Deadlock Checking by Data Race Detection[☆]

Ka I Pun, Martin Steffen, Volker Stolz

Dept. of Informatics, University of Oslo, Norway

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 analyse 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 analyses the data flow in connection with lock handling to find out control-points which 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.

Keywords: deadlock detection, race detection, type and effect system, concurrency, formal method

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

[☆]Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`) and the ARV grant of the Macao Science and Technology Development Fund.

Email addresses: `violet@ifi.uio.no` (Ka I Pun), `msteffen@ifi.uio.no` (Martin Steffen), `stolz@ifi.uio.no` (Volker Stolz)

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* [11] and *deadlocks* [19]. 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 [16]. 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. Our target language has explicit locks, i.e. we address *non-block structured* locking, and we can certify programs as safe which cannot be certified by approaches that use a static lock order (see Section 7 on related work).

The analysis consists of two phases. The first phase statically calculates information about lock usages per thread. Since deadlocks are a global phenomenon, i.e., involving more than one thread, the derived information is used in the second phase to instrument the program with additional variables to signal a race at control points that potentially are involved in a deadlock. The formal type and effect system for lock information in the first phase uses a constraint based flow analysis as proposed by [33]. The effects, using the flow information, capture in an approximate manner on how often different locks are being held and is likewise formulated using constraints. This information roughly corresponds to the notion of lock-sets in that at each point in the program, the analysis gives approximate information which locks are held. In the presence of re-entrant locks, an upper bound on how many times the locks are being held is given, which corresponds to a "may"-over-approximation. In contrast, the notion of lock-sets as used in many

race-freedom analyses, represents sets of locks which are necessarily held, which dually corresponds to a "must"-approximation.

Despite the fact that races, in contrast to deadlocks, are binary global concurrency errors 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 larger than 2, the transformation adds additional gate locks to check possible interleavings to a race (deadlock) pairwise.

Our approach widens the applicability of freely available state-of-the-art static race checkers: *Goblint* [45] 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* [34] 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 organized as follows. Section 2 presents syntax and operational semantics of the calculus. Afterwards, Section 4 formalizes the data flow analysis in the form of a (constraint-based) effect system. The obtained information is used in Sections 5 and 6 to instrument the program with race variables and additional locks. The sections also prove the soundness of the transformation. We conclude in Section 7 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 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.

P	::=	$\emptyset \mid p\langle t\rangle \mid P \parallel P$	program
t	::=	v	value
	$\mid$	$\texttt{let}\ x{:}T = e\ \texttt{in}\ t$	local variables and sequ. composition
e	::=	t	thread
	$\mid$	$v\ v$	application
	$\mid$	$\texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e$	conditional
	$\mid$	$\texttt{spawn}\ t$	spawning a thread
	$\mid$	$\texttt{new L}$	lock creation
	$\mid$	$v.\,\texttt{lock}$	acquiring a lock
	$\mid$	$v.\,\texttt{unlock}$	releasing a lock
v	::=	x	variable
	$\mid$	l^r	lock reference
	$\mid$	$\texttt{true} \mid \texttt{false}$	truth values
	$\mid$	$\texttt{fn}\ x{:}T.t$	function abstraction
	$\mid$	$\texttt{fun}\ f{:}T.x{:}T.t$	recursive function abstraction

Table 1: Abstract syntax

Expressions, as said, are given by e, and threads count among 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 returns a reference to it (the 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 $\texttt{fun}\ f{:}T_1.x{:}T_2.t$ 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* [25]. Obviously, the more "general" expressions like $e_1\ e_2$ or $e.\,\texttt{lock}$ etc. can straightforwardly be transformed into a-normal form, by adding local variables, in case of the application, e.g., by writing $\texttt{let}\ x_1 = e_1\ \ \texttt{in}\ \ (\texttt{let}\ x_2 = e_2\ \texttt{in}\ x_1\ x_2)$. We use this representation to slightly simplify the formulation of the operational semantics and in particular of the type systems, without sacrificing expressivity.

The grammar for types and type schemes, effects, and annotations is given Table 2, where π represents labels (used to label program points where locks

Y	$::= \rho \mid X$	type-level variables
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	$::= \bullet \mid X \mid \Delta, r{:}n \mid \Delta \oplus \Delta \mid \Delta \ominus \Delta$	lock env./abstract state
C	$::= \emptyset \mid \rho \sqsupseteq r,\, C \mid X \geq \Delta, C$	constraints

Table 2: Types

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 lock sets 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 $T, T_1, T_2, \ldots$ for the underlying types, and $\hat{T}$ and its syntactic variants for the annotated types, as given in the grammar. 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 environments) Δ are of the form $r_0{:}n_0, r_1{:}n_1, \ldots$. We use X for variables representing lock environments. 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 is an integer value including ∞ and $-\infty$. 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 $\bullet$. Furthermore, lock environments can be formed using $\oplus$ and $\ominus$. The definition of these binary operators will be given later (cf. Definition 3.1). Constraints C finally are finite sets of subset inclusions of

the form $\rho \sqsupseteq r$ and of constraints of the form $X \geq \Delta$. To allow a context-sensitive analysis we use type schemes $\hat{S}$, i.e., prefix-quantified types of the form $\forall \vec{Y}{:}C.\hat{T}$, where Y are variables ρ or X.

2.1. 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

$$t_1 \rightarrow t_2 \ . \tag{1}$$

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).

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

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

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

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

$$\sigma \vdash P \rightarrow \sigma' \vdash P' \ , \tag{2}$$

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, modeling 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; for P_1 and P_2 to be composed in parallel, the $\|$-operator requires $dom(P_1)$ and $dom(P_2)$ to be disjoint, which assures global uniqueness. A new lock is created by `new L` (cf. rule R-NEWL) which allocates a fresh lock reference in the heap. Initially, the lock is free. A lock l is acquired 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.

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

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

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

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

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

Table 4: Global steps

To analyse deadlocks and races, we specify which locks are meant statically by labeling 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. To formulate properties and the corresponding proofs later, relating the semantics with the static type system, we assume further that lock references l are also annotated, i.e. of the form l^ρ; the labeling is done by the type system presented next. The labeling is, as said, for proof-theoretic purposes only and does not influence the semantics.

3. Type system

Next we present the type and effect system, which later then is turned into an algorithmic version in Section 4. The typing part included flow-annotated types for locks, which represents a basic flow analysis keeping track of where locks are created resp. where they are used. The type based flow-analysis uses constraints and basically is an adapdation of flow analysis techniques proposed in [33] (not for locks, but for a functional, higher-order calculus). Besides the flow information about lock definition and usage, *effects* take care of estimating an upper bound of the lock-counter values, which may change by locking resp. unlocking. Also this part of the static analysis is based on constraints, using known techniques (see e.g. [49] and [9]). To enhance precision, the type and effect analysis is *context-sensitive*, i.e., uses *polymorphic* types. To assure that type inference is feasible later, polymorphic types allow prefix-quantification only, i.e., are based on the well-known notion of *type schemes* and let-polymorphism.

The judgments of the type system are of the form

$$C;\Gamma \vdash e : \hat{T} :: \varphi \tag{3}$$

where φ represents $\Delta_1 \to \Delta_2$. The judgment asserts that e is of type $\hat{T}$, where for annotated lock types of the form L^ρ where ρ 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 and hence the judgement is interpreted relative to solutions of the set of constraints C.

The rules for the type system are given in Table 5. The type (scheme) of a variable is determined by its declaration in the context Γ (cf. rule T-VAR) and it has no effect, i.e., its pre- and post-condition are identical. As a general observation and as ususal, values have no effect. Also lock creation in rule T-NEWL does not have an effect. As for the flow: π labels the point of creation of the lock; hence it must be a consequence of the constraints that π is contained in the annotation ρ of the lock type, written as $C \vdash \rho \sqsupseteq \{\pi\}$ in the premise of the rule. The

case for annotated lock references l^ρ in rule T-LREF works analogously, where the constraints ensure that the annotation ρ of the lock variable is contained in the annotation ρ' of the lock type. For function abstraction in rule T-ABS$_1$, the premise checks the body e of the function with the typing context appropriately extended. Note that in the function definition, the type of the formal parameter is declared as (un-annotated) type T, the declaration is remembered in the context as the binding $x{:}\lceil T \rceil$. The operation $\lceil T \rceil$ turns all occurrences of lock types L in T into annotated counter-parts, i.e., lock types L are annotated as $\mathtt{L}^\rho$ and arrow-types $T_1 \to T_2$ are annotated to $\hat{T}_1 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_2$. The treatment of recursive functions in T-ABS$_2$ works similarly. The treatment of function application in rule T-APP is straightforward: as function and argument are values, they have themselves no effect, and the post-condition is directly taken from the function's latent effect. The treatment of conditionals is standard (cf. rule T-COND), where two types and effects of the two branches must agree with each other. For sequential composition (cf. rule T-LET), the post-condition of the first part serves as pre-condition of the second. As far as the type is concerned, the (annotated) type schene S_1 as derived for e_1 must be compatible with the type T_1 as declared. The operation $\lfloor \hat{S}_1 \rfloor$ simply erases all annotations (including quantifications of the type-schemes therefore) and gives back the corresponding un-annotated type. Spawning a thread in rule T-SPAWN has no effect, where the premise of the rule checks well-typedness of the expression being spawned. Note that for that expression, all locks are assumed to be free, assuming $\bullet$ as pre-condition. The two rules T-LOCK and T-UNLOCK deal with locking and unlocking, simply counting up, resp. down the lock counter, requiring the post-condition to be larger than $\Delta_1 \oplus (\rho{:}1)$, resp. $\Delta_1 \ominus (\rho_1{:}1)$ (cf. Definition 3.1).

Definition 3.1 (Operations on Δ)**.** *$\Delta_1 \oplus \Delta_2$ is defined point-wise, i.e., for $\Delta = \Delta_1 \oplus \Delta_2$, we have $\Delta(\rho) = \Delta_1(\rho) + \Delta_2(\rho)$, for all ρ. Remember that, for the syntactic representation of abstract states, variables which are not mentioned are assumed to be 0, e.g., for the "empty" abstract state, $\bullet(\rho) = 0$ for all ρ. The difference operation $\Delta_1 \ominus \Delta_2$ is defined analogously using $-$. We also use $\Delta \oplus \rho$ as abbreviation for $\Delta \oplus (\rho{:}1)$, analogously for $\Delta \ominus \rho$. The order on abstract states, written $\Delta_1 \leq \Delta_2$, is defined point-wise. Analogously the least upper bound $\Delta_1 \vee \Delta_2$ and the greatest lower bound $\Delta_1 \wedge \Delta_2$. Based on that, the judgment $C \vdash \Delta_1 \leq \Delta_2$ is given by the rules of Table 7.*

The two dual rules of generalization and instantiation T-GEN and T-INST introduce, resp. eliminate type *schemes.* Together with a standard rule of subsump-

tion T-SUB, these rules are not syntax-directed and need to be eliminated to obtain an algorithmic version of the analysis.

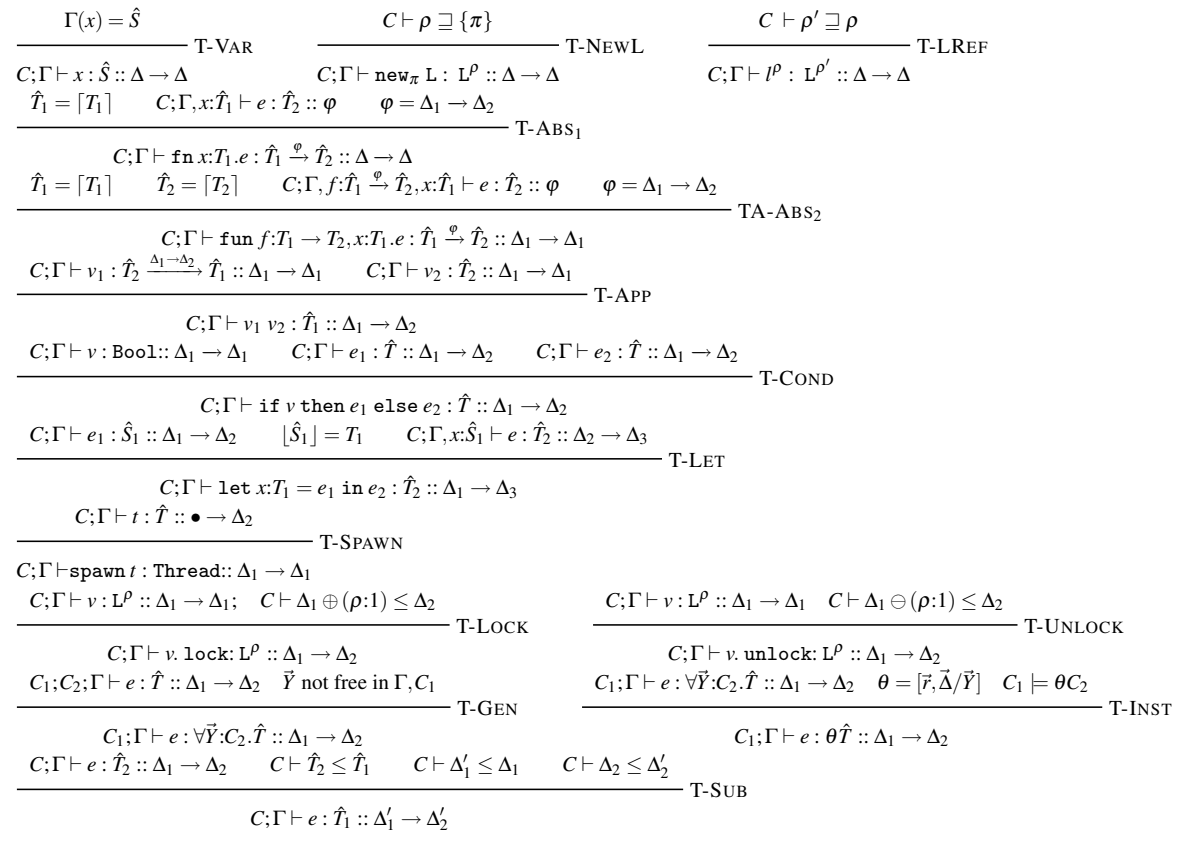

Table 5: Type and effect system

Definition 3.2 (Subtyping)**.** *The subtyping relation with judgements of the form $C \vdash \hat{T}_1 \leq \hat{T}_2$ is given inductively in Table 6:*

The typing rules from Table 5 work on the thread local level. Keeping track of the lock-counter is basically a single-threaded problem, i.e., each thread can be considered in isolation. This is a consequence of the fact that, even if shared, locks are obviously protected from interference. For subject reduction later, 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 for one

$$C \vdash \hat{T} \leq \hat{T} \quad \text{S-REFL} \qquad \frac{C \vdash \hat{T}_1 \leq \hat{T}_2 \qquad C \vdash \hat{T}_2 \leq \hat{T}_3}{C \vdash \hat{T}_1 \leq \hat{T}_3}\ \text{S-TRANS} \qquad \frac{C \models r_1 \subseteq r_2}{C \vdash \mathrm{L}^{r_1} \leq \mathrm{L}^{r_2}}\ \text{S-LOCK}$$

$$\frac{C \vdash \hat{T}_1' \leq \hat{T}_1 \qquad C \vdash \hat{T}_2 \leq \hat{T}_2' \qquad C \vdash \Delta_1' \leq \Delta_1 \qquad C \vdash \Delta_2 \leq \Delta_2'}{C \vdash \hat{T}_1 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_2 \leq \hat{T}_1' \xrightarrow{\Delta_1' \to \Delta_2'} \hat{T}_2'}\ \text{S-ARROW}$$

