UNIVERSITY OF OSLO
Department of Informatics

Behaviour Inference for Deadlock Checking

July 2012

Research Report No. 416

Ka I Pun, Martin Steffen, and Volker Stolz

ISBN 82-7368-379-6
ISSN 0806-3036

July 2012

Behaviour Inference for Deadlock Checking

9. July 2012

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

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

Abstract. This report extends our behavioral type and effect system for detecting dealocks in [8] by polymorphism and formalizing type inference (wrt. the lock types. Our inference is defined for a simple concurrent, first-order language. From the inferred effects, after suitable abstractions to keep the state space finite, we either obtain the verdict that the program will not deadlock, or that it may deadlock. We show soundness and completeness of the type inference.

1 Introduction

Deadlocks are a well-known problem for concurrent programs with shared resources. As the scheduling at run-time affects the occurrence of a deadlock, deadlocks may only occur occasionally, and therefore are difficult to detect. Whether or not a deadlock occurs in a specific run in a particular program mainly depends on if the running program encounters a number of processes forming a circular chain, where each process waits for shared resources held by the others [4].

Deadlock *prevention* (as opposed to deadlock avoidance) statically ensures that deadlock do not occur, typically by preventing cyclic waits by enforcing an order on lock acquisitions. This idea has, e.g., been formalized in a type-theoretic setting in the form of deadlock types [3]. The static system presented in [3] supports also type *inference* (and besides deadlocks, prevents race conditions, as well). Deadlock types are also used in [1], but not for static deadlock prevention, but for improving the efficiency for deadlock avoidance at run-time.

In contrast, in [8] used a behavioural type and effect system [2,7] to capture lock interaction and use that behavioural description to explore an abstraction of the state space to detect potential deadlocks. An effect system commonly captures phenomena that may occur *during* evaluation, for instance exceptions. Expressive effects, which can represent the behaviour of a program, are important for concurrent programs. In particular, the effects of our system express the relevant behaviour of a concurrent program with regard to re-entrant locks. To detect potential deadlocks, we execute the abstraction of the actual behaviour to spot cyclic waiting for shared locks among parallel threads in the program.

Compared to our earlier work [8], an algorithmic behavioural type and effect inference system [6,5] is proposed to provide polymorphism for first-order programming, and to enhance user-friendliness and usefulness: the most general type and the most specific abstract behaviour of an implicitly-typed input program are reconstructed. The

algorithmic inference is proven to be both sound and complete with respect to the specification of the type system, while the abstraction of the behaviour is shown to preserve potential deadlocks in the original program, i.e., if the abstraction is deadlock free, then the corresponding program is also deadlock free, but not vice versa.

Overview

As said, this technical paper extends the previous [8] by "type-inference" or "type-reconstruction". In doing so we concentrate on effect-part, i.e., we ignore the underlying, standard typing part when dealing with the inference problem; that part is is standard and would only notationally complicate the derivation rules without really adding to the result.

Effect reconstruction is practically motivated. For deadlock detection we use a behavioral effect type system that captures sequences of lock interactions of the program. This abstraction captured in the behavioral effects then can be used in a second stage to potentially detect deadlocks. The type and effect system is based on explicit typing, i.e., the user is required to in particular specify the expected locking behavior when introducing variables; it is preferable not to burden the programmer with this but to *infer* that behavior. Type inference or type reconstruction, also for effects, is a known problem and orthogonal to the problem of the proposed method of deadlock checking, which we develop in detail in [8] and which is based in a special form of simulation relation (called deadlock-sensitive simulation). Therefore, we omitted effect inference in [8] and provide details of an effect type inference algorithms in this report.

Technically, and as usual, the key to allow effective type reconstruction is the addition of type level variables. Concentrating on the effects, it basically means effect variables, here for locks. That is not only a technical means to enable algorithmic type reconstruction (using unification), but renders also the language more expressive by allowing (universally) polymorphic functions. In our development later, we allow universally polymorphic functions, polymorphic lets, but don't cover polymorphic recursion.

As a starting point, i.e., as specification for the algorithmic inference, we use the previous type system with the following three changes. First, of course, the type system now uses implicit typing, i.e., we switch from Curry-style typing to Church-style. Secondly, we simplify the type system in that we disallow "lock types" to denote *sets* of (abstract) locks which as a consequence does away with *subtyping* as far as lock types are concerned. This restriction is compensated by the fact that now we allow lock polymorphic functions. Another way of understanding the change is that we replaced subtype polymorphism which we used in [8] by universal polymorphism as far as lock arguments are concerned. This restriction is also in parts technically motivated: it allowed to separate the treatment if sub-type polymorphism from the treatment of universal polymorphism (see also below where we describe the development slightly more detailed). Note that here we still support subtyping as "imported" from the sub-effecting relations to the effect-annotated function types. Finally, we restrict the development to first-order functions. This restriction is technically motivated. In principle, as far as type inference is concerned, higher-order functions are unproblematic of course, after all, type inference as most commonly used today was developed in the context of

the λ-calculus/functional languages. In our setting with behavioral effects for deadlock checking, we restrict to the first-order case not because of the treatment of type level variables, but because of *sub-effecting*. In the presence of sub-effecting, an algorithmic version of the type system requires to be able to calculate the *minimal* effect and, as a consequence, be able to calculate the least upper bound of, for instance, two effects. In the language for behavioral effects, which can be understood to specify traces or sequences of lock interactions, this least upper bound of two effects φ_1 and φ_2 is represented as the non-deterministic choice $\varphi_1 + \varphi_2$. Due to the contra/co-variant typing of functions, the question of least upper bound dualizes to finding the greatest lower bound when dealing with sub-effecting on contra-variant positions. It is, however, unclear whether one could/should have a construction which corresponds to the dual for $+$ and which is useful in finding deadlocks. For that technical reason, we leave the treatment of higher-order functions for further research.

The main problem for an algorithmic treatment of the given specification of the type system is to overcome the inherent *non-determinism* of the typing rules, when interpreting them in a goal-directed manner. The two central problematic phenomena, which are treated by non-syntax directed rules, are the two forms of *polymorphism*: subtyping in the form of sub-effecting and universal polymorphism. The standard way to turn a type system into the presence of these forms of polymorphism is to let the algorithm calculate not non-deterministically an unnecessarily specific type, but to calculate, at each point, the "best possible" one in the sense of committing in the least possible way to any specific type. To be able to do so, it is necessary to avoid *backtracking* which would yield at least an unpractical implementation, if not outright result in undecidability. Concerning sub-effecting, this amounts to determining the *minimal* type which in our setting is the type with the minimal effect. Concerning type variables and type schemes, the best possible type is the most general type, also known as principal type, from which all others can be obtained by substitution. As usual, *unification* is the mechanism to determine the most general type if it exists. To deal with both mechanisms, we adopt a layered approach. In a first step, we tackle the problem of sub-typing/sub-effecting which yields as an intermediate system which is equivalent to the specification but where the only non-syntax directed rules are the ones dealing with universal polymorphism. In a second step, we get rid of the remaining non-deterministic rules (generalization and instantiation) following standard techniques. For the formulation of soundness and in particular completeness, we follows the classic formulation of [5], which provided an inductive proof of completeness of "algorithm *W*" for Hindley/Milner/Damas kind of let-polymorphism (captured by type schemes).

In Section 2, we introduce the syntax and semantics of our calculus. Section 3 presents a non-deterministic specification of a type- and effect system and the corresponding inference algorithm which reconstructs the type of an implicitly-typed concrete program, and captures the abstract behaviour of the program, and the conclusion is given in Section 4.

2 Calculus

The abstract syntax for a small concurrent calculus with functions, thread creation, and re-entrant locks is given in Table 1.

A program P consists of a parallel composition of processes $p\langle t\rangle$, where p identifies the process and t is a thread, i.e., the code being executed. The empty program is denoted as $\emptyset$. As usual, we assume $\|$ to be associative and commutative, with $\emptyset$ as neutral element. As for the code we distinguish threads t and expressions e, where t basically is a sequential composition of expressions. Values are denoted by v, and $\texttt{let}\ x{:}T = e\ \texttt{in}\ t$ represents the sequential composition of e followed by t, where the eventual result of e, i.e., once evaluated to a value, is bound to the local variable x. Expressions, as said, are given by e, and threads are among possible expressions. Further expressions are function application $e_1\ e_2$, conditionals, and the spawning of a new thread, written $\texttt{spawn}\ t$. The last three expressions deal with lock handling: `new L` creates a new lock (initially free) and gives 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. To keep track of lock creations, the corresponding expressions are annotated, where we assume an infinite reservoir of locations or labels π.

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

Table 1. Abstract syntax

The types and the effects are given in Table 2, resp. in Table 5 for the effects. Besides basic types for integers and booleans, the calculus supports types $\texttt{L}^r$ for lock references and function types $\vec{U} \xrightarrow{\varphi} U$, under the restriction to first-order functions. As abbreviation, the type Unit represents the empty product with $()$ as corresponding value, i.e., the empty tuple. For type inference or type reconstruction, we concentrate on the part we are most interested in, namely the behavior part, in particular, the locations. To do so,

we assume that the user provides the underlying types, i.e., without location and effect annotations and that they are not reconstructed by type inference. It would be straightforward with standard techniques, to incorporate conventional type reconstruction for the underlying types. For the current presentation, we omit that part not to clutter the presentation. So, the reconstruction just concerns the omitted locations for lock creation. In abuse of notation, we use T resp. U both for the annotated types from the grammar of Table 1 and the underlying types with the annotations stripped; which is meant should be clear from the context. Note further the simplification entailed by the restriction to first-order functions: in absence of higher-order functions, the only variables of function type are let-bound variables, but not formal parameters of functions. Basically it means that the type reconstruction algorithm does not need to *guess* (using type-level variables) the effect annotation φ on functional types $T_1 \xrightarrow{\varphi} T_2$; for let-bound variable, such a guess is not needed as the effect is *immediately* available upon analysis of the definition of the code bound to the variable.

Polymorphism for function definitions is captured by type-level variables. In our setting, the only type-level variables are location variables ρ representing locations π. They may show up in the location types $\mathtt{L}^{\rho}$ as well as in the effects φ on the latent effects of functions. The universally quantified type (corresponding to type schemes or polytypes) captures functions polymorphic in the locations. We abbreviate $\forall\rho_1.\ldots\forall\rho_n.T$ by $\forall\vec{\rho}.T$.

$U ::=$	$\mathtt{Bool} \mid \mathtt{Int} \mid \mathtt{Thread} \mid \mathtt{L}^{r}$	basic types
$T ::=$	$U \mid \vec{U} \xrightarrow{\varphi} U$	types
$S ::=$	$T \mid \forall\rho.S$	type schemes
$r ::=$	$\rho \mid \pi$	location annotations

Table 2. Types and type schemes

The effects of Table 5 form a behavioral abstraction of the behavior of the concurrent programs wrt. the lock usage. The behavior language can be seen as a small process algebra, for which certain algebraic laws will hold and for which we will give a semantics in the form of operational rules as well. We give the definition of effects φ later in Section 3.

The set of free and bound variables of a type are defined as usual, where $\forall$ acts as binder; bound variables are considered up-to renaming. We write $fv(T)$ for the set of free variables in T. Substitutions, with typical element θ, are mappings from variables (location) variables ρ to location annotations r. We write θT for the application of θ to a type T, replacing all free variables of T according to θ, with renaming of bound variables, if necessary. The domain $dom(\theta)$ of θ is defined the set of all variables where $\theta(\rho) \neq \rho$. Concrete substitutions we write in the form $[r_1/\rho_1]\ldots[r_k/\rho_k]$ or $[\vec{r}/\vec{\rho}]$.

2.1 Semantics

The small-step operational semantics given below is straightforward. We distinguish between local and global steps (cf. Tables 3 and 4). The local level deals with execution steps of one single thread, where the steps specify reduction steps in the following form:

$$t \to t' \ . \tag{1}$$

Rule R-RED is the basic evaluation step, replacing in the continuation thread t the local variable by the value v (where $[v/x]$ is understood as capture-avoiding substitution). Rule R-LET restructures a nested let-construct. As the let-construct generalizes sequential composition, the rule expresses 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 other rule, which performs a case distinction of the first basic expression in a let construct, that 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).

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

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

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

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

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

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

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

where P is a program, i.e., the parallel composition of a finite number of threads running in parallel, and σ contains the *locks*, i.e., it is a finite mapping from lock identifiers to the status of each lock (which can be either free or taken by a thread where a natural number indicates how often a thread has acquired the lock, modelling re-entrance). A thread-local step is 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 $\sigma + l_p$ is defined as follows: If $\sigma(l) = \mathit{free}$, then $\sigma + l_p = \sigma[l \mapsto p(1)]$ and if $\sigma(l) = p(n)$, then $\sigma + l_p = \sigma[l \mapsto p(n+1)]$. Dually $\sigma - l_p$ is defined as follows: if $\sigma(l) = p(n+1)$, then $\sigma - l_p = \sigma[l \mapsto p(n)]$, and if $\sigma(l) = p(1)$, then $\sigma - l_p = \sigma[l \mapsto \mathit{free}]$. 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.

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

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

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

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

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

Table 4. Global steps

3 Type system

The type and effect system for expressions is given in Table 8. The judgments for expressions are of the form

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

where the typing context $\Gamma = x_1{:}T_1, \ldots, x_n{:}T_n$ associates (annotated) types to variables. As assume that all variables x_i are different, so Γ can be seen as a finite mapping; we write $\Gamma(x)$ for the type of x in Γ and $\mathit{dom}(\Gamma)$ for the set of variable typed in Γ.

3.1 Effects

In specifying the syntax, we postponed the exact definition of the effects, which played no role in formulating the operational semantics. We do that next (see Table 5), before we can present the rules of the type and effect system in Section 3.3.

The effects are split between a global level Φ and a (thread-)local level φ. The empty effect ε represents behavior without lock operations. Sequential composition and

$\Phi ::= \mathbf{0} \mid p\langle\varphi\rangle \mid \Phi \parallel \Phi$		effects (global)
$\varphi ::= \varepsilon \mid \varphi;\varphi \mid \varphi+\varphi \mid \alpha$		effects (local)
$\mid X \mid rec\,X.\varphi$		recursive behavior
$a ::= \texttt{spawn}\,\varphi \mid \nu\mathrm{L}^r \mid \mathrm{L}^r.\texttt{lock} \mid \mathrm{L}^r.\texttt{unlock}$		labels/basic effects
$\alpha ::= a \mid \tau$		transition labels

Table 5. Effects

non-deterministic choice are represented by $\varphi_1;\varphi_2$ and $\varphi_1+\varphi_2$, respectively. Recursive behaviour is introduced through $rec\,X.\varphi$, where $rec\,X$ binds the recursion variable in φ. Recursion *is not polymorphic*, i.e. location variables in the effect depend entirely on the types that introduce them.

Labels a capture four basic effects: $\texttt{spawn}\,\varphi$ represents the effect of creating a new process with behaviour φ, while $\nu\mathrm{L}^r$ means the effect of creating a new lock at program point r. The effects of lock manipulations are captured by $\mathrm{L}^r.\texttt{lock}$ and $\mathrm{L}^r.\texttt{unlock}$, meaning acquiring a lock and releasing a lock, respectively, where r is again referring to the point of creation. τ is used later to label silent transitions.

