Deadlock Checking by Data Race Detection

Ka Pun, Martin Steffen, Volker Stolz

Deadlock Checking by Data Race Detection

Ka I Pun[1], Martin Steffen[1], and Volker Stolz[1,2]

[1] University of Oslo, Norway
[2] United Nations University—Intl. Inst. for Software Technology, Macao

Abstract. Deadlocks are a common problem in programs with lock-based concurrency and are hard to avoid or even to detect. One way for deadlock prevention is to statically analyze the program code to spot sources of potential deadlocks. We reduce the problem of deadlock checking to race checking, another prominent concurrency-related error for which good (static) checking tools exist. The transformation uses a type and effect-based static analysis, which analyzes the data flow in connection with lock handling to find out control-points that are potentially part of a deadlock. These control-points are instrumented appropriately with additional shared variables, i.e., race variables injected for the purpose of the race analysis. To avoid overly many false positives for deadlock cycles of length longer than two, the instrumentation is refined by adding "gate locks". The type and effect system and the transformation are formally given. We prove our analysis sound using a simple, concurrent calculus with re-entrant locks.

1 Introduction

Concurrent programs are notoriously hard to get right and at least two factors contribute to this fact: Correctness properties of a parallel program are often global in nature, i.e., result from the correct interplay and cooperation of multiple processes. Hence also violations are non-local, i.e., they cannot typically be attributed to a single line of code. Secondly, the non-deterministic nature of concurrent executions makes concurrency-related errors hard to detect and to reproduce. Since typically the number of different interleavings is astronomical or infinite, testing will in general not exhaustively cover all behavior and errors may remain undetected until the software is in use.

Arguably the two most important and most investigated classes of concurrency errors are *data races* [3] and *deadlocks* [9]. A data race is the simultaneous, unprotected access to mutable shared data with at least one write access. A deadlock occurs when a number of processes are unable to proceed, when waiting cyclically for each other's non-shareable resources without releasing one's own [7]. Deadlocks and races constitute equally pernicious, but complementary hazards: locks offer protection against races by ensuring mutually exclusive access, but may lead to deadlocks, especially using fine-grained locking, or are at least detrimental to the performance of the program by decreasing the degree of parallelism. Despite that, both share some commonalities, too: a race, respectively a deadlock, manifests itself in the execution of a concurrent program, when two processes (for a race) resp. two or more processes (for a deadlock) reach respective control-points that when reached *simultaneously*, constitute an unfortunate interaction: in case of a race, a read-write or write-write conflict on a shared variable, in case of a deadlock, running jointly into a cyclic wait.

In this paper, we define a static analysis for multi-threaded programs which allows reducing the problem of deadlock checking to race condition checking. The analysis is based on a type and effect system [2] which formalizes the data-flow of lock usages and, in the effects, works with an over-approximation on how often different locks are being held. The information is used to instrument the program with additional variables to signal a race at control points that potentially are involved in a deadlock. Despite the fact that races, in contrast to deadlocks, are a binary global concurrency error in the sense that only two processes are involved, the instrumentation is not restricted to deadlock cycles of length two. To avoid raising too many spurious alarms when dealing with cycles of length > 2, the transformation adds additional locks, to prevent that already parts of a deadlock cycle give raise to a race, thus falsely or prematurely indicating a deadlock by a race.

Our approach widens the applicability of freely available state-of-the-art static race checkers: *Goblint* [20] for the C language, which is not designed to do any deadlock checking, will report appropriate data races from programs instrumented through our transformation, and thus becomes a deadlock checker as well. *Chord* [15] for Java only analyses deadlocks of length two for Java's `synchronized` construct, but not explicit locks from `java.util.concurrent`, yet through our instrumentation reports corresponding races for longer cycles *and* for deadlocks involving explicit locks.

The remainder of the paper is organised as follows. Section 2 presents syntax and operational semantics of the calculus. Afterwards, Section 3 formalizes the data flow analysis in the form of a (constraint-based) effect system. The obtained information is used in Sections 4 and 5 to instrument the program with race variables and additional locks. The sections also prove the soundness of the transformation. We conclude in Section 6 discussing related and future work.

2 Calculus

In this section we present the syntax and (operational) semantics for our calculus, formalizing a simple, concurrent language with dynamic thread creation and higher-order functions. Locks likewise can be created dynamically, they are re-entrant and support non-lexical use of locking and unlocking. The abstract syntax is given in Table 1. A program P consists of a parallel composition of processes $p\langle t\rangle$, where p identifies the process and t is a thread, i.e., the code being executed. The empty program is denoted as $\emptyset$. As usual, we assume $\|$ to be associative and commutative, with $\emptyset$ as neutral element. As for the code we distinguish threads t and expressions e, where t basically is a sequential composition of expressions. Values are denoted by v, and $\texttt{let}\ x{:}T = e\ \texttt{in}\ t$ represents the sequential composition of e followed by t, where the eventual result of e, i.e., once evaluated to a value, is bound to the local variable x. Expressions, as said, are given by e, and threads are among possible expressions. Further expressions are function application, conditionals, and the spawning of a new thread, written $\texttt{spawn}\ t$. The last three expressions deal with lock handling: $\texttt{new L}$ creates a new lock (initially free) and gives a reference to it (the $\texttt{L}$ may be seen as a class for locks), and furthermore $v.\texttt{lock}$ and $v.\texttt{unlock}$ acquires and releases a lock, respectively. Values, i.e., evaluated expressions, are variables, lock references, and function abstractions, where we use $\texttt{fun}\ f{:}T_1.x{:}T_2.t$

Listing 1. Dining Philosophers

```
let l1 = new L; ...; ln = new L   /* create all locks */
  phil = fun F(x,y) . ( x.lock; y.lock;     /* eat */
                        y.unlock; x.unlock; /* think */
                        F(x,y) )
  in spawn(phil(l1,l2)); ... ; spawn(phil(ln,l1))
```

for recursive function definitions. Note that the grammar insists that, e.g., in an application, both the function and the arguments are values, analogously when acquiring a lock, etc. This form of representation is known as *a-normal form* [11].

Listing 1 shows the paraphrased code for the well-known Dining Philosopher example. The recursive body used for each philosopher is polymorphic in the lock locations.

