ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verifying Fine-Grained Concurrent Data Structures

Master Thesis

Felix Wolf

August 21, 2018

Advisors: Prof. Dr. Peter Müller, Dr. Malte Schwerhoff

Department of Computer Science, ETH Zürich

Abstract

There are many verifier for fine-grained concurrent code out there. However, many of them have a one sided bias either towards fully-automation or towards complete declaration of proofs. With our proof outline checker named Voila, we attempt to reap the benefits from both sides by letting the user focus on the key steps of a verification while the tool deals with the rest. Voila is a verifier for the TaDA logic, a powerful verification language for fine-grained concurrent programs.

Contents

Chapter 1

Introduction

Verification of concurrent code is hard, because different threads may interfere with each other. Consider the code `z := x.val; x.val := z + 2` where `x.val` is the value of some shared heap location. From the point of view of a single thread the value of `x.f` can change arbitrarily in between field read and field write because the execution of multiple threads running this code may overlap.

On the programming side, locks and atomic hardware instructions are some of the existing solutions to combat interference. Locks are used to enforce exclusive memory access: in the above example, interference in between the field read and field write cannot occur if a lock associated with the memory location is acquired before the field read and then released after the field write. However, many high-performance applications eschew the potential runtime overhead of coarse-grained concurrent locks and instead rely on fine-grained concurrent data structures and algorithms, implemented using atomic hardware instructions such as *compare-and-swap*.

These more flexible synchronization techniques make reasoning about shared state and thus reasoning about interference more complicated, because, in general, the heap locations manipulated by some code can be modified by another thread virtually at any moment. Typically, developer of fine-grained concurrent code think about self-imposed restriction to control the interference by limiting its capabilities. The task of concurrent code verification is to specify those self-imposed restrictions and reason about them. Those specifications are not part of the program state, but instead part of the auxiliary state, also referred to as *ghost code*, that aids the verification of the code.

Rely-guarantee reasoning [9] and *separation logic* [16] are two approaches that aid reasoning about heap locations are accessed and how access to those heap locations is distributed, respectively. For example regarding rely-guarantees on heap accesses, protocols specifying permitted resource transitions can es-

$$l \xrightarrow{s} a \bullet b \;\wedge\; l \xrightarrow{o} c \Rightarrow (l \xrightarrow{s} a \;\wedge\; l \xrightarrow{o} b \bullet c) \circledast (l \xrightarrow{s} b \;\wedge\; l \xrightarrow{o} a \bullet c)$$

Figure 1.1: FCSL [15] part of the subjective separation conjunction definition.

tablish invariants on heap values. If for the code `z := x.val; x.val := z + 2` the field `x.val` is associated with a protocol only permitting even values then we know, even in the presence of interference, that the field value directly after the write is even. For resource distribution, separation logic can be applied to reason about the ownership of resources, i.e. which resources are exclusively held and which resources might be shared with the environment. Exclusively owned resources are by definition not exposed to interference. FCSL [15] introduced the subjective separation conjunction $\circledast$ to reason about the resources distributed between multiple threads. For example the formula in figure 1.1 illustrates how resources are redistributed before the fork of two threads; The resources first exclusively owned ($l \xrightarrow{s} a \bullet b$) are distributed between two threads so that the part exclusive in one thread (e.g. $l \xrightarrow{s} a$) belongs to the environment in the other thread (e.g. $l \xrightarrow{o} a \bullet c$).

As mentioned before access protocols on heap resources can establish invariants. However, it is hard to establish two-state invariants, i.e. specifying how a resource changes over time. For the code `z := x.val; x.val := z + 2`, we were able to enforce the invariant that the field value is even. But now consider the two-state invariant that the field value is only permitted to increase. The code is not able to satisfy this condition because interference can take place in between the read and write, hence the write can actually decrease the value, for example when the write shadows two other writes. Such interleaving can be avoided with atomicity. *Atomicity* describes that an operation happens in a single, discrete instant in time, without possible interference from other threads. Hence, atomic operations allow to satisfy two-state invariants in a concurrent setting. However, atomicity is too strict for more complex operations such as operations on data structures because in those cases the operations are typically not atomic. Furthermore, it is often enough to know that a set of operations appears to behave atomic relative to each other. We describe this concept as *abstract atomicity*.

Iris [10], FCSL [15], and TaDA [1] are just a few of the programming logics that incorporate a synthesis of rely-guarantee reasoning, separation logic, and abstract atomicity. All of those logics already have a tool support to varying degrees: Iris and FCSL are verified in Coq, a declarative proof checker. The versatility of declarative proof checker allows Coq to thoroughly verify programs. As a trade-off, Coq offers less automation and hence requires more developer effort. On the other side, Caper [4] supports a subset of TaDA with a noteworthy high degree of automation. However, its high degree of automation comes at the cost of termination issues since the missing user input results in a larger proof search space.

We developed Voila, a verification language heavily based on TaDA, together with the Voila outline checker that takes as input a source program enriched with proof annotations and outputs if the verification was successful and, in the case of failure, emits meaningful error messages. With an outline checker the user focuses on the key steps of a proof and lets the tool handle the rest. The main goal is to find a good trade-off between fully-automatic verifier, where the tool tries to search for the correctness proof on its own, and a declarative proof checker, where the tool only checks the validity of a correctness proof given by the user.

We did not build the tool from scratch, but built it on top of the Viper intermediate verification language [14], a verification infrastructure for permission-based reasoning. This choice was motivated by Viper's claim to provide a backend that simplifies the development of more complex, permission-based verifiers.

Our contribution is the development of the Voila verifier that can verify a bigger subset of TaDA core mechanics than existing verifier, and illustrates the benefits of an outline checker. Furthermore, we tested the capabilities of Viper as an intermediate verification language for more complex logics. Finally, we also provide a soundness sketch for the majority of the encoding.

Chapter 2

TaDA Basics

In the introduction we introduced several constructs derived from a synthesis of rely-guarantee reasoning, separation logic, and time abstraction, namely protocols, resource ownership, and abstract atomicity. In this chapter we present how TaDA incorporates these constructs.

