$

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

$$
\begin{array}{rcl}
B(Y) & : & (\mathit{LockVar} \to \mathit{Var}) \cup (\mathit{LockSetVar} \to 2^{\mathit{Var}}) \\
B(Y)(\alpha) & = & B(Y(\alpha)) \\
B(Y)(\beta) & = & B(Y(\beta))
\end{array}
$$

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

Theorem 5. *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 A. □

3.3. 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 of the form:

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

Such Boolean formulas can be solved by a standard propositional satisfiability solver such as Chaff [14]. This translation from Boolean constraints into Boolean formulas is as follows::

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

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

To illustrate this translation, Fig. 2(e) presents the Boolean 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 6. *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, given a program P with locking variables, our type inference algorithm:

(1) generates from P a collection of constraints $\bar{C}$ over the locking variables;
(2) extracts a conditional assignment Y from $\bar{C}$;
(3) generates Boolean constraints $\bar{D} = Y(\bar{C})$;
(4) generates a corresponding Boolean formula $F = [\![\bar{D}]\!]$;
(5) uses a propositional satisfiability solver to determine a truth assignment B for F (in the case where F is satisfiable); and
(6) generates the explicitly typed program $A(P)$, where the assignment A is given by $A = B(Y)$.

That the generated program $A(P)$ is well-typed follows from Theorem 3, since $B \models \bar{D}$ by Theorem 6 and $A \models \bar{C}$ by Theorem 5. 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 type inference algorithm in the Rcc/Sat checker. This checker supports the full Java programming language, although it does not currently detect race conditions on array accesses. It 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 proceeds by first adding a predetermined number of `ghost` parameters to classes and methods lacking user-specified parameters. Next, for each unguarded, non-volatile field, Rcc/Sat adds the annotation `guarded_by` α, where α is fresh. Rcc/Sat also adds any missing `requires` annotations and ghost parameters, again using fresh locking variables. Rcc/Sat then applies 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. If the generated constraints are not satisfiable, Rcc/Sat reports the potential synchronization problems to the user, as described in Section 4.2 below.

4.1. Java features

The RFJ2 type system extends in a fairly natural manner to the full Java language. In this section, we highlight a few of the more interesting and substantial extensions.

Scope constraints. In the RFJ2 language, the only valid lock expressions are variables in scope. In real Java programs, however, more complex lock expressions are often used. For example, the following declaration uses the lock expression `this.lock`:

```
class Counter {
    final Object lock = new Object();
    int num /*# guarded_by this.lock */;
    ...
}
```

If Rcc/Sat only considered variable names when constructing the scope constraints, we could not infer the `guarded_by` annotation for this program, since `this.lock` would not be included in the scope constraint for the `num` field. To handle cases like this, we extend the definition of valid lock names to be any *final* object references. This set of expressions includes:

(1) `this`;
(2) ghost parameters;
(3) `final` variables;
(4) `final` static fields; and
(5) field dereferences of the form $e.f$, where e is a final object reference expression and f is a `final` field.

Rcc/Sat generates the set of expressions matching this characterization while constructing scope constraints. Since this set may be infinite, we heuristically limit it to expressions with at most two field accesses.

Inheritance, subtyping, and interfaces. Given the declaration

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

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 and methods. Static members may not refer to type parameters of the enclosing class 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 `thread_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 it is assumed to be held when `run` is invoked. A potential for unsoundness exists if multiple threads invoke the `run` method on the same object concurrently, although this unsoundness could be removed with additional static analyses or dynamic checks.

Escape mechanisms. We provide an escape 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 unchecked, as in C, rather than dynamically checked, as in Java. The soundness guarantees of our type system do not apply when these mechanisms are used.

4.2. Reporting errors

We introduce two important improvements to our type inference algorithm that enable Rcc/Sat to pinpoint likely errors in the program when the generated constraints are unsatisfiable.

Identifying field declarations with race conditions. Rcc/Sat checks each field declaration separately, in order to identify fields with potential race conditions. To check a particular field declaration, Rcc/Sat generates the constraints as before, except that it only adds field access constraints for accesses to this field. If the resulting constraints are satisfiable, then this field is race-free.

There is a possibility that a type instantiation C⟨α⟩ of a class C⟨ghost x⟩ will yield different solutions, say C⟨l1⟩ and C⟨l2⟩, when checking different fields. In this case, we can compose the results of the separate analyses by introducing additional type parameters in to the class declaration, such as C⟨ghost x1, x2⟩, and instantiating the class as C⟨l1,l2⟩ at the conflicting location.