$P ::= \emptyset \mid p\langle t\rangle \mid P \parallel P$	program
$t ::= v \mid \mathtt{let}\, x{:}T = e\, \mathtt{in}\, t$	thread
$e ::= t \mid v\, v \mid \mathtt{if}\, v\, \mathtt{then}\, e\, \mathtt{else}\, e \mid \mathtt{spawn}\, t \mid \mathtt{new\ L} \mid v.\,\mathtt{lock} \mid v.\,\mathtt{unlock}$	expr.
$v ::= x \mid l^r \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{fn}\, x{:}T.t \mid \mathtt{fun}\, f{:}T.x{:}T.t$	values

Table 1. Abstract syntax

The grammar for types, effects, and annotations is given Table 2, where π represents labels (used to label program points where locks are created), r represents (finite) sets of πs, where ρ is a corresponding variable. Labels π are an abstraction of concrete lock references which exist at run-time (namely all those references created at that program point) and therefore we refer to labels π as well as lock sets r also as *abstract locks*. Types include basic types B such as integers, booleans, etc., left unspecified, function types $\hat{T}_1 \xrightarrow{\varphi} \hat{T}_2$, and in particular lock types $\mathtt{L}$. To capture the data flow concerning locks, the lock types are annotated with a lock set r, i.e., they are of the form $\mathtt{L}^r$. This information will be inferred, and the user, when using types in the program, uses types without annotations (the “underlying” types). We write $T, T_1, T_2, \ldots$ as meta-variables for the underlying types, and $\hat{T}$ and its syntactic variants for the annotated types, as given in the grammar. Furthermore, polymorphism for function definition is captured by type schemes $\hat{S}$, i.e., types prefix-quantified over variables ρ and X, under some constraints. We let Y abbreviate either variables ρ or X, where X is a variable for effect which is introduced later. Any specialization of the type scheme $\forall\vec{Y}{:}C.\hat{T}$ has to satisfy the constraints C. For the deadlock and race analysis we need not only information which locks are used where, but also an estimation about the “value” of the lock, i.e., how often the abstractly represented locks are taken.

Estimation of the lock values, resp. their change is captured in the behavioral *effects* φ in the form of pre- and post-specifications $\Delta_1 \to \Delta_2$. Abstract states (or lock envi-

r	$::=$	$\rho \mid \{\pi\} \mid r \sqcup r$	lock/label sets
$\hat{T}$	$::=$	$B \mid \mathtt{L}^r \mid \hat{T} \xrightarrow{\varphi} \hat{T}$	types
$\hat{S}$	$::=$	$\forall \vec{Y}{:}C.\, \hat{T}$	type schemes
φ	$::=$	$\Delta \to \Delta$	effects/pre- and post specification
Δ	$::=$	$\bullet \mid X \mid \Delta, r{:}n$	lock env./abstract state
C	$::=$	$\emptyset \mid \rho \sqsupseteq r,\, C \mid X \geq \Delta, C$	constraints

Table 2. Types

ronments) Δ are of the form $r_0{:}n_0, r_1{:}n_1, \ldots$. The constraint based type system works on lock environments using variables only, i.e., the Δ are of the form $\rho_0{:}n_0, \rho_1{:}n_1, \ldots$, maintaining that each variable occurs at most once. Thus, in the type system, the environments Δ are mappings from variables ρ to lock counter values n, where n ranges from $+\infty$ to $-\infty$. As for the syntactic representation of those mappings: we assume that a variable ρ *not* mentioned in Δ corresponds to the binding $\rho{:}0$, e.g. in the empty mapping $\bullet$. Constraints C finally are finite sets of subset inclusions of the forms $\rho \sqsupseteq r$ and $X \geq \Delta$. We assume that the user provides the underlying types, i.e., without location and effect annotation, while our type system in Section 3 derives the smallest possible type in terms of originating locations for each variable of lock-type `L` in the program.

Semantics

Next we present the operational semantics, given in the form of a small-step semantics, distinguishing between local and global steps (cf. Tables 3 and 4). The local semantics deals with reduction steps of one single thread of the form $t_1 \to t_2$. Rule R-RED is the basic evaluation step which replaces the local variable in the continuation thread t by the value v (where $[v/x]$ represents capture-avoiding substitution). The Let-construct generalizes sequential composition and rule R-LET restructures a nested let-construct expressing associativity of that construct. Thus it corresponds to transforming $(e_1;t_1);t_2$ into $e_1;(t_1;t_2)$. Together with the first rule, it assures a deterministic left-to-right evaluation within each thread. The two R-IF-rules cover the two branches of the conditional and the R-APP-rules deals with function application (of non-recursive, resp. recursive functions).

The global steps are given in Table 4, formalizing transitions of configurations of the form $\sigma \vdash P$, i.e., the steps are of the form $\sigma \vdash P \to \sigma' \vdash P'$, where P is a program, i.e., the parallel composition of a finite number of threads running in parallel, and σ is a finite mapping from lock identifiers to the status of each lock (which can be either free or taken by a thread where a natural number indicates how often a thread has acquired the lock, modelling re-entrance). Thread-local steps are lifted to the global level by R-LIFT. Rule R-PAR specifies that the steps of a program consist of the steps of the individual threads, sharing σ. Executing the spawn-expression creates a new thread with a fresh identity which runs in parallel with the parent thread (cf. rule R-SPAWN). Globally, the process identifiers are unique. A new lock is created by `newL` (cf. rule R-NEWL) which allocates a fresh lock reference in the heap. Initially, the lock is free. A lock l is acquired

$\texttt{let}\ x{:}T = v\ \texttt{in}\ t\ \rightarrow\ t[v/x]$ R-RED

$\texttt{let}\ x_2{:}T_2 = (\texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1)\ \texttt{in}\ t_2\ \rightarrow \texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ (\texttt{let}\ x_2{:}T_2 = t_1\ \texttt{in}\ t_2)$ R-LET

$\texttt{let}\ x{:}T = \texttt{if}\ \texttt{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t\ \rightarrow \texttt{let}\ x{:}T = e_1\ \texttt{in}\ t$ R-IF$_1$

$\texttt{let}\ x{:}T = \texttt{if}\ \texttt{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t\ \rightarrow \texttt{let}\ x{:}T = e_2\ \texttt{in}\ t$ R-IF$_2$

$\texttt{let}\ x{:}T = (\texttt{fn}\ x'{:}T'.t')\ v\ \texttt{in}\ t \rightarrow \texttt{let}\ x{:}T = t'[v/x']\ \texttt{in}\ t$ R-APP$_1$

$\texttt{let}\ x{:}T = (\texttt{fun}\ f{:}T_1.x'{:}T_2.t')\ v\ \texttt{in}\ t\ \rightarrow \texttt{let}\ x{:}T = t'[v/x'][\texttt{fun}\ f{:}T_1.x'{:}T_2.t'/f]\ \texttt{in}\ t$ R-APP$_2$

Table 3. Local steps

by executing $l.\texttt{lock}$. There are two situations where that command does not block, namely the lock is free or it is already held by the requesting process p. The heap update $\sigma +_p l$ is defined as follows: If $\sigma(l) = \mathit{free}$, then $\sigma +_p l = \sigma[l \mapsto p(1)]$ and if $\sigma(l) = p(n)$, then $\sigma +_p l = \sigma[l \mapsto p(n+1)]$. Dually $\sigma -_p l$ is defined as follows: if $\sigma(l) = p(n+1)$, then $\sigma -_p l = \sigma[l \mapsto p(n)]$, and if $\sigma(l) = p(1)$, then $\sigma -_p l = \sigma[l \mapsto \mathit{free}]$. Unlocking works correspondingly, i.e., it sets the lock as being free resp. decreases the lock count by one (cf. rule R-UNLOCK). In the premise of the rules it is checked that the thread performing the unlocking actually holds the lock.