Memory Resources We start with how TaDA models memory resources. TaDA distinguishes between shared and exclusive resources, i.e. whether or not multiple threads can hold them at the same time. An exclusive resource is denoted as $x.f \mapsto v$ where f is a field with value v of a reference x. Since the assertion is exclusive, $x.f \mapsto v$ also expresses ownership of the field. Furthermore, $x.f$ is not exposed to interference and is hence by definition stable. In concurrent programs shared state is necessary because otherwise multiple threads would not be able to synchronize their actions. Therefore, every verification language for concurrent code requires some way to represent shared state. In Voila, shared state is modeled by shared regions which can be held by multiple threads. Formally, a shared region assertion $\mathbf{t}_a^\lambda(\vec{e},\ s)$ consists of a region type $\mathbf{t}$, a region id a, a level λ, parameters $\vec{e}$, and an abstract state s. The region id is used as a unique identifier of a region instance, hence the following holds:

$$\mathbf{t}_a^\lambda(\vec{e},\ s)\ \land\ \mathbf{t'}_a^{\lambda'}(\vec{e'},\ s') \Rightarrow t = t'\ \land\ \lambda = \lambda'\ \land\ \vec{e} = \vec{e'}$$

The region interpretation defines resources contained inside a region together with invariants on those resources as well as the composition of the abstract region state. For example, the shared region type **ECounter** with the interpretation $I\left(\mathbf{ECounter}_a^\lambda(x,\ v)\right) \triangleq x.val \mapsto v * \mathsf{even}(v)$ encapsulates a points-to predicate $x.val \mapsto v$ together with the invariant of the value being even. Furthermore, the interpretation defines the abstract state of **ECounter** as the value of the contained field resource $x.val$. It might seem contradicting that points-to predicates are one side exclusive, but on the other side can be placed into shared regions. The solution to that dilemma is given by threads can interact with shared regions: first, the resources contained in the interpretation can only be retrieved by opening a region. Second, a region cannot be opened again before it is closed. Lastly, opening a region is only allowed at an abstractly discrete instant in time, i.e. when it appears as if no other thread has opened the region. As a consequence of these constraints on shared regions, points-to predicates inside a region interpretation can still be treated as if they were actually exclusive. In the next chapter we discuss how to open a region and how to change the state of a region.

Abstract Atomicity Judgment Consider we want to specify a method that operated on the previously defined **ECounter** region. In figure 2.1 two different kind of method specifications offered by TaDA are shown. They are a non-atomic and abstract atomic specification, respectively. As described in the introduction, for non-atomic code it is hard to prove interesting properties in a fine-grained concurrent setting. The non-atomic specification states that if before a **ECounter** instance existed with some region state, then after the call the state of the region instance will be at least two. The assertion $[G]_a$ denotes a *regioned guard*, an auxiliary resource that does not model program state and is only used to verify a program. Intuitively, in TaDA, guards represent permissions to perform a state change on a region. For example, the provided non-atomic specification cannot be verified if the guard $[G]_r$ does not permit a transition of the **ECounter** state to a state larger or equal to 2. More interesting is the abstract atomic specification. Intuitively, it specifies that the region state of **ECounter** will increase by two as if the transition happened atomically. More formally, the specification states that, if the function terminates, as long as the state of the **ECounter** instance is contained in the set V an update to $v + 2$ will be performed by the callee where v is the state in V that the region had at the point of the update. The method level λ' specifies the group of functions for which the update will appear to have happened atomically, namely all functions that have a method level of at least λ'. Regarding the terminology, we refer to Γ as the functional environment that contains the specifications of all TaDA methods in the current program. As mentioned before λ' is the method level. The set V is the interference rely-guarantee of the method. It specifies an upper-bound of interference caused by the environment under which a method can still satisfy its specification. Finally, the atomicity context $\mathcal{A}$ stores which updates on region state might currently happen, i.e. if $a : X \rightsquigarrow Y$ is an entry of the atomicity context, then the region state belonging to a has to be guaranteed to be in X until the currently pending update will be performed that changes the region state to a value in Y. TaDA offers the same specifications for statements. In this case we refer to λ, $\mathcal{A}$, and V as the level, atomicity context, and interference rely-guarantee for the judgment on the statement.

$$\Gamma; \lambda'; \mathcal{A} \vdash \begin{array}{c} \{\exists v \in V.\ \mathbf{ECounter}^{\lambda}_{a}(x,\ v) * [\mathbf{G}]_a\} \\ \texttt{addEven} \\ \{\exists v \in V.\ \mathbf{ECounter}^{\lambda}_{a}(x,\ v) * [\mathbf{G}]_a * v \geq 2\} \end{array}$$

$$\Gamma; \lambda'; \mathcal{A} \vdash \forall\!\!\forall v \in V. \begin{array}{c} \langle \mathbf{ECounter}^{\lambda}_{a}(x,\ v) * [\mathbf{G}]_a \rangle \\ \texttt{addEven} \\ \langle \mathbf{ECounter}^{\lambda}_{a}(x,\ v+2) * [\mathbf{G}]_a \rangle \end{array}$$

Figure 2.1: Non-atomic and abstract atomic specification of a TaDA method `addEven`.

Abstract atomic specification in general is referred to as an atomic triple whereas non-atomic specification is referred to as a non-atomic triple.

Guard-labeled State Transition Systems and Stability As we have seen before, in TaDA guard denotes the permission to perform a state transition on a region. A guard is an element of a *guard algebra* $(\mathbb{G}, \bullet, 0, 1)$ which is a partially commutative monoid with the *guard conjunction* $\bullet$, the unit 0, and the max element 1. $\mathbb{G}$ is the carrier set every guard is contained in. A resource order on guards G is defined in the usual way:

$$G_1 \leq G_2 \Leftrightarrow \exists G_3.\ G_1 \bullet G_3 = G_2$$

Since 1 is the max element of a guard algebra $G \leq 1$ always holds. Different guard algebras are discussed in section 6.2.7. For now we only showcase two of them: first, we call a guard $\mathbf{G}$ *unique* if it is from the algebra $(\{0, \mathbf{G}\}, \bullet, 0, \mathbf{G})$ where $\mathbf{G} \bullet \mathbf{G}$ is always undefined. Second, we call a guard $\mathbf{G}$ *duplicable* if it is from the algebra $(\{\mathbf{G}\}, \bullet, \mathbf{G}, \mathbf{G})$. Each region instance $\mathbf{t}^{\lambda}_{a}(\vec{e},\ s)$ has its own guards written as $[G]_a$, as seen before, we refer to those as regioned guards. The guard conjunction $\bullet$ is lifted to regioned guards $[g_1]_a \bullet [g_2]_a = [g_1 \bullet g_2]_a$.