Table 6: Subtyping

$$C \vdash \Delta \leq \Delta \quad \text{S-REFL} \qquad \frac{C \vdash \Delta_1 \leq \Delta_2 \qquad C \vdash \Delta_2 \leq \Delta_3}{C \vdash \Delta_1 \leq \Delta_3}\ \text{S-TRANS} \qquad C, \Delta \leq X \vdash \Delta \leq X \quad \text{S-AX}$$

$$\frac{\Delta_1 \leq \Delta_2}{C \vdash \Delta_1 \leq \Delta_2}\ \text{S-BASE} \qquad \frac{C \vdash \Delta_1 \geq \bullet}{C \vdash \Delta_2 \oplus \Delta_1 \geq \Delta_2}\ \text{S-PLUS}_1 \qquad \frac{C \vdash \Delta_1 \leq \bullet}{C \vdash \Delta_2 \oplus \Delta_1 \leq \Delta_2}\ \text{S-PLUS}_2$$

$$\frac{C \vdash \Delta_1 \geq \bullet}{C \vdash \Delta_2 \ominus \Delta_1 \leq \Delta_2}\ \text{S-MINUS}_1 \qquad \frac{C \vdash \Delta_1 \leq \bullet}{C \vdash \Delta_2 \ominus \Delta_1 \geq \Delta_2}\ \text{S-MINUS}_2$$

Table 7: Order on abstract states

thread, $p\langle t \rangle : p\langle \varphi; C \rangle$, if $C \vdash t : \hat{T} :: \varphi$ for some type $\hat{T}$ (cf. Table 8). We will abbreviate $p_1\langle \varphi_1; C_1 \rangle \parallel \ldots \parallel p_k\langle \varphi_k; C_k \rangle$ by Φ. Note that for a named thread $p\langle t \rangle$ to be well-typed, the actual type $\hat{T}$ of t is irrelevant. We assume that the variables used in the constraint sets C_1 and C_2 are disjoint, and the same for φ_1 and φ_2. Under this assumption $\varphi_1 \parallel \varphi_2$ is the independent combination of φ_1 and φ_2, i.e., for $\varphi_1 = \Delta_1 \to \Delta_1'$ and $\varphi_2 = \Delta_2 \to \Delta_2'$, then their parallel combination is given by $\Delta \to \Delta'$ with Δ is the parallel combination of the functions Δ_1 and Δ_2; analogously for the post-condition. Furthermore, a running thread at the global level does not contain free variables (as the semantics is based in substitutions; cf. rule R-RED). Therefore, the premise uses an empty typing context Γ to analyse t.

$$\frac{C; \vdash t : \hat{T} :: \varphi}{\vdash p\langle t \rangle :: p\langle \varphi; C \rangle}\ \text{T-THREAD} \qquad \frac{\vdash P_1 :: \Phi_1 \qquad \vdash P_2 :: \Phi_2}{\vdash P_1 \parallel P_2 :: \Phi_1 \parallel \Phi_2}\ \text{T-PAR}$$

Table 8: Type and effect system (global)

4. Constraint generation

Next we turn the type system from Section 3 into an algorithm. To do so, it requires to get rid of the sources of non-determinism in the type system when using it an a goal-directed manner. In addition, instead of assuming a fixed set of constraints given a priori and checked at appropriate places in the derivation, constraints are generated (at those places) on the fly. The judgments of the system are now of the form

$$\Gamma \vdash e : \hat{T} :: \varphi; C \,. \tag{4}$$

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 algorithm proceeds in a syntax-directed manner and besides that generates the weakest constraints. Its rules are given in Table 9. 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 form the variable's type scheme, are the most general. The rules TA-NEWL and TA-LREF correspond to their counterparts from Table 5, except the rules now generate the corresponding constraint instead of checking the constraints against a given constraint set C. 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 L in T into their annotated counter-parts using *fresh* variables, and likewise using fresh variables to annotate the latent effect for function types. In rule T-ABS$_1$, a fresh variable is also used for the pre-condition of the function body as well as for the post-condition in the latent effect of the functional type. In rule TA-ABS$_2$, checking the body e under assuming a type for the variable f representing the recursive function generates new constraints, requiring that the type $\hat{T}_2'$ derived for the body is a subtype of the guessed return type in the latent effect; that's represented by the premise $\hat{T}_2 \geq \hat{T}_2' \vdash C_2$. Analogously for the comparison of the post-condition Δ_2 with X_2. Also 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 generating additional constraints. Furthermore, the precondition Δ of the application 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. Note that rule TA-ABS$_1$ resp. TA-ABS$_2$ ensure that Δ_1 is actually a variable. 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 $\hat{T}$ and 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. $\Delta \ominus \rho$.

As mentioned, instead of checking given constraints, the algorithm generates constraints on the fly. This is done for subtyping (corresponding to checking subtyping from Definition 3.2) and for the order-relation on lock environments (corresponding to checking that relation from Definition 3.1). The corresponding judgment can be seen as partial functions on pairs of types.

Definition 4.1 (Constraint generation)**.** *The judgment $\hat{T}_1 \leq \hat{T}_2 \vdash C$ (read as "requesting $\hat{T}_1 \leq \hat{T}_2$ generates, if satisfiable, the constraints C") is inductively given as follows:*

$$B \leq B \vdash \emptyset \quad \text{C-BASIC} \qquad \mathrm{L}^{\rho_1} \leq \mathrm{L}^{\rho_2} \vdash \{\rho_1 \sqsubseteq \rho_2\} \quad \text{C-LOCK}$$

$$\frac{\hat{T}_1' \leq \hat{T}_1 \vdash C_1 \qquad \hat{T}_2 \leq \hat{T}_2' \vdash C_2 \qquad X_1' \leq X_1 = C_3 \qquad X_2 \leq X_2' = C_4}{\hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 \leq \hat{T}_1' \xrightarrow{X_1' \to X_2'} \hat{T}_2' \vdash C_1, C_2, C_3, C_4} \text{C-ARROW}$$

For uniformity of notation,[1] *we write $\Delta \leq X \vdash C$ if $C = \Delta \leq X$.*

Note that for arrow types in C-ARROW, the latent effects are or the form $X_1 \to X_2$, resp. $X_1' \to X_2'$, i.e., formulated using variables. In the algorithm, this is assured by the introduction rules TA-ABS$_1$ and TA-ABS$_2$ for arrow types. To determine the type and the effect for conditionals, the algorithm has to determine the least upper bound of the types resp. of the post-conditions of the two branches of the conditional. This is formulated by generating appropriate constraints with the help of *fresh* variables:

[1] For abstract states, we need the definition of constraint generation only for the trivial case requiring $\Delta \leq X$.

$$\frac{\Gamma(x) = \forall \vec{Y}{:}C.\hat{T} \qquad \vec{Y}' \textit{ fresh} \qquad \theta = [\vec{Y}'/\vec{Y}]}{\Gamma \vdash x : \theta\hat{T} :: \Delta \to \Delta; \theta C}\ \text{TA-VAR}$$

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

$$\frac{\hat{T}_1 = \lceil T_1 \rceil_A \qquad \Gamma, x{:}\hat{T}_1 \vdash e : \hat{T}_2 :: X_1 \to \Delta_2; C \qquad X_1, X_2 \textit{ fresh}}{\Gamma \vdash \texttt{fn}\, x{:}T_1.e : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 :: \Delta_1 \to \Delta_1; C, X_2 \geq \Delta_2}\ \text{TA-ABS}_1$$

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

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

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

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

$$\frac{\Gamma \vdash t : \hat{T} :: \bullet \to \Delta_2; C}{\Gamma \vdash \texttt{spawn}\, t : \texttt{Thread} :: \Delta_1 \to \Delta_1; C}\ \text{TA-SPAWN}$$

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

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

Table 9: Algorithmic formulation with constraint generation

Definition 4.2 (Least upper bound)**.** *The partial operation $\vee$ on types, abstract states, and on effects, giving back a set of constraints plus a type, an abstract state, and an effect, respectively, is inductively given by the rules of Table 10. The operations $\wedge$ are defined dually.*

4.1. Equivalence of the two formulations

Before we connect the static analysis to the operational semantics, proving that it gives a static over-approximation, we show that the two alternative formulations are equivalent. Notationally, we refer to judgements and derivations in the system from Section 3 using $\vdash_s$ (for "specification") and $\vdash_a$ for the one where the constraints are generated by $\vdash_a$ (for "algorithm"). Soundness of $\vdash_a$ (wrt. $\vdash_s$) states

$$\frac{B_1 = B_2}{B_1 \vee B_2 = B_1;\emptyset}\ \text{LT-BASIC} \qquad \frac{\rho\ \mathit{fresh} \qquad \mathrm{L}^{\rho_1} \leq \mathrm{L}^{\rho} \vdash C_1 \qquad \mathrm{L}^{\rho_2} \leq \mathrm{L}^{\rho} \vdash C_2}{\mathrm{L}^{\rho_1} \vee \mathrm{L}^{\rho_2} = \mathrm{L}^{\rho};C_1,C_2}\ \text{LT-LOCK}$$

$$\frac{\hat{T}_1' \wedge \hat{T}_1'' = \hat{T};C_1 \qquad \hat{T}_2' \vee \hat{T}_2'' = \hat{T}';C_2 \qquad \varphi_1 \vee \varphi_2 = \varphi;C_3}{\hat{T}_1' \xrightarrow{\varphi_1} \hat{T}_2' \vee \hat{T}_1'' \xrightarrow{\varphi_2} \hat{T}_2'' = \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2;C_1,C_2,C_3}\ \text{LT-ARROW}$$

$$\frac{X\ \mathit{fresh} \qquad \Delta_1 \leq X \vdash C_1 \qquad \Delta_2 \leq X \vdash C_2}{\Delta_1 \vee \Delta_2 = X;C_1,C_2}\ \text{LE-STATES} \qquad \frac{\Delta_1' \wedge \Delta_1'' = \Delta_1;C_1 \qquad \Delta_2' \vee \Delta_2'' = \Delta_2;C_2}{\Delta_1 \to \Delta_2 \vee \Delta_1' \to \Delta_2' = \Delta_1 \to \Delta_2;C_1,C_2}\ \text{LE-ARROW}$$

Table 10: Least upper bound

that everything derivable in the $\vdash_a$-system is analogously derivable in the original one. We start with relating constraint checking with constraint generation in the following lemma.

Lemma 4.3 (Constraint generation)**.**

1. (a) $\hat{T}_1 \leq \hat{T}_2 \vdash C$, *then* $C \vdash \hat{T}_1 \leq \hat{T}_2$.
   (b) $\Delta \leq X \vdash C$, *then* $C \vdash \Delta \leq X$.
2. (a) *If* $C \vdash \theta\hat{T}_1 \leq \theta\hat{T}_2$, *then* $\hat{T}_1 \leq \hat{T}_2 \vdash C'$ *with* $C \models_\theta C'$.
   (b) *If* $C \vdash \theta\Delta \leq \theta X$, *then* $\Delta \leq X \vdash C'$ *with* $C \models_\theta C'$.

Proof. Straightforward. □

Lemma 4.4 (Soundness)**.** *Given* $\Gamma \vdash_a t : \hat{T} :: \Delta_1 \to \Delta_2;C$, *then* $C;\Gamma \vdash_s t : \hat{T} :: \Delta_1 \to \Delta_2$.

Proof. We are given a derivation of $\Gamma \vdash_a t : \hat{T} :: \Delta_1 \to \Delta_2;C$. The proof proceeds by straightforward induction on the derivation.

Case: TA-VAR
We are given $\Gamma \vdash_a x : \theta\hat{T} :: \Delta \to \Delta;\theta C$ where $\Gamma(x) = \hat{S} = \forall\vec{Y}{:}C.\hat{T}$ and $\theta = [\vec{Y}'/\vec{Y}]$ for some fresh variables $\vec{Y}'$. The case follows by T-VAR and T-INST:

$$\frac{\dfrac{\Gamma(x) = \hat{S}}{\theta C;\Gamma \vdash_s x : \hat{S} :: \Delta \to \Delta}\ \text{T-VAR} \qquad \theta = [\vec{Y}'/\vec{Y}] \qquad \theta C \models \theta C}{\theta C;\Gamma \vdash_s x : \theta\hat{T} :: \Delta \to \Delta}\ \text{T-INST}$$

Case: TA-NEWL
We are given $\Gamma \vdash_a \mathtt{new}_\pi\colon \mathrm{L}^{\rho} :: \Delta \to \Delta;C$ with $C = \rho \sqsupseteq \{\pi\}$ and ρ fresh. The case follows directly from T-NEWL. The case for TA-LREF works analogously.

Case: TA-ABS$_1$
We are given

$$\frac{\hat{T}_1 = \lceil T_1 \rceil_A \qquad \Gamma, x{:}\hat{T}_1 \vdash_a e : \hat{T}_2 :: X_1 \to \Delta_2; C \qquad X_1, X_2 \; \mathit{fresh}}{\Gamma \vdash_a \mathtt{fn}\, x{:}T_1.e : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 :: \Delta_1 \to \Delta_1; C'}$$

where $C' = C, X_2 \geq \Delta_2$. By induction, we get $C; \Gamma, x{:}\hat{T}_1 \vdash_s e : \hat{T}_2 :: X_1 \to \Delta_2$. Then, by strengthening the constraint set from C to C' and since $\lfloor \lceil T_1 \rceil_A \rfloor = T_1$, the case follows by T-SUB and T-ABS$_1$:

$$\dfrac{\dfrac{\dfrac{C'; \Gamma, x{:}\hat{T}_1 \vdash_s e : X_1 \to \Delta_2 \qquad C' \vdash X_2 \geq \Delta_2}{C'; \Gamma, x{:}\hat{T}_1 \vdash_s e : X_1 \to X_2}\,\text{T-SUB} \qquad \lfloor \hat{T}_1 \rfloor = T_1}{C'; \Gamma \vdash_s \mathtt{fn}\, x{:}T_1.e : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 :: \Delta_1 \to \Delta_1}\,\text{T-ABS}_1$$

Case: TA-ABS$_2$
We are given

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

where $C = C_1, C_2, C_3$. By induction and strengthening the constraint sets to C we get $C; \Gamma, f{:}\hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2, x{:}\hat{T}_1 \vdash_a e : \hat{T}_2 :: X_1 \to \Delta_2$, and since $\lfloor \lceil T_1 \rceil_A \rfloor = T_1$ and $\lfloor \lceil T_2 \rceil_A \rfloor = T_2$, the case follows by T-SUB and T-ABS$_2$

$$\dfrac{\dfrac{\dfrac{C; \Gamma, f{:}\hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2, x{:}\hat{T}_1 \vdash_s e : \hat{T}_2' :: X_1 \to \Delta_2 \quad C \vdash \hat{T}_2 \geq \hat{T}_2' \quad C \vdash X_2 \geq \Delta_2}{C; \Gamma, f{:}\hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2, x{:}\hat{T}_1 \vdash_s e : \hat{T}_2 :: X_1 \to X_2}\,\text{T-SUB}}{C; \Gamma \vdash_s \mathtt{fun}\, f{:}T_1 \to T_2, x{:}T_1.e : \hat{T}_1 \xrightarrow{X_1 \to X_2} \hat{T}_2 :: \Delta_1 \to \Delta_1}\,\text{T-ABS}_2$$