$$\frac{t_1 \rightarrow t_2}{\sigma \vdash p\langle t_1\rangle \rightarrow \sigma \vdash p\langle t_2\rangle}\ \text{R-LIFT} \qquad \frac{\sigma \vdash P_1 \rightarrow \sigma' \vdash P_1'}{\sigma \vdash P_1 \parallel P_2 \rightarrow \sigma' \vdash P_1' \parallel P_2}\ \text{R-PAR}$$

$$\sigma \vdash p_1\langle \texttt{let}\ x{:}T = \texttt{spawn}\ t_2\ \texttt{in}\ t_1\rangle \rightarrow \sigma \vdash p_1\langle \texttt{let}\ x{:}T = p_2\ \texttt{in}\ t_1\rangle \parallel p_2\langle t_2\rangle \quad \text{R-SPAWN}$$

$$\frac{\sigma' = \sigma[l \mapsto \mathit{free}] \qquad l \text{ is fresh}}{\sigma \vdash p\langle \texttt{let}\ x{:}T = \texttt{new L}\ \texttt{in}\ t\rangle \rightarrow \sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle}\ \text{R-NEWL}$$

$$\frac{\sigma(l) = \mathit{free} \vee \sigma(l) = p(n) \qquad \sigma' = \sigma +_p l}{\sigma \vdash p\langle \texttt{let}\ x{:}T = l.\ \texttt{lock}\ \texttt{in}\ t\rangle \rightarrow \sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle}\ \text{R-LOCK}$$

$$\frac{\sigma(l) = p(n) \qquad \sigma' = \sigma -_p l}{\sigma \vdash p\langle \texttt{let}\ x{:}T = l.\ \texttt{unlock}\ \texttt{in}\ t\rangle \rightarrow \sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle}\ \text{R-UNLOCK}$$

Table 4. Global steps

To analyze deadlocks and races, we specify which locks are meant statically by labelling the program points of lock creations with π, i.e., lock creation statements `new L` are augmented to $\texttt{new}_\pi\ \texttt{L}$ where the annotations π are assumed unique for a given program. We assume further that the lock references l are also labelled l^ρ; the labelling is done by the type system presented next.

3 Type and effect system

Next we present a constraint-based type and effect system for information which locks are being held at various points in the code. The analysis works thread-locally, i.e., it analyzes the code of one thread. In Section 4, we will use this information to determine points in a program, that globally may lead to deadlocks and which are then instrumented appropriately by additional race variables. The judgments of the system are of the form

$$\Gamma \vdash e : \hat{T} :: \varphi; C \,, \tag{1}$$

where φ is of the form $\Delta_1 \to \Delta_2$. Equivalently, we write also $\Gamma; \Delta_1 \vdash e : \hat{T} :: \Delta_2; C$ for the judgment. The judgment expresses that e is of type $\hat{T}$, where for annotated lock types of the form $\mathtt{L}^r$ the r expresses the potential points of creation of the lock. The effect $\varphi = \Delta_1 \to \Delta_2$ expresses the change in the lock counters, where Δ_1 is the pre-condition and Δ_2 the post-condition (in a partial correctness manner). The types and the effects contain variables ρ and X; hence the judgement is interpreted relative to the solutions of the set of constraints C. Given Γ and e, the constraint set C is generated during the derivation. Furthermore, the pre-condition Δ_1 is considered as given, whereas Δ_2 is derived.

The rules for the type system are given in Table 5. The rule TA-VAR combines looking up the variable from the context with instantiation, choosing fresh variables to assure that the constraints θC, where C is taken from the variable's type scheme, are most general. As a general observation and as usual, values have no effect, i.e., its pre- and post-condition are identical. Also lock creation in rule TA-NEWL does not have an effect. As for the flow: π labels the point of creation of the lock; hence a new constraint is generated, requiring $\rho \sqsupseteq \{\pi\}$ for the ρ-annotation in the lock type. The case for lock references l^ρ in rule TA-LREF works analogously, where the generated constraint uses the lock variable ρ instead of the concrete point of creation.

For function abstraction in rule TA-ABS$_1$, the premise checks the body e of the function with the typing context extended by $x{:}\lceil T \rceil_A$, where the operation $\lceil T \rceil_A$ turns all occurrences of lock types $\mathtt{L}$ in T into their annotated counter-parts using *fresh* variables, as well as introducing state variables for the latent effects of higher-order functions. Also for the pre-condition of the function body, a fresh variable is used. The recursive function is also formulated similarly. It uses in addition a *fresh* variable for the post-condition of the function body, and constraints requiring $X_2 \geq \Delta_2$ and $\hat{T}_2 \geq \hat{T}_2'$ are generated. For function application (cf. rule TA-APP), the subtyping requirement between the type $\hat{T}_2$ of the argument and the function's input type $\hat{T}_2'$ is used to generate additional constraints. Furthermore, the precondition Δ of the application is connected with the precondition of the latent effect Δ_1 and the post-condition of the latent effect with the post-condition of the application, the latter one using again a fresh variable. The corresponding two constraints $\Delta \leq \Delta_1$ and $\Delta_2 \leq X$ represent the control flow when calling, resp. when returning to the call site. The treatment of conditionals is standard (cf. rule TA-COND). To assure that the resulting type is an upper bound for the types of the two branches, two additional constraints C and C' are generated.

The let-construct (cf. rule TA-LET) is combined with the rule for generalization, such that for checking the body e_2, the typing context is extended by a type scheme

$\hat{S}_1$ which generalizes the type $\hat{T}_1$ of expression e_1. The close-operation is defined as $close(\Gamma, C, \hat{T}) = \forall \vec{Y}{:}C.\hat{T}$ where the quantifier binds all variables occurring free in C and $\hat{T}$ but not in Γ. Spawning a thread in rule TA-SPAWN has no effect, where the premise of the rule checks well-typedness of the thread being spawned. The last two rules deal with locking and unlocking, simply counting up, resp. down the lock counter, setting the post-condition to over-approximate $\Delta \oplus \rho$, resp. $\Delta \ominus \rho$.