3.2 Sub-effecting

The behaviour of an effect expression describes its possible traces to over-approximate the actual behaviour. The effects are *ordered* and the corresponding *sub-effect* relation is formalized in Table 7. Sub-effecting, i.e. the order on effects, leads to an order $\leq$ on types, in particular on function types.

Definition 1 (Subtyping and sub-effecting). *The binary relations $\equiv$ (equivalence) and $\leq$ (sub-effecting) on effects are given inductively by the rules of Table 7. In abuse of notation, the* subtyping *relation types, relative to a given context Γ, is written $S_1 \leq S_2$, as well and given in Table 6. Furthermore, we define $\vee$ by induction on types: $S \xrightarrow{\varphi_1} T \vee S \xrightarrow{\varphi_2} T$ is defined as $S \xrightarrow{\varphi_1+\varphi_2} T$. Else $\forall\rho.S \vee \forall\rho.T = \forall\rho.S \vee T$. Else $T \vee T$ equals T and $\vee$ is undefined otherwise.*

Sequential composition is associative, with ε as neutral element (cf. rules E-UNIT and EE-ASSOC$_s$). The non-deterministic choice is commutative, associative, and distributes over sequential composition (cf. rule EE-COMM, EE-ASSOC$_c$, and EE-DISTR). Idempotence of choice is described by EE-CHOICE. Sub-effecting is reflexive (modulo $\equiv$) and transitive. Note that sequential composition and choice are "monotone" wrt. $\leq$ (by SE-CHOICE$_2$ and SE-SEQ) and the premise of SE-CHOICE$_1$ makes sure that all variables occurring free in the effect are covered by the context (for the other sub-effecting rules, that is assured by induction). Monotonicity of the spawn-construct and recursion is covered by SE-SPAWN and SE-REC. As for subtyping: Rule SE-REFL expresses the reflexivity of the order. The order of two effects is lifted to function types and universally quantified types in rule S-ARROW and S-ALL. Transitivity of subtyping is straightforward.

$$\Gamma \vdash T \leq T \quad \text{S-REFL} \qquad \frac{\Gamma \vdash \varphi \leq \varphi'}{\Gamma \vdash T_1 \xrightarrow{\varphi} T_2 \leq T_1 \xrightarrow{\varphi'} T_2}\ \text{S-ARROW} \qquad \frac{\Gamma \vdash S_1 \leq S_2}{\Gamma \vdash \forall \rho.S_1 \leq \forall \rho.S_2}\ \text{S-ALL}$$

Table 6. Subtyping

$$recX.\varphi \equiv [recX.\varphi/X]\varphi \quad \text{EE-REC}$$
$$\varepsilon;\varphi \equiv \varphi \quad \text{EE-UNIT} \qquad \varphi_1;(\varphi_2;\varphi_3) \equiv (\varphi_1;\varphi_2);\varphi_3 \quad \text{EE-ASSOC}_S$$
$$\varphi + \varphi \equiv \varphi \quad \text{EE-CHOICE} \qquad (\varphi_1 + \varphi_2);\varphi_3 \equiv \varphi_1;\varphi_3 + \varphi_2;\varphi_3 \quad \text{EE-DISTR}$$
$$\varphi_1 + \varphi_2 \equiv \varphi_2 + \varphi_1 \quad \text{EE-COMM} \qquad \varphi_1 + (\varphi_2 + \varphi_3) \equiv (\varphi_1 + \varphi_2) + \varphi_3 \quad \text{EE-ASSOC}_C$$

$$\frac{\varphi_1 \equiv \varphi_2}{\Gamma \vdash \varphi_1 \leq \varphi_2}\ \text{SE-REFL} \qquad \frac{\Gamma \vdash \varphi_1 \leq \varphi_2 \qquad \Gamma \vdash \varphi_2 \leq \varphi_3}{\Gamma \vdash \varphi_1 \leq \varphi_3}\ \text{SE-TRANS}$$

$$\frac{fv(\varphi_1 + \varphi_2) \subseteq fv_T(\Gamma)}{\Gamma \vdash \varphi_1 \leq \varphi_1 + \varphi_2}\ \text{SE-CHOICE}_1 \qquad \frac{\Gamma \vdash \varphi_1 \leq \varphi_1' \qquad \Gamma \vdash \varphi_2 \leq \varphi_2'}{\Gamma \vdash \varphi_1 + \varphi_2 \leq \varphi_1' + \varphi_2'}\ \text{SE-CHOICE}_2$$

$$\frac{\Gamma \vdash \varphi_1 \leq \varphi_1' \qquad \Gamma \vdash \varphi_2 \leq \varphi_2'}{\Gamma \vdash \varphi_1;\varphi_2 \leq \varphi_1';\varphi_2'}\ \text{SE-SEQ} \qquad \frac{\Gamma \vdash \varphi_1 \leq \varphi_2}{\Gamma \vdash \texttt{spawn}\ \varphi_1 \leq \texttt{spawn}\ \varphi_2}\ \text{SE-SPAWN} \qquad \frac{\Gamma \vdash \varphi_1 \leq \varphi_2}{\Gamma \vdash recX.\varphi_1 \leq recX.\varphi_2}\ \text{SE-REC}$$

Table 7. Sub-effecting

3.3 Typing rules

The rules of the type system are given in Table 8. Variables, thread names, and lock references are all values and thus have no effect (cf. rules T-VAR, T-PREF, and T-LREF). Both branches of a conditional must agree on their type and their effect (cf. rule T-COND). The let-construct generalizes sequential composition and its effect $\varphi_1;\varphi_2$ is the sequential composition of the effects of the constituent parts (cf. rule T-LET). Note further that the type T_1 given by the user for the variable x is from the underlying types, i.e., without annotations, whereas the result of analyzing e_1 may be annotated (with locations and effects). So the user-given T_1 must correspond to the annotated type scheme S_1 with all annotations stripped, written $\lfloor S_1 \rfloor$. Note that the operator $\lfloor \_ \rfloor$ applies also to quantified types by stripping the quantifier and the related bound variables. The analysis of the body e_2 of the let-construct continues assuming the annotated type scheme S_1 for the local variable, not the underlying, declared type. Abstractions are considered as values and therefore have empty effect (cf. rule T-ABS). Similar to the let-construct, the types in $\vec{T_1}$ given by the user for the variables $\vec{x}$ are without annotations, and must correspond to the annotated $\vec{T}$. The analysis of the function body is based on the assumption of the annotated type $\vec{T}$ for the input parameters. The effect of the body in the premise is the *latent* effect of the overall function, and is annotated on the function type of the abstraction in the conclusion of the rule. Note that for the recursive function (cf. rule T-ABS$_{rec}$), the function type $\vec{T_1} \to T_2$ given by the user for the function f correspond to

the annotated type $\vec{T_1}' \to T_2'$ whose latent effect is guessed as the recursive effect variable X in the context of the premise. The overall type of the recursive function is the function type annotated with the recursive effect of the function body $recX.\varphi$. In the rule T-APP, the function as well the tuple of arguments in an application are considered as values and have empty effect. Therefore, the overall effect is the latent effect of the function body.

$$\frac{\Gamma(x) = S}{\Gamma \vdash x : S :: \varepsilon}\ \text{T-VAR} \qquad \frac{}{\Gamma \vdash p : \texttt{Thread} :: \varepsilon}\ \text{T-PREF} \qquad \frac{}{\Gamma \vdash l^{\pi} : \mathrm{L}^{\pi} :: \varepsilon}\ \text{T-LREF}$$

$$\frac{\Gamma \vdash v : \texttt{Bool} :: \varepsilon \qquad \Gamma \vdash e_1 : T :: \varphi \qquad \Gamma \vdash e_2 : T :: \varphi}{\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T :: \varphi}\ \text{T-COND}$$

$$\frac{\Gamma \vdash e_1 : S_1 :: \varphi_1 \qquad \lfloor S_1 \rfloor = T_1 \qquad \Gamma, x{:}S_1 \vdash e_2 : T_2 :: \varphi_2}{\Gamma \vdash \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : T_2 :: \varphi_1;\varphi_2}\ \text{T-LET}$$

$$\frac{\lfloor \vec{T} \rfloor = \vec{T_1} \qquad \Gamma, \vec{x}{:}\vec{T} \vdash e : T_2 :: \varphi}{\Gamma \vdash \texttt{fn}\ \vec{x}{:}\vec{T_1}.e : \vec{T} \xrightarrow{\varphi} T_2 :: \varepsilon}\ \text{T-ABS} \qquad \frac{\lfloor \vec{T_1'} \rfloor = \vec{T_1} \qquad \lfloor T_2' \rfloor = T_2 \qquad \Gamma, f{:}\vec{T_1}' \xrightarrow{X} T_2', \vec{x}{:}\vec{T_1}' \vdash e : T_2' :: \varphi}{\Gamma \vdash \texttt{fun}\ f{:}\vec{T_1} \to T_2.\vec{x}{:}\vec{T_1}.e : \vec{T_1}' \xrightarrow{recX.\varphi} T_2' :: \varepsilon}\ \text{T-ABS}_{rec}$$

$$\frac{\Gamma \vdash v_1 : \vec{T_2} \xrightarrow{\varphi} T :: \varepsilon \qquad \Gamma \vdash \vec{v_2} : \vec{T_2} :: \varepsilon}{\Gamma \vdash v_1\ \vec{v_2} : T :: \varphi}\ \text{T-APP}$$

$$\frac{\Gamma \vdash e : S :: \varphi}{\Gamma \vdash \texttt{spawn}\ e : \texttt{Thread}{::}\texttt{spawn}\ \varphi}\ \text{T-SPAWN} \qquad \Gamma \vdash \texttt{new}_{\pi}\ \mathrm{L}{:}\ \mathrm{L}^{\pi} :: \nu \mathrm{L}^{\pi}\ \ \text{T-NEWL}$$

$$\frac{\Gamma \vdash v : \mathrm{L}^{r} :: \varphi}{\Gamma \vdash v.\,\texttt{lock}{:}\ \mathrm{L}^{r} :: \varphi; \mathrm{L}^{r}.\texttt{lock}}\ \text{T-LOCK} \qquad \frac{\Gamma \vdash v : \mathrm{L}^{r} :: \varphi}{\Gamma \vdash v.\,\texttt{unlock}{:}\ \mathrm{L}^{r} :: \varphi; \mathrm{L}^{r}.\texttt{unlock}}\ \text{T-UNLOCK}$$

$$\frac{\Gamma \vdash e : S' :: \varphi' \qquad \Gamma \vdash S' \leq S \qquad \Gamma \vdash \varphi' \leq \varphi}{\Gamma \vdash e : S :: \varphi}\ \text{T-SUB}$$

$$\frac{\Gamma \vdash e : S :: \varphi \qquad \rho \notin fv_T(\Gamma) \qquad \rho \in fv_T(S)}{\Gamma \vdash e : \forall \rho.S :: \varphi}\ \text{T-GEN} \qquad \frac{\Gamma \vdash e : \forall \rho.S :: \varphi \qquad \rho = dom(\theta)}{\Gamma \vdash e : \theta S :: \varphi}\ \text{T-INST}$$

Table 8. Type and effect system

The spawn-statement is of type `Thread`, provided the spawned expression is well-typed (cf. rule T-SPAWN) and the effect just expresses that the effect φ of the body t is executed by a new thread. The treatment of creating a new lock at location π is straightforward: the expression is of (annotated) type L^{π} and of effect $\nu \mathrm{L}^{\pi}$. The non-syntax-directed T-SUB is a standard rule of subsumption, allowing to relax types and effects (cf. also Definition 1). The two remaining, dual rules of generalization and specialization or instantiation introduce, resp. eliminate polymorphic types (cf. rules T-GEN and T-INST). As usual, to introduce a universally-quantified type, the typing of the corresponding expression must not depend (via Γ) on the variable (here ρ) being quantified

over. We also assume that we only quantify over variables actually occurring in the type. The function $fv_T(\_)$ in the premise of the generalization rule returns the free variables which occur in the types, but *not* in latent effects.

3.4 Algorithmic formulation

Next we turn the type and effect system of Section 3 into an algorithm. The reason, why the type system in itself is non-algorithmic are the non-syntax-directed rules of Table 8 which are the rules dealing with *polymorphism* of the type system. The system supports two forms of polymorphism, subtype polymorphism (or rather sub-effecting) captured by the subsumption rule T-SUB and universal polymorphism (in the form of let-polymorphism for location variables) captured by the generalization and instantiation rules T-GEN and T-INST. To obtain an algorithm, those rules need to be replaced by syntax-directed counter-parts; for subtype polymorphism, the algorithm must calculate the most specific type, i.e., minimal type wrt. the subtype/sub-effecting order. For the universal polymorphism, the most general type needs to be determined. As usual, unification is the key for that.

The definition of unification on types is standard, ignoring the latent effects for the arrow types in the unification. Instead of unifying the latent effects φ_1 and φ_2 of two arrow types, their choice $\varphi_1 + \varphi_2$ is used, over-approximating both φ_1 and φ_2.

Definition 2 (Unification). *The most general unifier of two types is given by Table 9. We write $T = T_1 \wedge_\theta T_2$ if θ is an mgu of T_1 and T_2 and $T = \theta T_2 (= \theta T_2)$.*

$$
\begin{array}{rcl}
\mathscr{U}(\mathtt{Int},\mathtt{Int}) & = & id \\
\mathscr{U}(\mathtt{Bool},\mathtt{Bool}) & = & id \\
\mathscr{U}(T_1 \xrightarrow{\varphi} T_2, T_1' \xrightarrow{\varphi'} T_2') & = & \mathtt{let}\ \theta_1 = \mathscr{U}(T_1, T_1') \\
 & & \quad\ \theta_2 = \mathscr{U}(\theta_1 T_2, \theta_1 T_2') \\
 & & \mathtt{in}\ \ \theta_2 \circ \theta_1 \\
 & & \\
\mathscr{U}(\mathtt{L}^{r_1}, \mathtt{L}^{r_2}) & = & \mathtt{let}\ \theta_0 = \mathscr{U}(r_1, r_2) \\
 & & \quad\ \theta_1 = \mathscr{U}(\theta_0\, \mathtt{L}^{r_1}, \theta_0\, \mathtt{L}^{r_2}) \\
 & & \mathtt{in}\ \ \theta_1 \circ \theta_0 \\
\mathscr{U}(T_1, T_2) & = & \text{fail} \qquad \text{in all other cases} \\
 & & \\
\mathscr{U}(r, \rho) & = & [\rho \mapsto r] \\
\mathscr{U}(\rho, r) & = & \text{symmetrically} \\
\mathscr{U}(r_1, r_2) & = & \text{fail} \qquad \text{in all other cases}
\end{array}
$$

Table 9. Unification

As mentioned earlier, the generalization rule T-GEN allows to introduce universal quantification over a variable provided the variable is not mentioned in the typing con-

text. For the formulation of the algorithm, the following closure operation is useful, which generalizes over all variables, to which T-GEN applies.