Case: TA-APP
We are given

$$\frac{\Gamma \vdash_a v_1 : \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_1 :: \Delta \to \Delta; C_1 \quad \Gamma \vdash_a v_2 : \hat{T}_2' :: \Delta \to \Delta; C_2 \quad \hat{T}_2' \leq \hat{T}_2 \vdash C_3 \quad X \; \mathit{fresh}}{\Gamma \vdash_a v_1 \; v_2 : \hat{T}_1 :: \Delta \to X; C}$$

where $C = C_1, C_2, C_3, \Delta \leq \Delta_1, \Delta_2 \leq X$. Induction on the first subgoal gives $C_1; \Gamma \vdash_s v_1 : \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_1 :: \Delta \to \Delta$. Strengthening the constraint set from C_1 to C yields

$C;\Gamma \vdash_s v_1 : \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_1 :: \Delta \to \Delta$. Similarly, induction on the second subgoal and strengthen the constraint set from C_2 to C gives $C;\Gamma \vdash_s v_2 : \hat{T}_2' :: \Delta \to \Delta$. Then, the case follows by T-SUB and T-APP:

$$\dfrac{\dfrac{C;\Gamma \vdash_s v_1 : \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_1 :: \Delta \to \Delta \quad C \vdash \Delta \le \Delta_1 \quad C \vdash \Delta_2 \le X}{C;\Gamma \vdash_s v_1 : \hat{T}_2 \xrightarrow{\Delta \to X} \hat{T}_1 :: \Delta \to \Delta}\ \text{T-SUB} \qquad \dfrac{C;\Gamma \vdash_s v_2 : \hat{T}_2' :: \Delta \to \Delta \quad C \vdash \hat{T}_2' \le \hat{T}_2}{C;\Gamma \vdash_s v_2 : \hat{T}_2 :: \Delta \to \Delta}\ \text{T-SUB}}{C;\Gamma \vdash_s v_1 v_2 : \hat{T}_1 :: \Delta \to X}$$

Case: TA-COND
In this case, we are given

$$\dfrac{\begin{array}{c} T = \lfloor \hat{T}_1 \rfloor = \lfloor \hat{T}_2 \rfloor \qquad \hat{T}; C_3 = \hat{T}_1 \vee \hat{T}_2 \qquad \Delta'; C_4 = \Delta_1 \vee \Delta_2 \\ \Gamma \vdash_a v : \texttt{Bool} :: \Delta_0 \to \Delta_0; C_0 \quad \Gamma \vdash_a e_1 : \hat{T}_1 :: \Delta_0 \to \Delta_1; C_1 \quad \Gamma \vdash_a e_2 : \hat{T}_2 :: \Delta_0 \to \Delta_2; C_2 \end{array}}{\Gamma \vdash_a \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \hat{T} :: \Delta_0 \to \Delta'; C}$$

where $C = C_0, C_1, C_2, C_3, C_4$. By induction and also stregthening the constraint sets to C,

$$C;\Gamma \vdash_s v : \texttt{Bool} :: \Delta_0 \to \Delta_0,\ C;\Gamma \vdash_s e_1 : \hat{T}_1 :: \Delta_0 \to \Delta_1 \ \text{ and } \ C;\Gamma \vdash_s e_2 : \hat{T}_2 :: \Delta_0 \to \Delta_2 \tag{5}$$

Since $C \vdash \hat{T}_1 \le \hat{T}$ and $C \vdash \hat{T}_2 \le \hat{T}$ and furthermore $C \vdash \Delta_1 \le \Delta'$ and $C \vdash \Delta_2 \le \Delta'$, we can conclude the case with subsumption and T-COND:

$$\dfrac{C;\Gamma \vdash_s v : \texttt{Bool} :: \Delta_0 \to \Delta_0 \qquad \dfrac{C;\Gamma \vdash_s e_1 : \hat{T}_1 :: \Delta_0 \to \Delta_1}{C;\Gamma \vdash_s e_1 : \hat{T} :: \Delta_0 \to \Delta'} \qquad \dfrac{C;\Gamma \vdash_s e_2 : \hat{T}_2 :: \Delta_0 \to \Delta_2}{C;\Gamma \vdash_s e_2 : \hat{T} :: \Delta_0 \to \Delta'}}{C;\Gamma \vdash_s \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \hat{T} :: \Delta_0 \to \Delta'}$$

Case: TA-LET
We are given

$$\dfrac{\begin{array}{c} \lfloor \hat{T}_1 \rfloor = T_1 \qquad \hat{S}_1 = close(\Gamma, C_1, \hat{T}_1) \\ \Gamma \vdash_a e_1 : \hat{T}_1 :: \Delta_1 \to \Delta_2; C_1 \qquad \Gamma, x{:}\hat{S}_1 \vdash_a e : \hat{T}_2 :: \Delta_2 \to \Delta_3; C_2 \end{array}}{\Gamma \vdash_a \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2 :: \Delta_1 \to \Delta_3; C_2}$$

Induction on the first subgoal gives $C_1;\Gamma \vdash_s e_1 : \hat{T}_1 :: \Delta_1 \to \Delta_2$, which gives by T-GEN $\emptyset;\Gamma \vdash_s e_1 : \hat{S}_1 :: \Delta_1 \to \Delta_2$ which further implies $C_2;\Gamma \vdash_s e_1 : \hat{S}_1 :: \Delta_1 \to \Delta_2$. This together with induction on the second subgoal concludes the case, using T-LET:

$$\dfrac{C_2;\Gamma \vdash_s e_1 : \hat{S}_1 :: \Delta_1 \to \Delta_2 \qquad C_2, \Gamma, x{:}\hat{S}_1 \vdash_s e : \hat{T}_2 :: \Delta_2 \to \Delta_3}{C_2;\Gamma \vdash_s \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2 :: \Delta_1 \to \Delta_3}$$

Case: TA-SPAWN
Straightforward.

Case: TA-LOCK
In this case we have

$$\frac{\Gamma \vdash_a v : \mathtt{L}^{\rho} :: \Delta \to \Delta; C_1 \qquad X \textit{ fresh} \qquad C_2 = X \geq \Delta \oplus (\rho{:}1)}{\Gamma \vdash_a v.\ \mathtt{lock}\colon \mathtt{L}^{\rho} :: \Delta \to X; C_1, C_2}$$

Induction on the first subgoal and strengthening the constraint set from C_1 to C_1, C_2 gives $C_1, C_2; \Gamma \vdash_s \mathtt{L}^{\rho} :: \Delta \to \Delta$. Then, the case follows by T-LOCK:

$$\frac{C_1, C_2; \Gamma \vdash_s v : \mathtt{L}^{\rho} :: \Delta \to \Delta; \qquad C_1, C_2 \vdash X \geq \Delta \oplus (\rho{:}1)}{C_1, C_2 \Gamma \vdash_s v.\ \mathtt{lock}\colon \mathtt{L}^{\rho} :: \Delta \to X}$$

The unlocking case is analogous. □

Completeness is the inverse; in general we cannot expect that the constraints generated by $\vdash_a$ are the ones used when assuming a derivation in $\vdash_s$. Since $\vdash_a$ generates as little constraints as possible, the ones given back by $\vdash_a$ are weaker, less restrictive than the ones assumed in $\vdash_s$. An analogous relationship holds for the types and the post-conditions. The sources of non-determinism in the specification are: instantiation, generalization, and weakening the result by subsumption. For the proof of completeness, we first tackle the sources of non-determinism.

As an intermediate step, we *remove* the non-determinism from Table 5 by "embedding" subsumption, and instantiation and generalization into those rules where it is necessary. To remove subsumption, we build-in the weakening into rules T-APP and T-COND; to remove instantiation and generalization, we only instantiate when we look up the type of a variable and generalize only when we put a variable into the context in the `let`-bound expression. The syntax directed version is shown in Table 11.

Definition 4.5 (Generic instance)**.** *A type scheme $\forall \vec{Y_1}{:}C_1.\hat{T}_1$ is a generic instance of $\forall \vec{Y_2}{:}C_2.\hat{T}_2$, written as $\forall \vec{Y_1}{:}C_1.\hat{T}_1 \lesssim^g \forall \vec{Y_2}{:}C_2.\hat{T}_2$, iff there exists a substitution θ where $dom(\theta) \subseteq \vec{Y_2}$ such that*

1. $C_1 \vdash \theta C_2$
2. $C_1 \vdash \theta \hat{T}_2 \leq \hat{T}_1$
3. *No y in $\vec{Y_1}$ is free in $\forall \vec{Y_2}{:}C_2.\hat{T}_2$.*

The following lemmas are used in the proof of completeness.

$$\frac{\Gamma(x) = \forall \vec{Y}{:}C'.\hat{T} \qquad \theta = [\vec{r}, \vec{\Delta}/\vec{Y}] \qquad C \models \theta C'}{C;\Gamma \vdash x : \theta\hat{T} :: \Delta \to \Delta}\ \text{T-VAR}$$

$$\frac{C \vdash \rho \sqsupseteq \{\pi\}}{C;\Gamma \vdash \texttt{new}_\pi\ \texttt{L} :\ \texttt{L}^\rho :: \Delta \to \Delta}\ \text{T-NEWL} \qquad \frac{C \vdash \rho' \sqsupseteq \rho}{C;\Gamma \vdash l^\rho :\ \texttt{L}^{\rho'} :: \Delta \to \Delta}\ \text{T-LREF}$$

$$\frac{\hat{T}_1 = \lceil T_1 \rceil \qquad C;\Gamma, x{:}\hat{T}_1 \vdash e : \hat{T}_2 :: \varphi \qquad \varphi = \Delta_1 \to \Delta_2}{C;\Gamma \vdash \texttt{fn}\, x{:}T_1.e : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \Delta \to \Delta}\ \text{T-ABS}_1$$

$$\frac{\hat{T}_1 = \lceil T_1 \rceil \qquad \hat{T}_2 = \lceil T_2 \rceil \qquad C;\Gamma, f{:}\hat{T}_1 \xrightarrow{\varphi} \hat{T}_2, x{:}\hat{T}_1 \vdash e : \hat{T}_2 :: \varphi \qquad \varphi = \Delta_1 \to \Delta_2}{C;\Gamma \vdash \texttt{fun}\, f{:}T_1 \to T_2, x{:}T_1.e : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \Delta_1 \to \Delta_1}\ \text{TA-ABS}_2$$

$$\frac{C \vdash \hat{T}_2' \le \hat{T}_2 \qquad C \vdash \Delta \le \Delta_1 \qquad C \vdash \Delta_2 \le \Delta' \qquad C;\Gamma \vdash v_1 : \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_1 :: \Delta \to \Delta \qquad C;\Gamma \vdash v_2 : \hat{T}_2' :: \Delta \to \Delta}{C;\Gamma \vdash v_1\ v_2 : \hat{T}_1 :: \Delta \to \Delta'}\ \text{T-APP}$$

$$\frac{C \vdash \hat{T}_1 \le \hat{T} \qquad C \vdash \hat{T}_2 \le \hat{T} \qquad C \vdash \Delta_1 \le \Delta' \qquad C \vdash \Delta_2 \le \Delta' \qquad C;\Gamma \vdash v : \texttt{Bool} :: \Delta \to \Delta \qquad C;\Gamma \vdash e_1 : \hat{T}_1 :: \Delta \to \Delta_1 \qquad C;\Gamma \vdash e_2 : \hat{T}_2 :: \Delta \to \Delta_2}{C;\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \hat{T} :: \Delta \to \Delta'}\ \text{T-COND}$$

$$\frac{C_1, C_2;\Gamma \vdash e_1 : \hat{T}_1 :: \Delta_1 \to \Delta_2 \qquad \vec{Y} \text{ not free in } \Gamma, C_2 \qquad C_2;\Gamma, x{:}\forall \vec{Y}{:}C_1.\hat{T}_1 \vdash e_2 : \hat{T}_2 :: \Delta_2 \to \Delta_3}{C_2;\Gamma \vdash \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2 :: \Delta_1 \to \Delta_3}\ \text{T-LET}$$

$$\frac{C;\Gamma \vdash e : \hat{T} :: \bullet \to \Delta_2}{C;\Gamma \vdash \texttt{spawn}\, e : \texttt{Thread} :: \Delta_1 \to \Delta_1}\ \text{T-SPAWN}$$

$$\frac{C;\Gamma \vdash v : \texttt{L}^\rho :: \Delta_1 \to \Delta_1; \quad C \vdash \Delta_1 \oplus (\rho{:}1) \le \Delta_2}{C;\Gamma \vdash v.\ \texttt{lock} : \texttt{L}^\rho :: \Delta_1 \to \Delta_2}\ \text{T-LOCK} \qquad \frac{C;\Gamma \vdash v : \texttt{L}^\rho :: \Delta_1 \to \Delta_1 \quad C \vdash \Delta_1 \ominus (\rho{:}1) \le \Delta_2}{C;\Gamma \vdash v.\ \texttt{unlock} : \texttt{L}^\rho :: \Delta_1 \to \Delta_2}\ \text{T-UNLOCK}$$

Table 11: Type and effect system (syntax directed)

Lemma 4.6 (Characterization of subtypes)**.** *If* $C \vdash \hat{T} \le \hat{T}_1 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_2$*, then* $\hat{T} = \hat{T}_1' \xrightarrow{\Delta_1' \to \Delta_2'} \hat{T}_2'$ *with* $C \vdash \hat{T}_1 \le \hat{T}_1'$*,* $C \vdash \hat{T}_2' \le \hat{T}_2$*,* $C \vdash \Delta_1 \le \Delta_1'$*, and* $C \vdash \Delta_2' \le \Delta_2$*.*

Proof. Immediate. □

Lemma 4.7. *If* $C \models C_1$ *and* $C \models C_2$*, then* $C \models C_1, C_2$*.*

Proof. Straightforward. □

Lemma 4.8. *Assume* $C_2, \Gamma \vdash_n \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : T_2 :: \Delta_1 \to \Delta_2$ *with* $C_1;\Gamma \vdash e_1 : \hat{T}_1 :: \Delta_1 \to \Delta_2$ *and* $C_2;\Gamma, x{:}\forall \vec{Y}{:}C_1.\hat{T}_1 \vdash_n e_2 : \Delta_2 \to \Delta_3 ::$ *as premises of* T-LET*. If* x *occurs free in* e_2*, then* $C_2 \models \theta C_1$ *for some substitution* θ*.*

Proof. Straightforward, by inspection of rule TA-VAR. □

Lemma 4.9 (Weakening (type schemes))**.** *Assume* $C;\Gamma,x{:}\hat{S}_1 \vdash e : \hat{T}_2 :: \Delta_1 \to \Delta_2$ *and* $\hat{S}_1 \lesssim^g \hat{S}'_1$*. Then,* $C;\Gamma,x{:}\hat{S}'_1 \vdash e : \hat{T}_2 :: \Delta_1 \to \Delta_2$*.*

Proof. Straightforward. □

Lemma 4.10. *Assume* $C;\Gamma \vdash_n e : \hat{T} :: \Delta_1 \to \Delta_2$ *and* $C \vdash \Delta'_1 \leq \Delta_1$*. Then,* $C;\Gamma \vdash_n e : \hat{T} :: \Delta'_1 \to \Delta'_2$ *and* $C \vdash \Delta'_2 \leq \Delta_2$*.*

Proof. Straightforward. □

Lemma 4.11 (Completeness)**.** *Assume* $\Gamma \lesssim_\theta \Gamma'$*,* $\Delta_1 \lesssim_\theta \Delta'_1$*, and* $C;\Gamma \vdash_n t : \hat{T} :: \Delta_1 \to \Delta_2$*, then* $\Gamma \vdash_a t : \hat{T}' :: \Delta'_1 \to \Delta'_2; C'$ *such that*

1. $C \models_{\theta'} C'$*,*
2. $C \vdash \theta'\hat{T}' \leq \hat{T}$*, and*
3. $C \vdash \theta'\Delta'_2 \leq \Delta_2$*,*

for some $\theta' = \theta, \theta''$*.*

Proof. Assume $C;\Gamma \vdash_n t : \hat{T} :: \Delta_1 \to \Delta_2$. The proof then proceeds by induction on the structure of e. Since the system is syntax-directed, each case corresponds to the use of exactly one rule of Table 11.