The guard-labeled transition system then defines state transitions on regions that are permitted if the associated guard is owned. For example consider the **ECounter** region from before. Because we want an increasing counter, a possible transition system can be specified by $\mathbf{G} : \forall n,\ m.\ n \rightsquigarrow m \mid n < m \ \wedge\ even(m)$ where $\mathbf{G}$ is a unique guard. In other words, the unique guard $\mathbf{G}$ permits a transition on **ECounter** if the state is only increased and if the new state is even. Formally, the transition system of a region type $\mathbf{t}$ is defined by $\mathbb{T}_{\mathbf{t}}(g)$ which maps guards associated with the region to a subset of permitted transitions. For **ECounter** we get $\mathbb{T}_{\mathbf{ECounter}}(\mathbf{G}) = \{\ (n, m) \mid n < m\ \wedge\ even(m)\}$. Transition can be guarded by more than one guard, but $\mathbb{T}_{\mathbf{t}}(g_1) \subseteq \mathbb{T}_{\mathbf{t}}(g_1 \bullet g_2)$ has to hold to satisfy the resource order on guards.

$$\left.\begin{array}{l}
\varphi \vDash P \\
\varphi \vDash \mathbf{t}_a^\lambda(\vec{e},\ s) \\
\varphi \sim [g]_a \\
(s, s') \in \mathbb{T}_{\mathbf{t}}^\star(g) \\
\varphi \vDash a \ \Rrightarrow\ \blacklozenge \Rightarrow s' \in dom(\mathcal{A}(a))
\end{array}\right\} \Rightarrow P \sim \mathbf{t}_a^\lambda(\vec{e},\ s')$$

Figure 2.2: Formal definition of stability of an assertion P. Regarding the notation, φ entailing Q ($\varphi \vDash Q$) is defined as $\varphi \in [\![Q]\!]$, φ compatible with Q ($\varphi \sim Q$) is defined as $\exists \varphi' \in [\![Q]\!].\ \varphi \bullet \varphi'$, and $Q_1 \sim Q_2$ is defined accordingly where $[\![\cdot]\!]$ is the assertion semantics of TaDA.

With a state transition system we can now define stability. Intuitively, an assertion P is stable, if for each state φ satisfying P where some region $\mathbf{t}_a^\lambda(\vec{e},\ \cdot)$ is in state s and where the environment can perform the transition from s to s' in φ, then P should not contradict the region $\mathbf{t}_a^\lambda(\vec{e},\ \cdot)$ being in state s'. This condition corresponds to P holding even in the presence of interference because all states that can be reached by the environment still satisfy P. A transition from s to s' on a region $\mathbf{t}_a^\lambda(\vec{e},\ \cdot)$ in a state φ can be performed by the environment if the environment may hold a guard $[g]_a$ which permits the transition and if s' satisfies current rely-guarantees on the region state. As seen before, rely-guarantees on the region state is expressed through the atomicity context $\mathcal{A}$ that specifies for the region states currently being updated in which set the region state has to be guaranteed to be in. The *diamond resource* $r \Rrightarrow \blacklozenge$ denotes that oneself is currently performing an update on the region instance belonging to the region identifier r. Hence, stability also has to guarantee that condition. A formalization of stability is given in figure 2.2. Regarding the notation, $\varphi \vDash Q$ expresses state φ satisfying the assertion Q. Furthermore, $\varphi \sim Q$ states φ not contradicting Q holding for the environment. For example $\varphi \sim [g]_a$ from the stability definition holds if the environment might own the guard $[g]_a$. Therefore, $\varphi \vDash [\mathbf{G}]_a$ would be false for $\varphi \vDash [\mathbf{G}]_a$ where $\mathbf{G}$ is a unique guard, but would hold if $\mathbf{G}$ is a duplicable guard.

Chapter 3

TaDA Proof Outline

Consider the `addEven` method from the previous chapter. As specified formally in figure 2.1, it abstract atomically increases the state of the region **ECounter** by 2. In this chapter we will proof two different implementation of the `addEven` method in TaDA. As a reminder, the interpretation of **ECounter** is $I\left(\mathbf{ECounter}^{\lambda}_{a}(x,\, v)\right) \triangleq x.val \mapsto v * \mathsf{even}(v)$ and the region is guarded by a single guard **G** which permits all transitions to even integers that increase the state.

3.1 Single Statement Implementation

The first implementation of `addEven` given in figure 3.1 consists of a single call to the fetch-and-add primitive written as `FAA_val` where `val` specifies the field that is modified. The primitive atomically increases the value of a field location by the provided argument. Our goal is to prove that the first implementation adheres to the abstract atomic specification of `addEven` given in figure 2.1.

Such a proof of an atomic triple in TaDA has two objectives: proving functional correctness and proving abstract atomicity. For functional correctness, we will have to open the region to justify the change of the abstract state through a change inside the region interpretation. On the other side, the outline of the abstract atomicity proof depends always on the implementation. The proofs can be categorized into two groups, referred to as delegation and abstraction, both of which are presented in this section and in the next one respectively.

If the implementation, as in our case, consists of a single statement, then we can delegate the atomicity proof to a lower level; note that whereas `addEven` operates on **ECounter** regions, fetch-and-add operates on heap locations directly. In TaDA the delegation of an atomicity proof to a lower level is

```
method addEven(x) {
  FAA_val(x, z, z+2)
}
```

Figure 3.1: A trivial implementation for `addEven`.

$$\frac{a \notin \mathcal{A} \qquad \{(s,y) \mid s \notin S, y \in Y\} \subseteq \mathbb{T}^{\star}_{\mathbf{R}}(G) \qquad \Gamma;\lambda;\mathcal{A} \vdash \mathbb{V}s \in S. \begin{array}{c} \langle I\left(\mathbf{R}^{\lambda}_{a}(\vec{e},s)\right) * [G]_a * P(s)\rangle \\ \mathbb{C} \\ \langle \exists y \in Y.\, I\left(\mathbf{R}^{\lambda}_{a}(\vec{e},y)\right) * Q(s)\rangle \end{array}}{\Gamma;\lambda+1;\mathcal{A} \vdash \mathbb{V}s \in S. \begin{array}{c} \langle \mathbf{R}^{\lambda}_{a}(\vec{e},s) * [G]_a * P(s)\rangle \\ \mathbb{C} \\ \langle \exists y \in Y.\, \mathbf{R}^{\lambda}_{a}(\vec{e},y) * Q(s)\rangle \end{array}} \text{ UseAtomic}$$

Figure 3.2: The UseAtomic TaDA proof rule.

```
method addEven(x) {
$\mathbb{V}v \in V.$
$\langle \mathbf{ECounter}^{\lambda}_{r}(x,\ v) * [\mathbf{G}]_r\rangle$
  $\langle$x.val $\mapsto v * \mathsf{even}(v)\rangle$
    FAA_val(x, z, z+2)
  $\langle$x.val $\mapsto v + 2 * \mathsf{even}(v+2)\rangle$
$\langle \mathbf{ECounter}^{\lambda}_{r}(x,\ v+2) * [\mathbf{G}]_r$
}
```

Figure 3.3: A TaDA proof for the trivial implementation of `addEven`.

captured by UseAtomic rule shown in figure 3.2. This rule allows to open a region **t** and then to perform an update on it if the performed state change of **t** is permitted by an owned guard and if no other modification of **t** is tracked. A TaDA proof for the `addEven` implementation using the UseAtomic is given in 3.3. The proof proceeds as directed by the UseAtomic rule: While the **ECounter** region is opened, the `fetch-and-add` rule is applied. The state transition is allowed because the state has increased and stayed even.

3.2 Abstract Atomic Implementation

Typically an implementation will consist of more than one method call, thus requires more advanced reasoning about abstract atomicity. The second implementation of `addEven` given in figure 3.4 illustrates such a case: Inside a loop the value of the heap location x is successively read until it is successfully modified. The modification is done by the `compare-and-swap` primitive, which in case the heap value is equal to the provided argument, updates a heap location to some value and returns true, or otherwise returns false without a modification.