Identifying field accesses with race conditions. When there are race conditions on a field, it is often desirable to infer the most likely lock protecting that field and to generate errors for accesses where that lock is not held. Consider, for example, the following program, which has a race condition on the field c:

```
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 could produce an error indicating that no valid guard for the field c exists:

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

However, a more useful warning would identify the single field access that violates the most likely locking discipline:

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

To pinpoint likely error locations, we express type inference as a MAX-SAT optimization problem over weighted constraints, instead of a SAT decision problem. A *weighted constraint* $W = C|_w$ associates a weight w with a constraint C. The MAX-SAT optimization problem is, given regular constraints $\bar{C}$ and weighted constraints $\bar{W}$, to compute the optimal assignment A that (1) satisfies all constraints in $\bar{C}$ and (2) maximizes the sum:

$$\sum \{w \mid C|_w \in \bar{W} \wedge A \models C\}$$

For the example program above, we generate the following collection of regular and weighted constraints:

$\alpha \in \{\mathtt{y}, \mathtt{this}, \mathtt{no\_lock}\}$	Scope constraint for c
$\alpha \in \{\mathtt{y}, \mathtt{this}\}\,\vert_5$	Requirement that c is guarded by a valid lock
$\alpha \in \{\mathtt{y}, \mathtt{no\_lock}\}\,\vert_2$	Access constraint for c from f1
$\alpha \in \{\mathtt{y}, \mathtt{no\_lock}\}\,\vert_2$	Access constraint for c from f2
$\alpha \in \{\mathtt{this}, \mathtt{no\_lock}\}\,\vert_2$	Access constraint for c from f3

The lock name no_lock is used by the checker to indicate that no reasonable guarding lock can be found for a field. The first (regular) constraint asserts that the guard for c is visible at the field declaration; the second (weighted) constraint asserts that the guard for c is a real lock and not no_lock; and the last three (weighted) constraints assert that the lock is held on each access.

For these constraints, the maximal solution A is the assignment $\alpha = \mathtt{y}$, with a value of 9. We then generate error messages for all constraints in $\bar{W}$ that are not satisfied by A. Specifically, the constraint $\alpha \in \{\mathtt{this}, \mathtt{no\_lock}\}\,|_2$ is not satisfied by this maximal assignment A, which results in the above warning message for the field access at line 5. Conversely, if the optimal assignment did not satisfy the constraint $\alpha \in \{\mathtt{y}, \mathtt{this}\}\,|_5$, we would generate the

corresponding error message:

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

Rcc/Sat assigns constraints for field declarations the weight 5 and constraints for field accesses the weight 2. Therefore, Rcc/Sat reports potential race conditions on accesses to a field if that field has fewer than three race conditions under the most likely protecting lock. If a field has three or more race conditions under any potential protecting lock, Rcc/Sat reports a "no consistent protecting lock" warning on the field declaration. We have found that this heuristic works well in practice.

We solve the MAX-SAT constraint optimization problem over $\bar{W}$ and $\bar{C}$ by translating it into the input for the tool PBS [4]. The translation is similar to the case without weights. PBS 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 [14].

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.

The checker currently uses quite basic implementations of rapid type analysis [15], escape analysis [16], 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 eight benchmark programs: `elevator`, a discrete event simulator [17]; `tsp`, a Traveling Salesman Problem solver [17]; `sor`, a scientific computing program [17]; the `mtrt` ray-tracing program and `jbb` business objects simulator benchmarks [18]; and the `moldyn`, `montecarlo`, and `raytracer` benchmarks [19]. We ran these experiments on a 3.06 GHz Pentium 4 processor with 2 GB of memory, with Rcc/Sat configured to insert one `ghost` parameter on classes, interfaces, and instance methods and two parameters on static methods. The checker assumes no race conditions occur in library code called from the benchmark programs.

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.[3]

We include performance measurements for two different underlying solvers, the Chaff SAT solver [14] and the PBS weighted MAX-SAT solver [4]. The running times for Rcc/Sat with Chaff are significantly better because Chaff solves a SAT problem (rather than a weighted MAX-SAT problem). Also, Rcc/Sat takes advantage of several features of Chaff. First, Rcc/Sat uses Chaff's incremental solving interface to solve the many near-identical SAT problems generated when processing fields one at a time. Second, Chaff can run in the same process as Rcc/Sat, avoiding the need to communicate to the solver via files. In contrast, when using PBS to solve weighed MAX-SAT problems to generate precise error messages, writing and reading these files accounts for up to 25%–35% of Rcc/Sat's running time. While these differences make it difficult to directly compare the performance of Chaff and PBS in this setting, they do illustrate some of the engineering and performance trade-offs in the design of Rcc/Sat.

[3] The measurements differ from our preliminary results [20] due to various improvements in the Rcc/Sat implementation.

Table 1
Benchmark programs