Case: $e = x$
We are given

$$\frac{\Gamma(x) = \forall\vec{Y}{:}\tilde{C}.\hat{T} \qquad C \models \tilde{\theta}\tilde{C}}{C;\Gamma \vdash_n x : \tilde{\theta}\hat{T} :: \Delta \to \Delta}$$

The assumption $\Gamma \lesssim_\theta \Gamma'$ implies $\Gamma'(x) = \forall\vec{Y}{:}\tilde{C}'.\hat{T}'$ for some $\tilde{C}'$ and $\hat{T}'$ where $\tilde{C} = \theta\tilde{C}'$ and $\hat{T} = \theta\hat{T}'$. We are furthermore given $\Delta' \lesssim_\theta \Delta$. By TA-VAR,

$$\frac{\Gamma'(x) = \forall\vec{Y}{:}\tilde{C}'.\hat{T}' \qquad \theta_2 = [\vec{Y}'/\vec{Y}] \qquad \vec{Y}' \text{ fresh} \qquad C \models \tilde{\theta}'\tilde{C}'}{\Gamma \vdash_a x : \tilde{\theta}'\hat{T} :: \Delta' \to \Delta'; \tilde{\theta}'\tilde{C}'}$$

Since $\tilde{\theta}\tilde{C} = \tilde{\theta}\theta\tilde{C}' = \tilde{\theta}\theta{\tilde{\theta}'}^{-1}\tilde{\theta}'\tilde{C}'$ and then letting $\theta' = \tilde{\theta}\theta{\tilde{\theta}'}^{-1}$, the premise $C \models \tilde{\theta}\tilde{C}$ of T-VAR can be written as $C \models_{\theta'} \tilde{\theta}'\tilde{C}'$, as required. Further $\theta'\tilde{\theta}'\hat{T}' = \tilde{\theta}\theta{\tilde{\theta}'}^{-1}\tilde{\theta}'\hat{T}' = \tilde{\theta}\theta\hat{T}' = \tilde{\theta}\hat{T}$ and hence, by reflexivity $C \vdash \theta'\tilde{\theta}'\hat{T}' \leq \tilde{\theta}'\hat{T}$, as required. Finally, likewise by reflexivity, $C \vdash \theta'\Delta' \leq \Delta'$, as required.

Case: $e = \texttt{new}_\pi\ \texttt{L}$
We are given $C;\Gamma \vdash_n \texttt{new}_\pi\ \texttt{L}\colon \texttt{L}^\rho :: \Delta \to \Delta$ where $C \vdash \rho \sqsupseteq \{\pi\}$. In the algorithm, TA-NEWL gives

$$\frac{\rho' \textit{ fresh}}{\Gamma' \vdash_a \texttt{new}_\pi\ \texttt{L} :\ \texttt{L}^{\rho'} :: \Delta' \to \Delta';\rho' \sqsupseteq \{\pi\}}$$

and with setting $\theta' = \theta, [\rho/\rho']$ the case is immediate, using reflexivity. The case for references works analogously.

Case: $e = \texttt{fn}\ x{:}T_1.e'$
We are given

$$\frac{\hat{T}_1 = \lceil T_1 \rceil \qquad C;\Gamma, x{:}\hat{T}_1 \vdash_n e' : \hat{T}_2 :: \varphi \qquad \varphi = \Delta_1 \to \Delta_2}{C;\Gamma \vdash_n \texttt{fn}\ x{:}T_1.e' : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \Delta \to \Delta}$$

and furthermore $\Gamma \lesssim_\theta \Gamma'$ and $\Delta \lesssim_\theta \Delta'$. Now let $\hat{T}_1' = \lceil T_1 \rceil_A$, i.e., an annotation of T_1 using fresh variables and let also X_1 be fresh. Thus $\hat{T}_1 = \theta_1 \hat{T}_1'$ and $\Delta_1 = \theta_1 X_1$ where $dom(\theta_1) = fv(\hat{T}_1') \cup \{X_1\}$. Now let $\tilde{\theta} = \theta, \theta_1$ and hence

$$\Gamma, x{:}\hat{T}_1 \lesssim_{\tilde{\theta}} \Gamma', x{:}\hat{T}_1' \quad \text{and} \quad \Delta_1 \lesssim_{\tilde{\theta}} X_1\ . \tag{6}$$

Thus, by induction, $\Gamma', x{:}\hat{T}_1' \vdash_a e' : \hat{T}_2' :: X_1 \to \Delta_2'; C'$, where in addition

$$C \models_{\theta'} C', \quad C \vdash \theta' \hat{T}_2' \le \hat{T}_2, \quad C \vdash \theta' \Delta_2' \le \Delta_2, \quad \text{and} \quad \theta' = \tilde{\theta}, \tilde{\theta}''\ , \tag{7}$$

for some $\tilde{\theta}''$. Using TA-ABS$_1$ gives

$$\frac{\hat{T}_1' = \lceil T_1 \rceil_A \qquad \Gamma', x{:}\hat{T}_1' \vdash_a e' : \hat{T}_2' :: X_1 \to \Delta_2'; C' \qquad \varphi' = X_1 \to X_2 \qquad X_1, X_2 \textit{ fresh}}{\Gamma' \vdash_a \texttt{fn}\ x{:}T_1.e' : \hat{T}_1' \xrightarrow{\varphi'} \hat{T}_2' :: \Delta' \to \Delta'; C', X_2 \ge \Delta_2'}$$

Now the conditions 1 – 3 of the completeness formulation remain to be checked. For constraints, let $\theta'' = \theta', [\Delta_2/X_2]$, and induction (cf. equation (7)) gives $C \vdash \theta' \Delta_2' \le \Delta_2$. This implies $C \vdash \theta'' \Delta_2' \le \theta'' X_2$ which means $C \models_{\theta''} \Delta_2' \le X_2$. We have further from induction that (cf. equation (7)) $C \models_{\theta'} C'$ which means $C \models_{\theta''} C'$, and therefore $C \models_{\theta''} C', \Delta_2' \le X_2$, as required. Now, $\theta'' \hat{T}_1' = \theta' \hat{T}_1' = \tilde{\theta}, \tilde{\theta}'' \hat{T}_1' = \theta, \theta_1, \tilde{\theta}'' \hat{T}_1' = \hat{T}_1$, hence $C \vdash \theta'' \hat{T}_1' \ge \hat{T}_1$ by reflexivity. By induction, $C \vdash \theta' \hat{T}_2' \le \hat{T}_2$ (cf. again equation (7)) and since $\theta'' \hat{T}_2' = \theta' \hat{T}_2'$, that implies $C \vdash \theta'' \hat{T}_2' \le \hat{T}_2$. Since

$\theta'' X_1 = \theta' X_1 = \Delta_1$, reflexivity gives $C \vdash \theta'' X_1 \geq \Delta_1$. Since $\theta'' X_2 = \Delta_2$, again reflexivity gives $C \vdash \theta'' X_2 \leq \Delta_2$. Together that yields $C \vdash \theta''(\hat{T}_1' \xrightarrow{\varphi'} \hat{T}_2') \leq \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2$:

$$\frac{C \vdash \theta'' \hat{T}_1' \geq \hat{T}_1 \qquad C \vdash \theta'' \hat{T}_2 \leq \hat{T}_1 \qquad C \vdash \theta'' X_1 \geq \Delta_1 \qquad C \vdash \theta'' X_2' \leq \Delta_2}{C \vdash \theta''(\hat{T}_1' \xrightarrow{\varphi'} \hat{T}_2') \leq \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2}$$

as required for part 2. For part 3, $\theta'' \Delta' = \theta' \Delta' = \tilde{\theta}, \tilde{\theta}'' \Delta' = \theta, \theta_1, \tilde{\theta}'' \Delta' = \Delta$, and thus $C \vdash \theta'' \Delta' \leq \Delta$ follows by reflexivity.

Case: $e = \texttt{fun}\ f{:}T_1 \to T_2, x{:}T_1.e'$

We are given in this case

$$\frac{\hat{T}_1 = \lceil T_1 \rceil \qquad \hat{T}_2 = \lceil T_2 \rceil \qquad C; \Gamma, f{:}\hat{T}_1 \xrightarrow{\varphi} \hat{T}_2, x{:}\hat{T}_1 \vdash_n e' : \hat{T}_2 :: \varphi \qquad \varphi = \Delta_1 \to \Delta_2}{C; \Gamma \vdash_n \texttt{fun}\ f{:}T_1 \to T_2, x{:}T_1.e' : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \Delta_1 \to \Delta_1}$$

and furthermore $\Gamma \lesssim_\theta \Gamma'$ and $\Delta_1 \lesssim_\theta \Delta_1'$. Now let $\hat{T}_1' = \lceil T_1 \rceil_A$ and $\hat{T}_2' = \lceil T_2 \rceil_A$, i.e., an annotation of T_1 and T_2 using fresh variables and let also X_1 and X_2 be fresh. Thus $\hat{T}_1 = \theta_1 \hat{T}_1'$, $\theta_1 \hat{T}_2'$, $\Delta_1 = \theta_1 X_1$, and $\Delta_2 = \theta_1 X_2$ where $dom(\theta_1) = fv(\hat{T}_1') \cup fv(\hat{T}_2') \cup \{X_1\} \cup \{X_2\}$. Now let $\tilde{\theta} = \theta, \theta_1$ and hence

$$\Gamma, f{:}\hat{T}_1 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_2, x{:}\hat{T}_1 \lesssim_{\tilde{\theta}} \Gamma', f{:}\hat{T}_1' \xrightarrow{X_1 \to X_2} \hat{T}_2', x{:}\hat{T}_1' \quad \text{and} \quad \Delta_1 \lesssim_{\tilde{\theta}} X\ . \tag{8}$$

Then, by induction, $\Gamma', f{:}\hat{T}_1' \xrightarrow{X_1 \to X_2} \hat{T}_2', x{:}\hat{T}_1' \vdash_a e' : \hat{T}_2'' :: X_1 \to \Delta_2''; C_1'$, where in addition

$$C \models_{\theta'} C_1', \quad C \vdash \theta' \hat{T}_2'' \leq \hat{T}_2, \quad C \vdash \theta' \Delta_2' \leq \Delta_2, \quad \text{and} \quad \theta' = \tilde{\theta}, \tilde{\theta}''\ , \tag{9}$$

for some $\tilde{\theta}''$. By the second judgement in equation (9), $C \vdash \theta' \hat{T}_2'' \leq \tilde{\theta} \hat{T}_2' = \theta' \hat{T}_2'$. By Lemma 4.3, we get

$$\hat{T}_2'' \leq \hat{T}_2' \vdash C_2' \quad \text{and} \quad C \models_{\theta'} C_2'\ . \tag{10}$$

Furthermore, with the third judgement in equation (9), $C \vdash \theta' \Delta_2' \leq \tilde{\theta} X_2 = \theta' X_2$. By Lemma 2(2b), we get

$$C \models_{\theta'} \Delta_2' \leq X_2\ . \tag{11}$$

Now, using TA-ABS$_2$ gives

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

For part 1, the first judgement from induction (cf. equation (9)) and the second judgements from equations (10) and (11), and by Lemma 4.7 gives $C \models_{\theta'} C'_1, C'_2, C'_3$. Finally, $C \vdash \theta'(\hat{T}'_1 \xrightarrow{X_1 \to X_2} \hat{T}'_2) \leq \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2$ for part 2 and $C \vdash \theta'\Delta'_1 \leq \Delta_1$ for part 3 follows immediately by reflexivity.

Case: $e = \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2$
We are given

$$\frac{C_1; \Gamma \vdash_n e_1 : \hat{T}_1 :: \Delta_1 \to \Delta_2 \quad C_2; \Gamma, x{:}\forall\vec{Y}{:}C_1.\hat{T}_1 \vdash_n e_2 : \hat{T}_2 :: \Delta_2 \to \Delta_3}{C_2, \Gamma \vdash_n \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2 :: \Delta_1 \to \Delta_2} \tag{12}$$

where $\vec{Y} \notin fv(\Gamma, C_2)$ and furthermore $\Gamma \lesssim_\theta \Gamma'$ and $\Delta_1 \lesssim_\theta \Delta'_1$. Induction on e_1 gives $\Gamma' \vdash_a e_1 : \hat{T}'_1 :: \Delta'_1 \to \Delta'_2; C'_1$, where in addition

$$C_1 \models_{\theta_1} C'_1, \quad C_1 \vdash \theta_1\hat{T}'_1 \leq \hat{T}_1, \quad C_1 \vdash \theta_1\Delta'_2 \leq \Delta_2, \quad \text{and} \quad \theta_1 = \theta, \theta'_1\ , \tag{13}$$

for some θ'_1. Now let $\hat{S}_1 = \forall\vec{Y}{:}C_1.\hat{T}_1$ and $\hat{S}_2 = close(\Gamma', C'_1, \hat{T}'_1) = \forall\vec{Y}'{:}C'_1.\hat{T}'_1$. Since $\hat{S}_1 \lesssim^g \hat{S}_2$, the second premise of (12) can be weakened with Lemma 4.9 to

$$C_2; \Gamma, x{:}\hat{S}_2 \vdash_n e_2 : \hat{T}_2 :: \Delta_2 \to \Delta_3\ . \tag{14}$$

Assuming that x occurs free in e_2, Lemma 4.8 gives

$$C_2 \models \theta' C_1\ , \tag{15}$$

for some θ'. The case where x does not occur free in e_2 is omitted: it is simpler, since e_2 can be typed with the context Γ alone. Now, applying θ' to the third assertion from equation (13) yields $\theta'C_1 \vdash \theta'\theta_1\Delta'_2 \leq \theta'\Delta_2$, and strengthening the constraints, using equation (15), yields

$$C_2 \vdash \theta'\theta_1\Delta'_2 \leq \theta'\Delta_2\ . \tag{16}$$

Since the domain of the substitution θ' are the variables $\vec{Y}$ (bound in $\hat{S}_1$), this implies

$$C_2 \vdash \theta_1\Delta'_2 \leq \Delta_2\ . \tag{17}$$

With this inequation, we can weaken the judgement from equation (14) by strengthening the pre-condition with Lemma 4.10 into

$$C_2; \Gamma, x{:}\hat{S}_2 \vdash_n e_2 : \hat{T}_2 :: \theta_1\Delta'_2 \to \Delta''_3 \quad \text{with} \quad C_2 \vdash \Delta''_3 \leq \Delta_3\ . \tag{18}$$