$$\frac{\Gamma(x) = \forall \vec{Y}{:}C.\hat{T} \qquad \vec{Y}'\ \mathit{fresh} \qquad \theta = [\vec{Y}'/\vec{Y}]}{\Gamma \vdash x : \theta\hat{T} :: \Delta \to \Delta; \theta C}\ \text{TA-VAR} \qquad \frac{\Gamma \vdash t : \hat{T} :: \bullet \to \Delta_2; C}{\Gamma \vdash \texttt{spawn}\, t : \texttt{Thread} :: \Delta_1 \to \Delta_1; C}\ \text{TA-SPAWN}$$

$$\frac{\rho\ \mathit{fresh}}{\Gamma \vdash \texttt{new}_\pi\ \texttt{L} :\ \texttt{L}^\rho :: \Delta \to \Delta; \rho \sqsupseteq \{\pi\}}\ \text{TA-NEWL} \qquad \frac{\rho'\ \mathit{fresh}}{\Gamma \vdash l^\rho : \texttt{L}^{\rho'} :: \Delta \to \Delta; \rho' \sqsupseteq \rho}\ \text{TA-LREF}$$

$$\frac{\hat{T}_1 = \lceil T_1 \rceil_A \qquad \Gamma, x{:}\hat{T}_1 \vdash e : \hat{T}_2 :: X \to \Delta_2; C \qquad X\ \mathit{fresh}}{\Gamma \vdash \texttt{fn}\, x{:}T_1.e : \hat{T}_1 \xrightarrow{X \to \Delta_2} \hat{T}_2 :: \Delta_1 \to \Delta_1; C}\ \text{TA-ABS}_1$$

$$\frac{\begin{array}{c}\hat{T}_1 = \lceil T_1 \rceil_A \qquad \hat{T}_2 = \lceil T_2 \rceil_A \qquad X_1, X_2\ \mathit{fresh} \\ \Gamma, f{:}\hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2, x{:}\hat{T}_1 \vdash e : \hat{T}_2' :: X_1 \to \Delta_2; C_1 \qquad \hat{T}_2' \leq \hat{T}_2 \vdash C_2 \qquad C_3 = \Delta_2 \leq X_2\end{array}}{\Gamma \vdash \texttt{fun}\, f{:}T_1 \to T_2, x{:}T_1.e : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 :: \Delta_1 \to \Delta_1; C_1, C_2, C_3}\ \text{TA-ABS}_2$$

$$\frac{\Gamma \vdash v_1 : \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_1 :: \Delta \to \Delta; C_1 \qquad \Gamma \vdash v_2 : \hat{T}_2' :: \Delta \to \Delta; C_2 \qquad \hat{T}_2' \leq \hat{T}_2 \vdash C \qquad X\ \mathit{fresh}}{\Gamma \vdash v_1\ v_2 : \hat{T}_1 :: \Delta \to X; C_1, C_2, C, \Delta \leq \Delta_1, \Delta_2 \leq X}\ \text{TA-APP}$$

$$\frac{\begin{array}{c}T = \lfloor \hat{T}_1 \rfloor = \lfloor \hat{T}_2 \rfloor \qquad \hat{T}; C = \hat{T}_1 \vee \hat{T}_2 \qquad \Delta'; C' = \Delta_1' \vee \Delta_2' \\ \Gamma \vdash v : \texttt{Bool} :: \Delta_0 \to \Delta_0; C_0 \quad \Gamma \vdash e_1 : \hat{T}_1 :: \Delta_0 \to \Delta_1; C_1 \quad \Gamma \vdash e_2 : \hat{T}_2 :: \Delta_0 \to \Delta_2; C_2\end{array}}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \hat{T} :: \Delta_0 \to \Delta'; C_0, C_1, C_2, C, C'}\ \text{TA-COND}$$

$$\frac{\Gamma \vdash e_1 : \hat{T}_1 :: \Delta_1 \to \Delta_2; C_1 \qquad \lfloor \hat{T}_1 \rfloor = T_1 \qquad \hat{S}_1 = close(\Gamma, C_1, \hat{T}_1) \qquad \Gamma, x{:}\hat{S}_1 \vdash e_2 : \hat{T}_2 :: \Delta_2 \to \Delta_3; C_2}{\Gamma \vdash \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2 :: \Delta_1 \to \Delta_3; C_2}\ \text{TA-LET}$$

$$\frac{\Gamma \vdash v : \texttt{L}^\rho :: \Delta \to \Delta; C_1 \qquad X\ \mathit{fresh} \qquad C_2 = X \geq \Delta \oplus (\rho{:}1)}{\Gamma \vdash v.\ \texttt{lock} : \texttt{L}^\rho :: \Delta \to X; C_1, C_2}\ \text{TA-LOCK}$$

$$\frac{\Gamma \vdash v : \texttt{L}^\rho :: \Delta \to \Delta; C_1 \qquad X\ \mathit{fresh} \qquad C_2 = X \geq \Delta \ominus (\rho{:}1)}{\Gamma \vdash v.\ \texttt{unlock} : \texttt{L}^\rho :: \Delta \to X; C_1, C_2}\ \text{TA-UNLOCK}$$

Table 5. Constraint based type and effect system

The type system is basically a single-threaded analysis. For subject reduction later and soundness of the analysis, we also need to analyse processes running in parallel. The definition is straightforward, since a global program is well-typed simply if all its threads are. For one thread, $p\langle t \rangle : p\langle \varphi; C \rangle$, if $\vdash t : \hat{T} :: \varphi; C$ for some type $\hat{T}$. We will abbreviate $p_1\langle \varphi_1; C_1 \rangle \parallel \ldots \parallel p_k\langle \varphi_k; C_k \rangle$ by Φ.

Constraints C come in two forms: $r \sqsubseteq \rho$ and $X_1 \leq X_2 \oplus (\rho{:}n)$ resp. $X_1 \leq X_2 \ominus (\rho{:}n)$. We consider both kinds of constraints as independent, in particular a constraint of the form $X_1 \leq X_2 \oplus (\rho{:}n)$ is considered as a constraint between the two variables X_1 and X_2 and not as a constraint between X_1, X_2, *and* ρ. Given C, we write C^ρ for the ρ-constraints in C and C^X for the constraints concerning X-variables. Solutions to the constraints are ground substitutions; we use θ to denote substitutions. analogous to the distinction for the constraints, we write θ^ρ for substitutions concerning the ρ-variables and θ^X for substitutions concerning the X-variables. A ground θ^ρ-substitution maps ρ's to finite sets $\{\pi_1, \ldots, \pi_n\}$ of labels and a ground θ^X-substitution maps X's to Δ's (which are of the form $\rho_1{:}n_1, \ldots, \rho_k{:}n_k$); note that the range of the ground θ^X-substitution still contains ρ-variables. We write $\theta^\rho \models C$ if θ^ρ solves C^ρ and analogously $\theta^X \models C$ if θ^X solves C^X. For a $\theta = \theta^X \theta^\rho$, we write $\theta \models C$ if $\theta^\rho \models C$ and $\theta^X \models C$. Furthermore we write $C_1 \models C_2$ if $\theta \models C_1$ implies $\theta \models C_2$, for all ground substitutions θ. For the simple super-set constraints of the form $\rho \sqsupseteq r$, constraints always have a unique minimal solution. Analogously for the C^X-constraints. A heap σ satisfies an abstract state Δ, if Δ over-approximates the lock counter for all locks in σ: Assuming that Δ does not contain any ρ-variables and that the lock references in σ are labelled by π's, $\sigma \models \Delta$ if $\sum_{\pi \in r} \sigma(l^\pi) \leq \Delta(r)$ (for all r in $dom(\Delta)$). Given a constraint set C, an abstract state Δ (with lock references l^ρ labelled by variables) and a heap σ, we write $\sigma \models_C \Delta$ ("σ satisfies Δ under the constraints C"), iff $\theta \models C$ implies $\theta\sigma \models \theta\Delta$, for all θ. A heap σ satisfies a global effect Φ (written $\sigma \models \Phi$), if $\sigma \models_{C_i} \Delta_i$ for all $i \leq k$ where $\Phi = p_1\langle\varphi_1; C_1\rangle \parallel \ldots \parallel p_k\langle\varphi_k; C_k\rangle$ and $\varphi_i = \Delta_i \to \Delta_i'$.