Program	Size (LOC)	Time w/PBS (s)		Time w/Chaff (s)		Number of constraints	SAT problem size	
		Total	per field	Total	per field		variables	clauses
elevator	529	3.3	0.14	2.9	0.12	267	1,449	3,831
tsp	723	4.4	0.11	3.16	0.08	286	1,846	5,183
sor	687	2.6	0.08	2.4	0.07	117	679	400
raytracer	1,982	15.0	0.19	5.7	0.04	689	6,745	17,263
moldyn	1,408	8.4	0.07	4.4	0.04	816	2,352	4,713
montecarlo	3,674	17.9	0.15	7.9	0.07	820	6,515	15,020
mtrt	11,315	155.2	0.85	37.1	0.20	3,769	39,609	112,493
jbb	30,519	2022.5	2.56	343.1	0.43	11,148	96,130	259,352

Table 2
Results of Rcc/Sat inference

Program	Number of manual annotations	Fields			
		Total	Read-shared	Race-free	No guard
elevator	0	23	17	6	0
tsp	3	37	21	13	3
sor	1	29	22	7	0
raytracer	2	77	45	28	4
moldyn	3	107	57	44	6
montecarlo	1	110	68	42	0
mtrt	6	181	114	63	4
jbb	40	787	472	295	20

When using Chaff on the larger benchmarks, the preliminary analyses described in Section 4.3 typically consumed less than 5%–10% of the time. In general, Rcc/Sat spent roughly 4%–10% of the overall time performing the constraint translations and 30%–60% of the overall time solving SAT problems. The remaining time was divided among standard type checking, generating the initial constraints, and writing annotated versions of the code to disk.

In Table 2, the "Manual Annotations" column reflects the number of annotations manually inserted into each program to guide the analysis. We added these few annotations to suppress warnings in situations where immediately identifiable local properties ensured correctness. The manual annotations were inserted, for example, to delineate single-threaded parts of the program; to explicitly instantiate classes in two places where the scope constraint generation heuristics did not consider the appropriate locks; to declare classes as parameterized by more ghost parameters than the default; 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* [7], meaning that a lock protects 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 unguarded fields. The analyses described in Section 4.3 reduced the number of unguarded fields by 20%–75%, a significant percentage.

We also performed the same experiments with Rcc/Sat configured to insert zero or two type parameters for classes instead of one. When no type parameters were added, the number of unguarded fields increased dramatically because thread-local data could not be properly identified. When two parameters were added to each class (and three to static methods), the results for `jbb` improved slightly over the standard configuration.

Rcc/Sat identified three fields in the `tsp` benchmark on which there are intentional races [21,7]. 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`. We believe that the remaining warnings for `modlyn`, `mtrt`, and `jbb` are spurious and could be eliminated by inserting 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 [12]. 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 intra-procedural context yields equality constraints over lock variables, which can be efficiently solved using union-find. We believe it may be possible to extend our inter-procedural type inference algorithm to accommodate ownership types. Grossman has developed a race-free type system for Cyclone, a statically safe variant of C [22]. 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 [23,24], 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 permit polynomial time algorithms for other constraint-based analyses (see, for example, Aiken's survey [25]). Cardelli [26] 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 [27] is a tool for detecting race conditions in unannotated programs dynamically (though it may fail to detect certain errors because of insufficient test coverage). Recent work extends the Eraser algorithm to handle object-oriented programming language features [28] and to leverage static analysis for improved precision and performance [29]. Agarwal and Stoller [30] 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 [31] and intra-procedural ownership inference [12] to construct annotations. These dynamic analyses complement our static approach, and it may be possible to leverage their results to facilitate type inference.

A variety of other approaches have been developed for race prevention; these include abstract interpretation [32], dataflow analyses [33,34], model checking [35–38], and type systems for process calculi [39,40].

A common and significant problem with many type inference techniques is the inability to construct meaningful error messages when inference fails (see, for example, [41–43]). 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.

Heine and Lam [44] generate meaningful error messages for a constraint-based unique pointer analysis by solving 0-1 inequality constraints with a specialized tool that processes constraints incrementally, starting with those most likely to be correct. One option for future work is to develop such a specialized solver for our constraints, which may improve performance as well as error reporting.

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 this approach to 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 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 the Rcc/Sat checker. Fadi Aloul and Lintao Zhang provided valuable assistance with PBS and Chaff, respectively. This work was partly supported by the National Science Foundation under Grants CCR-0341179 and CCR-0341387; by faculty research funds granted by the University of California at Santa Cruz and by Williams College; and by a Sloan Fellowship.