The assumption $\Gamma \lesssim_\theta \Gamma'$ implies $\Gamma, x{:}\hat{S}_2 \lesssim_\theta \Gamma', x{:}\hat{S}_2$. Since $\theta_1'\Gamma' = \Gamma'$, that implies $\Gamma, x{:}\hat{S}_2 \lesssim_{\theta,\theta_1'} \Gamma', x{:}\hat{S}_2$, which means $\Gamma, x{:}\hat{S}_2 \lesssim_{\theta_1} \Gamma', x{:}\hat{S}_2$ (since with equation (13), $\theta_1 = \theta, \theta_1'$). Since furthermore by definition, $\theta_1\Delta_2' \lesssim_{\theta_1} \Delta_2'$, we can use induction on e_2, resp. on the judgement from equation (18), yielding $\Gamma', x{:}\hat{S}_2 \vdash_a e_2 : \hat{T}_2' :: \Delta_2' \to \Delta_3'; C_2'$, where in addition

$$C_2 \models_{\theta_2} C_2', \quad C_2 \vdash \theta_2\hat{T}_2' \leq \hat{T}_2, \quad C_2 \vdash \theta_2\Delta_3' \leq \Delta_3'', \quad \text{and} \quad \theta_2 = \theta, \theta_2' \,, \tag{19}$$

for some θ_2'. By TA-LET, we get, with $\hat{S}_2 = close(\Gamma', C_1', \hat{T}_1')$, as defined above,

$$\frac{\Gamma' \vdash_a e_1 : \hat{T}_1' :: \Delta_1' \to \Delta_2'; C_1' \qquad \Gamma', x{:}\hat{S}_2 \vdash_a e_2 : \hat{T}_2' :: \Delta_2' \to \Delta_3'; C_2'}{\Gamma' \vdash_a \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2' :: \Delta_1' \to \Delta_3'; C_2'} \tag{20}$$

as required, and the conditions 1 – 2 follow directly from induction (cf. equation (19)). Finally, for part 2, the second judgement in equation (18) and the third judgement in equation (19), and by transitivity gives $C_2 \vdash \theta_2\Delta_3' \leq \Delta_3$, as required.

Case: $e = v_1 v_2$

In this case we are given

$$\frac{C \vdash \hat{T}_2' \leq \hat{T}_2 \qquad C \vdash \Delta_0 \leq \Delta_1 \qquad C \vdash \Delta_2 \leq \Delta \qquad C;\Gamma \vdash_n v_1 : \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_1 :: \Delta_0 \to \Delta_0 \qquad C;\Gamma \vdash_n v_2 : \hat{T}_2' :: \Delta_0 \to \Delta_0}{C;\Gamma \vdash_n v_1\ v_2 : \hat{T}_1 :: \Delta_0 \to \Delta} \tag{21}$$

where $\Gamma \lesssim_\theta \Gamma'$ and $\Delta_0 \lesssim_\theta \Delta_0'$. Induction on v_1 yields $\Gamma' \vdash_a v_1 : \hat{T}' :: \Delta_0' \to \Delta_1'; C_1'$ where

$$C \models_{\theta_1'} C_1', \quad C \vdash \theta_1'\hat{T}' \leq \hat{T}_2 \xrightarrow{\Delta_1 \to \Delta_2} \hat{T}_1, \quad C \vdash \theta_1'\Delta_1' \leq \Delta, \quad \text{and} \quad \theta_1' = \theta, \theta_1'' \,. \tag{22}$$

As v_1 is a value, $\Delta_1' = \Delta_0'$. By the characterization of subtyping from Lemma 4.6, $\theta_1'\hat{T}' = \tilde{\hat{T}}_2 \xrightarrow{\tilde{\Delta_1} \to \tilde{\Delta_2}} \tilde{\hat{T}}_1 = \theta_1'\hat{T}_2'' \xrightarrow{\theta_1'\Delta_1'' \to \theta_1'\Delta_2''} \theta_1'\hat{T}_1''$ where

$$C \vdash \hat{T}_2 \leq \theta_1'\hat{T}_2'', \quad C \vdash \theta_1'\hat{T}_1'' \leq \hat{T}_1, \quad C \vdash \Delta_1 \leq \theta_1'\Delta_1'', \quad \text{and} \quad C \vdash \theta_1'\Delta_2'' \leq \Delta_2 \,. \tag{23}$$

Induction on v_2 yields $\Gamma' \vdash_a v_2 : \hat{T}_2''' :: \Delta_0' \to \Delta_2'; C_2'$ where

$$C \models_{\theta_2'} C_2', \quad C \vdash \theta_2'\hat{T}_2''' \leq \hat{T}_2', \quad C \vdash \theta_2'\Delta_2' \leq \Delta, \quad \text{and} \quad \theta_2' = \theta, \theta_2'' \,. \tag{24}$$

As v_2 is a value, $\Delta_2' = \Delta_0'$. Wlog. $dom(\theta_1'') \cap dom(\theta_2'') = \emptyset$. Now define $\tilde{\theta} = \theta, \theta_1'', \theta_2''$.

By transitivity, the second judgement of (24), the first premise of (21), and the first judgment of (23) give $C \vdash \theta_2' \hat{T}_2''' \leq \theta_1' \hat{T}_2''$, which implies

$$C \vdash \tilde{\theta} \hat{T}_2''' \leq \tilde{\theta} \hat{T}_2'' \ . \tag{25}$$

By Lemma 4.3, that means

$$T_2''' \leq \hat{T}_2'' \vdash C_3' \quad \text{with} \quad C \models_{\tilde{\theta}} C_3' \ . \tag{26}$$

Now, by TA-APP

$$\frac{\begin{array}{c}\hat{T}_2''' \leq \hat{T}_2'' \vdash C_3' \quad X \textit{ fresh} \\ \Gamma' \vdash_a v_1 : \hat{T}_2'' \xrightarrow{\Delta_1'' \to \Delta_2''} \hat{T}_1'' :: \Delta_0' \to \Delta_0'; C_1' \quad \Gamma' \vdash_a v_2 : \hat{T}_2''' :: \Delta_0' \to \Delta_0'; C_2'\end{array}}{\Gamma' \vdash_a v_1\ v_2 : \hat{T}_1'' :: \Delta_0' \to X; C'} \tag{27}$$

where $C' = C_1', C_2', C_3', \Delta_0' \leq \Delta_1'', \Delta_2'' \leq X$ and where the two typing premises are given by induction and the subtyping premise is covered by (26).

The second premise of (21) and the third judgment of (23) give $C \vdash \Delta_0 \leq \theta_1' \Delta_1''$, using transitivity. This implies $C \vdash \theta_1' \Delta_0' \leq \theta_1' \Delta_1''$ and thus further $C \vdash \tilde{\theta} \Delta_0' \leq \tilde{\theta} \Delta_1''$, which means $C \models_{\tilde{\theta}} \Delta_0' \leq \Delta_1''$.

Likewise with transitivity, the third premise of (21) and the fourth judgment of (23) give $C \vdash \theta_1' \Delta_2'' \leq \Delta$, and further $C \vdash \tilde{\theta} \Delta_2'' \leq \Delta$. With setting $\tilde{\theta}' = \tilde{\theta}, [\Delta/X]$, $C \models \tilde{\theta}' \Delta_2'' \leq \tilde{\theta}' X$, i.e., $C \models_{\tilde{\theta}'} \Delta_2'' \leq X$. The conditions concerning constraints C can be summed up as follows:

$$C \models_{\tilde{\theta}'} C_1', \quad C \models_{\tilde{\theta}'} C_2', \quad C \models_{\tilde{\theta}'} C_3' \quad C \models_{\tilde{\theta}'} (\Delta_0' \leq \Delta_1'') \quad C \models_{\tilde{\theta}'} (\Delta_2'' \leq X) \tag{28}$$

which means with Lemma 4.7 $C \models_{\tilde{\theta}'} C'$, as required in part 1. For part 2, $C \vdash \tilde{\theta}' \hat{T}_2'' \leq \hat{T}_1$ follows from the second judgment of equation (23). For part 3, $C \vdash \tilde{\theta}' X \leq \Delta'$ follows by reflexivity and the definition of $\tilde{\theta}'$ as $\tilde{\theta}, [\Delta'/X]$.

Case: $e =$ `if` v `then` e_1 `else` e_2
We are given

$$\frac{\begin{array}{cccc} C \vdash \hat{T}_1 \leq \hat{T} & C \vdash \hat{T}_2 \leq \hat{T} & C \vdash \Delta_1 \leq \Delta & C \vdash \Delta_2 \leq \Delta \\ C; \Gamma \vdash v : \texttt{Bool} :: \Delta_0 \to \Delta_0 & C; \Gamma \vdash e_1 : \hat{T}_1 :: \Delta_0 \to \Delta_1 & C; \Gamma \vdash e_2 : \hat{T}_2 :: \Delta_0 \to \Delta_2 & \end{array}}{C; \Gamma \vdash_n \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \hat{T} :: \Delta_0 \to \Delta} \tag{29}$$

and further $\Gamma \lesssim_\theta \Gamma'$ and $\Delta_0 \lesssim_\theta \Delta_0'$. Induction on the subterm v gives $\Gamma' \vdash_a v :$ `Bool`$:: \Delta_0' \to \Delta_0'; C_0'$, where

$$C \models_{\theta_0'} C_0', \quad C \vdash \theta_0' \,\texttt{Bool} \leq \texttt{Bool}, \quad C \vdash \theta_0' \Delta_0' \leq \Delta_0, \quad \text{and} \quad \theta_0' = \theta, \theta_0'' \ , \tag{30}$$

for some substitution θ_0''. Induction on the second subterm gives $\Gamma' \vdash_a e_1 : \hat{T}_1' :: \Delta_0' \to \Delta_1'; C_1'$, where

$$C \models_{\theta_1'} C_1', \quad C \vdash \theta_1' \hat{T}_1' \leq \hat{T}_1, \quad C \vdash \theta_1' \Delta_1' \leq \Delta_1, \quad \text{and} \quad \theta_1' = \theta, \theta_1'' , \tag{31}$$

for some substitution θ_1''. By transitivity, the second resp. third judgement of (31), and first resp. third premise of (29) gives

$$C \vdash \theta_1' \hat{T}_1' \leq \hat{T}, \quad \text{resp.} \quad C \vdash \theta_1' \Delta_1' \leq \Delta . \tag{32}$$

Similarly, induction on the last subterm e_2 gives $\Gamma' \vdash_a e_2 : \hat{T}_2' :: \Delta_0' \to \Delta_2'; C_2'$, where

$$C \models_{\theta_2'} C_2', \quad C \vdash \theta_2' \hat{T}_2' \leq \hat{T}_2, \quad C \vdash \theta_2' \Delta_2' \leq \Delta_2, \quad \text{and} \quad \theta_2' = \theta, \theta_2'' , \tag{33}$$

for some substitution θ_2''. By transitivity, the second, resp. the third judgment of (33), and the first, resp. the third premise of (29) give

$$C \vdash \theta_2' \hat{T}_2' \leq \hat{T}, \quad \text{resp.} \quad C \vdash \theta_2' \Delta_2' \leq \Delta . \tag{34}$$

By rule TA-COND, we get

$$\frac{\begin{array}{c} T = \lfloor \hat{T}_1' \rfloor = \lfloor \hat{T}_2' \rfloor \qquad \hat{T}'; C_3' = \hat{T}_1' \vee \hat{T}_2' \qquad \Delta'; C_4' = \Delta_1' \vee \Delta_2' \\ \Gamma' \vdash_a v : \texttt{Bool} :: \Delta_0' \to \Delta_0'; C_0' \quad \Gamma' \vdash_a e_1 : \hat{T}_1' :: \Delta_0' \to \Delta_1'; C_1' \quad \Gamma' \vdash_a e_2 : \hat{T}_2' :: \Delta_0' \to \Delta_2'; C_2' \end{array}}{\Gamma' \vdash_a \texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2 : \hat{T}' :: \Delta_0' \to \Delta'; C'}$$

where $C' = C_0', C_1', C_2', C_3', C_4'$ with $\hat{T}_1' \leq \hat{T}', \hat{T}_2' \leq \hat{T}' \vdash C_3'$ and $\Delta_1' \leq \Delta', \Delta_2' \leq \Delta' \vdash C_4'$. Wlog. the domains of the substitutions θ_0'', θ_1'', and θ_2'' are pairwise disjoint. By definition of $\vee$ on types, $\hat{T}'$ is annotated with *fresh* variables, and hence $(dom(\theta) \cup dom(\theta_0'') \cup dom(\theta_1'') \cup dom(\theta_2'')) \cap fv(\hat{T}') = \emptyset$. Furthermore, $\hat{T} = \tilde{\theta}' \hat{T}'$ where $dom(\tilde{\theta}') = fv(\hat{T}')$. Similarly, by the definition of $\vee$, Δ' is a fresh variable. Hence, $(dom(\theta) \cup dom(\theta_0'') \cup dom(\theta_1'') \cup dom(\theta_2'')) \cap fv(\Delta') = \emptyset$. We further know that $\Delta = \tilde{\theta}'' \Delta'$ where $dom(\tilde{\theta}'') = fv(\Delta')$. Now setting $\theta' = \theta, \theta_0'', \theta_1'', \theta_2'', \tilde{\theta}', \tilde{\theta}''$, the first judgments of (32) resp. (34) imply

$$C \vdash \theta' \hat{T}_1' \leq \theta' \hat{T}' \quad \text{resp.} \quad C \vdash \theta' \hat{T}_2' \leq \theta' \hat{T}' \tag{35}$$

By Lemma 4.3, that means

$$\hat{T}_1' \leq \hat{T}', \ \hat{T}_2' \leq \hat{T}' \vdash C_3' \quad \text{with} \quad C \models_{\theta'} C_3' . \tag{36}$$

Similary, the second judgements of (32) resp. (34) implies

$$C \vdash \theta'\Delta_1' \leq \theta'\Delta' \quad \text{resp.} \quad C \vdash \theta'\Delta_2' \leq \theta'\Delta' \tag{37}$$

That implies, again by Lemma 4.3,

$$\Delta_1' \leq \Delta',\ \Delta_2' \leq \Delta' \vdash C_4' \quad \text{with} \quad C \models_{\theta'} C_4' \,. \tag{38}$$

Furthermore, the first judgements of the equations (30), (31) and (33) gives

$$C \models_{\theta'} C_0', \quad C \models_{\theta'} C_1' \quad \text{and} \quad C \models_{\theta''} C_2' \,. \tag{39}$$

Hence, equations (39), (36) and (38) give $C \models_{\theta'} C'$, as required in part 1. For part 2 resp. 3, $C \vdash \theta'\hat{T}' \leq \hat{T}$ resp. $C \vdash \theta'\Delta' \leq \Delta$ by reflexivity, as required.