Definition 3 (Closure). *The closure of a type T with respect to context Γ is given as $close_\Gamma(T) = \forall\vec{\rho}.T$, where $\vec{\rho} = fv(T) \setminus fv(\Gamma)$.*

Definition 4 (Instantiation with fresh variables). *Assume $\Gamma \vdash e : \ T :: \varphi$. We define a fresh instance of S (wrt. Γ) by replacing all universally quantified variables of S by fresh ones. I.e., for $S = \forall\vec{\rho}.T$, then $INST_\Gamma(S) = \theta T$, where $\theta = [\vec{\rho}/\vec{\rho}']$ where $\vec{\rho}'$ are fresh variables.*

The judgements of the type reconstruction algorithm look as follows

$$\Gamma \vdash_A e : T :: \varphi, \theta \tag{4}$$

and is interpreted as follows: under the typing context Γ, expression e is of type T and with effect φ, under a substitution θ. Substitutions θ are finite and partial mappings from location variables ρ to location annotations r. We assume that the type T and effect φ already have the substitution θ applied. Note also that T is a type, not a type scheme, which differs from the judgments used in the type system. As for the bindings in the typing context, variables are bound to *type schemes* (as in the type system). The rules for the algorithm are given in Table 10.

The type of a variable is looked up from the context Γ and a variable introduces no constraints, i.e., the substitution is the identity *id*. As all values, variables have no effect (cf. rule TA-VAR. Similarly the treatment of thread names and lock references in rules TA-THREAD and TA-LREF). Note that S, as fetched from the typing environment, may be a type scheme, whereas the instance T is a type (cf. Definition 4). As values, also abstractions have no effect (cf. rule TA-ABS). The types $\vec{T}_1$ given by the user are without annotations. The operator $\lceil\_\rceil_A$ used in the premise of the rule annotates T_1 with *fresh* variables, and function body is checked in the premise under the assumption of the thus annotated type $\vec{T}_1'$ of the input parameters $\vec{x}$. The latent effect of the function is the effect of the function body. The substitution obtained from analyzing the function body is propagated in the conclusion, with the fresh variables from the user-provided type removed, as they are local to the body. Recursive function definitions in TA-ABS_{rec} use an additional fresh recursion variable X in place for the latent effect of a recursive invocation, which is bound when entering the recursion. Note that we do not allow polymorphic recursion: the recursive invocation of a function with parameters of type L will fix the type of the formal parameters.

For applications (cf. rule TA-APP), the function as well as the arguments are already evaluated, and therefore both abstraction and argument have empty effect and identity substitution. The type of the abstraction is a function type annotated with the latent effect. We use unification to ensure that the type of the argument is equal to the argument type of the abstraction. The overall type and effect of the application are respectively the output type and the latent effect of the abstraction which are specified by the unification substitution.

The definition of conditionals (cf. rule TA-COND) first ensures the types of the two branches are equal modulo latent effects with unification. The overall type and effect are the least upper bound of the two types resp. two effects from the two branches.

The sequential composition of two expressions is constructed by the rule TA-LET. The closure of the type of the `let`-bound variable is added to the context to give polymorphism. The overall effect $\varphi_1;\varphi_2$ is the sequential composition of the expressions. The remaining rules TA-SPAWN, TA-NEWL, TA-LOCK and TA-UNLOCK are straightforward.

$$\frac{\Gamma(x) = S}{\Gamma \vdash_A x : INST_\Gamma(S) :: \varepsilon, id}\ \text{TA-VAR} \qquad \frac{}{\Gamma \vdash_A p : \texttt{Thread} :: \varepsilon, id}\ \text{TA-PREF} \qquad \frac{}{\Gamma \vdash_A l^\pi : \mathrm{L}^\pi :: \varepsilon, id}\ \text{TA-LREF}$$

$$\frac{\vec{T}_1' = \lceil \vec{T}_1 \rceil_A \qquad \Gamma, \vec{x}{:}\vec{T}_1' \vdash_A e : T_2 :: \varphi, \theta \qquad \vec{\rho} = fv(\vec{T}_1')}{\Gamma \vdash_A \texttt{fn}\, \vec{x}{:}\vec{T}_1.e : (\theta\vec{T}_1') \xrightarrow{\varphi} T_2 :: \varepsilon, \theta \setminus \vec{\rho}}\ \text{TA-ABS}$$

$$\frac{\begin{array}{c}\vec{T}_1' = \lceil \vec{T}_1 \rceil_A \quad T_2' = \lceil T_2 \rceil_A \quad \Gamma, f{:}\vec{T}_1' \xrightarrow{X} T_2', x{:}\vec{T}_1' \vdash_A e : T :: \varphi, \theta \\ X\ fresh \quad \lfloor T \rfloor = T_2 \quad \vec{\rho} = fv(\vec{T}_1' \to T_2') \quad \theta_1 = \mathscr{U}(T, \theta T_2')\end{array}}{\Gamma \vdash_A \texttt{fun}\, f{:}\vec{T}_1 \to T_2.x{:}\vec{T}_1.e : (\theta_1\theta\vec{T}_1') \xrightarrow{\theta_1(rec X.\varphi)} (\theta_1\theta T_2') :: \varepsilon, \theta_1 \circ \theta \setminus \vec{\rho}}\ \text{TA-ABS}_{rec}$$

$$\frac{\Gamma \vdash_A v_1 : \vec{T}_2' \xrightarrow{\varphi} T' :: \varepsilon, id \qquad \Gamma \vdash_A v_2 : \vec{T}_2, \varepsilon, id \qquad \theta = \mathscr{U}(\vec{T}_2', \vec{T}_2)}{\Gamma \vdash_A v_1\ v_2 : \theta T' :: \theta\varphi, \theta}\ \text{TA-APP}$$

$$\frac{\Gamma \vdash_A v : \texttt{Bool} :: \varepsilon, id \qquad \Gamma \vdash_A e_1 : T_1 :: \varphi_1, \theta_1 \qquad \theta_1\Gamma \vdash_A e_2 : T_2 :: \varphi_2, \theta_2 \qquad \theta_3 = \mathscr{U}(\theta_2\, T_1, T_2)}{\Gamma \vdash_A \texttt{if}\, v\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \theta_3\theta_2 T_1 \vee \theta_3 T_2 :: \theta_3\theta_2\varphi_1 + \theta_3\varphi_2,\ \theta_3 \circ \theta_2 \circ \theta_1}\ \text{TA-COND}$$

$$\frac{\Gamma \vdash_A e_1 : T_1' :: \varphi_1, \theta_1 \qquad S_1 = close_{\theta_1\Gamma}(T_1') \qquad \lfloor S_1 \rfloor = T_1 \qquad \theta_1\Gamma, x{:}S_1 \vdash_A e_2 : T_2 :: \varphi_2, \theta_2}{\Gamma \vdash_A \texttt{let}\, x{:}T_1 = e_1\, \texttt{in}\, e_2 : T_2 :: \theta_2\varphi_1;\varphi_2, \theta_2 \circ \theta_1}\ \text{TA-LET}$$

$$\frac{\Gamma \vdash_A e : T :: \varphi, \theta}{\Gamma \vdash_A \texttt{spawn}\, e : \texttt{Thread} :: \texttt{spawn}\, \varphi, \theta}\ \text{TA-SPAWN} \qquad \Gamma \vdash_A \texttt{new}_\pi\, L : \mathrm{L}^\pi :: \nu\mathrm{L}^\pi, id\ \ \text{TA-NEWL}$$

$$\frac{\Gamma \vdash_A v : \mathrm{L}^r :: \varepsilon, id}{\Gamma \vdash_A v.\, \texttt{lock} : \mathrm{L}^r :: \mathrm{L}^r.\texttt{lock}, id}\ \text{TA-LOCK} \qquad \frac{\Gamma \vdash_A v : \mathrm{L}^r :: \varepsilon, id}{\Gamma \vdash_A v.\, \texttt{unlock} : \mathrm{L}^r :: \mathrm{L}^r.\texttt{unlock}, id}\ \text{TA-UNLOCK}$$

Table 10. Algorithmic effect inference

3.5 Soundness and completeness

Next we prove soundness and completeness of the type reconstruction algorithm from Table 10 wrt. its specification (cf. Table 8). We start with a few technical properties, mainly about the subtyping/sub-effecting relation, substitutions, and instantiation, which are needed for establishing those results.

3.5.1 Preliminaries The algorithm treats subjecting by giving back the best possible effect (in the sense of being minimal wrt. $\leq$). For conditionals, in particular, the algorithm gives back $\varphi_1 + \varphi_2$ as latent effect, where φ_1 and φ_2 are the latent effects of

the two conditional branches. The next lemma establishes that this corresponds to the minimal effect, and further that the corresponding type is minimal.

Lemma 1 (Least upper bound).

1. $\varphi_1 + \varphi_2$ *is the least upper bound (wrt. sub-effecting) of* φ_1 *and* φ_2
2. $T_1 \vee T_2$ *is the least upper bound (wrt. subtyping) of* T_1 *and* T_2.

Proof. For the effects in part 1: that $\varphi_1 + \varphi_2$ is an upper bound of φ_1 and φ_2 is immediate by SE-CHOICE$_1$ (plus commutativity and transitivity of $\equiv$). That it is the least upper bound is shown as follows: Assume a φ' s.t. $\varphi_1 \leq \varphi'$ and $\varphi_2 \leq \varphi'$. By SE-CHOICE$_2$, this implies $\varphi_1 + \varphi_2 \leq \varphi' + \varphi'$. Since further $\varphi' + \varphi' \equiv \varphi'$ by EE-CHOICE, the claim follows. Part 2 for the types is an immediate consequence. □

The following simple lemmas are needed later (in Lemma 29) as part of the completeness proof.

Lemma 2 ($\leq$ and $\vee$). *If* $\Gamma \vdash S_1 \leq T_1$ *and* $\Gamma \vdash S_2 \leq T_2$*, then* $\Gamma \vdash S_1 \vee S_2 \leq T_1 \vee T_2$.

Proof. Straightforward from the definitions of $\vee$ and $\leq$: The only interesting case is where the S_i's and T_i's are arrow types, and the lemma follows by rule SE-CHOICE$_2$. □

Lemma 3 ($\leq$ and substitution).

1. $\Gamma \vdash S_1 \leq S_2$ *implies* $\theta\Gamma \vdash \theta S_1 \leq \theta S_2$.
2. $\varphi_1 \leq \varphi_2$ *implies* $\theta\varphi_1 \leq \theta\varphi_2$.

Proof. Straightforward. □

The following two lemma provides a characterization of subtypes of a type of a given form (arrow type or universally quantified type scheme).

Lemma 4 (Characterization of subtypes).

1. *If* $\Gamma \vdash T \leq T_1 \xrightarrow{\varphi} T_2$*, then* $T = T_1 \xrightarrow{\varphi'} T_2$ *with* $\varphi' \leq \varphi$.
2. *If* $\Gamma \vdash S \leq \forall\rho.S_2$*, then* $S = \forall\rho.S_1$ *with* $S_1 \leq S_2$.

Proof. Straightforward, (under the restriction to first-order), inverting the subtyping rules from Table 6. □

Lemma 5 (Minimal effect). *If* $\Gamma \vdash \varphi_1 \leq \varphi_2 \equiv \varepsilon$ *then* $\varphi_1 \equiv \varepsilon$.

Proof. By straightforward induction on the sub-effecting rules of Table 7. The only rules which apply are the following: For SE-REFL, the result is immediate and the case for transitivity follows by straightforward induction. The remaining cases are straightforward, too. □

Thus ε is a minimal element wrt. $\leq$ (modulo $\equiv$), but note that it is not the least, i.e., we do not have $\varepsilon \leq \varphi$ for all φ.

Lemma 6. *If $\Gamma \vdash S_1 \leq S_2$. Then $fv_T(S_1) = fv_T(S_2)$.*

Proof. Obvious. Note that fv_T gives back the free variables without those variables occurring in the latent effects. □

The following is a simple observation about which variables can occur free in the typing derivations (in the specification).

Lemma 7 (Sub-effecting and free variables).

1. *If $\Gamma \vdash \varphi_1 \leq \varphi_2$, and $fv(\varphi_1) \subseteq fv_T(\Gamma)$, then $fv(\varphi_2) \subseteq fv_T(\Gamma)$.*
2. *If $\Gamma \vdash \varphi_1 \leq \varphi_2$, and $fv(\varphi_1) \subseteq fv_T(\Gamma) \cup fv_T(S)$ for some type S, then $fv(\varphi_2) \subseteq fv_T(\Gamma) \cup fv_T(S)$.*

Proof. Directly from the rules of Table 7; cf. especially rule SE-CHOICE$_1$. □

The algorithm does not only give back the "best" type wrt. subtyping/sub-effecting. Unification is used to synthesize the "best" type in the sense of the most general. To express that order, we introduce the following definitions.

Definition 5 (Instantiation and ordering). *We write $S_1 \lesssim_\theta S_2$ for $S_1 = \theta S_2$ and $S_1 \lesssim S_2$, if $S_1 \lesssim_\theta S_2$ for some θ. In abuse of notation we use the same notation for the order on substitutions, i.e., $\theta_1 \lesssim_\theta \theta_2$ if $\theta_1 = \theta \circ \theta_2$ and $\theta_1 \lesssim \theta_2$, if $\theta_1 \lesssim_\theta \theta_2$, for some θ. $\forall \vec{\rho}.S$ is a* generic *instance of $\forall \vec{\rho}'.S'$, written $\forall \vec{\rho}.S \lesssim^g \forall \vec{\rho}'.S'$, iff $S = [T_i/\rho_i']S'$ for some types T_i, and ρ_j does not occur free in $\forall \vec{\rho}'.S'$.*

Lemma 8 ($\vee$ and instantiation). *Let $S = S_1 \vee S_2$. If $T \in INST_\Gamma(S)$, then $T = T_1 \vee T_2$, for some $T_1 \in INST_\Gamma(S_1)$ and $T_2 \in INST_\Gamma(T_2)$. I.e., wlog. $INST_\Gamma(S_1 \vee S_2) = INST_\Gamma(S_1) \vee INST_\Gamma(S_2)$.*

Proof. Straightforward. □

Lemma 9. *If $\vec{T}_1 \xrightarrow{\varphi} T_2 \lesssim^g S$, then $S = \forall \vec{\rho}.\vec{T}_1' \xrightarrow{\varphi'} T_2'$, s.t. $\vec{T}_1 \xrightarrow{\varphi} T_2 = [S_i/\rho_i](\vec{T}_1' \xrightarrow{\varphi} T_2')$ for some types S_i.*

Proof. Straightforward by the definition of generic instance. □

Lemma 10 (Generic instance). *If $S_1 \lesssim^g S_2$, then $INST_\Gamma(S_1) \subseteq INST_\Gamma(S_2)$.*

Proof. Straightforward by the definition of $\lesssim^g$ and *INST*. □

Lemma 11 (Generic instance and substitution). *Assume $dom(\theta) \subseteq fv(S)$ and $dom(\theta) \cap fv(\Gamma) = \emptyset$. Then $\theta\ INST_\Gamma(S) = INST_\Gamma(\theta S)$.*

Proof. Straightforward. Note that $INST_\Gamma(S)$ uses only (fresh) variables not occurring in Γ for instantiation for the bound variables in S and that the substitution θ affects only the free variables in S. □

Corollary 1. *Assume $dom(\theta) \subseteq fv_T(S_2)$ and $dom(\theta) \cap fv(\Gamma) = \emptyset$. If $S_1 \lesssim^g \theta S_2$, then $INST_\Gamma(S_1) \subseteq \theta INST_\Gamma(S_2)$.*

Proof. By Lemma 10 and 11, $INST_\Gamma(S_1) \subseteq INST_\Gamma(\theta S_2) = \theta INST_\Gamma(S_2)$. □

Lemma 12 (Generic instance and closure). *$T \lesssim^g close_\Gamma(T)$, for all T and Γ.*

Proof. Immediate. □

Lemma 13 (Free variables and substitution).