Appendix A. Proofs

This section presents the proofs for the lemmas and theorems presented in the paper (except for Theorem 4, whose proof is presented in Appendix B).

Restatement of Lemma 2. *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

$$\begin{array}{ll} & A \models C \\ \text{iff} & A(l_1) = A(l_2) \\ \text{iff} & A'(A(l_1)) = A'(A(l_2)) \quad \text{by Lemma 7} \\ \text{iff} & A' \models (A(l_1) = A(l_2)) \\ \text{iff} & A' \models A(C) \end{array}$$

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

The following technical lemma states that a lock expression l is insensitive to assignment application if it does not contain any locking variables:

Lemma 7. *Given assignments A and A' and lock expression l, $A'(A(l)) = A(l)$.*

Proof. By case analysis on the structure of l. □

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

Lemma 8. *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 expression and a lock set expression, respectively. The last case holds since $t = cn\langle l_1, \ldots, l_n\rangle$. □

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 1. *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 $P \vdash E\ \&\ \bar{C}$ then $A(P) \vdash A(E)\ \&\ A(\bar{C})$;
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 $A(P); A(E) \vdash A(\mathit{meth})\ \&\ A(\bar{C})$; and
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 statement follows by a case analysis. We show a few representative cases.

- Consider statement (1). The only way to derive $P \vdash \bar{C}$ is by rule [PROG], based on the antecedents:

  no class is declared twice in P
  no field name appears more than once per class
  no method name appears more than once per class
  $P = defn_{1..n}\ e$
  $P \vdash defn_i \ \&\ \bar{C}_i \qquad \forall i \in 1..n$
  P; ghost main_lock; {main_lock} $\vdash e : t \ \&\ \bar{C}'$

  where $\bar{C} = \bar{C}_{1..n} \cup \bar{C}'$. Since all field, class, and method names remain the same when constructing $A(P) = A(defn_{1..n})\ A(e)$, it is clear that the first three antecedents also hold for $A(P)$. By the inductive hypotheses:

  $A(P) \vdash A(defn_i) \ \&\ A(\bar{C}_i) \qquad \forall i \in 1..n$
  $A(P)$; A(ghost main_lock); A({main_lock}) $\vdash A(e) : A(t) \ \&\ A(\bar{C}')$

  and we may conclude $A(P) \vdash A(\bar{C})$ by rule [PROG].
- Consider statement (5), and suppose

  $P; E \vdash t\ fn$ guarded_by $l \ \&\ \bar{C}$

  is derived by the rule [FIELD] based on the antecedents $P; E \vdash t \ \&\ \bar{C}'$ and $\bar{C} = \bar{C}' \cup \{l \in dom(E)\}$. By the inductive hypothesis, $A(P); A(E) \vdash A(t) \ \&\ A(\bar{C}')$. By the rule [FIELD], we can conclude:

  $A(P); A(E) \vdash A(t\ fn$ 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

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

  By the inductive hypotheses:

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

  Also, we know that

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

  Note that $A(\theta(t')) = (A(\theta))(A(t'))$ and $(A(\theta(l)) = (A(\theta))(A(l))$ from Lemma 8. From these, we have

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

  and we can conclude that

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

Let $\varphi = [x_1 := L_1, \ldots, x_n := L_n]$ be a *conditional substitution* that maps variables to conditional lock expressions.

Lemma 9. *For all boolean assignments B, conditional substitutions φ, conditional lock expressions L, and conditional lock set expressions S:*

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

Proof. (1) By structural induction 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 10. *For all boolean assignments B, conditional assignments Y, lock expressions l, and lock set expressions s:*

(1) $(B(Y))(l) = B(Y(l))$.
(2) $(B(Y))(s) = B(Y(s))$.

Proof. (1) By structural induction 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

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

Similarly,

$$\begin{aligned} & B(Y(l)) \\ = \ & B(Y(l' \cdot \theta)) \\ = \ & B(Y(\theta)(Y(l'))) \\ = \ & B(Y(\theta)(B(Y(l')))) && \text{by Lemma 9} \\ & && \text{replacing } \varphi \text{ and } L \text{ with } Y(\theta) \text{ and } Y(l') \\ = \ & B(Y(\theta)(x)) && \text{by definition of } x \text{ above} \\ = \ & B(Y(\theta(x))) && \text{by definition of the application of} \\ & && \text{a conditional assignment } Y \text{ to substitution } \theta \end{aligned}$$

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

(2) The proof for s is similar. □

Restatement of Theorem 5. *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

$$\begin{array}{ll} & (B(Y)) \models C \\ \text{iff} & (B(Y)) \models l_1 = l_2 \\ \text{iff} & (B(Y))(l_1) = (B(Y))(l_2) \\ \text{iff} & B(Y(l_1)) = B(Y(l_2)) \qquad \text{by Lemma 10} \\ \text{iff} & B \models Y(l_1) = Y(l_2) \\ \text{iff} & B \models Y(l_1 = l_2) \\ \text{iff} & B \models Y(C) \end{array}$$

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

Appendix B. Proof of Theorem 4

Restatement of Theorem 4. *For an arbitrary* RFJ2 *program P, the problem of finding an assignment A such that $A(P)$ is explicitly typed and well-typed is NP-complete.*

Proof. This follows from the following Lemmas 11 and 12, 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. □

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

B.1. Translation

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

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

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 the following classes `Object` and `F`:

```
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 placeholders 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;
}
```

```
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; }
    }
  }
}
```

Fig. B.1. Class S generated from a 3-SAT instance.

Finally, P also contains the following class S appearing in Fig. B.1. In this definition, we use $p_{j,k}$ to refer to a particular lock, depending on $d_{j,k}$:

$$p_{j,k} = \begin{cases} \mathtt{p}i & \text{if } d_{i,j} = v_i \\ \mathtt{n}i & \text{if } d_{i,j} = \neg v_i \end{cases}$$

In addition, we use the notation synchronized(e_1, e_2) { e } as an abbreviation for

synchronized(e_1) { synchronized(e_2) { e } }

In essence, the code in part (a) enforces that the lock $\alpha_{j,k}$ corresponding to term $d_{j,k}$ is set to either pi or ni. It will be pi if v_i is true in the truth assignment we construct and ni 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 pi or all ni. Part (c) ensures that at least one term in each clause of T is satisfied.

B.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 $i \in 1..u$,

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

Lemma 11. *This truth assignment satisfies T.*

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

(1) Suppose $d_{j,k}$ is v_i. Then $v_i =$ **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

$$\gamma_j = \mathtt{a}k \ \wedge \ d_{j,k} = \neg v_i$$
$$\gamma'_j = \mathtt{a}k' \ \wedge \ d_{j',k'} = v_i$$

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 pi and ni are held when cj.zk and cj'.zk' are accessed, we have that

$$\alpha_{j,k} \in \{\mathtt{p}i, \mathtt{n}i\}$$
$$\alpha_{j',k'} \in \{\mathtt{p}i, \mathtt{n}i\}$$

Also, the type of cj.zk is $\mathtt{F}\langle\alpha_{j,k}\rangle$, and the type of c$j'$.z$k'$ is $\mathtt{F}\langle\alpha_{j',k'}\rangle$. 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 $\mathtt{F}\langle\alpha_{j,k}\rangle$ is the same type as $\mathtt{F}\langle\alpha_{j',k'}\rangle$. 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 Cj is guarded by γ_j and $\gamma_j = \mathtt{a}k$, the lock $\alpha_{j,k}$ must be held when cj.y is accessed in the statement above. Since $d_{j,k} = \neg v_i$, then ni is in the lock set $\{p_{j,1}, p_{j,2}, p_{j,3}\}$ However, pi is not in this set since we assume no SAT clause contains both a variable and its negation. Therefore, $\alpha_{j,k} = \mathtt{n}i$.

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; }
```