Case: $e = \texttt{spawn}\, e'$
We are given

$$\frac{C;\Gamma \vdash_n e' : \hat{T} :: \bullet \to \Delta_2}{C;\Gamma \vdash_n \texttt{spawn}\, e' : \texttt{Thread}:: \Delta_1 \to \Delta_1}$$

and further $\Gamma \lesssim_\theta \Gamma'$ and $\Delta_1 \lesssim_\theta \Delta_1'$. By induction on e', $\Gamma' \vdash_a e' : \hat{T}' :: \bullet \to \Delta_2';C'$ where additionally

$$C \models_{\theta'} C', \quad C \vdash \theta'\hat{T}' \leq \hat{T}, \quad C \vdash \theta'\Delta_2' \leq \Delta_2, \quad \text{and} \quad \theta' = \theta, \theta'' \tag{40}$$

for some substitution θ''. Applying rule TA-SPAWN gives

$$\frac{\Gamma' \vdash_a e' : \hat{T}' :: \bullet \to \Delta_2';C'}{\Gamma' \vdash_a \texttt{spawn}\, e' :\texttt{Thread} :: \Delta_1' \to \Delta_1';C'}$$

Immediately $C \models_{\theta'} C'$ by induction, and $C \vdash \texttt{Thread} \leq \texttt{Thread}$ and $C \vdash \theta'\Delta_1 \leq \Delta_1$ by reflexivity, which concludes the case.

Case: $e = v.\ \texttt{lock}$
In this case, we have

$$\frac{C;\Gamma \vdash v : \mathrm{L}^{\rho} :: \Delta_1 \to \Delta_1; \qquad C \vdash \Delta_1 \oplus (\rho{:}1) \leq \Delta_2}{C;\Gamma \vdash v.\ \texttt{lock}\colon \mathrm{L}^{\rho} :: \Delta_1 \to \Delta_2} \tag{41}$$

and further $\Gamma \lesssim_\theta \Gamma'$ and $\Delta_1 \lesssim_\theta \Delta_1'$. We get by induction that $\Gamma' \vdash_a v : \mathrm{L}^{\rho'} :: \Delta_1' \to \Delta_1'';C'$ where $\Delta_1'' = \Delta_1'$ as v is a value. In addition, we have

$$C \models_{\theta'} C', \quad C \vdash \theta'\mathrm{L}^{\rho'} \leq \mathrm{L}^{\rho} \quad C \vdash \theta'\Delta_1' \leq \Delta_1 \quad \text{and} \quad \theta' = \theta, \theta'' \tag{42}$$

By TA-LOCK, we get

$$\frac{\Gamma' \vdash_a v : \mathtt{L}^{\rho'} :: \Delta_1' \to \Delta_1'; C' \qquad X' \textit{ fresh} \qquad \Delta_1' \oplus (\rho':1) \leq X' \vdash C''}{\Gamma' \vdash_a v.\,\mathtt{lock} : \mathtt{L}^{\rho'} :: \Delta_1' \to X'; C', C''}$$

Setting $\tilde{\theta} = \theta', [\Delta_2/X]$. The second judgement of (41) gives $C \vdash \tilde{\theta}\Delta_1' \oplus (\rho':1) \leq \tilde{\theta}X'$ and therefore $C \models_{\tilde{\theta}} C''$. Then, together with the first judgement in (42) and by Lemma 4.7, we get $C \models_{\tilde{\theta}} C', C''$, as required. Then, $C \vdash \tilde{\theta}\mathtt{L}^{\rho'} \leq \mathtt{L}^{\rho}$ by induction, and the case follows by reflexivity for $C \vdash \tilde{\theta}X' \leq \Delta_2$.

It is analogous for the unlocking case. □

4.2. Soundness of the static analysis and subject reduction

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 derive 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, in particular its pre-condition, 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 variable ρ; 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 and 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 $p\langle \rho.\mathtt{unlock} \rangle$ and all other steps are labelled by τ, denoting internal steps. As a side remark: as for now, τ steps do not change the σ. Nonetheless, subject reduction in Lemma 4.13(1) is formulated in a way that mentions σ_2 as a state after the step possibly different from the state σ_1 before the step. If our language featured mutable state (apart from the lock counters), which we left out as orthogonal for the issues at hand, the more general formulation would be more adequate. Also later, when introducing race variables, which are mutable shared variables, τ-steps may change σ, and so we chose the more general formulation already here, even if strictly speaking not needed yet. 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 simula-

tion. 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 4.12 (Subject reduction (local))**.** *Assume* $C;\Gamma \vdash t_1 : \hat{T} :: \Delta_1 \to \Delta_2$ *and* $t_1 \xrightarrow{\tau} t_2$*, then* $C;\Gamma \vdash t_2 : \hat{T} :: \Delta_1 \to \Delta_2$.

Proof. Straightforward, by case analysis of the rules of Table 3. Note that the derivation steps do not change the state of the locks.

□

Constraints C are of the forms $r \sqsubseteq \rho$ and $\Delta \leq X$. We consider both kinds of constraints as independent, in particular a constraint of the form, for instance, $X \geq \Delta \oplus (\rho{:}n)$ is considered as a constraint between the abstract states, independent from solving constraints concerning 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 $\varphi_i = \Delta_i \to \Delta_i'$.

Lemma 4.13 (Subject reduction (global))**.** *Assume* $\Gamma \vdash P \parallel p\langle t_1\rangle :: \Phi \parallel p\langle\Delta_1 \to \Delta_2;C\rangle$*, and furthermore* $\theta \models C$ *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* $\Gamma \vdash P \parallel p\langle t_2\rangle :: \Phi \parallel p\langle\Delta_1 \to \Delta_2;C\rangle$ *where* $\sigma_2 \models \theta\Delta_1$ *and* $\sigma_2 \models \Phi$.

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

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

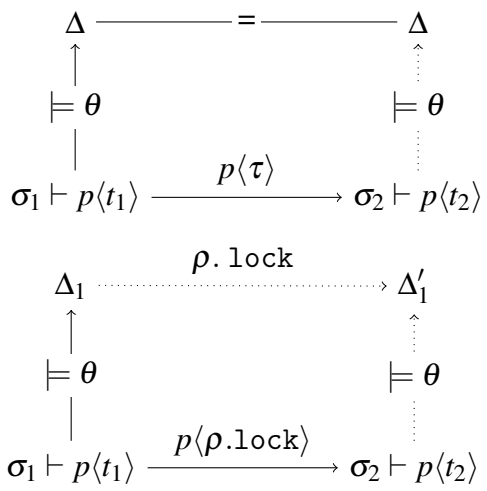

Figure 1: Subject reduction (case of unlocking analogous)

Proof. Concentrating on a single thread, assume $\Gamma \vdash p\langle t_1\rangle :: p\langle \Delta_1 \to \Delta_2; C\rangle$, and furthermore $\theta \models C_1$ and $\sigma_1 \models \theta\Delta_1$. Part 1 for τ-steps follows from subject reduction for local steps from Lemma 4.12.

For part 2 and part 3, to simplify the presentation of the proof, we make the proof wrt. the normalized system (cf. Table 11), i.e. all the sources of non-determinism are removed. The formulation of the lemma remains unchanged.

In part 2, we are given $\sigma_1 \vdash p\langle t_1\rangle \xrightarrow{p\langle\rho.\mathtt{lock}\rangle} \sigma_2 \vdash p\langle t_2\rangle$; the only rule justifying that step is R-LOCK in Table 4:

Case: R-LOCK:
$\sigma_1 \vdash p\langle \mathtt{let}\ x{:}T = l^{\rho}.\ \mathtt{lock}\ \mathtt{in}\ t\rangle \xrightarrow{p\langle\rho.\mathtt{lock}\rangle} \sigma_2 \vdash p\langle \mathtt{let}\ x{:}T = l^{\rho}\ \mathtt{in}\ t\rangle$
where $\sigma_1(l) = \mathit{free}$ or $\sigma_1(l) = p(n)$ and $\sigma_2 = \sigma_1 +_p l$. The assumption of well-

typedness and inverting rules T-THREAD, T-LET, T-LOCK, and T-LREF gives

$$\dfrac{\dfrac{\dfrac{C \vdash \rho' \sqsupseteq \rho}{C;\Gamma \vdash l^{\rho} : \mathrm{L}^{\rho'} :: \Delta_1 \to \Delta_1} \quad C \vdash \Delta_1 \oplus \rho' \leq \Delta_1'}{C;\Gamma \vdash l^{\rho}.\,\mathtt{lock} : \mathrm{L}^{\rho'} :: \Delta_1 \to \Delta_1'}\,\text{T-LOCK} \qquad C;\Gamma, x{:}\mathrm{L}^{\rho'} \vdash t : \hat{T} :: \Delta_1' \to \Delta_2}{C;\Gamma \vdash \mathtt{let}\ x{:}T = l^{\rho}.\,\mathtt{lock}\ \mathtt{in}\ t : \mathrm{L}^{\rho'} :: \Delta_1 \to \Delta_2}\,\text{T-LET}}{\Gamma \vdash p\langle \mathtt{let}\ x{:}T = l^{\rho}.\,\mathtt{lock}\ \mathtt{in}\ t\rangle :: p\langle \Delta_1 \to \Delta_2; C\rangle}$$

For the configuration after the step, applying rules T-LREF, T-LET, and T-THREAD gives:

$$\dfrac{\dfrac{\dfrac{C \vdash \rho' \sqsupseteq \rho}{C;\Gamma \vdash l^{\rho} : \mathrm{L}^{\rho'} :: \Delta_1' \to \Delta_1'}\,\text{T-LREF} \qquad C;\Gamma, x{:}\mathrm{L}^{\rho'} \vdash t : \hat{T} :: \Delta_1' \to \Delta_2}{C;\Gamma \vdash \mathtt{let}\ x{:}T = l^{\rho}\ \mathtt{in}\ t : \mathrm{L}^{\rho'} :: \Delta_1' \to \Delta_2}\,\text{T-LET}}{\Gamma \vdash p\langle \mathtt{let}\ x{:}T = l^{\rho}\ \mathtt{in}\ t\rangle :: p\langle \Delta_1' \to \Delta_2; C\rangle}$$

Given $\sigma_1 \models \theta\Delta_1$ and $\sigma_2 = \sigma_1 +_p l$ together with $C \vdash \Delta_1' \geq \Delta_1 \oplus \rho'$ and $C \vdash \rho' \sqsupseteq \rho$ gives $\sigma_2 \models \theta\Delta_1'$. Since $\sigma_2 = \sigma_1 +_p l$ means that process p is holding the lock l, and does not affect the local states of the other processes, therefore $\sigma_2 \models \Phi$, which concludes the case.

Part 3 for unlocking works analogously. □

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 4.14 (Soundness of the approximation)**.**

1. *If* $\Gamma \vdash t : \hat{T} :: \Delta_1 \to \Delta_2; C$ *and* $t \longrightarrow^* t'$*, then* $\Gamma \vdash t' : \hat{T} :: \Delta_1 \to \Delta_2; C$.
2. *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 where* $\sigma \models \theta\Delta_i$ *(for all i).*

Proof. By induction of the number of steps using Lemma 4.13. Since initially, all locks in σ_0 are free, $\sigma_0 \models \theta\Delta_0$ for all C and all $\theta \models C$. □

Next we carry over subject reduction and soundness of the type system to the algorithmic formulation. We start with subject reduction, which corresponds to Lemma 4.13 (see also Figure 1). Again we concentrate on the effect part. Since now the type system calculates the minimal effect, in particular, given a pre-condition, a minimal post-condition, reduction may lead to an stricter post-condition. Similarly for the set of constraints.

Lemma 4.15 (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* $C_1 \models C_2$ *and* $\sigma_2 \models \theta\Delta_1$ *and* $\sigma_2 \models \Phi$ *and furthermore* $C_1 \vdash \Delta_1 \le \Delta_1'$ *and* $C_1 \vdash \Delta_2' \le \Delta_2$*.*
2. $\sigma_1 \vdash P \parallel p\langle t_1\rangle \xrightarrow{p\langle \rho.\mathtt{lock}\rangle} \sigma_2 \vdash P \parallel p\langle t_2\rangle$*, then* $\Gamma \vdash P \parallel p\langle t_2\rangle :: \Phi \parallel p\langle \Delta_1' \to \Delta_2, C_2\rangle$ *where* $C_1 \vdash \Delta_1 \oplus \rho \le \Delta_1'$*. Furthermore* $C_1 \models C_2$*,* $\sigma_2 \models \theta\Delta_1'$ *and* $\sigma_2 \models \Phi$*.*
3. $\sigma_1 \vdash P \parallel p\langle t_1\rangle \xrightarrow{p\langle \rho.\mathtt{unlock}\rangle} \sigma_2 \vdash P \parallel p\langle t_2\rangle$*, then* $\Gamma \vdash P \parallel p\langle t_2\rangle :: \Phi \parallel p\langle \Delta_1' \to \Delta_2, C_2\rangle$ *where* $C_1 \vdash \Delta_1 \ominus \rho \le \Delta_1'$*. Furthermore,* $C_1 \models C_2$*,* $\sigma_2 \models \theta\Delta_1'$ *and* $\sigma_2 \models \Phi$*.*

Proof. Basically a consequence of the corresponding subject reduction Lemma 4.13 plus soundness and completeness: We are given $\Gamma \vdash_a P \parallel p\langle t_1\rangle :: \Phi \parallel p\langle \Delta_1 \to \Delta_2; C_1\rangle$, which implies by soundness from Lemma 4.4 that also $\Gamma \vdash_s P \parallel p\langle t_1\rangle :: \Phi \parallel p\langle \Delta_1 \to \Delta_2; C_1\rangle$.

In part 1, the corresponding part of Lemma 4.13 gives for the configuration after the step

$$\Gamma \vdash_s P \parallel p\langle t_2\rangle :: \Phi \parallel p\langle \Delta_1 \to \Delta_2; C_1\rangle \tag{43}$$

and furthermore $\sigma_2 \models \theta\Delta_1$ and $\sigma_2 \models \Phi$, as required. Furthermore, derivability of (43) implies with completeness from Lemma 4.11 that $\Gamma \vdash_a P \parallel p\langle t_2\rangle :: \Phi \parallel p\langle \Delta_1' \to \Delta_2'; C_2\rangle$, where $C_1 \models C_2$ and $C_1 \vdash \Delta_2' \le \Delta_2$, which discharges two further claims and concludes part 1. Parts 2 and 3 work analogously. □

5. 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 analyse 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. In this way, deadlock detection is reduced to the problem of race detection. 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 $z, z', z_1, \ldots$ 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.

5.1. Deadlocks and races

We start by formally defining the notion of deadlock used here, which is fairly standard (see also [38]): a program is deadlocked, if a number of processes are cyclically waiting for each other's locks.

Definition 5.1 (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 5.2 (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 5.3 (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 5.4 (Second lock point (static))**.** *Given a static thread $t_0[t]$, a process identifier p and $\Delta_0 \vdash t_0 : \Delta$, with $\Delta_0 = \bullet$. The occurrence of t in t_0 is a* static slp *if:*

1. $t = \texttt{let}\ x{:}\mathrm{L}^{\{\ldots,\pi,\ldots\}} = v.\ \texttt{lock}\ \texttt{in}\ t'$.