1. *$fv(S) \subseteq fv(T)$ implies $fv(\theta S) \subseteq fv(\theta T)$.*
2. *$fv(S) \subseteq fv(\Gamma)$ implies $fv(\theta S) \subseteq fv(\theta\Gamma)$.*

Proof. Straightforward. □

Lemma 14 (Free variables and substitution). *Let $dom(\theta) = fv(T) \cap X$. Then $fv(\theta T) = fv(ran(\theta)) \cup (fv(T) \setminus X)$.*

Proof. Immediate. □

Lemma 15 (Free variables and substitution). *Assume $dom(\theta) = fv(T) \cap X$ and furthermore $fv(ran(\theta)) \subseteq X$. Then $fv(\theta T) \setminus X = fv(T) \setminus X$.*

Proof. The free variables of T can be split into $fv(T) \cap X$ and $fv(T) \setminus X$.

$$\begin{aligned} fv(\theta T) \setminus X &= (fv(ran(\theta)) \cup (fv(T) \setminus X)) \setminus X && \text{(Lemma 14)} \\ &= (fv(ran(\theta)) \setminus X) \cup (fv(T) \setminus X) \\ &= fv(T) \setminus X\ . && \text{(by assumption } fv(ran(\theta)) \subseteq X) \end{aligned}$$

□

Lemma 16 (Closure and substitution). *Let $dom(\theta) = fv(T) \cap fv(\Gamma)$. If $fv(T) \setminus fv(\Gamma) = fv(\theta T) \setminus fv(\Gamma)$ then $\theta close_\Gamma(T) = close_\Gamma(\theta T)$.*

Proof. Immediate from the definition of closure. □

Lemma 17 (Closure and substitution). *Assume $dom(\theta) = fv(T) \cap fv(\Gamma)$ and furthermore $fv(ran(\theta)) \subseteq fv(\Gamma)$. Then $\theta close_\Gamma(T) = close_\Gamma(\theta T)$.*

Proof. By Lemma 15, $fv(\theta T) \setminus fv(\Gamma) = fv(T) \setminus fv(\Gamma)$, and the result follows by Lemma 16. □

Lemma 18. *Let T be a type and $T \lesssim^g \theta \forall \vec{\rho}.S$, then $T = \tilde{\theta}\theta S$ for some $\tilde{\theta}$.*

Proof. Obvious. □

The following are simple facts concerning free variables in type schemes.

Lemma 19 (Free variables and closure). *$fv(close_\Gamma(T)) \subseteq fv(\Gamma)$ and $fv(close_\Gamma(T)) \subseteq fv(T)$.*

Proof. Straightforward by the definition of closure. □

Lemma 20 ($\lesssim^g$ and instantiation). *If $T_1 \lesssim^g T_2$, then $\theta T_1 \lesssim^g \theta T_2$*

Proof. Straightforward.

Lemma 21 (Generalization). *If $S_1 \lesssim^g S_2$ and $\rho \notin fv(S_2)$, then $\forall \rho . S_1 \lesssim^g S_2$.*

Proof. Straightforward. □

Lemma 22. *For all types, $T \lesssim \lceil \lfloor T \rfloor \rceil_A$.*

Proof. Straightforward. □

The following is a simple fact about substitution. Let's write $\theta \downarrow_{\vec{\rho}}$ for the projection of θ to the variables $\vec{\rho}$. Furthermore, if $dom(\theta_1) \cap dom(\theta_2) = \emptyset$, we write $\theta_1 \parallel \theta_2$ for the substitution θ s.t. $\theta(\rho) = \theta_1(\rho)$ when $\rho \in dom(\theta_1)$ and $\theta(\rho) = \theta_2(\rho)$ when $\rho \in dom(\theta_2)$.

Lemma 23. *If $\theta_1 T = \theta_2 T$, then $\theta_1 \downarrow_{fv(T)} = \theta_2 \downarrow_{fv(T)}$.*

Proof. Obvious. □

Lemma 24 (Disjoint domains and unifier). *Assume $fv(T_1) \cap fv(T_2) = \emptyset$. If $\theta_1 T_1 = S = \theta_2 T_2$ for some substitutions θ_1 and θ_2, then $\theta T_1 = \theta T_2 = S$ for some θ.*

Proof. Wlog., $dom(\theta_1) = fv(T_1)$ and $dom(\theta_2) = fv(T_2)$, i.e., also the domains of θ_1 and θ_2 are disjoint. Then setting $\theta = \theta_1 \parallel \theta_2$ gives $\theta T_1 = \theta_1 T_1 = S = \theta_2 T_2 = \theta T_2$, as required. □

The following lemma is ultimately crucial for the induction step in the proof of completeness of the algorithm and is, in broad terms, a refinement of the key property of unification: applying the most general unifier of two types yields a more general type than any other type which results from unification.

Lemma 25 (Unification and $\lesssim$).

$$\frac{\theta_1 T_1' = \theta_2 \theta_2' T_1' \qquad T \lesssim_{\theta_1} T_1' \qquad T \lesssim_{\theta_2} T_2' \qquad T_3' = \theta_2' T_1' \wedge_{\theta_3'} T_2'}{\theta_2 = \theta_3 \theta_3' \qquad T \lesssim_{\theta_3} T_3'} \tag{5}$$

Proof. The following chain $\theta_2 \theta_2' T_1' = \theta_1 T_1' = T = \theta_2 T_2'$ shows that θ_2 is a unifier of $\theta_2' T_1'$ and T_2'. Since θ_3' is the most general one, $\theta_2 \lesssim_{\theta_3} \theta_3'$ for some θ_3.

Using that, the second result in the conclusion is shown as follows:

$$\begin{aligned} \theta_3 T_3' &= \theta_3 \theta_3' \theta_2' T_1' && \text{(by definition of } T_3') \\ &= \theta_2 \theta_2' T_1' && \text{(equation for } \theta_2) \\ &= \theta_1 T_1' && \text{(first premise)} \\ &= T\ . && \text{(second premise)} \end{aligned}$$

Note that alternatively, the equality can be proven using $T_3' = \theta_3' T_2$ and the second premise instead. □

The next result is a generalization of the previous, taking into account also generic instantiation.

Lemma 26 (Unification, $\lesssim$ and $\lesssim^g$).

$$\frac{S \lesssim^g \theta_1 close_{\theta_1'\Gamma'}(T_1') \quad S \lesssim^g \theta_2 close_{\theta_2'\theta_1'\Gamma'}(T_2') \quad T_3' = \theta_2' T_1' \wedge_{\theta_3'} T_2' \quad \theta_1 T_1' = \theta_2 \theta_2' T_1'}{S \lesssim^g \theta_3 close_{\theta_3'\theta_2'\theta_1'\Gamma'}(T_3')} \tag{6}$$

Proof. By assumption we have

$$S \lesssim^g \theta_1 close_{\theta_1'\Gamma'}(T_1') \quad \text{and} \quad S \lesssim^g \theta_2 close_{\theta_2'\theta_1'\Gamma'}(T_2') \; . \tag{7}$$

The type scheme S is of the form $\forall \vec{\rho}.T$ for some type T. Since $T \lesssim^g \forall \vec{\rho}.T = S$, equation (7) implies with transitivity of $\lesssim^g$ that also

$$T \lesssim^g \theta_1 close_{\theta_1'\Gamma'}(T_1') \quad \text{and} \quad T \lesssim^g \theta_2 close_{\theta_2'\theta_1'\Gamma'}(T_2') \; . \tag{8}$$

By Lemma 18, this is equivalent to

$$T = \tilde{\theta}_1 \theta_1 T_1' \quad \text{and} \quad T = \tilde{\theta}_2 \theta_2 T_2' \; . \tag{9}$$

for some substitutions $\tilde{\theta}_1$ and $\tilde{\theta}_2$. As wlog the domains of $\tilde{\theta}_1$ and $\tilde{\theta}_2$ are disjoint, that implies with the help of Lemma 24

$$T = \tilde{\theta} \theta_1 T_1' \quad \text{and} \quad T = \tilde{\theta} \theta_2 T_2' \; . \tag{10}$$

for some common substitution $\tilde{\theta}$. The first assumption from equation (6) implies

$$\tilde{\theta} \theta_1 T_1' = \tilde{\theta} \theta_2 \theta_2' T_1' \; . \tag{11}$$

Now the previous Lemma 25 applies, yielding

$$\tilde{\theta} \tilde{\theta}_2 = \tilde{\theta}_3 \theta_3' \quad \text{and} \quad T \lesssim_{\tilde{\theta}_3} T_3' \tag{12}$$

for some substitution $\tilde{\theta}_3$. Now the variables of T_3' (covered by $\tilde{\theta}_3$) can be split into those in $\theta_3' \theta_2' \theta_1' \Gamma'$ (non-generic) and those *not* in $\theta_3' \theta_2' \theta_1' \Gamma'$, i.e., the generic ones. Thus, $\tilde{\theta}_3$ can be split into θ_3 with $dom(\theta_3) = fv(T_3') \cap fv(\theta_3' \theta_2' \theta_1' \Gamma')$ and $dom(\theta_3'') = fv(T_3') \setminus fv(\theta_3' \theta_2' \theta_1' \Gamma')$. Furthermore,

$$ran(\theta_3) \cap dom(\theta_3'') = \emptyset \quad \text{and} \quad fv(ran(\theta_3)) \subseteq fv(\theta_3' \theta_2' \theta_1' \Gamma') \; . \tag{13}$$

The left-hand equation gives

$$T = \theta_3'' \theta_3 T_3' \; . \tag{14}$$

Hence by definition of closure

$$T \lesssim^g close_{\theta_3'\theta_2'\theta_1'\Gamma'}(\theta_3 T_3') \; . \tag{15}$$

and further using the right-hand side of equation (13) and Lemma 17

$$T \lesssim^g \theta_3 close_{\theta_3'\theta_2'\theta_1'\Gamma'}(T_3') \; , \tag{16}$$

Finally, with the generalization Lemma 21

$$S \lesssim^g \theta_3 close_{\theta_3'\theta_2'\theta_1'\Gamma'}(T_3') \; , \tag{17}$$

as required. □

3.5.2 Soundness As usual, soundness is basically straightforward, whereas completeness later requires a careful formulation of the relationship between specification and algorithm to allow an inductive proof.

The next lemma captures a straightforward property of the type and effect system, namely preservation of typing under substitution.

Lemma 27 (Substitution). *If* $\Gamma \vdash e : S :: \varphi$*, then* $\theta\Gamma \vdash e : \theta S :: \theta\varphi$ *for all* θ*.*

Proof. By straightforward induction on the the derivation. □

Lemma 28 (Soundness). *If* $\Gamma \vdash_A e : T :: \varphi, \theta$*, then* $\theta\Gamma \vdash e : T :: \varphi$*.*

Proof. Assume $\Gamma \vdash_A e : T :: \varphi, \theta$ and proceed by induction on the derivation by the rules of Table 10.

Case: TA-VAR: $\Gamma \vdash_A x : \mathit{INST}_\Gamma(S) :: \varepsilon, \mathit{id}$
where $\Gamma(x) = S = \forall\vec{\rho}.T$. The instance $\mathit{INST}_\Gamma(S)$ uses fresh variables. Thus, wlog., $\mathit{INST}_\Gamma(S) = \theta' T$ where $\mathit{dom}(\theta') = \vec{\rho}$ and the case follows by T-VAR and an appropriate number of applications of T-INST:

$$\dfrac{\dfrac{\Gamma(x) = S}{\Gamma \vdash x : S :: \varepsilon}\,\text{T-VAR} \qquad S = \forall\vec{\rho}.T \qquad \vec{\rho} = \mathit{dom}(\theta') \qquad \mathit{INST}_\Gamma(S) = \theta' T}{\Gamma \vdash x : \mathit{INST}_\Gamma(S) :: \varepsilon}\,\text{T-INST}^*$$

Case: TA-PREF and TA-LREF
Both cases are trivial, with $\theta = \mathit{id}$.

Case: TA-ABS:
We are given

$$\dfrac{\vec{T}_1 = \lceil\vec{T}\rceil_A \qquad \Gamma, \vec{x}{:}\vec{T}_1 \vdash_A e : T_2 :: \varphi, \theta \qquad \vec{\rho} = \mathit{fv}(\vec{T}_1)}{\Gamma \vdash_A \mathtt{fn}\,\vec{x}{:}\vec{T}.e : (\theta\vec{T}_1) \xrightarrow{\varphi} T_2 :: \varepsilon, \theta \setminus \vec{\rho}}$$

and we need to prove $(\theta \setminus \vec{\rho})\Gamma \vdash \mathtt{fn}\, x{:}T.e : (\theta\vec{T}_1) \xrightarrow{\varphi} T_2 :: \varepsilon$. Since the $\vec{\rho}$ are fresh, $(\theta \setminus \vec{\rho})\Gamma = \theta\Gamma$. So by induction on the second premise, we conclude

$$\dfrac{\lfloor\theta\vec{T}_1\rfloor = \vec{T} \qquad \theta(\Gamma, x{:}\vec{T}_1) \vdash e : T_2 :: \varphi}{\theta\Gamma \vdash \mathtt{fn}\, x{:}T.e : (\theta\vec{T}_1) \xrightarrow{\varphi} T_2 :: \varepsilon}\,\text{T-ABS}$$

Case: TA-ABS$_{rec}$:
We are given

$$\dfrac{\begin{array}{c}\vec{T}_1' = \lceil\vec{T}_1\rceil_A \quad T_2' = \lceil T_2\rceil_A \quad \Gamma, f{:}\vec{T}_1' \xrightarrow{X} T_2', x{:}\vec{T}_1' \vdash_A e : T :: \varphi, \theta \\ X\ \mathit{fresh} \quad \lfloor T\rfloor = T_2 \quad \vec{\rho} = \mathit{fv}(\vec{T}_1' \to T_2') \quad \theta_1 = \mathscr{U}(T, \theta T_2')\end{array}}{\Gamma \vdash_A \mathtt{fun}\, f{:}\vec{T}_1 \to T_2.x{:}\vec{T}_1.e : (\theta_1\theta\vec{T}_1') \xrightarrow{\theta_1(\mathit{rec}X.\varphi)} (\theta_1\theta T_2') :: \varepsilon,\ \theta_1 \circ \theta \setminus \vec{\rho}}$$

and need to prove $(\theta_1\theta \setminus \vec{\rho})\Gamma \vdash \mathtt{fun}\, f{:}\vec{T}_1 \to T_2.x{:}\vec{T}_1.e : (\theta_1\theta\vec{T}_1') \xrightarrow{\theta_1(\mathit{rec}X.\varphi)} (\theta_1\theta T_2') :: \varepsilon$. Since the variables $\vec{\rho}$ are fresh, $\theta \setminus \vec{\rho}\Gamma = \theta\Gamma$. Induction on the typing subgoal yields