The lock $\alpha_{j',k'}$ must be held when cj'.y is accessed in this statement. Since $d_{j',k'} = v_i$, then pi is in the set $\{p_{j,1}, p_{j,2}, p_{j,3}\}$. However, n$i$ is not in this set since we assume no clause contains both a variable and its negation. Therefore, $\alpha_{j',k'} = \mathtt{p}i$.

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 $j \in 1..n$,

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

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

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

Lemma 12. *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

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

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

Next, consider each statement

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

```
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. B.2. Program corresponding to the example SAT instance.

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 $\mathtt{c}j\mathtt{.w}k$ and $\mathtt{c}j'\mathtt{.w}k'$ are held, and that the types of $\mathtt{c}j\mathtt{.w}k$ and $\mathtt{c}j'\mathtt{.w}k'$ are equal. The locks are clearly held since the code synchronizes on them. The type of $\mathtt{c}j\mathtt{.w}k$ is $\mathtt{F}\langle\alpha_{j,k}\rangle$ and the type of $\mathtt{c}j'\mathtt{.w}k'$ is $\mathtt{F}\langle\alpha_{j',k'}\rangle$. Given the construction of the substitution above, $\alpha_{j,k}$ and $\alpha_{j',k'}$ will either both be $\mathtt{p}i$ or both be $\mathtt{n}i$.

Finally, consider each statement

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

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

(1) $\alpha_{j,k} = \mathtt{p}i$: Then $d_{j,k}$ refers to v_i and $v_i =$ **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 $v_i =$ **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. □

B.3. Example

Consider the SAT problem:

$$v_1 \vee \neg v_2 \vee v_3$$
$$\neg v_1 \vee v_2 \vee v_3$$

This SAT problem reduces to the type and effect inference problem for the program in Fig. B.2.

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

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

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 straightforward to construct substitutions for the type variables that make the program well-typed for any satisfying assignment of the 3-SAT problem.

References

[1] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Transactions on Computers 28 (9) (1979) 690–691.
[2] C. Flanagan, S.N. Freund, Type-based race detection for Java, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2000, pp. 219–232.
[3] C. Flanagan, S.N. Freund, Detecting race conditions in large programs, in: Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering, 2001, pp. 90–96.
[4] F.A. Aloul, A. Ramani, I.L. Markov, 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, 2002, pp. 346–353.
[5] C. Flanagan, S. Qadeer, Types for atomicity, in: Proceedings of the ACM Workshop on Types in Language Design and Implementation, 2003, pp. 1–12.
[6] C. Flanagan, S. Qadeer, A type and effect system for atomicity, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2003, pp. 338–349.
[7] C. Flanagan, S.N. Freund, Atomizer: A dynamic atomicity checker for multithreaded programs, in: Proceedings of the ACM Symposium on the Principles of Programming Languages, 2004, pp. 256–267.
[8] S.D. Stoller, Model-checking multi-threaded distributed Java programs, International Journal on Software Tools for Technology Transfer 4 (1) (2002) 71–91.
[9] S.D. Stoller, E. Cohen, Optimistic synchronization-based state-space reduction, in: H. Garavel, J. Hatcliff (Eds.), Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, in: Lecture Notes in Computer Science, vol. 2619, Springer-Verlag, 2003, pp. 489–504.
[10] C. Flanagan, S. Qadeer, Transactions for software model checking, in: Proceedings of the Workshop on Software Model Checking, 2003, pp. 338–349.
[11] M.B. Dwyer, J. Hatcliff, Robby, V.P. Ranganath, Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs, Formal Methods in System Design 25 (2–3) (2004) 199–240.
[12] C. Boyapati, 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, 2001, pp. 56–69.
[13] M. Abadi, C. Flanagan, S.N. Freund, Types for safe locking: Static race detection for Java, ACM Transactions on Programming Languages and Systems 28 (2) (2006).
[14] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, in: Proceedings of the Design Automation Conference, 2001, pp. 530–535.
[15] D.F. Bacon, 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, 1996, pp. 324–341.