In TaDA to proof a implementation consisting of multiple statements the MakeAtomic rule from figure 3.5 is required. This rule captures the core mechanic of abstract atomicity. Similar to the UseAtomic rule, MakeAtomic allows an update to the region state if an owned guard permits the performed state change. The difference is that the MakeAtomic rule allows the update to be performed in multiple steps on the same level, expressed through the transition from atomic-triple to Hoare-triple. Performing the region update in more than one step leads to two requirements: First, because the atomic-triples in TaDA specify a single linearization point, only one region update is allowed. Second, because interference may happen after the linearization point, the region update is tracked. As mentioned in the TaDA basics chapter, the two tracking resources $a \Mapsto \blacklozenge$ and $a \Mapsto (x, y)$ track the uniqueness, and the source and target of the update transition respectively. The atomic tracking context $\mathcal{A}$ is updated with the mapping $a : x \in X \rightsquigarrow Y$ to store the permitted transitions and to ensure that the region is not updated without the diamond resource, for example through the use of *use_atomic*.

The actual state change is proven with the UpdateRegion rule from figure 3.5. The rule allows to open a region and to update the state if the state transition is contained in the atomicity tracking context. The atomicity context together with the diamond resource take the role of the guard to permit transitions. The rule considers the case in which an update happens, as well as the case in which an update did not happen, to be coherent with typical concurrent programming semantics employed in primitives such as `compare-and-swap`, in which an update may or may not take place.

The complete proof sketch of the second `addEven` method can be seen in figure 3.6. The MakeAtomic rule switches the context from atomic-triple to Hoare-triple. The OpenRegion rule, which can be seen as a variation of the UseAtomic that does not change the abstract state of the region, opens the region to write the value of the heap location into the variable `v`. Lastly, the region is updated depending on the success of the `compare-and-swap` primitive with UpdateRegion.

```
method addEven(x) {
  do {
    z := x.val
    b := CAS_val(x, z, z+2)
  } while(b);
}
```

Figure 3.4: A more advanced implementation for `addEven`.

$$\frac{\Gamma;\lambda;\mathcal{A} \vdash \forall s \in S.\ \langle\ I\left(\mathbf{R}_a^\lambda(\vec{e},s)\right) * P(s)\ \rangle\ \mathbb{C}\ \langle\ I\left(\mathbf{R}_a^\lambda(\vec{e},s)\right) * Q(s)\ \rangle}{\Gamma;\lambda+1;\mathcal{A} \vdash \forall s \in S.\ \langle\ \mathbf{R}_a^\lambda(\vec{e},s) * P(s)\ \rangle\ \mathbb{C}\ \langle\ \mathbf{R}_a^\lambda(\vec{e},s)\ *\ Q(s)\ \rangle}\ \textsc{OpenRegion}$$

$$\frac{\begin{array}{c} a \notin \mathcal{A} \qquad \{(s,y) \mid s \notin S, y \in Y\} \subseteq \mathbb{T}^\star_{\mathbf{R}}(G) \\ \Gamma;\lambda;a : s \in S \rightsquigarrow Y, \mathcal{A} \vdash \begin{array}{c} \{\exists s \in S.\ \mathbf{R}_a^\lambda(\vec{e},s) * a \Rrightarrow \blacklozenge\ \} \\ \mathbb{C} \\ \{\exists s \in S.\ \exists y \in Y.\ a \Rrightarrow (s,y)\} \end{array}\end{array}}{\Gamma;\lambda;\mathcal{A} \vdash \forall s \in S.\ \langle\ \mathbf{R}_a^\lambda(\vec{e},s) * [G]_a\ \rangle\ \mathbb{C}\ \langle\ \exists y \in Y.\ \mathbf{R}_a^\lambda(\vec{e},y) * [G]_a\ \rangle}\ \textsc{MakeAtomic}$$

$$\frac{\begin{array}{c} a \notin \mathcal{A} \qquad \{(s,y) \mid s \notin S, y \in Y\} \subseteq \mathbb{T}^\star_{\mathbf{R}}(G) \\ \Gamma;\lambda;\mathcal{A} \vdash \forall s \in S.\ \begin{array}{c} \langle I\left(\mathbf{R}_a^\lambda(\vec{e},s)\right) * P(s)\rangle \\ \mathbb{C} \\ \left\langle \begin{array}{c} \exists y \in Y.\ I\left(\mathbf{R}_a^\lambda(\vec{e},y)\right) * Q_1(s,y) \vee \\ I\left(\mathbf{R}_a^\lambda(\vec{e},s)\right) * Q_2(s) \end{array} \right\rangle \end{array}\end{array}}{\begin{array}{c} \Gamma;\lambda+1;a : s \in S \rightsquigarrow Y, \mathcal{A} \vdash \forall s \in S.\ \langle \mathbf{R}_a^\lambda(\vec{e},s) * a \Rrightarrow \blacklozenge * P(s)\rangle \\ \mathbb{C} \\ \left\langle \begin{array}{c} \exists y \in Y.\ \mathbf{R}_a^\lambda(\vec{e},y) * Q_1(s,y) * a \Rrightarrow (s,y) \vee \\ \mathbf{R}_a^\lambda(\vec{e},s) * Q_2(s) * a \Rrightarrow \blacklozenge \end{array} \right\rangle \end{array}}\ \textsc{UpdateRegion}$$

Figure 3.5: The MakeAtomic, UpdateRegion, and OpenRegion proof rule from TaDA.

```
method addEven(x) {
$\mathbb{\forall} v \in V.$
$\langle \mathbf{ECounter}^{\lambda}_{r}(x,\ v) * [\mathbf{G}]_r \rangle$
  $r :\ v \in V \rightsquigarrow v + 2$
  $\{\exists v \in V.\ \mathbf{ECounter}^{\lambda}_{r}(x,\ v) * r \Rrightarrow \blacklozenge\}$
  do {
    $\{\exists v \in V.\ \mathbf{ECounter}^{\lambda}_{r}(x,\ v) * r \Rrightarrow \blacklozenge\}$
     $\mathbb{\forall} v \in V.$
     $\langle \texttt{x.val} \mapsto v * \mathsf{even}(v) \rangle$
       z := x.val
     $\langle \texttt{x.val} \mapsto v * \mathsf{even}(v) * \mathsf{even}(z) \rangle$
    $\{\exists v \in V.\ \mathbf{ECounter}^{\lambda}_{r}(x,\ v) * r \Rrightarrow \blacklozenge * \mathsf{even}(\texttt{z})\rangle\}$
     $\mathbb{\forall} v \in V.$
     $\langle \texttt{x.val} \mapsto v * \mathsf{even}(v) * \mathsf{even}(\texttt{z}) \rangle$
       b := CAS_val(x, z, z + 2)
     $\langle \texttt{b ? x.val} \mapsto v : \texttt{x.val} \mapsto v + \texttt{a} * \mathsf{even}(v) \rangle$
    $\{\exists v \in V.\ \texttt{b ? } \mathbf{ECounter}^{\lambda}_{r}(x,\ v) * r \Rrightarrow \blacklozenge\ :\ r \Rrightarrow (v, v+2) * \mathsf{even}(v+2)\rangle\}$
  } while(b);
  $\{\exists v \in V.\ r \Rrightarrow (v, v+2) * \mathsf{even}(v+2)\rangle\}$
$\langle \mathbf{ECounter}^{\lambda}_{r}(x,\ v+2) * [\mathbf{G}]_r \rangle$
}
```

Figure 3.6: A proof of the more advanced implementation for `addEven`.