$\theta(\Gamma, f{:}\vec{T}_1' \xrightarrow{X} T_2', x{:}\vec{T}_1') \vdash e : T :: \varphi$. The substitution Lemma 27 and using $\theta_1 T = \theta_1 \theta T_2'$ —θ_1 is a unifier of T and $\theta T_2'$— further gives $\theta_1\theta\Gamma, f{:}\theta_1\theta\vec{T}_1' \xrightarrow{\theta_1\theta X} \theta_1\theta T_2', x{:}\theta_1\theta\vec{T}_1' \vdash e : \theta_1\theta T_2' :: \theta_1\varphi$, and we can conclude with T-ABS$_{rec}$, and using $\theta \setminus \vec{\rho}\ \Gamma = \Gamma$:

$$\frac{\theta_1\theta\Gamma, f{:}\theta_1\theta\vec{T}_1' \xrightarrow{\theta_1\theta X} \theta_1\theta T_2', x{:}\theta_1\theta\vec{T}_1' \vdash e : \theta_1\theta T_2' :: \theta_1\varphi}{\theta_1\theta\Gamma \vdash \texttt{fun}\ f{:}\vec{T}_1 \to T_2.x{:}\vec{T}_1.e : (\theta_1\theta\vec{T}_1') \xrightarrow{\theta_1(rec X.\varphi)} (\theta_1\theta T_2') :: \varepsilon}\ \text{T-ABS}_{rec}$$

Case: TA-APP:
We are given

$$\frac{\Gamma \vdash_A v_1 : \vec{T}_2' \xrightarrow{\varphi} T' :: \varepsilon, id \qquad \Gamma \vdash_A v_2 : \vec{T}_2, \varepsilon, id \qquad \theta = \mathscr{U}(\vec{T}_2', \vec{T}_2)}{\Gamma \vdash_A v_1\ v_2 : \theta T' :: \theta\varphi, \theta}$$

and by induction on the first premise we get

$$\Gamma \vdash v_1 : \vec{T}_2{}' \xrightarrow{\varphi} T' :: \varepsilon\ .$$

which implies with the substitution Lemma 27 $\theta\Gamma \vdash v_1 : \theta(\vec{T}_2{}' \xrightarrow{\varphi} T') :: \varepsilon$. Using induction and the substitution lemma on the second premise gives $\theta\Gamma \vdash v_2 : \theta\vec{T}_2 :: \varepsilon$, and the case follows by rule T-APP:

$$\frac{\theta\Gamma \vdash v_1 : \theta(\vec{T}_2' \xrightarrow{\varphi} T') :: \varepsilon \quad \theta\Gamma \vdash v_2 : \theta\vec{T}_2 :: \varepsilon \qquad \theta\vec{T}_2' = \theta\vec{T}_2}{\theta\Gamma \vdash v_1 v_2 : \theta T' :: \theta\varphi}$$

Case: TA-COND:
We are given

$$\frac{\begin{array}{c}\Gamma \vdash_A v : \texttt{Bool} :: \varepsilon, id \quad \Gamma \vdash_A e_1 : T_1 :: \varphi_1, \theta_1 \quad \theta_1\Gamma \vdash_A e_2 : T_2 :: \varphi_2, \theta_2 \\ \theta_3 = \mathscr{U}(\theta_2 T_1, T_2) \quad \theta = \theta_3 \circ \theta_2 \circ \theta_1\end{array}}{\Gamma \vdash_A \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \theta_3\theta_2 T_1 \vee \theta_3 T_2 :: \theta_3\theta_2\varphi_1 + \theta_3\varphi_2,\ \theta}$$

Induction on the first three premises yields

$$\Gamma \vdash v : \texttt{Bool} :: \varepsilon, \quad \theta_1\Gamma \vdash e_1 :: T_1, \varphi_1, \text{ and } \quad \theta_2\theta_1\Gamma \vdash e_2 : T_2 :: \varphi_2\ ,$$

and further with the help of the substitution Lemma 27

$$\theta\Gamma \vdash v : \texttt{Bool} :: \varepsilon, \quad \theta\Gamma \vdash e_1 : \theta_3\theta_2 T_1 :: \theta_3\theta_2\varphi_1, \text{ and } \quad \theta\Gamma \vdash e_2 : \theta_3 T_2 :: \theta_3\varphi_2\ .$$

Letting $T = \theta_3\theta_2 T_1 \vee \theta_3 T_2$ and abbreviating $\varphi = \theta_3(\theta_2\varphi_1 + \varphi_2)$ we conclude the case by subsumption and rule T-COND:

$$\frac{\theta\Gamma \vdash v : \texttt{Bool} :: \varepsilon \qquad \dfrac{\dfrac{\theta_3\theta_2 T_1 \le T \quad \theta_3\theta_2\varphi_1 \le \varphi}{\theta\Gamma \vdash e_1 :: \theta_3\theta_2 T_1, \theta_3\theta_2\varphi_1}}{\theta\Gamma \vdash_A e_1 : T :: \varphi} \qquad \dfrac{\dfrac{\theta_3 T_2 \le T \quad \theta_3\varphi_2 \le \varphi}{\theta\Gamma \vdash e_2 : \theta_3 T_2 :: \theta_3\varphi_2}}{\theta\Gamma \vdash e_2 : T :: \varphi}}{\theta\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T :: \varphi}$$

Case: TA-LET:
We are given

$$\frac{\Gamma \vdash_A e_1 : T_1' :: \varphi_1, \theta_1 \quad S_1 = close_{\theta_1\Gamma}(T_1') \quad \lfloor S_1 \rfloor = T_1 \qquad \theta_1\Gamma, x{:}S_1 \vdash_A e_2 : T_2 :: \varphi_2, \theta_2}{\Gamma \vdash_A \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : T_2 :: \theta_2\varphi_1; \varphi_2, \theta_2 \circ \theta_1}$$

where $S_1 = \forall\vec{\rho}.T_1'$ for $\vec{\rho} = fv(T_1') \setminus fv(\theta_1\Gamma)$. Induction on the first subgoal and a number of instances of the generalization rule T-GEN from Table 8 gives $\theta_1\Gamma \vdash e_1 : S_1 :: \varphi_1$, and specializing with the substitution Lemma 27 gives $\theta_2\theta_1\Gamma \vdash e_1 : \theta_2 S_1 :: \theta_2\varphi_1$. The second subgoal $\lfloor S_1 \rfloor = T_1$ implies $\lfloor \theta_2 S_1 \rfloor = T_1$. By induction on the right-most subgoal, we have $\theta_2(\theta_1\Gamma, x{:}S_1) \vdash e_2 : T_2 :: \varphi_2$. So by T-LET,

$$\frac{\theta_2\theta_1\Gamma \vdash e_1 : \theta_2 S_1 :: \theta_2\varphi_1 \qquad \lfloor \theta_2 S_1 \rfloor = T_1 \qquad \theta_2\theta_1\Gamma, x{:}\theta_2 S_1 \vdash e_2 : T_2 :: \varphi_2}{\theta_2\theta_1\Gamma \vdash \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : T_2 :: \theta_2\varphi_1; \varphi_2}$$

which concludes the case.

Case: TA-SPAWN: $\Gamma \vdash_A \texttt{spawn}\ e : \texttt{Thread} :: \texttt{spawn}\ \varphi, \theta$
By straightforward induction on the premise of TA-SPAWN and using T-SPAWN.

Case: TA-LOCK:
We are given

$$\frac{\Gamma \vdash_A v : \mathrm{L}^r :: \varphi, \theta}{\Gamma \vdash_A v.\,\texttt{lock}{:}\mathrm{L}^r :: \mathrm{L}^r.\,\texttt{lock}, \theta}\ \text{TA-LOCK}$$

By induction, $\theta\Gamma \vdash v : \mathrm{L}^r :: \varphi$, whence the result follows by T-LOCK.

Case: TA-UNLOCK: $\Gamma \vdash_A v.\,\texttt{unlock}{:}\mathrm{L}^r :: \mathrm{L}^r.\,\texttt{unlock}, \theta$
Analogously to TA-LOCK. □

3.5.3 Completeness For completeness in the inverse direction we need to prove in principle that all typing judgments derivable by the specification are found by the algorithm as well. As the algorithm is deterministic whereas the specification is not, not all typings are literally given by the algorithm. Instead, and as usual, the algorithm gives back a type and effect that represents all possible typings from the specification. In our setting there are two sources of non-determinism in the specification. Weakening the result by subsumption, and the non-determinism inherent in the specialization and generalization rules. For the proof of completeness, we tackle both sources of non-determinism separately, eliminating subtyping/sub-effecting first.

As intermediate step, we *remove* the subsumption rule from Table 8 and "build in" the weakening by subsumption into those rules where it is needed, namely for application and for conditions. More precisely, apart from removing subsumption, we replace rule T-APP and T-COND by the versions of Table 11.

The following is a simple property of the intermediate system, needed in the minimal typing Lemma 30.

Lemma 29 (Strengthening). *Given $\Gamma, x{:}S_1 \vdash_2 e : S_2 :: \varphi$, where $\Gamma \vdash S_1' \leq S_1$ for some S_1', then $\Gamma, x{:}S_1' \vdash_2 e : S_2' :: \varphi'$ for some S_2' and φ' where $S_2' \leq S_2$ and $\varphi' \leq \varphi$.*

$$\frac{\Gamma \vdash_2 v : \texttt{Bool} :: \varepsilon \qquad \Gamma \vdash_2 e_1 : T_1 :: \varphi_1 \qquad \Gamma \vdash_2 e_2 : T_2 :: \varphi_2}{\Gamma \vdash_2 \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T_1 \vee T_2 :: \varphi_1 + \varphi_2}\ \text{T-COND}_2$$

$$\frac{\Gamma \vdash_2 v_1 : T_2 \xrightarrow{\varphi} T :: \varepsilon \qquad \Gamma \vdash_2 v_2 : T_2' :: \varepsilon \qquad \Gamma \vdash T_2' \leq T_2}{\Gamma \vdash_2 v_1\ v_2 : T :: \varphi}\ \text{T-APP}_2$$

Table 11. Intermediate type and effect system

Proof. Proceed by induction on the derivation of the typing judgement. The case of T-VAR is immediate, and likewise the ones for T-PREF and T-LREF. The case for T-SPAWN follows by induction, the one for T-NEWL is immediate. The case for T-LET follows by induction and with the help of SE-SEQ. Similarly, rules T-ABS and T-ABS$_{rec}$ follow straightforwardly by induction. The cases for T-LOCK and T-UNLOCK follow by induction and using SE-SEQ. Also T-GEN follows by induction and with the help of S-ALL.

Case: T-INST
We are given

$$\frac{\Gamma, x{:}S_1 \vdash e : \forall \rho . S_2 :: \varphi \qquad \theta = [\rho \to r]}{\Gamma, x{:}S_1 \vdash e : \theta S_2 :: \varphi}\ \text{T-INST}$$

By induction $\Gamma, x{:}S_1' \vdash e : S' :: \varphi'$ where $\varphi' \leq \varphi$ and $S' \leq \forall \rho . S_2$. The latter implies $S' = \forall \rho . S_2'$ with $S_2' \leq S_2$ by Lemma 4(2), and the result follows by preservation of $\leq$ under substitution (Lemma 3(1).

Case: T-COND
We are given

$$\frac{\Gamma, x{:}T \vdash_2 v : \texttt{Bool} \qquad \Gamma, x{:}T \vdash_2 e_1 : T_1 :: \varphi_1 \qquad \Gamma, x{:}T \vdash_2 e_2 : T_2 :: \varphi_2}{\Gamma, x{:}T \vdash_2 \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T_1 \vee T_2 :: \varphi_1 + \varphi_2}$$

By induction $\Gamma, x{:}T' \vdash_2 e_1 : T_1' :: \varphi_1'$ and $\Gamma, x{:}T' \vdash_2 e_2 : T_2' :: \varphi_2'$, where $T_1' \leq T_1$, $T_2' \leq T_2$, and further $\varphi_1' \leq \varphi_1'$ and $\varphi_2' \leq \varphi_2$. By Lemma 2, $T_1' \vee T_2' \leq T_1 \vee T_2$. Furthermore, by rule SE-CHOICE$_2$ $\varphi_1' + \varphi_2' \leq \varphi_2 + \varphi_2$, from which the result follows.

Case: T-APP
from Table 11 follows by induction. □

Lemma 30 (Minimal typing). *If $\Gamma \vdash e : S_1 :: \varphi_1$, then $\Gamma \vdash_2 e : S_2 :: \varphi_2$ where $S_2 \leq S_1$ and $\varphi_2 \leq \varphi_1$.*

Proof. By induction on derivations with the rules of $\vdash$. from Table 8. The cases for T-VAR, T-PREF, T-LREF, and T-NEWL are immediate by reflexivity of $\leq$.

Case: T-ABS
We are given

$$\frac{\lfloor \vec{T} \rfloor = \vec{T_1} \qquad \Gamma, \vec{x}{:}\vec{T} \vdash e : T_2 :: \varphi}{\Gamma \vdash \texttt{fn}\ \vec{x}{:}\vec{T_1} . e : \vec{T} \xrightarrow{\varphi} T_2 :: \varepsilon}$$

By induction we get $\Gamma, \vec{x}{:}\vec{T} \vdash_2 e : T_2' :: \varphi'$ with $T_2' \leq T_2$ and $\varphi' \leq \varphi$. Since our restriction to first-order, $T_2' = T_2$. That implies $\vec{T} \xrightarrow{\varphi'} T_2 \leq \vec{T} \xrightarrow{\varphi} T_2$ by rule S-ARROW, and the result follows by T-ABS.

Case: T-ABS$_{rec}$
We are given

$$\frac{\lfloor \vec{T}_1' \rfloor = \vec{T}_1 \qquad \lfloor T_2' \rfloor = T_2 \qquad \Gamma, f{:}\vec{T}_1' \xrightarrow{X} T_2', \vec{x}{:}\vec{T}_1' \vdash e : T_2' :: \varphi}{\Gamma \vdash \mathtt{fun}\ f{:}\vec{T}_1 \to T_2.\vec{x}{:}\vec{T}_1.e : \vec{T}_1' \xrightarrow{recX.\varphi} T_2' :: \varepsilon}$$

By induction, $\Gamma, f{:}\vec{T}_1' \xrightarrow{X} T_2', \vec{x}{:}\vec{T}_1' \vdash e : T_2'' :: \varphi'$ where $T_2'' \leq T_2'$ and $\varphi' \leq \varphi$. Under the restriction to first-order, $T_2'' = T_2'$ and hence $\vec{T}_1 \xrightarrow{recX.\varphi} T_2 \leq \vec{T}_1' \xrightarrow{recX.\varphi'} T_2$ by rule S-ARROW, and the case follows by T-ABS$_{rec}$.

Case: T-APP
We are given

$$\frac{\Gamma \vdash v_1 : \vec{T}_2 \xrightarrow{\varphi} T :: \varepsilon \qquad \Gamma \vdash \vec{v}_2 : \vec{T}_2 :: \varepsilon}{\Gamma \vdash v_1 \vec{v}_2 : T :: \varphi}$$

Induction of the first subgoal gives $\Gamma \vdash_2 v_1 : T_1' :: \varphi_1'$ where $T_1' \leq \vec{T}_2 \xrightarrow{\varphi} T$ and $\varphi_1' \leq \varepsilon$. By Lemma 4(1) and minimality of ε from Lemma 5, this implies $T_1' = \vec{T}_2 \xrightarrow{\varphi'} T$, where $\varphi' \leq \varphi$ and $\varphi_1' \equiv \varepsilon$. By induction on the second subgoal, $\Gamma \vdash_2 \vec{v}_2 : \vec{T}_2' :: \varphi_2'$ with $\vec{T}_2' \leq \vec{T}_2$ and, again with Lemma 5, $\varphi_2' \equiv \varepsilon$. Hence, by T-APP$_2$ of Table 11

$$\frac{\Gamma \vdash_2 v_1 : \vec{T}_2 \xrightarrow{\varphi'} T :: \varepsilon \qquad \Gamma \vdash_2 \vec{v}_2 : \vec{T}_2' \qquad T_2' \leq T_2 :: \varepsilon}{\Gamma \vdash_2 v_1\ \vec{v}_2 : T :: \varphi'}$$

which concludes the case.