Soundness

Next we prove soundness of the analysis wrt. the semantics. The core of the proof is the preservation of well-typedness under reduction ("subject reduction"). The static analysis does not only give back types (as an abstraction of resulting *values*) but also effects (in the form of pre- and post-specification). While types are preserved, we cannot expect that the effect of an expression remains unchanged under reduction. As the pre- and post-conditions specify (upper bounds on) the allowed lock values, the only steps which change are locking and unlocking steps. To relate the change of pre-condition with the steps of the system we assume the transitions to be labelled. Relevant is only the lock set variable ρ; the identity p of the thread, the label π and the actual identity of the lock are not relevant for the formulation of subject reduction, hence we do not include that information in the labels here. The steps for lock-taking are of the form $\sigma_1 \vdash p\langle t_1\rangle \xrightarrow{p\langle\rho.\mathtt{lock}\rangle} \sigma_2 \vdash p\langle t_2\rangle$; unlocking steps analogously are labelled by $\rho.\mathtt{unlock}$ and all other steps are labelled by τ, denoting internal steps. The formulation of subject reduction can be seen as a form of *simulation* (cf. Figure 1): The concrete steps of the system —for one process in the formulation of subject reduction— are (weakly) simulated by changes on the abstract level; weakly, because τ-steps are ignored in the simulation. To make the parallel between simulation and subject reduction more visible, we write $\Delta_1 \xrightarrow{\rho.\mathtt{lock}} \Delta_2$ for $\Delta_2 = \Delta_1 \oplus \rho$ (and analogously for unlocking).

Lemma 1 (Subject reduction). *Assume $\Gamma \vdash P \parallel p\langle t_1\rangle :: \Phi \parallel p\langle\Delta_1 \to \Delta_2; C_1\rangle$, and furthermore $\theta \models C_1$ for some ground substitution and $\sigma_1 \models \theta\Delta_1$ and $\sigma_1 \models \Phi$.*

1. *$\sigma_1 \vdash P \parallel p\langle t_1\rangle \xrightarrow{p\langle\tau\rangle} \sigma_2 \vdash P \parallel p\langle t_2\rangle$, then $\Gamma \vdash P \parallel p\langle t_2\rangle :: \Phi \parallel p\langle\Delta_1' \to \Delta_2', C_2\rangle$ where $C_1 \vdash \Delta_1 \leq \Delta_1'$ and $C_1 \vdash \Delta_2' \leq \Delta_2$. Furthermore, $C_1 \models C_2$ and $\sigma_2 \models \theta\Delta_1$ and $\sigma_2 \models \Phi$.*

2. *$\sigma_1 \vdash P \parallel p\langle t_1\rangle \xrightarrow{p\langle \rho.\texttt{lock}\rangle} \sigma_2 \vdash P \parallel p\langle t_2\rangle$, then $\Gamma \vdash P \parallel p\langle t_2\rangle :: \Phi \parallel p\langle \Delta_1' \to \Delta_2, C_2\rangle$ where $C_1 \vdash \Delta_1 \oplus \rho \leq \Delta_1'$ and $C_1 \vdash \Delta_2' \leq \Delta_2$. Furthermore $C_1 \models C_2$ and $\sigma_2 \models \theta\Delta_1'$ and $\sigma_2 \models \Phi$.*
3. *$\sigma_1 \vdash P \parallel p\langle t_1\rangle \xrightarrow{p\langle \rho.\texttt{unlock}\rangle} \sigma_2 \vdash P \parallel p\langle t_2\rangle$, then $\Gamma \vdash P \parallel p\langle t_2\rangle :: \Phi \parallel p\langle \Delta_1' \to \Delta_2, C_2\rangle$ where $C_1 \vdash \Delta_1 \ominus \rho \leq \Delta_1'$ and $C_1 \vdash \Delta_2' \leq \Delta_2$. Furthermore $C_1 \models C_2$ and $\sigma_2 \models \theta\Delta_1'$ and $\sigma_2 \models \Phi$.*

The property of the lemma is shown pictorially in Figure 1.

As an immediate consequence, all configurations reachable from a well-typed initial configuration are well-typed itself. In particular, for all those reachable configurations, the corresponding pre-condition (together with the constraints) is a sound over-approximation of the actual lock counters in the heap.

Corollary 1 (Soundness of the approximation). *Let $\sigma_0 \vdash p\langle t_0\rangle$ be an initial configuration. Assume further $\Gamma \vdash p\langle t_0\rangle :: p\langle \Delta_0 \to \Delta_2; C\rangle$ and $\theta \models C$ and where Δ_0 is the empty context. If $\sigma_0 \vdash p\langle t_0\rangle \to^* \sigma \vdash P$, then $\Gamma \vdash P :: \Phi$, where $\Phi = p_1\langle \Delta_1 \to \Delta_1'; C_1\rangle \parallel \ldots \parallel p_k\langle \Delta_k \to \Delta_k'; C_k\rangle$ and where $\sigma \models \theta\Delta_i$ (for all i).*

4 Race variables for deadlock detection

Next we use the information inferred by the type system in the previous section to locate control points in a program which potentially give rise to a deadlock. As we transform the given program after analyzing it, for improved precision, we assume that in the following all non-recursive function applications are instantiated/ inlined: a unique call-site per function ensures the most precise type- and effect information for that function, and correspondingly the best suitable instrumentation. The polymorphic type system gives a context-sensitive representation, which can then be instantiated per call-site. Note that this way, we need to analyze only the original program, and each function in there once, although for the next step, we duplicate methods. Recursive functions are instantiated once with (minimal) effects capturing all call-sites.

Those points are instrumented appropriately with assignments to additional shared variables, intended to flag a race. To be able to do so, we slightly need to extend our calculus. The current formulation does not have shared variables, as they are irrelevant for the analysis of the program, which concentrates on the locks. In the following we assume that we have appropriate syntax for accessing shared variables; we use $z, z', z_1, \ldots$

$$\begin{tikzcd}
\Delta \arrow[r, dash, "="] & \Delta \\
\sigma_1 \vdash p\langle t_1\rangle \arrow[u, "\models \theta"] \arrow[r, "p\langle \tau\rangle"] & \sigma_2 \vdash p\langle t_2\rangle \arrow[u, dotted, "\models \theta"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\Delta_1 \arrow[r, dotted, "\rho.\texttt{lock}"] & \Delta_1' \\
\sigma_1 \vdash p\langle t_1\rangle \arrow[u, "\models \theta"] \arrow[r, "p\langle \rho.\texttt{lock}\rangle"] & \sigma_2 \vdash p\langle t_2\rangle \arrow[u, dotted, "\models \theta"']
\end{tikzcd}$$

Fig. 1. Subject reduction (case of unlocking analogous)

to denote shared variables, to distinguish them from the let-bound thread-local variables x and their syntactic variants. For simplicity, we assume that they are statically and globally given, i.e., we do not introduce syntax to declare them. Together with the lock references, their values are stored in σ. To reason about changes to those shared variables, we introduce steps of the form $\xrightarrow{p\langle !z\rangle}$ and $\xrightarrow{p\langle ?z\rangle}$, representing write resp. read access of process p to z. Alternatives to using a statically given set of shared variables, for instance using dynamically created pointers to the heaps are equally straightforward to introduce syntactically and semantically, without changing the overall story.