Chapter 4

Viper Basics

The Viper intermediate verification language [14] is a verification infrastructure for permission-based reasoning. The key concept is that access permissions are necessary to access the heap. As a consequence, because code can only manipulate heap memory that it has permission to and because access permissions of resources themselves are disjoint, local reasoning is enabled.

The ownership of the permission amount `p` to the field `f` of reference `x` is expressed by the *accessibility predicate* `acc(x.f, p)`. The permission `p` is a rational between 0 and 1, also notated as `none` and `write` respectively, where 1 corresponds to write permission and any non-zero amount corresponds to read permission. As comparison, the points-to predicate `x.val` $\mapsto s$ can be expressed as `acc(x.val, write) && x.val == s` where `&&` notates the Viper separating conjunction. Access permissions aggregate under the separation conjunction so `acc(x.f, 1/2) && acc(x.f, 1/2)` amounts to write permissions.

Permissions are gained either implicitly at the beginning of a method through the method precondition, or explicitly through the use of an `inhale` statement. Vice versa, the same for the loss of permissions through method postconditions or the `exhale` statement. Figure 4.1 shows two Viper examples: on the right side, permission to a field `x.f` is acquired and then 42 is written to the field. Note that `acc(x.val)` is a shorthand for `acc(x.val, write)`. Afterwards, the permission is first asserted, then removed, and finally acquired again, all of which succeeds. The `exhale` after the `assert` succeeds since `assert` does not remove permissions. Lastly, it is asserted that the field still has its old value. This assertion fails because the value of a field is chosen arbitrarily if permission to the field is inhaled when currently no permission to that field is held. This mechanic can be used to havoc the value of a field by successively exhaling and inhaling the permissions to that field. One way to access an old value of a field is with labels and `old` as shown on the right side: a label is placed before the field permission is

```
inhale acc(x.f)
x.f := 42
assert acc(x.f)
exhale acc(x.f)
inhale acc(x.f)
assert x.f == 42 // fail
```

```
inhale acc(x.f)
x.f := 42
label l
exhale acc(x.f)
assert old[l](x.f) == 42
assert perm(x.f) == none
```

Figure 4.1: Two Viper snippets with fields, field permissions, and inhale exhale statements.

exhaled, and after the exhale the label is used together with `old` to assert that the old value of the field is equal to 42, which succeeds. Furthermore, in the right example, the last assertion successfully checks that no permission to the field is held by using the `perm` predicate which queries the held permission amount. The `old` expression can be combined with `acc` and `perm` so we define the following shorthands: first, `perm[l](e)` is short for `old[l](perm(e))`. Second, `acc[l](e)` is a shorthand for `acc(e, perm[l](e))`.

Besides fields, Viper also offers *predicates* [17] and *heap-dependent functions* [7]. Figure 4.2 shows example declarations: the predicate `pair` has in its body two access predicates to the fields `x.f` and `x.g`. Intuitively, `pair` abstracts over part of the heap by representing a pair heap structure. The `getFirst` heap-dependent function then corresponds to a getter for that pair structure. The footprint of the heap-dependent function provided through the `requires` clause specifies the part of the heap on which the function depends on. For example `getFirst` depends only on `cell`. Another way to write a predicate expression `p(e)` is `acc(p(e))`. This notation is useful if we want only a fraction of the permissions specified in the predicates body: for example `getFirst` can only be used if write permission to `cell(x)` is held. If the precondition were `acc(cell(x), 1/2)` instead, then `getFirst` could also be applied with half permission to cell. In the encoding we often use access predicates around predicates to facilitate the distinction between predicates and functions.

The body of a heap-dependent function is an expression with the same type as the specified return type. Important is that heap-dependent function do not modify the heap, i.e. are side-effect free by definition. Inside the function body of `getFirst`, the predicate instance is unfolded, meaning the predicate instance is replaced with its corresponding predicate body, and the integer value of `x.f` is returned. In Viper predicate definitions are iso-recursive [22], meaning that the predicate instance is not treated equally to the corresponding body. Instead, predicate instances need to be explicitly folded and unfolded by the Viper statements `fold p(e)` and `unfold p(e)`, or inside an expression with `unfolding p(e) in (...)`. Figure

```
predicate pair(x: Ref) { acc(x.f) && acc(x.g) }

function getFirst(x: Ref): Int
  requires cell(x)
{ unfolding cell(x) in x.f }
```

Figure 4.2: Viper snippet with predicate and heap-dependent function declarations. `Ref` is the Viper reference type.

```
inhale cell(x)
unfold cell(x)
x.f := 42
fold cell(x)
assert getFirst(x) == 42
```

Figure 4.3: Viper snippet with predicates, heap-dependent functions, and fold and unfold.

```
predicate fp(x: Ref, y: Ref)

function f(x: Ref, y: Ref): Int
  requires acc(fp(x))
```

```
inhale acc(fp(x,y))
assume f(x,y) == 42
exhale acc(fp(x,y))
inhale acc(fp(x,y))
assume f(x,y) == 41
```

Figure 4.4: Two Viper snippets with abstract predicate and function declaration the left side, and use of those predicate and function on the right side.

4.3 illustrates the `cell` predicate and its getter in use: first, `cell` is inhaled. Then, the predicate is unfolded and folded where in between the field `x.f` originating from the predicate body is modified. Lastly, it is asserted with `getFirst` that the value was changed.

Abstract predicates[17] or *abstract heap-dependent functions*[7] do not have a body. Figure 4.4 shows on the left side a declaration of an abstract predicate `fp(x,y)` and an abstract heap-dependent function `f(x,y)` that depends on `fp(x,y)`. On the right side, we illustrate how we employ abstract predicates in the encoding: `f` is used similarly to a field with the difference of being parameterized in `x` and `y`. The value of a function instance is declared by assuming some constraint on the value of the function instance. In the example we assume that the value of `f(x,y)` is 42. Before reassignment, the function has to be havoced by exhaling and inhaling the predicate permission.