Case: T-COND
We are given

$$\frac{\Gamma \vdash v : \mathtt{Bool} :: \varepsilon \qquad \Gamma \vdash e_1 : T :: \varphi \qquad \Gamma \vdash e_2 : T :: \varphi}{\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : T :: \varphi}$$

By induction, we have $\Gamma \vdash_2 v : \mathtt{Bool}$, $\Gamma \vdash_2 e_1 : T_1 :: \varphi_1$ and $\Gamma \vdash_2 e_2 : T_2 :: \varphi_2$, where $T_1 \leq T$, $T_2 \leq T$, $\varphi_1 \leq \varphi$, and $\varphi_2 \leq \varphi$. Hence by Lemma 1, $T_1 \vee T_2 \leq T$ and $\varphi_1 + \varphi_2 \leq \varphi$ and the case follows by T-COND$_2$.

Case: T-LET
We are given

$$\frac{\Gamma \vdash e_1 : S_1 :: \varphi_1 \qquad \lfloor S_1 \rfloor = T_1 \qquad \Gamma, x{:}S_1 \vdash e_2 : T_2 :: \varphi_2}{\Gamma \vdash \mathtt{let}\ x{:}T_1 = e_1\ \mathtt{in}\ e_2 : T_2 :: \varphi_1; \varphi_2}$$

By induction, $\Gamma \vdash_2 e_1 : S_1' :: \varphi_1'$ and

$$\Gamma, x{:}S_1 \vdash_2 e_2 : T_2' :: \varphi_2' \,, \tag{18}$$

where $S_1' \leq S_1$, $T_2' \leq T_2$, $\varphi_1' \leq \varphi_1$, and $\varphi_2' \leq \varphi_2$. The equality $\lfloor S_1 \rfloor = T_1$ and $S_1' \leq S_1$ implies $\lfloor S_1' \rfloor = T$. Furthermore, by strengthening from Lemma 29, equation (18) implies that

$$\Gamma, x{:}S_1' \vdash_2 e_2 : T_2'' :: \varphi_2'' \,, \tag{19}$$

for some T_2'' and φ_2'' where $T_2'' \leq T_2'$ and $\varphi_2'' \leq \varphi_2'$. Hence by transitivity $T_2'' \leq T_2$ and $\varphi_2'' \leq \varphi_2$, and by T-LET of Table 11 we get

$$\frac{\Gamma \vdash_2 e_1 : S_1' :: \varphi_1' \qquad \lfloor S_1' \rfloor = T_1 \qquad \Gamma, x{:}S_1' \vdash_2 e_2 : T_2'' :: \varphi_2''}{\Gamma \vdash_2 \mathtt{let}\ x{:}T_1 = e_1\ \mathtt{in}\ e_2 : T_2'' :: \varphi_1'; \varphi_2''}$$

Rule SE-SEQ gives $\varphi_1'; \varphi_2'' \leq \varphi_1; \varphi_2$, as required.

Case: T-SPAWN
We are given

$$\frac{\Gamma \vdash e : S :: \varphi}{\Gamma \vdash \mathtt{spawn}\ e : \mathtt{Thread}{::}\mathtt{spawn}\ \varphi}$$

By induction, we get $\Gamma \vdash_2 e : S' :: \varphi'$, where $S' \leq S$ and $\varphi' \leq \varphi$. Hence the result follows by T-SPAWN and the fact that $\varphi' \leq \varphi$ implies $\mathtt{spawn}\ \varphi' \leq \mathtt{spawn}\ \varphi$ by SE-SPAWN.

Case: T-LOCK and T-UNLOCK
By induction and the rules for $\leq$ concerning sequential composition.

Case: T-SUB
By straightforward induction and transitivity.

Case: T-GEN
We are given

$$\frac{\Gamma \vdash e : S :: \varphi \qquad \rho \notin fv_T(\Gamma) \qquad \rho \in fv_T(S)}{\Gamma \vdash e : \forall \rho.S :: \varphi}$$

We get $\Gamma \vdash_2 e : S' :: \varphi'$ by induction where $S' \leq S$ and $\varphi' \leq \varphi$. Hence $\Gamma \vdash_2 e : \forall \rho.S' :: \varphi'$ by rule T-GEN; note that by Lemma 6 $\rho \in fv_T(S)$ and $S' \leq S$ implies that also $\rho \in fv_T(S')$. Finally $\forall \rho.S' \leq \forall \rho.S$ by S-ALL.

Case: T-INST
We are given

$$\frac{\Gamma \vdash e : \forall \rho.S_1 :: \varphi \qquad \rho = dom(\theta)}{\Gamma \vdash e : \theta S_1 :: \varphi}$$

and by induction $\Gamma \vdash_2 e : S_2 :: \varphi'$ where $S_2 \leq \forall \rho.S_1$ and $\varphi' \leq \varphi$. By Lemma 4(2), $S_2 = \forall \rho.S_2'$ with $S_2' \leq S_1$. Hence the case follows by Lemma 3, the instantiation rule T-INST and S-ALL. □

The second step in the completeness proofs gets rid of the non-determinism of the instantiation and generalization rules, by generalizing functions when they are introduced and specializing them when they are used, relying on unification.

Definition 6 (Well-formed context). *Let $fv_B(\Gamma)$ denote all free variables $fv_T(T)$ for all bindings of the form $x{:}T$ in Γ were T is not of the form $\forall \vec{\rho}.\vec{T}_1 \to T_2$. Well-formedness of a context Γ is defined inductively as follows: the empty context is well-formed. For $\Gamma = \Gamma', x{:}\forall \vec{\rho}.\vec{T}_1 \xrightarrow{\varphi} T_2$, Γ is well-formed if Γ' is well-formed and if*

1. $fv_T(T_2) \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma)$
2. $fv(\varphi) \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma)$.

If $\Gamma = \Gamma', x{:}T$ otherwise, Γ is well-formed if Γ' is well-formed.

Lemma 31 (Free variables). *Assume Γ is ok, and $\Gamma \vdash e : S :: \varphi'$ by the rules of the type system of Table 8. Then the following holds:*

1. *(a) If $S = \forall\vec{\rho}.\vec{T}_1 \xrightarrow{\varphi} T_2$, then*
   - *i. $fv_T(T_2) \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma)$*
   - *ii. $fv(\varphi) \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma)$*

   (b) Otherwise, $fv_T(T) \subseteq fv_B(\Gamma)$.
2. $fv(\varphi') \subseteq fv_B(\Gamma)$.

The same is true for judgments derived in the intermediate system of Table 11.

Proof. By induction on the derivation.

Case: T-VAR
We are given

$$\frac{\Gamma(x) = S}{\Gamma \vdash x : S :: \varepsilon}$$

Part 1a follows by well-formedness of Γ. Part 2 for the effects is trivial.

Case: T-PREF, T-LREF, and T-NEWL
Trivial, since no variables are involved.

Case: T-COND
All parts by straightforward induction.

Case: T-LET
We are given

$$\frac{\Gamma \vdash e_1 : S_1 :: \varphi_1 \quad \lfloor S_1 \rfloor = T_1 \quad \Gamma, x{:}S_1 \vdash e_2 : T_2 :: \varphi_2}{\Gamma \vdash \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : T_2 :: \varphi_1; \varphi_2}$$

Induction on the first subgoal gives: if $S_1 = \forall\vec{\rho}.\vec{T} \xrightarrow{\varphi} T'$ for some $\vec{T}$, T' and φ, then

$$fv_T(T') \subseteq fv_T(\vec{T}) \cup fv_B(\Gamma), \quad \text{and} \quad fv(\varphi) \subseteq fv_T(\vec{T}) \cup fv_B(\Gamma) \tag{20}$$

otherwise,

$$fv_T(S_1) \subseteq fv_B(\Gamma)\ . \tag{21}$$

In the case of 1a, that is, in case $T_2 = \forall\vec{\rho}'.\vec{T}_2' \xrightarrow{\varphi'} T_2''$ for some $\vec{T}_2'$, T_2'', and φ'. We have to show $fv_T(T_2'') \subseteq fv_T(\vec{T}_2') \cup fv_B(\Gamma)$ and $fv(\varphi') \subseteq fv_T(\vec{T}_2) \cup fv_B(\Gamma)$. Induction on the second subgoal gives

$$fv_T(T_2'') \subseteq fv_T(\vec{T}_2') \cup fv_B(\Gamma, x{:}S_1) \quad \text{and} \quad fv(\varphi') \subseteq fv_T(\vec{T}_2') \cup fv_B(\Gamma, x{:}S_1)\ . \tag{22}$$

If $S_1 = \forall\vec{\rho}.\vec{T} \xrightarrow{\varphi} T'$, $fv_B(\Gamma, x{:}S_1) = fv_B(\Gamma)$ (as free variables of the type scheme are ignored in fv_B); otherwise, equation (21) implies $fv_B(\Gamma, x{:}S_1) = fv_B(\Gamma)$. Therefore, we have that

$$fv_B(\Gamma, x{:}S_1) = fv_B(\Gamma) \tag{23}$$

This together with equation (22) covers part 1a.

In case of part 1b, i.e., where T_2 is a type scheme/arrow type, we have to show $fv_T(T_2) \subseteq fv_B(\Gamma)$. Induction on the second subgoals gives $fv_T(T_2) \subseteq fv_B(\Gamma, x{:}S_1)$, and equation (23) implies $fv_T(T_2) \subseteq fv_B(\Gamma)$, as required.

For part 2 for the effects, we have to show $fv(\varphi_1;\varphi_2) \subseteq fv_B(\Gamma)$. Induction on the left- and right-most subgoals gives $fv(\varphi_1) \subseteq fv_B(\Gamma)$, resp. $fv(\varphi_2) \subseteq fv_B(\Gamma, x{:}S_1)$. By equation (23), the latter implies $fv(\varphi_2) \subseteq fv_B(\Gamma)$, which concludes the case.

Case: T-ABS
We are given

$$\frac{\lfloor\vec{T}\rfloor = \vec{T}_1 \qquad \Gamma, \vec{x}{:}\vec{T} \vdash e : T_2 :: \varphi}{\Gamma \vdash \mathtt{fn}\,\vec{x}{:}\vec{T}_1.e : \vec{T} \xrightarrow{\varphi} T_2 :: \varepsilon}$$

Under the restriction to first-order, neither $\vec{T}$ nor T_2 are arrow type schemes. Under this restriction, induction on the right premise gives

$$fv_T(T_2) \subseteq fv_B(\Gamma, x{:}\vec{T}) \quad \text{and} \quad fv(\varphi) \subseteq fv_B(\Gamma, x{:}\vec{T})\ . \tag{24}$$

Part 1a follows directly from equation (24), while part 2 for the effects is trivially true.

Case: T-ABS$_{rec}$
We are given that

$$\frac{\lfloor\vec{T}_1'\rfloor = \vec{T}_1 \qquad \lfloor T_2'\rfloor = T_2 \qquad \Gamma, f{:}\vec{T}_1' \xrightarrow{X} T_2', \vec{x}{:}\vec{T}_1' \vdash e : T_2' :: \varphi}{\Gamma \vdash \mathtt{fun}\ f{:}\vec{T}_1 \to T_2.\vec{x}{:}\vec{T}_1.e : \vec{T}_1' \xrightarrow{rec X.\varphi} T_2' :: \varepsilon}$$

Under the restriction to first-order, neither $\vec{T}$ nor T_2 are arrow type schemes. By induction, we get

$$fv_T(T_2') \subseteq fv_B(\Gamma, f{:}\vec{T}_1' \xrightarrow{X} T_2', \vec{x}{:}\vec{T}_1') \quad \text{and} \quad fv(\varphi) \subseteq fv_B(\Gamma, f{:}\vec{T}_1' \xrightarrow{X} T_2', \vec{x}{:}\vec{T}_1') \tag{25}$$

For part 1a, $fv_T(T_2') \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma)$ follows directly from equation (25) (note that fv_B ignores free variables in function types in Γ). Part 2 for the effects is trivially true.

Case: T-APP
We are given

$$\frac{\Gamma \vdash v_1 : \vec{T}_2 \xrightarrow{\varphi} T :: \varepsilon \qquad \Gamma \vdash \vec{v}_2 : \vec{T}_2 :: \varepsilon}{\Gamma \vdash v_1\ \vec{v}_2 : T :: \varphi}$$

Under the the restriction to first-order, we need only to show that $fv_T(T) \subseteq fv_B(\Gamma)$ in part 1b. Induction on the first subgoal gives $fv_T(T) \subseteq fv_T(\vec{T}_2) \cup fv_B(\Gamma)$, and $fv(\varphi) \subseteq fv_T(\vec{T}_2) \cup fv_B(\Gamma)$. Furthermore, induction on the second subgoal gives $fv_T(\vec{T}_2) \subseteq fv_B(\Gamma)$. Therefore, $fv_T(T) \subseteq fv_T(\vec{T}_2) \cup fv_B(\Gamma) = fv_B(\Gamma)$, as required.

For part 2, we get from part 1 that $fv(\varphi) \subseteq fv_T(\vec{T}_2) \cup fv_B(\Gamma)$ and $fv_T(\vec{T}_2) \subseteq fv_B(\Gamma)$. Thus, $fv(\varphi) \subseteq fv_B(\Gamma)$, which concludes the case.

Case: T-SPAWN
Part 1 for the type is trivial as no variables are involved, and part 2 follows by straightforward induction.

Case: T-LOCK
We are given

$$\frac{\Gamma \vdash v : \mathtt{L}^r :: \varphi}{\Gamma \vdash v.\,\mathtt{lock}\colon \mathtt{L}^r :: \varphi; \mathtt{L}^r.\,\mathtt{lock}}$$

Part 1 for the type follows by straightforward induction. For part 2, induction yields $fv_T(\mathtt{L}^r) \subseteq fv_B(\Gamma)$ which implies $fv(\mathtt{L}^r.\,\mathtt{lock}) \subseteq fv_B(\Gamma)$. This together with $fv(\varphi) \subseteq fv_B(\Gamma)$ concludes the case. The case of T-UNLOCK works analogously.

Case: T-SUB
We are given t

$$\frac{\Gamma \vdash e : S' :: \varphi_1' \qquad \Gamma \vdash S' \leq S \qquad \Gamma \vdash \varphi_1' \leq \varphi_1}{\Gamma \vdash e : S :: \varphi_1}$$

Induction on the first subgoal gives: if $S' = \forall \vec{\rho}.\vec{T}_1 \xrightarrow{\varphi_2'} T_2$ for some T_1, T_2, and φ_2', then

$$fv_T(T_2) \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma), \quad \text{and} \quad fv(\varphi_2') \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma)\,; \tag{26}$$

otherwise,

$$fv_T(S') \subseteq fv_B(\Gamma) \tag{27}$$

For part 1a, it's easy to see that $S \geq S'$ implies $S = \forall \vec{\rho}'.\vec{T}_1 \xrightarrow{\varphi_2} T_2$ with $\Gamma \vdash \varphi_2' \leq \varphi_2$. Then, we get $fv_T(T_2) \subseteq fv_T(T_1) \cup fv_B(\Gamma)$ directly from equation (26) for the first part of 1a. For the second part, $fv(\varphi_2') \subseteq fv_T(T_1) \cup fv_B(\Gamma)$ from equation (26), and $\Gamma \vdash \varphi_2' \leq \varphi_2$ implies by Lemma 7(2) that $fv(\varphi_2) \subseteq fv_T(T_1) \cup fv_B(\Gamma)$, as required.