2. $\Delta_1 \vdash t :: \Delta_2$, *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 5.5 (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 4.14). □

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 5.6 (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 5.7 (Protection)**.** *A property* ψ *is protected by a variable* z *starting from configuration* $\sigma \vdash p\langle t \rangle$, *if* $\sigma \vdash p\langle t \rangle \rightarrow^* \xrightarrow{a} \sigma' \vdash p\langle t' \rangle$ *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 5.8 (Lifting)**.** *Assume two processes $p_1\langle t_1\rangle$ and $p_2\langle t_2\rangle$ and two thread-local properties ψ_1 and ψ_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.*

Proof. Straightforward. □

5.2. 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 9 such that the result of the derivation not only derives type and effect information, but transforms the program at the same time, with judgments of the form $\Gamma \vdash_p t \rhd t' : \hat{T} :: \varphi$, where t is transformed to t' in process p. 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 (5.9) from Definition 5.4).

Definition 5.9 (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:*

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

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.

Remark 5.10 (Re-entrant vs. binary locks)**.** *These definitions are applicable for both re-entrant and non-re-entrant, i.e., binary locks. Of course, "self-deadlock" where a process deadlocks in trying to acquire a lock it already has cannot be detected by adding race variables as races involve two processes. On the other hand, self-deadlocks (in a setting with binary locks) are straightforward to detect by standard techniques, compared to deadlocks involving cycles of length larger than two processes, as they can be checked thread-locally.* □

Lemma 5.11 (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).*

Proof. By construction, the transformation syntactically adds the race variable immediately in front of static second lock points. □

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

Lemma 5.12. *Given an abstract cycle Δ_C, and let P_0 be a transformed program according to Definition 5.9. 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 5.2), some deadlocked configuration $\sigma' \vdash P'$ is reachable from the initial configuration:

$$\sigma_0 \vdash P_0 \longrightarrow^* \sigma' \vdash P' \quad \text{where} \quad P' = \ldots p_i\langle t_i' \rangle \parallel \ldots \parallel p_j\langle t_j' \rangle \parallel \ldots \,, \tag{44}$$

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 5.3). Since those are over-approximated by their static analogues (cf. Lemma 5.5), the occurrence of t_i' in t_i^0 resp. of t_j' in t_j^0 is a *static* slp. By Lemma 5.11, 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 $\sigma_0 \vdash p_i\langle t_i^0 \rangle$ implies that the static slp in each process p_i is protected by the same variable x. Hence, by Lemma 5.8, $\sigma_0 \vdash P_0$ 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 5.12 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 5.13 (Transformation preserves behavior)**.** *P is deadlock-free iff P^T is deadlock-free, for arbitrary programs.*

Proof. Straightforward. □

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

Lemma 5.14 (Data races and deadlocks)**.** *P is deadlock-free if P^T is race-free, for arbitrary programs.*

Proof. A direct consequence of Lemma 5.12 and Lemma 5.13. □

In the next section, where we additionally add new locks to enhance the precision of the analysis, it becomes slightly more complex to establish that connection between the original and the transformed program.

6. 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, which removes false positives by allowing us to check reachability of slp-s pairwise. We call these locks *gate locks*. Adding new locks, however, needs to be done carefully so as not to change the behaviour of the program, in particular, not to break Lemma 5.13.

We first define another (conceptual) use of locks, denoted *short-lived locks*. A lock, resp. its usage is short-lived, if the execution between taking the lock and releasing it is atomic, i.e., free from interference. In particular, between acquisition and release, no further lock-acquisition is allowed, including taking the same lock again in a re-entrant manner. It is obvious that transforming a program by adding short-lived locks does not lead to more deadlocks. We will use short-lived locks as gate-locks to protect assignment to the added race variables.

A deadlock involving a short-lived lock g and any other lock l means that there exists two processes where one is holding l and tries to take g, while the other one is holding g and tries l. Since no locking step is allowed while one is holding a short-lived lock without first releasing it, such a deadlock does not exist.

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 locks, no new deadlocks will be introduced. Similar to the transformation in Definition 5.9, 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 at most *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. If the verdict here is "no race detected", it means that the designated process never reaches any slp (for a particular cycle) at the same time as one of the other processes, thus ruling out a deadlock at those slps.

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".

This can be illustrated through a variant of the Dining Philosophers example, where the commonly-known deadlock is avoided by one process not closing the cycle: for any possible cycle, one of the processes fails to participate. If we do not non-deterministically select that process as the special one, there will always be a race reported between two other processes, indicating that they are willing to contribute their share to cycle.

Therefore, we suggest to run the analysis with each process being designated as special in turn in parallel to find the optimal result. Note that checks for different cycles can also 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 (as we have shown through the concept of short-lived locks), and can be analysed in a single race checker-run.

Theorem 6.1 (Soundness)**.** *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.*

To test our approach, we generate a static number of the Dining Philosophers (see Appendix A.2) together with the required locks, calculate all potential cycles of length up to *n* that we have to check for, and instrument the C program according to our transformation with race accesses for each cycle and gate locks (each race variable is protected with at most one gate lock).

n	# potential cycles	# race vars. per phil.	time single run	peak heap (approx.)
2	2	1	<0.1s	5 mb
3	30	4	~0.1s	5 mb
4	732	54	<0.5s	11 mb
5	29780	1384	~4m	666 mb

Table 12: Checking the Dining Philosophers with *Goblint*

Table 12 shows the corresponding values for a statically optimal version of the philosophers, that is, no abstraction occurs and thus it is known which philosopher uses which locks. Measurements have been taken on a recent Linux machine with *Goblint* version 0.9.6 / OCaml 3.12.1. We also report peak memory consumption of *Goblint*. Given this perfect static information, e.g. in the case of four philosophers, the transformation uses out of 732 race variables (one per potential cycle from all combinations of 4 processes/4 locks/cycles up to length 4) 54 per process. Our generator did not terminate producing the instrumented code for six dining philosophers due to the large number of potential cycles.

In the case of the "fixed" dining philosophers problem, where one of the processes *avoids* to close the cycle by breaking the symmetry, we need the same effort to certify it as deadlock-free. As a further optimization, we could observe if for each potential cycle, all its "segments" have in fact been used in a transformation—if not, we can immediately discharge this cycle without having to check for any races.

For each size n we have to generate n programs rotating the "special" process for the gate lock, giving another linear factor for the number of race checker runs. An instance is reported as safe (e.g. in the case where one philosopher avoids to close the cycle) when *at least one of the n runs* does not report a (potential) race.

Interpreting counter-examples. A potential race report immediately allows to conclude which particular cycle may have been closed, as by default each race variable corresponds to one particular cycle. We expect that the race checker reports the variable the race has happened on (as is the case with e.g. *Goblint*), and additional output by the checker may indiciate at which source code-location the cycle occurred: the statement with the race access directly corresponds to a subsequent lock-statement, as indicated by our transformation. Likewise, multiple race reports can be interpreted individually in a similar fashion.

Bounded search. As the number of potential cycles grows quickly with the number of threads and (abstract) locks goes up (see again Table 12), users may choose a staged approach to deadlock detection. In a larger problem (e.g. five dining philosophers), the size of the program can be minimized by *collapsing race variables*. Due to the soundness of our mechanism, it is possible to partition the search into all cycles of length 2 individually, but collapsing all larger cycles. An analysis result could then in principle report deadlocks with cycles of length 2, and absence of any deadlock for larger cycles. Conversely, if a deadlock in the larger abstraction is reported, the check could be repeated with partitioning race variables into explicit detection of cycles of length 3, and collapsing sizes 4 and 5 into a single check. In a further step, sizes 4 and five could be split and checked separately if necessary.

7. Conclusion

We presented an approach to statically analyse 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.

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.

Compared to our earlier work [38], apart from using a race checker instead of formulation a search for deadlocks in terms of a model checking problem, our new approach can soundly handle locks that are (repeatedly) created from the same abstract location. Also, we present here a polymorphic effect-inference which does not share the restriction to first-order functions of [37]. Unlike the model checking-based approach with a growing call-stack that must be cut off at a finite depth, we now summarise recursive methods in the effect system and rely on the soundness, precision, and effectiveness of the race checker, which should scale linearly in the source code-size of the instrumented program. We also provide a more detailed theory and proofs compared to the conference contribution [39].

Related work. Numerous approaches have been investigated and implemented over the years to analyse concurrent and multi-threaded programs (cf. e.g. [41] for a survey of various static analyses). Not surprisingly, in particular approaches to prevent races [11] and/or deadlocks [17] have been extensively studied for various languages and based on different techniques.

A classic programming discipline to prevent deadlocks a priori is to impose an order on the locks [12]. Acquiring locks consistent with that given order eliminates circular waits as one of the four necessary conditions for deadlocks [16]. Unlike the work presented here, many static type system build upon that idea by incorporating information about lock orders in their types. For instance, [14] introduced "deadlock types" extending their earlier work [15] on race-free Java. These incorporate a lock-level into the types for locks; the lock levels are ordered and the type system is responsible to assure adherence to an order-consistent lock acquisition policy. Actually, the paper allows more flexibility than adherence to a fixed lock order in that the order may change at run-time, without of course violating acyclicity. To enhance precision, the type system supports lock level *polymorphism*. Furthermore, the paper sketches how type inference can be done in their system, however, it seems, it is not attempted for the lock-polymorphic part. Different from the work presented here, the language in [14] is based on a block-structured locking discipline (using Java's `synchronized`-keyword). Also [22], besides race freedom, covers checking for deadlock freedom using ordered lock levels in the type system. To prevent races, the work introduced a form of singleton types for locks, i.e., a form of dependent types. A singleton lock type represents a single lock and the type system keeps track of sets of singleton lock types, there called *permissions*, representing statically the locks which are neces-

sarily held at a given point, i.e., a permission corresponds to the notion of *lock sets*, which is a key ingredient in many static (and dynamic) analyses to assure race freedom. In comparison with the current work, they roughly correspond to our notion of lock environments or abstract states, except that we are dealing with re-entrant locks, i.e., the lock environment of the type system corresponds to multi-sets as they approximate the number of times a lock is held. Besides that, lock environments in our work correspond to a "may"-interpretation, whereas the lock sets or permissions represent the locks which are necessarily held. Another difference is that our language supports non-block-structured locking. Thus, our type system considers pre- and post-specifications of the lock environments (together forming the effect of an expression), whereas in the block-structured discipline of [22] allows a simpler treatment in the type system where it suffices to type check an expression under the assumption of a given lock set, but without the need of a post-condition as the life-time of a lock ends at the end of a block. Finally, our type system makes no use of singleton types, but represents locks abstractly by their point of creation. A type-based analysis for non block-structured locking disciplines and for binary locks is presented in [47]. As the previously mentioned works, deadlock freedom is assured by incorporating lock levels into the types for locks. To deal with mutable references and aliasing, the work uses furthermore ownership types but does not consider type inference. Type inference assuring deadlock freedom for non-block-structured locking disciplines is investigated in [51] for a low-level concurrent polymorphic calculus.

The strengths resp. weaknesses of static vs. dynamic approaches to avoid or prevent (concurrency) errors are complementary: static approaches, when insisting on soundness, necessarily over-approximate and achieving acceptable precision is problematic. Dynamic checking, on the other hand, may incur run-time overhead and, concentrating individual runs, may miss erroneous situations. So some work tries to achieve the best of both worlds by combining dynamic and static approaches. To prevent races, for instance [6, 7] present a novel extension of type inference in the context of parametrized race-free Java PRFJ [15]. The parametric polymorphism of PRFJ makes complete type inference infeasible, so, to infer lock annotations, the work resorts to an incomplete inference algorithm, which is aided by monitoring executions at run-time. The paper coins the term type *discovery* for this form of run-time assisted type inference. [42] improves on the results in covering not only polymorphic instantiation (as [6]) but also polymorphic generalization. [8] combines static and dynamic techniques in an opposite way: where in type discovery, run-time analysis is used to assist the static analysis, in particular type inference, [8] uses a static type system to reduce

the overhead of run-time checking, based on the well-known GoodLock algorithm [30], by inferring which run-time checks may safely be omitted. The paper focuses on deadlock analysis using deadlock types. Extending [44], it covers also race-freedom and absence of atomicity violations besides deadlock freedom. The paper deals with block-structured locking in a Java-like language. They also present a type *inference* algorithm for *basic*, i.e., non-polymorphic deadlock types (in contrast to [14]). Also [27, 28] present a combination of static and dynamic techniques for deadlock avoidance. In contrast to [8], the static phase is not used to improve a dynamic monitoring of the running system, but, based on the result of the static phase, to interfere with the execution to "schedule around" impending deadlocks. Targeting mainly low-level languages, the system extends [13] to cover non block-structured lock-usage. Especially for non-block-structured locking locking, aliasing is a major problem, as lock sets cannot adequately be calculated statically and so the calculation is deferred until run-time. In general, the approaches use effect systems to approximate future lock usage and the generalization beyond a block-structured locking discipline necessitates to consider behavioral effects, i.e., taking the order of future lock acquisitions and releases into account ("continuation effects"). The type and effect system uses singleton lock types and supports lock-polymorphic function.

As mentioned in the introduction, in a concurrency model based on shared memory and lock-based synchronization, races and deadlocks are related but complementary concurrency problems. As emphasized, most static analyses including the type-based ones for deadlock prevention are based on confirming or deriving an acyclic order on lock acquisition. In contrast, our static analysis is not order-based but derives information about which locks are potentially held per thread, i.e., independent from a global order, to defer the global task of deadlock detection to race checking, after an appropriate transformation. Our type and effect analysis therefore resembles thus also static techniques determining "lock sets" which are used for race detection [21] [1] [23] [24] [29] to name a few. 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.

[35] presents a model-checking approach to deadlock detection on the control-

flow graph which relies on the precision of the underlying analyses that are necessary to handle all features of Javaprograms in terms of abstract locks and threads. This approach checks deadlocks between a pair of abstract threads and a pair of locks. It abstracts threads and locks by their allocation sites. The approach is neither sound nor complete.

Future work. Our analysis summarizes the potential locations of recursive function arguments based on its call-sites, which is the reason for some of the (expected) imprecision. In earlier work, we investigated inference and polymorphism [37], but how the presence of a static second lock points can be ascertained in such a polymorphic setting needs further investigation. A natural extension of our work would be an implementation of our type and effect system to transform concurrent programs written in e.g. in C and Java. Complications in those languages like *aliasing* would need to be taken into account, although we expect that results from a *may-alias* analysis could directly be consumed by our analysis. For practical applications, the restriction on fixed number of processes will not fit every program. 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.

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] ACM. *ACM Conference on Programming Language Design and Implementation (PLDI)*, June 1993. In *SIGPLAN Notices* 28(6).

[3] ACM. 2001.

[4] ACM. *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '02 (Seattle, USA)*, Nov. 2002. In *SIGPLAN Notices*.

[5] ACM. *ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[6] R. Agarwal, A. Sasturkar, and S. D. Stoller. Type discovery for parameterized race-free Java. Technical report, Dept. of Computer Science, SUNY, Stony Brooks NY, Dec. 2004.

[7] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In Steffen and Levi [46].

[8] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with state analysis and run-time monitoring. In Ur et al. [50], pages 191–207.

[9] T. Amtoft, H. R. Nielson, and F. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

[10] J. C. Baeten and S. Mauw, editors. *CONCUR '99: Concurrency Theory (10th International Conference, Eindhoven, The Netherlands)*, volume 1664 of *Lecture Notes in Computer Science*. Springer-Verlag, Aug. 1999.