Chapter 5

Voila Language

As mentioned in the introduction, a proof outline only contains the interesting key steps of a full proof. Consider the TaDA proof sketch of the `addEven` method: we only gave the three important rule applications that require deeper knowledge about the program, for example, the position of the linearization point indicated by the UPDATEREGION rule. The other required rule applications that mainly transform the auxiliary state were omitted. Furthermore, multiple side conditions checks of the three important rule applications are omitted as well: we did not check that the transition is permitted by the guard and neither did we check that the atomicity context does not contain a mapping for the region identifier. As mentioned before, the goal of Voila is to enable the verification of similar proof outlines that only contain the interesting key steps. In this section, we discuss the Voila language and show how the concept of proof outlines was built into the language.

5.1 Voila Overview

Voila is a Java-like imperative language with a TaDA-like assertion language. Furthermore, Voila has additional statements for TaDA's core rules. We apply the existing TaDA terminology to Voila: for example, Voila also has interference rely-guarantees, a judgment level, or an atomicity context. Furthermore, we apply the TaDA logic to Voila as well: we call a well-defined Voila program correct if there exists a proof for the corresponding TaDA specification that applies the core rules in the same positions as annotated in the Voila implementation. Assertion stability is lifted in the same manner.

A verified Voila implementation for the `addEven` method is shown in figure 5.1. The specification consists of three parts: the pre- and postcondition of a method together with the interference clause, indicated by `requires`, `ensures`, and `interference`, respectively. Assertions for pre- and postconditions are the same as in TaDA specifications, only the notation differs slightly: region identifier and region level are written as normal region arguments. Furthermore, the regioned guard $[\texttt{G}]_{\texttt{r}}$ is written as `G@r`. The interference clause specifies the interference rely-guarantee of the `addEven` method that in TaDA would be bound by $\mathbb{V}$. Different from TaDA the interference context is bound for each region instance separately to simplify the encoding. Regarding the method body, `addEven` has the same implementation as given in chapter 3. However, the implementation is enriched with TaDA annotations for the applied proof rules together with an invariant for the while loop. The proof rules are almost the same as in the previously presented proof sketch with the difference that the targeted region, as well as required guards, are explicitly provided with the `using` and `with` clause, respectively. The invariant is new and expresses that as long as the compare-and-swap did not successfully perform a heap modification, the diamond resource is held, and otherwise the executed modification is captured in the transition resource. Regarding the TaDA rules, a Voila program is similar to a proof sketch as the user only has to use the four core rules shown in section 3: UseAtomic,MakeAtomic, UpdateRegion, and OpenRegion; the automatic verifier deals with all other required TaDA rule applications. Furthermore, Voila automatically proofs the side conditions.

The Voila implementation of `addEven` also uses an **ECounter** region, hence **ECounter** requires a region declaration as shown in figure 5.2. The region arguments are the region id, the region level, and the cell. The resource `x.val` $\mapsto$ `?s` together with the invariant `even(s)` contained in the interpretation of **ECounter** are specified inside the `interp` clause. The value of the field `x.val` is bound by the binding operator `?` so that it can be specified as the state of **ECounter** in the `state` clause. The **G** guard associated with **ECounter** is captured in the `guards` clause. The `duplicable` specifies the guard algebra the guard belongs to. Finally, the `actions` clause defines that the region state is allowed to be increased to an even number if the **ECounter** guard is held. Besides region declarations also structure declarations are necessary. Figure 5.3 shows a straightforward structure declaration for `cell` that is used inside the **ECounter** region.

5.2 Voila Syntax

In the previous section, we gave a rough overview of Voila. In this section, we show the formal syntax of Voila that will be used in the encoding.

Before we discuss the Voila syntax, we introduce some of our employed notations. The letters x, n, r, F, R, S range over variables, integers, region identifiers, fields, region types, and structure names, respectively. As a general guideline, capital case variables such as R, F, M are used for names, and

```
abstract_atomic procedure
   addEven(id r, int lvl, cell x)
  interference ?n in Nat on ECounter(r,lvl,x);
  requires ECounter(r,lvl,x,n) && G@r;
  ensures ECounter(r,lvl,x,n+2) && G@r;
{
  int z; bool b;
  make_atomic using ECounter(r,lvl,x) with G@r {
    do
      invariant ECounter(r,lvl,x,?s);
      invariant b ? r ⤇ ♦ : r ⤇ (s,s+a);
    {
        open_region using ECounter(r,lvl,x) {
          z := x.val;
        }
        update_region using ECounter(r,lvl,x) {
          b := CAS_val(x, z, z+2);
        }
    } while (b)
  }
}
```

Figure 5.1: The `addEven` implementation in Voila.

```
region ECounter(id r, int lvl, cell x)
  guards { duplicable G; }
  interp { x.val ↦ ?s && even(s) }
  state { s }
  actions
    { int n, m | n < m && even(m) | G: n  ⇝  m; }
```

Figure 5.2: Declaration of the **ECounter** region in Voila.

```
struct cell { int val; }
```

Figure 5.3: Declaration of a cell structure with a single field `val` of type `int` in Voila.

lowercase is used for other syntactic elements. Furthermore, $\vec{v}$ is a shorthand for the sequence $v_1, v_2, \ldots, v_n$, and operations to such sequences are applied pointwise, e.g. $\vec{x} : \vec{t}$ stands for $x_1 : t_1, \ldots, x_n : t_n$.

Expressions Similar to other verification languages, Voila distinguishes between expressions used by the implementation and assertions used to verify the implementation. We refer to the latter also as ghost code. A syntax definition of the basic elements of Voila is given in figure 5.4. Regarding types, Voila has ids, booleans, integers, structure types, rationals, sets, and tuples. The latter three, namely Rationals, sets, and tuples, are mathematical data types used in specifications. Structure types are the types of references. The syntax for Booleans and expressions is the standard for separation logic notation, but also contains operations for rationals, sets, and tuples: for example, $\mathsf{Set}(?x \mid b)$ is a set comprehension and $\mathsf{Get}_{\mathsf{i}}(e)$ is a getter for the i-th tuple element. Assertions are composed of standard separation logic assertions together with TaDA assertions: on one hand, points-to predicates $x.F \mapsto q$, guard expressions $g@r$, as well as both tracking resources $r \Mapsto \blacklozenge$, $r \Mapsto (q_{\mathsf{from}}, q_{\mathsf{to}})$ are directly taken from TaDA. On the other hand, region assertions $R(\vec{e}_{\mathsf{in}}; \vec{q}_{\mathsf{out}}; q_{\mathsf{state}})$ have an extended syntax: we distinguish between in- and out-arguments. In-arguments are the standard TaDA region arguments together with the region identifier and the region level. On the other side, out-arguments, known from VeriFast [8], are parameters that must be uniquely determined by the interpretation. They are used to bind a value of a region that is useful in the proof outline. Often out-arguments and the state are omitted in a region assertion, then $R(\vec{e}_{\mathsf{in}})$ is written instead. Another special type of expression is the binding expression $?x$. They have a similar use as in VeriFast: they allow to bind the value of a field or a region. For example $x.val \mapsto ?y$ `&&` $y > 42$ binds the value of $x.val$ to y and checks that this bound value is larger than 42. For simplicity we will assume that all binder variables are globally unique, i.e. only bound once, hence we do not have to worry about scopes.