[16] J. Bogda, U. Hölzle, Removing unnecessary synchronization in Java, in: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, 1999, pp. 35–46.
[17] C. von Praun, T. Gross, Static conflict analysis for multi-threaded object-oriented programs, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2003, pp. 115–128.
[18] Standard Performance Evaluation Corporation, SPEC benchmarks, Available from http://www.spec.org/, 2003.
[19] Java Grande Forum, Java Grande benchmark suite, Available from http://www.javagrande.org/, 2003.
[20] C. Flanagan, S.N. Freund, Type inference against races, in: Static Analysis Symposium, 2004, pp. 116–132.
[21] R. O'Callahan, J.-D. Choi, Hybrid dynamic data race detection, in: ACM Symposium on Principles and Practice of Parallel Programming, 2003, pp. 167–178.
[22] D. Grossman, Type-safe multithreading in Cyclone, in: Proceedings of the ACM Workshop on Types in Language Design and Implementation, 2003, pp. 13–25.
[23] M. Tofte, 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, 1994, pp. 188–201.
[24] J.-P. Talpin, P. Jouvelot, Polymorphic type, region and effect inference, Journal of Functional Programming 2 (3) (1992) 245–271.
[25] A. Aiken, Introduction to set constraint-based program analysis, Science of Computer Programming 35 (2) (1999) 79–111.

[26] L. Cardelli, Typechecking dependent types and subtypes, in: Lecture notes in computer science on Foundations of logic and functional programming, 1988, pp. 45–57.
[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T.E. Anderson, Eraser: A dynamic data race detector for multi-threaded programs, ACM Transactions on Computer Systems 15 (4) (1997) 391–411.
[28] C. von Praun, T. Gross, Object race detection, in: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, 2001, pp. 70–82.
[29] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, M. Sridhara, Efficient and precise datarace detection for multithreaded object-oriented programs, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2002, pp. 258–269.
[30] R. Agarwal, S.D. Stoller, Type inference for parameterized race-free Java, in: Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation, 2004, pp. 149–160.
[31] J. Aldrich, V. Kostadinov, C. Chambers, Alias annotations for program understanding, in: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, 2002, pp. 311–330.
[32] Polyspace Technologies, available at http://www.polyspace.com, 2003.
[33] M.B. Dwyer, L.A. Clarke, Data flow analysis for verifying properties of concurrent programs, Technical Report 94-045, Department of Computer Science, University of Massachusetts at Amherst, 1994.
[34] D.R. Engler, K. Ashcraft, RacerX: Effective, static detection of race conditions and deadlocks, in: ACM Symposium on Operating System Principles, 2003, pp. 237–252.
[35] A.T. Chamillard, L.A. Clarke, G.S. Avrunin, An empirical comparison of static concurrency analysis techniques, Technical Report 96-084, Department of Computer Science, University of Massachusetts at Amherst, 1996.
[36] J.C. Corbett, Evaluating deadlock detection methods for concurrent software, IEEE Transactions on Software Engineering 22 (3) (1996) 161–180.
[37] L. Fajstrup, E. Goubault, M. Raussen, Detecting deadlocks in concurrent systems, in: D. Sangiorgi, R. de Simone (Eds.), Proceedings of the International Conference on Concurrency Theory, in: Lecture Notes in Computer Science, vol. 1466, Springer-Verlag, 1998, pp. 332–347.
[38] E. Yahav, Verifying safety properties of concurrent Java programs using 3-valued logic, in: Proceedings of the ACM Symposium on the Principles of Programming Languages, 2001, pp. 27–40.
[39] N. Kobayashi, A partially deadlock-free typed process calculus, ACM Transactions on Programming Languages and Systems 20 (2) (1998) 436–482.
[40] N. Kobayashi, S. Saito, E. Sumii, An implicitly-typed deadlock-free process calculus, in: C. Palamidessi (Ed.), Proceedings of the International Conference on Concurrency Theory, in: Lecture Notes in Computer Science, vol. 1877, Springer-Verlag, 2000, pp. 489–503.
[41] M. Wand, Finding the source of type errors, in: Proceedings of the ACM Symposium on the Principles of Programming Languages, 1986, pp. 38–43.
[42] J. Yang, G. Michaelson, P. Trinder, J.B. Wells, Improved type error reporting, in: Proceedings of the International Workshop on Implementation of Functional Languages, 2000, pp. 71–86.
[43] C. Haack, J.B. Wells, Type error slicing in implicitly typed higher-order languages, in: European Symposium on Programming, 2003, pp. 284–301.
[44] D.L. Heine, M.S. Lam, A practical flow-sensitive and context-sensitive C and C++ memory leak detector, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2003, pp. 168–181.