[11] N. E. Beckman. A survey of methods for preventing race conditions. Available at `http://www.nelsbeckman.com/publications.html`, May 2006.

[12] A. D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Research Center, 1989.

[13] G. Boudol. A deadlock-free semantics for shared memory concurrency. In ICTAC09 [32], pages 140–154.

[14] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In OOPSLA'02 [4]. In *SIGPLAN Notices*.

[15] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In OOPSLA'01 [3].

[16] E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.

[17] J. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, Mar. 1996.

[18] P. Cousot, editor. *SAS'01: 8th International Static Analysis Symposium (Paris, France)*, volume 2126 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[19] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, 1965. Reprinted in [26].

[20] J. Ferrante, D. A. Padua, and R. L. Wexelblat, editors. *PPoPP'05: 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2005.

[21] C. Flanagan and M. Abadi. Object types against races. In Baeten and Mauw [10], pages 288–303.

[22] C. Flanagan and M. Abadi. Types for safe locking. In Swierstra [48], pages 91–108.

[23] C. Flanagan and S. Freund. Type-based race detection for Java. In *Proceedings of PLDI'00, ACM SIGPLAN Conference on ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, 2000.

[24] C. Flanagan and S. Freund. Type inference against races. In SAS'04 [43], pages 116–132.

[25] 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)* [2]. In *SIGPLAN Notices* 28(6).

[26] F. Genyus. *Programming Languages*. Academic Press, 1968.

[27] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type system for unstructured locking that guarantees deadlock freedomg without imposing a lock ordering. In *Pre-Proceedings of the Workshop on Programming Language Approaches to Concurrenct and Communication-Centric Software (PLACES 2009)*, volume 17 of *EPTCS*, pages 95–109, 2010.

[28] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type and effect system for deadlock avoidance in low-level languages. In *TLDI'11*. ACM, 2011.

[29] D. Grossman. Type-safe multithreading in Cyclone. In *TLDI'03: Types in Language Design and Implementation*, pages 13–25. ACM, 2003.

[30] K. Havelund. Using runtime analysis to guide model checking of Java programs. In Havelund et al. [31].

[31] K. Havelund, J. Penix, and W. Visser, editors. *Proceedings of the Spin 2000 Workshop in Model Checking of Software*, 2000.

[32] *Proceedings of the International Colloquium on Theoretical Aspects of Computing, ICTAC 2009*, volume 5684 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[33] C. Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1997. Technical Report DIKU-TR-97/1.

[34] 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)* [5], pages 308–319.

[35] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *31st International Conference on Software Engineering (ICSE09)*. IEEE, 2009.

[36] J. Palsberg and Z. Su, editors. *SAS'09*, volume 5673 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[37] K. I. Pun, M. Steffen, and V. Stolz. Behaviour inference for deadlock checking. Technical report 416, University of Oslo, Dept. of Informatics, July 2012.

[38] 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.

[39] K. I. Pun, M. Steffen, and V. Stolz. Deadlock checking by data race detection. In *Proc. of the 5th IPM International Conference on Fundamentals of Software Engineering (FSEN'13)*, 2013. To appear.

[40] G. Ramalingam, editor. *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[41] M. Rinard. Analysis of multithreaded programs. In Cousot [18], pages 1–19.

[42] J. Rose, N. Swamy, and M. Hicks. Dynamic inference of polymorphic lock types. In *Proceedings of the ACM Conference on Principles of Distributed Computing (PODC) Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, pages 18–25, July 2004.

[43] *SAS'04: 11th International Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[44] A. Sasturkar, R. Agarwal, L. Wang, and S. Stoller. Automated type-based analysis of data races and atomicity. In Ferrante et al. [20], pages 83–94.

[45] H. Seidl and V. Vojdani. Region analysis for race detection. In Palsberg and Su [36], pages 171–187.

[46] B. Steffen and G. Levi, editors. *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI) 2004*, volume 2937 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[47] K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In Ramalingam [40], pages 155–170.

[48] S. Swierstra, editor. *Proceedings of the 8th European Symposium on Programming (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*. Springer, 1999.

[49] J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *J. Funct. Program.*, 2(3):245–271, 1992.

[50] S. Ur, E. Bin, and Y. Wolfsthal, editors. *Proceedings of the Haifa Verification Conference 2005*, volume 3875 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[51] V. Vasconcelos, F. Martins, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *Post-Proceedings of the Workshop on Programming Language Approaches to Concurrenct and Communication-Centric Software (PLACES 2009)*, volume 17 of *EPTCS*, pages 95–109, 2010.

Appendix A. Code

Appendix A.1. Philosopher Generation (Haskell)

```
module Main where

{- $Id: MkPhils.hs 32648 2013-02-28 14:43:31Z stolz $

This should generate 'n' philosophers, where the 1st one is "special" in the
    sense that it doesn't need the gatelock,
and the 2nd can be made to be the "fixed" philosopher that breaks the cycle by
    choosing (r,l) instead of (l,r).

Usages:
  runhaskell MkPhils.hs 2 True 1
  runhaskell MkPhils.hs 2 False 1
  runhaskell MkPhils.hs 3 True 1
  runhaskell MkPhils.hs 3 False 1
  ...

Most notably, for one of the `MkPhils 3 True i` runs, it should NOT report a race
    . Currently i=2
because of the single cycle that we are looking for.

TODO:
```

```
 - print cycle for each racevar
 - don't declare unused variables
-}

import System.IO
import Data.List
import Data.Map (Map)
import qualified Data.Map (fromList,lookup)
import Data.Set (Set,size)
import qualified Data.Set (empty, insert)
import Control.Monad (when, unless)
import System.Environment (getArgs)

type RaceVar = String
type CycleInfo = Map Int (Int,Int)
data Cycle = Cycle RaceVar CycleInfo deriving (Show)

-- generate cycle with wrap around starting from oi/i:
-- Main> cycleFromTo 2 2 3 False
-- [(2,3),(3,1),(1,2)]
cycleFromTo :: Int -> Int -> Int -> Bool -> [(Int,Int)]
cycleFromTo oi i j finished
  | i == j && finished = []
  | i == j && (not finished) = (j,1):(cycleFromTo oi 1 oi True)
  | i < j = (i,i+1):cycleFromTo oi (i+1) j finished

cycleTo :: Int -> [(Int,Int)]
cycleTo t =  cycleFromTo 1 1 t False

main = do
  args@(numStr:fixStr:specialStr:_) <- getArgs
  let numLocks = (read numStr) :: Int
  -- unbreak one philosopher
  let fixPhil = (read fixStr) :: Bool
  -- designated phil not using gatelocks
  let specialPhil = (read specialStr) :: Int
  -- no more read-statements below.
  let locks = [1..numLocks]
  let cycles = zipWith (\i c -> Cycle ("z"++show i) c) [1..] (cycleInfo numLocks
      numLocks)
  putStrLn $ "/* generated through: ./MkPhils " ++ (concat $ intersperse " " args
      ) ++ " */"
  preamble cycles locks
  -- get all desired (l,r) combinations for phils
  let (special:fixed:rest) = cycleTo numLocks
  -- Am I the special one who doesn't need gatelocks?
  pS <- phil 1 cycles (1 == specialPhil) special
  -- Am I the philosopher who "breaks" the cycle? fixphil=False will create
      another "minion".
  -- If we have only 2 processes, don't use gate-locks.
  pF <- phil 2 cycles (numLocks <= 2 || 2 == specialPhil) ((\(l,r) -> if fixPhil
      then (r,l) else (l,r)) fixed)
  -- Our minions:
  ps <- mapM (\(n,lr) -> phil n cycles (n == specialPhil) lr) (zip [3..] rest)
  cMain (pS:pF:ps)

-- Filter only "correct" cycles (1,2 -> 2,.. -> ... -> n,1).
```

```
wf :: [(Int,Int)] -> Bool
wf c@((hd,_):_) = wf2 c
  where
    -- need to remember the head position for wrap-around
    wf2 ((_,x):(y,b):[]) = x == y && hd == b -- check if cycle is actually a
        cycle
    wf2 ((_,x):r@((y,_):_)) = x == y && wf2 r

nextInCycle :: Int -> [Int] -> [(Int,Int)]
nextInCycle cur allowed = [(cur,a) | a <- allowed, a /= cur]

-- Doesn't generate "true" cycles, need to filter with `wf`.
allCycles :: Int -> Int -> [[(Int,Int)]]
allCycles cnt n
 | n > cnt = error "Can't have longer cycles than number of locks!"
 | n == 1 = concat $ foldr (\cs s -> (map (\c@(_,to) -> [c]) cs):s) [] [ filter
     (\(l,r) -> l /= r) $ nextInCycle i [1..cnt] | i <- [1..cnt]]
 | otherwise = concatMap ((\c@((hd,_):_) -> let (_,tl) = last c in [c++[nic] |
     nic <- nextInCycle tl [1..cnt]])) (allCycles cnt (n-1))

allCyclesUpto :: Int -> Int -> [[(Int,Int)]]
allCyclesUpto cnt len = concat [filter wf (allCycles cnt l) | l <- [2..len]]

cycleInfo :: Int -> Int -> [CycleInfo]
-- maximum cycle length = # of locks
-- still contains duplicates, so need to nub
cycleInfo phils locks = Data.List.nub $ map Data.Map.fromList $ map (uncurry zip)
     [ (p,c) | p <- (permutations [1..phils]), c <- (allCyclesUpto locks locks)]

preamble :: [Cycle] -> [Int] -> IO ()
preamble zs locks = do
  putStrLn "#include <pthread.h>"
  putStrLn "#include <stdio.h>"
  putStrLn ""
  putStrLn $ "pthread_mutex_t "++ concat (intersperse "," (map (\(Cycle n _) -> "
      gate"++n) zs)) ++" = PTHREAD_MUTEX_INITIALIZER;"
  mapM_ (\l -> putStrLn $ "pthread_mutex_t mutex"++(show l)++" =
      PTHREAD_MUTEX_INITIALIZER;") locks
  putStrLn $ "int " ++ concat (intersperse "," (map (\(Cycle n _) -> n) zs)) ++ "
      ;"
  putStrLn ""

data Phil = Phil Int (Set RaceVar) String deriving (Show)

phil :: Int -> [Cycle] -> Bool -> (Int,Int) -> IO Phil
phil i cycs gated (l,r) = do
  putStrLn $ "void *phil"++(show i)++"(void *arg) {"
  putStrLn $ "  while(1) { pthread_mutex_lock(&mutex"++(show l)++");"
  -- Kalmer suggests to avoid addition to speed up things here:
  let (zs,gates) = foldr (\(Cycle z cs) old@(ozs,out) -> let raceAccess = "  "++z
      ++"=1; // RACE!"
                                  in maybe old (\(have,want) -> if (have == l
                                      && want == r)
                                       then (Data.Set.insert z ozs, (if gated
                                               then raceAccess
                                               else "  pthread_mutex_lock(&gate
                                                   "++z++");\n"++raceAccess++"\
```

```
                                                    n␣␣pthread_mutex_unlock(&
                                                    gate"++z++");"):out)
                                          else old) (Data.Map.lookup i cs)) (Data.
                                              Set.empty, []) cycs
  mapM_ putStrLn gates
  putStrLn $ "␣␣pthread_mutex_lock(&mutex"++(show r)++");"
  putStrLn $ "␣␣pthread_mutex_unlock(&mutex"++(show l)++");"
  putStrLn $ "␣␣pthread_mutex_unlock(&mutex"++(show r)++");"
  putStrLn $ "␣␣}"
  putStrLn "}"
  hPutStrLn stderr $ show (i, Data.Set.size zs)
  return $ Phil i zs ("phil" ++ (show i))

cMain :: [Phil] -> IO ()
cMain ps = do
  putStrLn "int␣main(void)␣{"
  putStrLn $ "␣␣pthread_t␣" ++ concat (intersperse "," (map (\(Phil i _zs n) -> "
      id"++n) ps)) ++ ";"
  mapM_ (\(Phil i _zs n) -> do
    putStrLn $ "␣␣pthread_create(&id"++n++",␣NULL,"++n++",␣NULL);") ps
  putStrLn "␣␣return␣0;"
  putStrLn "}"
```

Appendix A.2. Generated Philosophers (C)

Three philosophers, "instantiated" body, with gatelocks, instrumented for all possible 54 cycles. Observe how *after* applying the transformation it becomes obvious that only the cycle represented by race variable z7 can actually occur, as it is the only variable reference by more than a single process.

```
/* generated through: ./MkPhils 3 False 1 */
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t gatez1, gatez2, gatez3, gatez4, gatez5, gatez6, gatez7, gatez8, gatez9,
    gatez10, gatez11, gatez12, gatez13, gatez14, gatez15, gatez16, gatez17, gatez18,
    gatez19, gatez20, gatez21, gatez22, gatez23, gatez24, gatez25, gatez26, gatez27,
    gatez28, gatez29, gatez30 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex3 = PTHREAD_MUTEX_INITIALIZER;
int z1, z2, z3, z4, z5, z6, z7, z8, z9, z10, z11, z12, z13, z14, z15, z16, z17, z18, z19, z20, z21,
    z22, z23, z24, z25, z26, z27, z28, z29, z30;

void *phil1(void *arg) {
  pthread_mutex_lock(&mutex1);
  z1=1; // RACE!
  z7=1; // RACE!
  z17=1; // RACE!
  z27=1; // RACE!
  pthread_mutex_lock(&mutex2);
  pthread_mutex_unlock(&mutex1);
  pthread_mutex_unlock(&mutex2);
  return NULL;
```

```
}
void *phil2(void *arg) {
  pthread_mutex_lock(&mutex2);
  pthread_mutex_lock(&gatez6);
  z6=1; // RACE!
  pthread_mutex_unlock(&gatez6);
  pthread_mutex_lock(&gatez7);
  z7=1; // RACE!
  pthread_mutex_unlock(&gatez7);
  pthread_mutex_lock(&gatez16);
  z16=1; // RACE!
  pthread_mutex_unlock(&gatez16);
  pthread_mutex_lock(&gatez24);
  z24=1; // RACE!
  pthread_mutex_unlock(&gatez24);
  pthread_mutex_lock(&mutex3);
  pthread_mutex_unlock(&mutex2);
  pthread_mutex_unlock(&mutex3);
  return NULL;
}
void *phil3(void *arg) {
  pthread_mutex_lock(&mutex3);
  pthread_mutex_lock(&gatez7);
  z7=1; // RACE!
  pthread_mutex_unlock(&gatez7);
  pthread_mutex_lock(&gatez13);
  z13=1; // RACE!
  pthread_mutex_unlock(&gatez13);
  pthread_mutex_lock(&gatez23);
  z23=1; // RACE!
  pthread_mutex_unlock(&gatez23);
  pthread_mutex_lock(&gatez29);
  z29=1; // RACE!
  pthread_mutex_unlock(&gatez29);
  pthread_mutex_lock(&mutex1);
  pthread_mutex_unlock(&mutex3);
  pthread_mutex_unlock(&mutex1);
  return NULL;
}
int main(void) {
  pthread_t idphil1,idphil2,idphil3;
  pthread_create(&idphil1, NULL,phil1, NULL);
  pthread_create(&idphil2, NULL,phil2, NULL);
  pthread_create(&idphil3, NULL,phil3, NULL);
  return 0;
}
```