Structures As seen in the previous section, we use typed fields to access values on the heap. Each reference is of a structure type and provides field accesses according to the structure definition. The structure definition syntax is given in figure 5.5.

Regions As mentioned in chapter 2, in TaDA three different components define the semantics of a region: first, the region interpretation defines the region state based on lower level state together with resources and invariants on those resources hidden in the region. Second, the state transition system defines which state transitions are permitted by which guards. Lastly, the guard algebra defines the semantics of the used guards. In Voila we took

$$
\begin{array}{c}
t ::= \mathsf{id} \mid \mathsf{bool} \mid \mathsf{int} \mid \mathsf{frac} \mid S \mid \mathsf{set}\langle t \rangle \mid \mathsf{tuple}\langle \vec{t} \rangle \\
b ::= b_1 \;\texttt{\&\&}\; b_2 \mid b_1 \;\texttt{||}\; b_2 \mid !b \mid b_1 \Rightarrow b_2 \mid e_1 \odot e_2 \\
\texttt{where } \odot \in \{=, \neq, <, >, \in, \subseteq\} \\
e ::= x \mid n \mid b \mid \mathsf{Set}(\vec{e}) \mid \mathsf{Set}(?x \mid b) \mid \mathsf{Tuple}(\vec{e}) \mid \mathsf{Get_i}(e) \mid e_1 \oplus e_2 \\
\texttt{where } \oplus \in \{+, -, <, *, /, \cup\} \\
q ::= ?x \mid e \\
w ::= R(\vec{e}_{\mathsf{in}}; \vec{q}_{\mathsf{out}}; q_{\mathsf{state}}) \mid x.F \mapsto q \mid r \Mapsto \blacklozenge \mid r \Mapsto (q_{\mathsf{from}}, q_{\mathsf{to}}) \mid g@r \\
\mid w_1 \;\texttt{\&\&}\; w_2 \mid b \mid b \Rightarrow w
\end{array}
$$

Figure 5.4: Type, expression, and assertion syntax of Voila, respectively.

$$\textit{struct } S\{\vec{t}\;\vec{F}\}$$

Figure 5.5: Structure syntax.

the same approach as Caper [4]: A single region declaration defines the region interpretation together with the state transition system. Furthermore, predefined guard algebras are provided. In figure 5.6 the syntax for region declarations is given. A region definition consists of four components: first, the guard set declares the guards associated to the region. Moreover, the guard set defines for each associated guard the guard algebra the guard belongs to. For simplicity we will assume that guard names are unique in the Voila program. The predefined guard algebras are for now not important and will be discussed in section 6.2.7. Second and third, we split the region interpretation into an interpretation and state definition. The corresponding region interpretation in the TaDA style is $I\left(R_a^\lambda(\ldots, n)\right) \triangleq w_{\mathsf{interp}} \;\texttt{\&\&}\; n = e_{\mathsf{state}}$. We will discuss the reason for the separation in section 6.1. Lastly, the action set defines the state transition system through actions: an action $\vec{t}\;\vec{x} \mid c \mid g : \alpha \rightarrow \beta$ (where the action variables $\vec{x}$ can occur in c, α, β, and g) permits a transition from some a to some b if there exist some $\vec{x}$ such that $a = \alpha$, $b = \beta$, and c hold and g is owned. We require and check that actions are transitively closed. Therefore, the complete transition system of a region type R can be defined as follows:

$$
\begin{array}{c}
(a, b) \in \mathbb{T}_R^\star(g) \Leftrightarrow a = b \;\vee\; \exists A \in \textsc{Actions}(R).\; (a, b) \in \mathbb{T}_A(g) \\
(a, b) \in \mathbb{T}_A(g) \Leftrightarrow \exists \vec{x}_A.\; a = \alpha_A \;\wedge\; b = \beta_A \;\wedge\; c_A \;\wedge\; g_A \leq g
\end{array}
$$

The symbols g_A, $\vec{x}_A$, α_A, β_A, and c_A notate action guard, action variables, pre- and post-state, and action condition of an action A, respectively. Furthermore, $\textsc{Actions}(R)$, $\textsc{State}(R)$, $\textsc{Interp}(R)$, $\textsc{Formal}_{\mathsf{in}}(R)$, $\textsc{Formal}_{\mathsf{out}}(R)$,

$$\begin{array}{l}
\mathsf{region}\ R(\vec{t}_{\mathsf{in}}\ \vec{x}_{\mathsf{int}}; \vec{t}_{\mathsf{out}}\ \vec{x}_{\mathsf{out}}) \\
\quad \mathsf{guards}\{\vec{gdef}\} \\
\quad \mathsf{interp}\{w_{\mathsf{interp}}\} \\
\quad \mathsf{state}\{e_{\mathsf{state}}\} \\
\quad \mathsf{actions}\{\vec{action}\}
\end{array}$$

$$gdef ::= k\ G(\vec{t}\ \vec{x}) \text{ where } k \in \{\mathsf{unique},\ \mathsf{duplicable},\ \mathsf{divisible}\}$$
$$action ::= (\ \vec{t}\ \vec{x}\ \mid\ b\ \mid\ g : \alpha\ \rightarrow\ \beta\ )$$

Figure 5.6: Region syntax and an region definition example. In the example the types of the action variables are omitted.

Guards(R) refer to the region's action set, state, interpretation, formal in- and out- arguments, and associated guards, respectively.

Statements A syntax definition for statements is given in 5.7. We distinguish between abstract atomic and non-atomic statements: non-atomic statements are the standard sequential composition, if-else, while loop, variable assignment, and calls to non-atomic methods. Abstract atomic statements are the TaDA rule statements, heap read and write, and calls to the compare-and-swap primitive, as well as calls to abstract-atomic methods. Because a Voila program can have multiple fields, we define a compare-and-swap statement $x\ :=\ \mathsf{CAS}_F(e_1, e_2, e_3)$ for each field F. The distinction between both kinds of statements is motivated by the TaDA logic: as mentioned in 2, TaDA differentiates between atomic and non-atomic triples; because of our separation, it is unambiguous which triple kind is required for which statement: atomic triples are used to verify abstract-atomic statements, and non-atomic triples are used to verify non-atomic statements. In order to get this separation, we extended the syntax with an explicit $\mathsf{atomic}\{a\}$ statement that switches from non-atomic statements to atomic ones. The $\mathsf{atomic}\{a\}$ statement is not used in Voila programs because the Voila tool can detect the required triple transitions on its own. Similarly, we added footnotes to calls to indicate whether the callee is atomic or non-atomic. These annotations are also not required in a real Voila program.