For part 1b, by Lemma 6, $fv_T(S') = fv_T(S)$, this together with equation (27) implies $fv_T(S) \subseteq fv_B(\Gamma)$, as required.

Part 2 for the effects, $fv(\varphi_1') \subseteq fv_B(\Gamma)$ from the induction and the second subgoal $\Gamma \vdash \varphi_1' \leq \varphi_1$ implies $fv(\varphi_1) \subseteq fv_B(\Gamma)$ by the first part of Lemma 7, which concludes the case.

Case: T-GEN
We are given

$$\frac{\Gamma \vdash e : S :: \varphi \qquad \rho' \notin fv_T(\Gamma) \qquad \rho' \in fv_T(S)}{\Gamma \vdash e : \forall \rho'.S :: \varphi}$$

Induction on the first subgoal gives: if $S = \forall \vec{\rho}.\vec{T}_1 \xrightarrow{\varphi'} T_2$ for some T_1, T_2 and φ', then

$$fv_T(T_2) \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma), \quad \text{and} \quad fv(\varphi') \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma)\,; \tag{28}$$

otherwise,

$$fv_T(S) \subseteq fv_B(\Gamma)\,. \tag{29}$$

For part 1a, we have to show $fv_T(T_2) \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma)$ and $fv(\varphi') \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma)$ for $\Gamma \vdash e : \forall \rho'\, \forall \vec{\rho}.\vec{T}_1 \xrightarrow{\varphi'} T_2$, which follows directly from the induction.

By equation (29), part 1b does not apply.

Part 2 for the effects follows by straightforward induction.

Case: T-INST
In this case,

$$\frac{\Gamma \vdash e : \forall \rho'.S :: \varphi \qquad \rho' = dom(\theta)}{\Gamma \vdash e : \theta S :: \varphi}$$

In this case, $S = \forall \vec{\rho}.\vec{T}_1 \xrightarrow{\varphi'} T_2$ for some $\vec{T}_1$, T_2, and φ'. Induction on the left subgoal gives

$$fv_T(T_2) \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma)\ , \quad \text{and} \quad fv_T(\varphi') \subseteq fv_T(\vec{T}_1) \cup fv_B(\Gamma)\ . \tag{30}$$

Furthermore, $\theta S = \forall \vec{\rho}.\theta(\vec{T}_1 \xrightarrow{\varphi'} T_2)$ and since the substitution affects *only* ρ', i.e.. $\rho' = dom(\theta)$, equation (30) yields $fv_T(\theta T_2) \subseteq fv_T(\theta \vec{T}_1) \cup fv_B(\Gamma)$, and $fv_T(\theta\varphi') \subseteq fv_T(\theta\vec{T}_1) \cup fv_B(\Gamma)$, which concludes the case.

Part 2 for the effects follows by straightforward induction. □

Note that given $\Gamma \vdash e : T$, the inclusion $fv(T) \subseteq fv(\Gamma)$ does *not* hold: Free variables in T but not in Γ are the "fresh" ones rule T-GEN can quantify over. They are also the ones that are quantified over in the closure of a type (cf. Definition 3).

Let's write $\vdash_2'$ for derivations in a restricted system in which derivations do not *end* in a last step by using *instantiation* or *generalization*.

Lemma 32 (Transfer). *If $\vdash_A$ is complete wrt. $\vdash_2'$ (in the sense of Lemma 34), then so is it wrt. $\vdash_2$.*

Proof. We are given completeness of $\vdash_A$ wrt. the restricted derivations in $\vdash_2'$. Assume $\Gamma \vdash_2' e : T$ and further $\Gamma \lesssim \Gamma'$, i.e., Γ' is a generalization of Γ, resp. Γ is a non-generic instance of Γ'. Proceed by induction on the number n of instances of rule T-INST and T-GEN at the end of the derivation. The base case for $n = 0$ is immediate. So assume $n+1$. There are two cases, depending on which rule has been used in the last step.

Subcase: T-INST
We are given that

$$\frac{\Gamma \vdash_2' e : \forall \rho.S :: \varphi \qquad \rho = dom(\theta)}{\Gamma \vdash_2 e : \theta S :: \varphi}\ \text{T-INST}$$

i.e., $\theta S \lesssim^g \forall \rho.S$. By induction we get

$$\Gamma' \vdash_A e : T' :: \varphi', \theta_1' \text{ where } \Gamma \lesssim_{\theta_1} \theta_1'\Gamma',\ \forall \rho.S \lesssim^g \theta_1 close_{\theta'\Gamma'}(T'), \text{ and } \varphi \lesssim_{\theta_1} \varphi',$$

for some θ_1. Transitivity of $\lesssim^g$ gives $\theta S \lesssim^g \theta_1 close_{\theta'\Gamma'}(T')$ which concludes the case.

Subcase: T-GEN
By T-GEN, we are given that

$$\frac{\Gamma \vdash_2' e : S :: \varphi \qquad \rho \in fv_T(S) \qquad \rho \notin fv_T(\Gamma)}{\Gamma \vdash_2 e : \forall \rho.S :: \varphi}\ \text{T-GEN}$$

By induction, we get $\Gamma' \vdash_A e : T' :: \varphi', \theta'$ where $\Gamma \lesssim_\theta \theta'\Gamma'$ for some θ. Furthermore, we have $S \lesssim^g \theta\ close_{\theta'\Gamma'}(T')$ and $\varphi \lesssim_\theta \varphi'$. By Lemma 19, $fv(close_{\theta'\Gamma'}(T')) \subseteq fv(\theta'\Gamma')$, and therefore

$$fv(\theta\ close_{\theta'\Gamma'}(T')) \subseteq fv(\theta\theta'\Gamma') = fv(\Gamma) \tag{31}$$

by Lemma 13 and the induction hypothesis $\Gamma = \theta\theta'\Gamma'$. This implies $\rho \notin fv(\theta\ close_{\theta'\Gamma'}(T'))$ since $\rho \notin fv(\Gamma)$. This together with $S \lesssim^g \theta\ close_{\theta'\Gamma'}(T')$ and the generalization Lemma 21 yields $\forall\rho.S \lesssim^g \theta\ close_{\theta'\Gamma'}(T')$, as required. The rest of the conditions for completeness are covered directly by the induction hypothesis, which concludes the case. □

Lemma 33 (Weakening). *If $\Gamma, x{:}S_1 \vdash_2 e : S_2 :: \varphi$ and $S_1 \lesssim^g S_1'$, then $\Gamma, x{:}S_1' \vdash_2 e : S_2 :: \varphi$.*

Proof. In the derivation of $\Gamma, x{:}S_1 \vdash_2 e : S_2 :: \varphi$, instead of using T-VAR for $x{:}S_1$, we replace it with $x{:}S_1'$ followed by T-INST, and therefore $\Gamma, x{:}S_1' \vdash_2 e : S_2 :: \varphi$, as required. □

Lemma 34 (Completeness). *If $\Gamma \lesssim \Gamma'$ and $\Gamma \vdash_2' e : S :: \varphi$, then*

1. *$\Gamma' \vdash_A e : T' :: \varphi', \theta'$*
2. *There exists a θ, s.t.*
   (a) *$\Gamma \lesssim_\theta \theta'\Gamma'$*
   (b) *$S \lesssim^g \theta\ close_{\theta'\Gamma'}(T')$*
   (c) *$\varphi \lesssim_\theta \varphi'$.*

Proof. Assume $\Gamma \lesssim \Gamma'$ and $\Gamma \vdash_2' e : T :: \varphi$. The proof then proceeds by induction on the structure of e. Note that since the derivation in $\vdash_2'$ does not end by T-INST or T-GEN, the syntactic form of e determines the derivation rule used in the last step.

Case: $e = x$
With T-VAR as the only rule to justify the typing judgment, we know $\varphi = \varepsilon$ and $\Gamma(x) = S$. We are given that $\Gamma \lesssim \Gamma'$, i.e. $\Gamma = \tilde{\theta}\Gamma'$ for some $\tilde{\theta}$. Thus, $\Gamma'(x) = S'$ with $T = \tilde{\theta}S'$. Hence

$$\frac{\Gamma'(x) = S'}{\Gamma' \vdash_A e : INST_{\Gamma'}(S'), :: \varepsilon, id}\ \text{TA-VAR}$$

Wlog. $close_{\Gamma'}(INST_{\Gamma'}(S')) = S'$, thus $S = \tilde{\theta}\ close_{\Gamma'}(INST_{\Gamma'}(S'))$. Setting $\theta = \tilde{\theta}$, Part (2a) follows using $\theta' = id$. Part (2b) $S \lesssim^g \tilde{\theta}\ close_{\Gamma'}(INST_{\Gamma'}(S'))$ follows by reflexivity of $\lesssim^g$, and part (2c) is immediate.

Case: $e = l^r$
Straightforward.

Case: $e = \mathtt{fn}\,\vec{x}{:}\vec{T}_1.t$
In this case

$$\frac{\lfloor \vec{T} \rfloor = \vec{T}_1 \qquad \Gamma, \vec{x}{:}\vec{T} \vdash_2 t : T_2 :: \varphi}{\Gamma \vdash_2' \mathtt{fn}\,\vec{x}{:}\vec{T}_1.t : \vec{T} \xrightarrow{\varphi} T_2 :: \varepsilon}\ \text{T-ABS}$$

Let $\vec{T}' = \lceil \vec{T}_1 \rceil_A$, which implies $\vec{T} \lesssim \vec{T}'$ by Lemma 22. Since $\Gamma \lesssim \Gamma'$ the variables in $\vec{T}'$ are fresh, we have $\Gamma, x{:}\vec{T} \lesssim \Gamma', x{:}\vec{T}'$. Induction on t gives $\Gamma', x{:}\vec{T}' \vdash_A t : T_2' :: \varphi', \theta'$ and in addition,

$$\Gamma, x{:}\vec{T} \lesssim_\theta \theta'\Gamma', x{:}\theta'\vec{T}', \quad T_2 \lesssim^g \theta\ close_{\theta'\Gamma'}(T_2'), \quad \text{and} \quad \varphi \lesssim_\theta \varphi' \tag{32}$$

for some substitution θ. In our first-order setting, the second part of equation (32) can be replaced by the simpler $T_2 = \theta T_2'$. By rule TA-ABS,

$$\frac{\vec{T}' = \lceil \vec{T}_1 \rceil_A \qquad \Gamma', x{:}\vec{T}' \vdash_A t : T_2' :: \varphi', \theta' \qquad \vec{\rho} = fv(\vec{T}')}{\Gamma' \vdash_A \mathtt{fn}\, \vec{x}{:}\vec{T}_1.t : (\theta'\vec{T}') \xrightarrow{\varphi'} T_2' :: \varepsilon, \theta' \setminus \vec{\rho}}$$

From equation (32), $\Gamma \lesssim_\theta \theta'\Gamma'$, and since $\vec{\rho}$ are fresh, $\Gamma \lesssim_\theta (\theta' \setminus \vec{\rho})\Gamma'$, covering part (2a). Equation (32) furthermore gives $\vec{T} = \theta\theta'\vec{T}'$, and further $T_2 = \theta T_2'$ and $\varphi = \theta\varphi'$. Hence,

$$\vec{T} \xrightarrow{\varphi} T_2 = \theta(\theta'\vec{T}' \xrightarrow{\varphi'} T_2') \,. \tag{33}$$

By the closure Lemma 12, $\theta'\vec{T}' \xrightarrow{\varphi'} T_2' \lesssim^g close_{\theta'\Gamma'}(\theta'\vec{T}' \xrightarrow{\varphi'} T_2')$, and further with the substitution Lemma 20 $\theta(\theta'\vec{T}' \xrightarrow{\varphi'} T_2') \lesssim^g \theta\, close_{\theta'\Gamma'}(\theta'\vec{T}' \xrightarrow{\varphi'} T_2')$, and therefore equation (33) yields $\vec{T} \xrightarrow{\varphi} T_2 \lesssim^g \theta\, close_{\theta'\Gamma'}(\theta'\vec{T}' \xrightarrow{\varphi'} T_2')$, as required in part 2b. Part 2c is immediate.

Case: $e = \mathtt{fun}\, f{:}T.\vec{x}{:}\vec{T}.t$
In this case

$$\frac{\lfloor \vec{T}_1 \rfloor = \vec{T} \qquad \lfloor T_2 \rfloor = T' \qquad \Gamma, f{:}\vec{T}_1 \xrightarrow{X} T_2, \vec{x}{:}\vec{T}_1 \vdash_2 t : T_2 :: \varphi}{\Gamma \vdash_2' \mathtt{fun}\, f{:}\vec{T} \to T'.\vec{x}{:}\vec{T}.t : \vec{T}_1 \xrightarrow{recX.\varphi} T_2 :: \varepsilon}\ \text{T-ABS}_{rec}$$

Let $\vec{T}_1' = \lceil \vec{T} \rceil_A$ and $T_2' = \lceil T' \rceil_A$, which implies with Lemma 22 $\vec{T}_1 \lesssim \vec{T}_1'$ and $T_2 \lesssim T_2'$, and therefore further (since the introduced variables are fresh) $\Gamma, f{:}\vec{T}_1 \xrightarrow{X} T_2, x{:}\vec{T}_1 \lesssim \Gamma', f{:}\vec{T}_1' \xrightarrow{X} T_2', x{:}\vec{T}_1'$. Induction on the subterm t gives

$$\Gamma', f{:}\vec{T}_1' \xrightarrow{X} T_2', x{:}\vec{T}_1' \vdash_A t : T_2'' :: \varphi', \theta_1'$$

and in addition,

$$\begin{aligned} &\Gamma, f{:}\vec{T}_1 \xrightarrow{X} T_2, \vec{x}{:}\vec{T}_1 \lesssim_{\theta_1} \theta_1'(\Gamma', f{:}\vec{T}_1' \xrightarrow{X} T_2', x{:}\vec{T}_1'), \\ &T_2 \lesssim^g \theta_1\, close_{\theta_1'\Gamma'}(T_2''), \quad \text{and} \quad \varphi \lesssim_{\theta_1} \varphi' \end{aligned} \tag{34}$$

for some θ_1. Furthermore, the second inequation in (34) simplifies to $T_2 = \theta_1 T_2''$ in the first-order setting. Equation (34) and $\lfloor T_2 \rfloor = T'$ implies $\lfloor T_2'' \rfloor = T'$. Then, by rule TA-ABS$_{rec}$,

$$\frac{\begin{array}{c}\vec{T}_1' = \lceil \vec{T} \rceil_A \quad T_2' = \lceil T' \rceil_A \quad \Gamma', f{:}\vec{T}_1' \xrightarrow{X} T_2', x{:}\vec{T}_1' \vdash_A t : T_2'' :: \varphi', \theta_1' \\ X \text{ fresh} \quad \lfloor T_2'' \rfloor = T' \quad \vec{\rho} = fv(\vec{T}_1' \xrightarrow{X} T_2') \quad \theta_2' = \mathscr{U}(T_2'', \theta_1' T_2')\end{array}}{\Gamma' \vdash_A \mathtt{fun}\, f{:}\vec{T}_1 \to T_2.\vec{x}{:}\vec{T}_1.t : \theta_2'\theta_1'\vec{T}_1' \xrightarrow{\theta_2'(recX.\varphi')} \theta_2'\theta_1'T_2' :: \varepsilon, \theta_2' \circ \theta_1' \setminus \vec{\rho}}$$