4.1 Deadlocks and races

We start by formally defining the notion of deadlock used here, which is fairly standard (see also [16]): a program is deadlocked, if a number of processes are cyclically waiting for each other's locks.

Definition 1 (Waiting for a lock). *Given a configuration $\sigma \vdash P$, a process p* waits *for a lock l in $\sigma \vdash P$, written as waits$(\sigma \vdash P, p, l)$, if (1) it is not the case that $\sigma \vdash P \xrightarrow{p\langle l.\mathtt{lock}\rangle}$, and furthermore (2) there exists σ' s.t. $\sigma' \vdash P \xrightarrow{p\langle l.\mathtt{lock}\rangle} \sigma'' \vdash P'$. In a situation without (1), we say that in configuration $\sigma \vdash P$, process p* tries *for lock l (written tries$(\sigma \vdash P, p, l)$).*

Definition 2 (Deadlock). *A configuration $\sigma \vdash P$* is deadlocked *if $\sigma(l_i) = p_i(n_i)$ and furthermore waits$(\sigma \vdash P, p_i, l_{i+_k 1})$ (where $k \geq 2$ and for all $0 \leq i \leq k-1$). The $+_k$ is meant as addition modulo k. A configuration $\sigma \vdash P$* contains a deadlock*, if, starting from $\sigma \vdash P$, a deadlocked configuration is reachable; otherwise it is deadlock free.*

Thus, a process can only be deadlocked, i.e., being part of a deadlocked configuration, if *p holds* at least one lock already, and is *waiting* for another one. With re-entrant locks, these two locks must be different. Independent from whether it leads to a deadlock or not, we call such a situation —holding a lock and attempting to acquire another one— a *second lock point.* More concretely, given a configuration, where we abbreviate the situation where process p holds lock l_1 and *tries* l_2 by $slp(\sigma \vdash P)_p^{l_1 \to l_2}$. The abstraction in the analysis uses program points π to represent concrete locks, and the goal thus is to detect in an approximate manner cycles using those abstractions π. As stated, a concrete deadlock involves a cycle of processes and locks. We call an *abstract cycle* Δ_C a sequence of pairs $\vec{p}{:}\vec{\pi}$ with the interpretation that p_i is holding π_i and wants π_{i+1} (modulo the length of the cycle). Next we fix the definition for being a second lock point. At run-time a process is at a second lock point simply if it holds a lock and tries to acquire a another, different one.

Definition 3 (Second lock point (runtime)). *A local configuration $\sigma \vdash p\langle t\rangle$ is at a second point (holding l_1 and attempting l_2, when specific), written $slp(\sigma \vdash p\langle t\rangle)^{l_1 \to l_2}$, if $\sigma(l_1) = p(n)$ and tries$(\sigma \vdash p\langle t\rangle, l_2)$. Analogously for abstract locks and heaps over those: $slp(\sigma \vdash p\langle t\rangle)^{\pi_1 \to \pi_2}$, if $\sigma(\pi_1) = p(n)$ and tries$(\sigma \vdash p\langle t\rangle, \pi_2)$. Given an abstract cycle Δ_C a local configuration is at a second lock point of Δ_C, if $slp(\sigma \vdash p\langle t\rangle)^{\pi_1 \to \pi_2}$ where, as specified by Δ_C, p holds π_1 and wants π_2. Analogously we write for global configurations e.g., $slp(\sigma \vdash P)_p^{\pi_1 \to \pi_2}$, where p is the identity of a thread in P.*

Ultimately, the purpose of the static analysis is to derive (an over-approximation of the) second lock points as a basis to instrument with race variables. The type system works thread-locally, i.e., it derives potential second lock points *per thread*. Given a static thread, i.e., an expression t without run-time syntax, second lock points are control points where the static analysis derives the danger of attempting a second lock. A control-point in a thread t corresponds to the *occurrence* of a sub-expression; we write $t[t']$ to denote the occurrence of t' in t. As usual, occurrences are assumed to be unique.

Definition 4 (Second lock point (static)). *Given a static thread $t_0[t]$, a process identifier p and $\Delta_0 \vdash_p t_0 : \Delta$, where $\Delta_0 = \bullet$. The occurrence of t in t_0 is a* static slp *if:*

1. $t = \texttt{let}\, x{:}\texttt{L}^{\{\ldots,\pi,\ldots\}} = v.\, \texttt{lock in}\, t'$.
2. $\Delta_1 \vdash_p t :: \Delta_2$*, for some Δ_1 and Δ_2, occurs in a sub-derivation of $\Delta_0 \vdash t_0 :: \Delta$.*
3. *there exists $\pi' \in \Delta_1$ s.t. $\Delta_C \vdash p$ has π', and $\Delta_C \vdash p$ wants π .*

Assume further $\sigma_0 \vdash p\langle t_0\rangle \longrightarrow^* \sigma \vdash p\langle t\rangle$. We say $\sigma \vdash p\langle t\rangle$ is at a static second lock point if t occurs as static second lock point in t_0.

Lemma 2 (Static overapproximation of slp's). *Given Δ_C and $\sigma \vdash P$ be a reachable configuration where $P = P' \parallel p\langle t\rangle$ and where furthermore the initial state of p is $p\langle t_0\rangle$. If $\sigma \vdash p\langle t\rangle$ is at a dynamic slp (wrt. Δ_C), then t is a static slp (wrt. Δ_C).*

Proof. A direct consequence of soundness of the type system (cf. Corollary 1). $\square$

Next we define the notion of *race*. A race manifests itself, if at least two processes in a configuration attempt to access a shared variables at the same time, where at least one access is a write-access.

Definition 5 (Race). *A configuration $\sigma \vdash P$ has a (manifest)* race*, if $\sigma \vdash P \xrightarrow{p_1\langle !x\rangle}$, and $\sigma \vdash P \xrightarrow{p_2\langle !x\rangle}$ or $\sigma \vdash P \xrightarrow{p_2\langle ?x\rangle}$, for two different p_1 and p_2. A configuration $\sigma \vdash P$ has a race if a configuration is* reachable *where a race manifests itself. A program has a race, if its initial configuration has a race; it is race-free else.*

Race variables will be added to a program to assure that, if there is a deadlock, also a race occurs. More concretely, being based on the result of the static analysis, appropriate race variables are introduced for each *static* second lock points, namely immediately preceding them. Since static lock points over-approximate the dynamic ones and since being at a dynamic slp is a necessary condition for being involved in a deadlock, that assures that no deadlock remains undetected when checking for races. In that way, that the additional variables "protect" the second lock points.