The first inequation in (34) further implies $T_2 = \theta_1\theta_1'T_2'$. This together with $T_2 = \theta_1 T_2''$ means that θ_1 is unifier of $\theta_1'T_2'$ and T_2'', and since θ_2' is the most general one,

$$\theta_1 \lesssim_{\theta_2} \theta_2' \tag{35}$$

for some θ_2. With $\vec{\rho}$ fresh, $\theta_2'\theta_1'\Gamma' = (\theta_2'\theta_1' \setminus \vec{\rho})\Gamma'$, which gives together with left-most inequation of (34) $\Gamma \lesssim_{\theta_2} (\theta_2'\theta_1' \setminus \vec{\rho})\Gamma'$, as required in part 2a. Using (35), $\varphi \lesssim_{\theta_1} \varphi'$ can be re-written as $\varphi \lesssim_{\theta_2} \theta_2'\varphi'$ which implies $recX.\varphi_1 \lesssim_{\theta_2} \theta_2' recX.\varphi_1'$. The first inequation of (34) together with (35) gives further that $\vec{T}_1 \lesssim_{\theta_2} \theta_2'\theta_1'\vec{T}_1'$ and $T_2 \lesssim_{\theta_2} \theta_2'\theta_1'T_2'$ and thus

$$\vec{T}_1 \xrightarrow{recX.\varphi} T_2 = \theta_2(\theta_2'\theta_1'\vec{T}_1' \xrightarrow{\theta_2' recX.\varphi'} \theta_2'\theta_1'T_2') \,. \tag{36}$$

Let $\tilde{T} = \theta_2'\theta_1'\vec{T}_1' \xrightarrow{\theta_2' recX.\varphi'} \theta_2'\theta_1'T_2'$. By the closure Lemma 12, $\tilde{T} \lesssim^g close_{\theta_2'\theta_1'\Gamma'}(\tilde{T})$ and further with Lemma 20, $\theta_2\tilde{T} \lesssim^g \theta_2\ close_{\theta_2'\theta_1'\Gamma'}(\tilde{T})$. Then, $\theta_2'\theta_1'\Gamma' = (\theta_2'\theta_1' \setminus \vec{\rho})\Gamma'$ together with (36) gives $\vec{T}_1 \xrightarrow{recX.\varphi} T_2 \lesssim^g \theta_2 close_{(\theta_2'\theta_1' \setminus \vec{\rho})\Gamma'}(\tilde{T})$, as required in part 2b. Part 2c is immediate.

Case: $e = \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ t$
In this case

$$\frac{\Gamma \vdash_2 e_1 : S_1 :: \varphi_1 \qquad \lfloor S_1 \rfloor = T_1 \qquad \Gamma, x{:}S_1 \vdash_2 t : T_2 :: \varphi_2}{\Gamma \vdash_2' \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ t : T_2 :: \varphi_1;\varphi_2}\ \text{T-LET}$$

Induction on e_1 gives $\Gamma' \vdash_A e_1 : T_1' :: \varphi_1', \theta_1'$ and in addition we have

$$\Gamma \lesssim_{\theta_1} \theta_1'\Gamma', \quad S_1 \lesssim^g \theta_1\ close_{\theta_1'\Gamma'}(T_1'), \quad \text{and} \quad \varphi_1 = \theta_1\varphi_1' \tag{37}$$

for some θ_1. With the left-most inequation of (37), we can rewrite the judgement of the other subterm t as $\theta_1\theta_1'\Gamma', x{:}S_1 \vdash_2 t : T_2 :: \varphi_2$. By Lemma 33 and using the second inequation of (37), this can be weakened to

$$\theta_1\theta_1'\Gamma', x{:}\theta_1 S_1' \vdash_2 t : T_2 :: \varphi_2 \tag{38}$$

where $S_1' = close_{\theta_1'\Gamma'}(T_1')$. Since $\theta_1\theta_1'\Gamma', x{:}\theta_1 S_1' \lesssim_{\theta_1} \theta_1'\Gamma', x{:}S_1'$, induction on the subterm t in equation (38) gives $\theta_1'\Gamma', x{:}S_1' \vdash_A t : T_2' :: \varphi_2', \theta_2'$ where in addition

$$\theta_1(\theta_1'\Gamma', x{:}S_1') \lesssim_{\theta_2} \theta_2'(\theta_1'\Gamma', x{:}S_1') \,, \quad T_2 \lesssim^g \theta_2\ close_{\theta_2'\theta_1'\Gamma'}(T_2') \quad \text{and} \quad \varphi_2 = \theta_2\varphi_2' \tag{39}$$

for some θ_2. We have $\lfloor S_1 \rfloor = T_1$, which together with second inequation (37) implies $\lfloor S_1' \rfloor = T_1$. Then, by rule TA-LET,

$$\frac{\Gamma' \vdash_A e_1 : T_1' :: \varphi_1', \theta_1' \quad S_1' = close_{\theta_1'\Gamma'}(T_1') \quad \lfloor S_1' \rfloor = T_1 \quad \theta_1'\Gamma', x{:}S_1' \vdash_A t : T_2' :: \varphi_2', \theta_2'}{\Gamma' \vdash_A \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ t : T_2' :: \theta_1'\varphi_1';\varphi_2', \theta_2' \circ \theta_1'}$$

From the left-most inequations of (37) and of (39), we get $\Gamma = \theta_1\theta_1'\Gamma' = \theta_2\theta_2'\theta_1'\Gamma'$ covering part 2a. $T_2 \lesssim^g \theta_2\ close_{\theta_2'\theta_1'\Gamma'}(T_2')$ follows directly from the induction hypothesis in equation (39), covering 2b. Finally we have $\varphi_1 = \theta_1\varphi_1' = \theta_2\theta_2'\varphi_1'$, which yields $\varphi_1;\varphi_2 = \theta_2(\theta_2'\varphi_1';\varphi_2')$, covering part 2c, concluding the case.

Case: $e = \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$
In this case, we are given

$$\frac{\Gamma \vdash_2 v : \texttt{Bool} :: \varepsilon \qquad \Gamma \vdash_2 e_1 : T_1 :: \varphi_1 \qquad \Gamma \vdash_2 e_2 : T_2 :: \varphi_2}{\Gamma \vdash'_2 \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T_1 \vee T_2 :: \varphi_1 + \varphi_2}\ \text{T-COND}_2$$

First note that the existence of $T_1 \vee T_2$ implies that either $T_1 = T_2$ if T_1 and T_2 are no type schemes. Otherwise, $T_1 = \forall\vec{\rho}.U_1 \xrightarrow{\varphi_1} U_2$ and $T_2 = \forall\vec{\rho}.U_1 \xrightarrow{\varphi_2} U_2$ and $T_1 \vee T_2 = \forall\vec{\rho}.U_1 \xrightarrow{\varphi_1+\varphi_2} U_2$. Let's concentrate on the first case, the second one works analogously, remembering that unification on types does not take the annotations φ_1 and φ_2 into account. So let's set $T = T_1 \vee T_2 = T_1 = T_2$. Induction on the first subterm v gives $\Gamma' \vdash_A v : \texttt{Bool} :: \varepsilon, id$, and further $\Gamma \lesssim_\theta \Gamma'$ for some θ. Induction on the second subterm e_1 gives $\Gamma' \vdash_A e_1 : T'_1 :: \varphi'_1, \theta'_1$ and in addition

$$\Gamma \lesssim_{\theta_1} \theta'_1\Gamma', \quad T_1 \lesssim^g \theta_1\, close_{\theta'_1\Gamma'}(T'_1), \quad \text{and} \quad \varphi_1 \lesssim_{\theta_1} \varphi'_1 \tag{40}$$

for some θ_1. The left-most inequation (40) means $\Gamma \lesssim \theta'_1\Gamma'$ and furthermore the judgment of the subterm e_2 from the last premise can be rewritten as $\theta_1\theta'_1\Gamma' \vdash_2 e_2 : T_2 :: \varphi_2$. Hence, induction on e_2 gives $\theta'_1\Gamma' \vdash_A e_2 : T'_2 :: \varphi'_2, \theta'_2$, and further

$$\Gamma = \theta_1\theta'_1\Gamma' \lesssim_{\theta_2} \theta'_2\theta'_1\Gamma', \quad T_2 \lesssim^g \theta_2\, close_{\theta'_2\theta'_1\Gamma'}(T'_2), \quad \text{and} \quad \varphi_2 \lesssim_{\theta_2} \varphi'_2 \tag{41}$$

for some θ_2. Then, by TA-COND

$$\frac{\begin{array}{c}\Gamma' \vdash_A v : \texttt{Bool} :: \varepsilon, id \quad \Gamma' \vdash_A e_1 : T'_1 :: \varphi'_1, \theta'_1 \quad \theta'_1\Gamma' \vdash_A e_2 : T'_2 :: \varphi'_2, \theta'_2 \\ \theta'_3 = \mathscr{U}(\theta'_2 T'_1, T'_2) \quad T' = (\theta'_3\theta'_2 T'_1 \vee \theta'_3 T'_2)\end{array}}{\Gamma' \vdash_A \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T' :: \theta'_3\theta'_2\varphi'_1 + \theta'_3\varphi'_2, \theta'_3 \circ \theta'_2 \circ \theta'_1}$$

From equation (41), $\Gamma = \theta_1\theta'_1\Gamma' = \theta_2\theta'_2\theta'_1\Gamma'$, and since $fv(T'_1) \subseteq fv(\theta'_1\Gamma')$, this implies with Lemma 23

$$\theta'_1 T'_1 = \theta_2\theta'_2 T'_1 \tag{42}$$

and furthermore we have

$$T_2 = \theta_2 T'_2 \tag{43}$$

Hence θ_2 is a (another) unifier of $\theta'_2 T'_1$ and T'_2. Since θ'_3 is the most general one, $\theta_2 \lesssim_{\theta_3} \theta'_3$ for some substitution θ_3. This means $\Gamma \lesssim_{\theta_3} \theta'_3\theta'_2\theta'_1\Gamma'$, as required in part 2a.

The second conditions of the induction hypotheses from (40) and (41) together with equation (42) cover the premises of Lemma 26, which gives $T \lesssim^g \theta_3 close_{\theta'_3\theta'_2\theta'_1\Gamma'}(T'_3)$, as required in part 2b.

For the effect, we have $\varphi_1 = \theta_3\theta'_3\theta'_2\varphi'_1$ and $\varphi_2 = \theta_3\theta'_3\varphi'_2$. Therefore, $\varphi_1 + \varphi_2 = \theta_3(\theta'_3\theta'_2\varphi'_1 + \theta'_3\varphi'_2)$, as required for part 2c.

Case: $e = v\ \vec{v}$
Analogously.

Case: $e = \texttt{spawn}\, t$
In this case,

$$\frac{\Gamma \vdash_2 t : S :: \varphi}{\Gamma \vdash_2' \texttt{spawn}\, t : \texttt{Thread} :: \texttt{spawn}\, \varphi}\ \text{T-SPAWN}$$

Induction on the well-typed subterm t gives $\Gamma' \vdash_A t : T' :: \varphi', \theta'$, and further

$$\Gamma \lesssim_\theta \theta'\Gamma', \quad T \lesssim^g \theta close_{\theta'\Gamma'}(T'), \quad \text{and} \quad \varphi \lesssim_\theta \varphi'$$

for some θ. Then, by rule TA-SPAWN,

$$\frac{\Gamma' \vdash_A t : T' :: \varphi', \theta'}{\Gamma' \vdash_A \texttt{spawn}\, t : \texttt{Thread} :: \texttt{spawn}\, \varphi', \theta'}\ \text{TA-SPAWN}$$

The induction hypothesis $\varphi \lesssim_\theta \varphi'$ implies $\texttt{spawn}\, \varphi \lesssim_\theta \texttt{spawn}\, \varphi'$, which concludes the case.

Case: $e = \texttt{new}_\pi\, \texttt{L}$
Straightforward.

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

$$\frac{\Gamma \vdash_2 v : \texttt{L}^r :: \varphi}{\Gamma \vdash_2' v.\, \texttt{lock}: \texttt{L}^r :: \varphi; \texttt{L}^r.\, \texttt{lock}}$$

Induction on the subterm v gives $\Gamma' \vdash_A v : \texttt{L}^{r'} :: \varphi', \theta'$ where $\varphi' = \varepsilon$ and $\theta' = id$ by TA-VAR. In addition,

$$\Gamma \lesssim_\theta \Gamma', \quad \text{and} \quad \texttt{L}^r \lesssim^g \theta\ close_{\Gamma'}(\texttt{L}^{r'}) \tag{44}$$

for some substitution θ. With rule TA-LOCK, we get

$$\frac{\Gamma' \vdash_A v : \texttt{L}^{r'} :: \varepsilon, id}{\Gamma' \vdash_A v.\, \texttt{lock} : \texttt{L}^{r'} :: \texttt{L}^{r'}.\, \texttt{lock}, id}$$

Parts 2a and 2b are directly given by induction. For part 2c for the effects, the right induction hypothesis of (44) means $\texttt{L}^r = \theta\ \texttt{L}^{r'}$, and therefore $\texttt{L}^r.\, \texttt{lock} \lesssim_\theta \texttt{L}^{r'}.\, \texttt{lock}$, as required.

Case: $e = v.\, \texttt{unlock}$
Analogously. □

4 Conclusion

We have presented an algorithmic inference type- and effect system to reconstruct the type for an implicitly-typed program, and derives the abstract behaviour for a calculus supporting multi-threading concurrency, functions, and re-entrant locks. We have

proven the algorithm is sound and complete with regard to a non-deterministic specification of the type system. To check for potential deadlocks on the abstract level, different interleavings of the threads must be considered which easily leads to an explosion of state space. To render a finite state space, we put an upper limit on re-entrant lock counters, and bound the non-tail recursive function calls to abstract the behavioural effect into a coarser, tail-recursive one. The correctness of the abstraction with regard to preserving potential deadlocks which occur in the original program can be shown analogously to the one in [8]. The reverse does not hold, i.e., a deadlock in the abstraction does not necessarily exist in the concrete program, as the abstraction over-approximates the actual behaviour of the program.

Related Work

Acknowledgements We thank Axel Simon for fruitful discussion and making available his copy of [5].

References

1. R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with state analysis and run-time monitoring. In S. Ur, E. Bin, and Y. Wolfsthal, editors, *Proceedings of the Haifa Verification Conference 2005*, volume 3875 of *Lecture Notes in Computer Science*, pages 191–207. Springer-Verlag, 2006.
2. T. Amtoft, H. R. Nielson, and F. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.
3. C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation (PLDI) (San Diego, California)*. ACM, June 2003.
4. E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.
5. L. Damas. *Type Assignment in Programming Languages*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1985. CST-33-85.
6. L. Damas and R. Milner. Principal type-schemes for functional programming languages. In *Ninth Annual Symposium on Principles of Programming Languages (POPL) (Albuquerque, NM)*, pages 207–212. ACM, January 1982.
7. F. Nielson, H.-R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
8. 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.