Definition 6 (Protection). *A property ψ is protected by a variable z starting from configuration $\sigma \vdash p\langle t\rangle$, if $\sigma \vdash p\langle t\rangle \to^* \xrightarrow{a} \sigma' \vdash p\langle t'\rangle$ and $\psi(p\langle t'\rangle)$ implies that $a = !z$. We say, ψ is protected by z, if it is protected by z starting from an arbitrary configuration.*

Protection, as just defined, refers to a property and the execution of a single thread. For race checking, it must be assured that the local properties are protected by the same, i.e., shared variable are necessarily and commonly reached. That this is the case is formulated in the following lemma:

Lemma 3 (Lifting). *Assume two processes $p_1\langle t_1\rangle$ and $p_2\langle t_2\rangle$ and two thread-local properties φ_1 and φ_2 (for p_1 and p_2, respectively). If ψ_1 is protected by x for $p_1\langle t_1\rangle$ and ψ_2 for $p_2\langle t_2\rangle$ by the same variable, and a configuration $\sigma \vdash P$ with $P = p_1\langle t_1\rangle \parallel p_2\langle t_2\rangle \parallel P''$ is reachable from $\sigma' \vdash P'$ such that $\psi_1 \wedge \psi_2$ holds, then $\sigma' \vdash P'$ has a race.*

4.2 Instrumentation

Next we specify how to transform the program by adding race variables. The idea is simple: each static second lock point, as determined statically by the type system, is instrumented by an appropriate race variable, adding it in front of the second lock point. In general, to try to detect different potential deadlocks at the same time, different race variables may be added simultaneously (at different points in the program). The following definition defines where to add a race variable representing one particular cycle of locks Δ_C. Since the instrumentation is determined by the static type system, one may combine the derivation of the corresponding lock information by the rules of Table 5 such that the result of the derivation not only derives type and effect information, but also transforms the program at the same time, with judgments of the form $\Gamma \vdash t \rhd t' : \hat{T} :: \varphi$, where t is transformed to t'. Note that we assume that a solution to the *constraint set has been determined and applied* to the type and the effects. Since the only control points in need of instrumentation are where a lock is taken, the transformation for all syntactic constructs is trivial, leaving the expression unchanged, except for $v.\texttt{lock}$-expressions, where the additional assignment is added if the condition for static slp is satisfied (cf. equation (7) from Definition 4).

Definition 7 (Transformation). *Given an abstract cycle Δ_C. For a process p from that cycle, the control points instrumented by a $!z$ are defined as follows:*

$$\frac{\dfrac{\Gamma \vdash v : \texttt{L}^r :: \Delta_1 \to \Delta_1 \quad \Delta' = \Delta_1 \oplus r \quad \pi \in r \quad \pi' \in \Delta_1 \quad \Delta_C \vdash p \textit{ wants } \pi \quad \Delta_C \vdash p \textit{ has } \pi'}{\Gamma \vdash v.\texttt{lock} : \texttt{L}^r :: \Delta_1 \to \Delta_2} \qquad \Gamma, x{:}\texttt{L}^r \vdash t \rhd t' : T :: \Delta_2 \to \Delta_3}{\Gamma \vdash \texttt{let}\, x{:}T = v.\texttt{lock}\,\texttt{in}\, t \ \rhd \texttt{let}\, x{:}T = (!z; v.\texttt{lock})\,\texttt{in}\, t' : T :: \Delta_1 \to \Delta_3}$$

By construction, the added race variable protects the corresponding static slp, and thus, ultimately the corresponding dynamic slp's, as the static ones over-approximate the dynamic ones.

Lemma 4 (Race variables protect slp's). *Given a cycle Δ_C and a corresponding transformed program. Then all static second lock points in the program are protected by the race variable (starting from the initial configuration).*

The next lemma shows that there is a race "right in front of" a deadlocked configuration for a transformed program.

Lemma 5. *Given an abstract cycle Δ_C, and let P_0 be a transformed program according to Definition 7. If the initial configuration $\sigma_0 \vdash P_0$ has a deadlock wrt. Δ_C, then $\sigma_0 \vdash P_0$ has a race.*

Proof. By the definition of deadlock (cf. Definition 2), some deadlocked configuration $\sigma' \vdash P'$ is reachable from the initial configuration:

$$\sigma_0 \vdash P_0 \longrightarrow^* \sigma' \vdash P' \quad \text{where} \quad P' = \ldots p_i\langle t'_i\rangle \parallel \ldots \parallel p_j\langle t'_j\rangle \parallel \ldots\ , \tag{2}$$

where by assumption, the processes p_i and the locks they are holding, resp. on which they are blocked are given by Δ_C, i.e., $\sigma(l_i) = p_i(n_i)$ and $waits(\sigma' \vdash P', p_i, l_{i+_k 1})$. Clearly, each participating process $\sigma' \vdash p_i\langle t'_i\rangle$ is at a *dynamic* slp (cf. Definition 3). Since those are over-approximated by their static analogues (cf. Lemma 2), the occurrence of t'_i in t^0_i resp. of t'_j in t^0_j is a *static* slp. By Lemma 4, all static slp (wrt. the given cycle) are protected, starting from the initial configuration, by the corresponding race variable. This together with the fact that $\sigma' \vdash p_i\langle t'_i\rangle$ is reachable from $\sigma_0 \vdash p_i\langle t^0_i\rangle$ implies that the static slp in each process p_i is protected by the same variable x. Hence, by Lemma 3, $\sigma_0 \vdash P_0$ has a race between p_i and p_j. □

The previous lemma showed that the race variables are added at the "right places" to detect deadlocks. Note, however, that the property of the lemma was formulated for the transformed program while, of course, we intend to detect deadlocks in the original program. So to use the result of Lemma 5 on the original program, we need to convince ourselves that the transformation does not change (in a relevant way) the behavior of the program, in particular that it neither introduces nor removes deadlocks. Since the instrumentation only adds variables which do not influence the behavior, this preservation behavior is obvious. The following lemma shows that transforming programs by instrumenting race variables preserves behavior.

Lemma 6 (Transformation preserves behavior). *P is deadlock-free iff P^T is deadlock-free, for arbitrary programs.*

Next, we state that with the absence of data race in a transformed program, the corresponding original one is deadlock-free:

Lemma 7 (Data races and deadlocks). *P is deadlock-free if P^T is race-free, for arbitrary programs.*

5 Gate locks

Next we refine the transformation to improve its precision. By definition, races are inherently *binary*, whereas deadlocks in general are not, i.e., there may be more than two processes participating in a cyclic wait. In a transformed program, all the processes involved in a specific abstract cycle Δ_C share a common race variable. While sound, this would lead to unnecessarily many false alarms, because already if two processes as part of a cycle of length $n > 2$ reach simultaneously their race-variable-instrumented control-points, a race occurs, even if the cycle may never be closed by the remaining processes. In the following, we add not only race variables, but also *additional* locks, assuring that parts of a cycle do not already lead to a race; we call these locks *gate locks*. Adding new locks, however, needs to be done carefully so as not to change the behavior of the program, in particular, not to break Lemma 6.

We first define another (conceptual) use of locks, denoted *short-lived locks*. A process which is holding a short-lived lock has to first release it before trying any other lock. It is obvious to see that transforming a program by adding short-lived locks does not lead to more deadlocks.

A gate lock is a short-lived lock which is specially used to protect the access to race variables in a program. Since gate locks are short-lived, no new deadlocks will be introduced. Similar to the transformation in Definition 7, we still instrument with race variables at the static second lock points, but *also* wrap the access with locking/unlocking of the corresponding gate lock (there is one gate lock per Δ_C). However, we *pick one* of the processes in Δ_C which *only* accesses the race variable *without* the gate lock held. This transformation ensures that the picked process and exactly *one* of the other processes involved in a deadlock cycle may reach the static second lock points at the same time, and thus a race occurs. That is, only the race between the process which could close the deadlock cycle and any *one* of the other processes involved in the deadlock will be triggered.

Observe that depending on the chosen process, the race checker may or may not report a race—due to the soundness of our approach, we are obviously interested in the best result, which is "no race detected". Therefore, we suggest to run the analysis with all processes to find the optimal result. Note that checks for different cycles and with different "special" processes for the gate lock-based instrumentation can easily be run in parallel or distributed. It is also possible to instrument a single program for the detection of multiple cycles: even though a lock statement can be a second lock point for multiple abstract locks, the transformations for each of them do not interfere with each other, and can be analyzed in a single race checker-run.

Theorem 1. *Given a program P, P^T is a transformed program of P instrumenting with race variables and gate locks, P is deadlock-free if P^T is race-free.*

6 Conclusion

We presented an approach to statically analyze multi-threaded programs by reducing the problem of deadlock checking to data race checking. The type and effect system statically over-approximates program points, where deadlocks may manifest themselves and instruments programs with additional variables to signal a race. Additional locks are added to avoid too many spurious false alarms. We show soundness of the approach, i.e., the program is deadlock free, if the corresponding transformed program is race free.

Numerous approaches have been investigated and implemented over the years to analyze concurrent and multi-threaded programs (cf. e.g. [18] for a survey of various static analyses). Not surprisingly, in particular approaches to prevent races [3] and/or deadlocks [8] have been extensively studied for various languages and are based on different techniques. (Type-based) analyses for race detection include [1] [10] [6] [19] [13] to name a few. Partly based on similar techniques, likewise for the prevention of deadlocks are [21] [14]. Static detection of potential deadlocks is a recurring topic: traditionally, a lock-analysis is carried out to discover whether the locks can be *ordered*, such that subsequent locks can only be acquired following that order [4]. Then, a deadlock is immediately ruled out as this construction precludes any "deadly embrace". The lock order may be specified by the user, or inferred [5]. To the best of our knowledge, our contribution is the first formulation of (potential) deadlocks in terms of data races. Due to the number of race variables introduced in the transformation, and assuming that race checking scales linearly in their number, we expect an efficiency comparable to explicit-state model checking.

In general, races are prevented not just by protecting shared data via locks; a good strategy is to avoid also shared data in the first place. The biggest challenge for static analysis, especially when insisting on soundness of the analysis, is to achieve better approximations as far as the danger of shared, concurrent access is concerned. Indeed, the difference between an overly approximate analysis and one that is usable in practice lies not so much in obtaining more refined conditions for races as such, but to get a grip on the imprecision caused by aliasing, and the same applies to static deadlock prevention.

Future work A natural extension of our work would be an implementation of our type and effect system to transform concurrent programs written in e.g. C and Java. Complications in those languages like aliasing need to be taken into account, although results from a *may-alias* analysis could directly be consumed by our analysis. The potential blowup of source code-size through instantiation of function applications can be avoided by directly making use of context in the race-checker, instead of working on a source-based transformed program. As a first step, we intend to make our approach more applicable, to directly integrate the transformation-phase into *Goblint*, so that no explicit transformation of C programs needs to take place.

For practical applications, our restriction on a fixed number of processes will not fit every program, as will the required static enumeration of abstract cycle information. We presume that our approach will work best on code found e.g. in the realm of embedded system, where generally a more resource-aware programming style means that threads and other resources are statically allocated.

For lack of space, most of the proofs have been omitted here. Further details can be found in the accompanying technical report [17].

Acknowledgements We are grateful for detailed discussion of *Goblint* to Kalmer Apinis, and Axel Simon, from TU München, Germany.

References

1. M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.
2. T. Amtoft, H. R. Nielson, and F. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.
3. N. E. Beckman. A survey of methods for preventing race conditions. Available at `http://www.nelsbeckman.com/publications.html`, May 2006.
4. A. D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Research Center, 1989.
5. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '02 (Seattle, USA)*. ACM, Nov. 2002. In *SIGPLAN Notices*.

6. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '01*. ACM, 2001.
7. E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.
8. J. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, Mar. 1996.
9. E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, 1965. Reprinted in [12].
10. C. Flanagan and S. Freund. Type inference against races. In *Proceedings of SAS '04*, volume 3148 of *Lecture Notes in Computer Science*, pages 116–132. Springer-Verlag, 2004.
11. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM, June 1993. In *SIGPLAN Notices* 28(6).
12. F. Genyus. *Programming Languages*. Academic Press, 1968.
13. D. Grossman. Type-safe multithreading in Cyclone. In *TLDI'03: Types in Language Design and Implementation*, pages 13–25. ACM, 2003.
14. N. Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.
15. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *ACM Conference on Programming Language Design and Implementation (PLDI) (Ottawa, Ontario, Canada)*, pages 308–319. ACM, June 2006.
16. K. I. Pun, M. Steffen, and V. Stolz. Deadlock checking by a behavioral effect system for lock handling. *Journal of Logic and Algebraic Programming*, 81(3):331–354, Mar. 2012. A preliminary version was publiched as University of Oslo, Dept. of Computer Science Technical Report 404, March 2011.
17. K. I. Pun, M. Steffen, and V. Stolz. Deadlock checking by data race detection. Technical report 421, University of Oslo, Dept. of Informatics, Oct. 2012.
18. M. Rinard. Analysis of multithreaded programs. In P. Cousot, editor, *Proceedings of the 8th International Static Analysis Symposium, SAS '01*, volume 2126 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001.
19. A. Sasturkar, R. Agarwal, L. Wang, and S. Stoller. Automated type-based analysis of data races and atomicity. In J. Ferrante, D. A. Padua, and R. L. Wexelblat, editors, *PPoPP'05*, pages 83–94. ACM, 2005.
20. H. Seidl and V. Vojdani. Region analysis for race detection. In J. Palsberg and Z. Su, editors, *Proceedings of SAS '09*, volume 5673 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 2009.
21. V. Vasconcelos, F. Martins, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In A. R. Beresford and S. J. Gay, editors, *Pre-Proceedings of the Workshop on Programming Language Approaches to Concurrenct and Communication-Centric Software (PLACES 2009)*, volume 17 of *EPTCS*, pages 95–109, 2009.