Methods Similar to statements, we distinguish between abstract atomic and non-atomic methods. Again, the differentiation is motivated by TaDA's separation of atomic and non-atomic triples. The method syntax is given in figure 5.8. Both kinds of methods have the standard pre- and postcondition clauses. In contrast to non-atomic methods, abstract atomic methods

$$\begin{aligned} h ::= {} & h_1; h_2 \mid \mathsf{if}(b)\ \{h_1\}\ \mathsf{else}\ \{h_2\} \mid \mathsf{while}(b)\ \mathsf{invariant}\ w\ \{h\} \\ & \mid x := e \mid \vec{x} := M_H(\vec{e}) \mid \mathsf{atomic}\{a\} \end{aligned}$$

$$\begin{aligned} a ::= {} & \mathsf{use\_atomic\ using}\ R(\vec{e}_{\mathsf{in}})\ \mathsf{with}\ g\ \{a\} \mid \mathsf{make\_atomic\ using}\ R(\vec{e}_{\mathsf{in}})\ \mathsf{with}\ g\ \{h\} \\ & \mid \mathsf{update\_region\ using}\ R(\vec{e}_{\mathsf{in}})\ \{a\} \mid \mathsf{open\_region\ using}\ R(\vec{e}_{\mathsf{in}})\ \{a\} \\ & \mid x := y.F \mid x.F := e \mid x := \mathsf{CAS}_F(e_1, e_2, e_3) \mid \vec{x} := M_A(\vec{e}) \end{aligned}$$

Figure 5.7: Statement syntax; we distinguish between abstract atomic statements and non-atomic statements. As common for deductive verification techniques, while statements are extended with an invariant that holds before and after each loop iteration in which the loop condition is satisfied.

have an additional interference clause $?y$ in e_{set} on $R(\vec{e}_{\mathsf{in}})$ that specifies, for a specific region instance $R(\vec{e}_{\mathsf{in}})$, an upper-bound on the interference caused by the environment. The interference upper-bound is given in the form of the expression e_{set} that specifies the set of all values the environment may change the region state into. Interference clauses, as the name suggests, correspond to the TaDA interference rely-guarantee which is specified by the variables and their respective domains bound by the pseudo-quantifier $\mathbb{V}$. We allow the variable y bound in the interference clause to occur inside the set definition. This is useful to model the absence of interference: for example, the clause $?y$ in $Set(y)$ on $\mathbf{ECounter}(\vec{e}_{\mathsf{in}})$ declares that the environment is allowed to change the state only to itself, which is equivalent to no interference occurring at all.

Recall that a TaDA method specification for abstract atomic methods is of the form $\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}\vec{x} \in \vec{X}.\ \langle\ P(\vec{x}, \vec{z})\ \rangle\ f(\vec{z})$ returns $(\vec{r})\ \langle\ Q(\vec{x}, \vec{z}, \mathbf{r})\ \rangle$. As already discussed, the pre- and postcondition as well as the interference rely-guarantee is specified by the `requires`, `ensures`, and `interference` clause, respectively. The function environment is implicitly defined by the set of methods contained in a Voila program. The level λ and the atomicity context $\mathcal{A}$ are not specified explicitly by additional clauses, but instead through the precondition of the method. The method level is defined as the highest level occurring in w_{pre} plus one. The atomicity context of a method is then any mapping that is not defined on a region with a level lower than the method level. More formally, for a precondition w_{pre} the method level λ as well as the atomicity context $\mathcal{A}$ is defined as follows:

$$\begin{gathered} \lambda := 1 + \max \textsc{Levels}(w_{\mathsf{pre}}) \\ \mathcal{A} \in \{\ M \mid \forall r \in \mathsf{dom}(M).\ \text{level associated to } r \text{ is} \geq \lambda\ \ \} \end{gathered}$$

The function $\textsc{Levels}(\cdot)$ extracts region levels from an assertion, i.e. it is defined as $\textsc{Levels}(R(\vec{e}_{in})) = \{lvl(\vec{e}_{in})\}$ where $lvl(\cdot)$ returns the level from an

```
abstract_atomic procedure $M(\vec{t}_{\mathsf{in}}\ \vec{x}_{\mathsf{in}})$ returns $(\vec{t}_{\mathsf{out}}\ \vec{x}_{\mathsf{out}})$
    interference $?y$ in $e_{\mathsf{set}}$ on $R(\vec{e}_{\mathsf{in}})$
    requires $w_{\mathsf{pre}}$
    ensures $w_{\mathsf{post}}$
    $\{\vec{t}_{\mathsf{var}}\ \vec{t}_{\mathsf{var}};\ a_{\mathsf{body}}\}$
```

```
procedure $M(\vec{t}_{\mathsf{in}}\ \vec{x}_{in})$ returns $(\vec{t}_{\mathsf{out}}\ \vec{x}_{\mathsf{out}})$
    requires $w_{\mathsf{pre}}$
    ensures $w_{\mathsf{post}}$
    $\{\vec{t}_{\mathsf{var}}\ \vec{t}_{\mathsf{var}};\ h_{\mathsf{body}}\}$
```

Figure 5.8: Method syntax: we distinguish between abstract atomic and non-atomic methods. Abstract atomic methods can specify arbitrary many interference clauses.

argument sequence. Otherwise levels are aggregated. In Voila a verified method holds for all possible atomicity contexts that satisfy the aforementioned constraint. The method level is defined that way because the only constraint on judgment levels enforced by TaDA rules is that a judgment level has to be larger than the level of a used region. Similarly for the atomicity context, where the definition guarantees that no region occurring in the precondition has an entry in the atomicity context of the method. The functions $\textsc{Pre}(M)$, $\textsc{Post}(M)$, $\textsc{Inter}(M)$, $\textsc{Formal}_{\mathsf{in}}(M)$, and $\textsc{Formal}_{\mathsf{out}}(M)$ refer to the method's pre- and postcondition, interference clause, and formal in- and out-arguments, respectively.

Well-Definedness A Voila program has several well-definedness conditions: first, regarding stability, region interpretations, loop invariants, as well as method pre- and postconditions are required to be stable. This requirement comes directly from TaDA. Furthermore, the transition system defined be the action set of each region has to be transitively closed, i.e. for some states a, b, c, some guard g, and some actions A_1 and A_2, if $(a,b) \in \mathbb{T}_{A_1}(g)$ and $(b,c) \in \mathbb{T}_{A_2}(g)$ hold, then there has to be a guard A_3 with $(a,c) \in \mathbb{T}_{A_3}(g)$. As a consequence, our definition of the state transition system $\mathbb{T}^{\star}_{R}(g)$ is sound. Variables that are bound by the binding operator $?$ are not allowed to occur before they are bound. Lastly, all expressions have to be well-typed and all variables have to be properly declared. All well-definedness conditions can be checked by the Voila verifier.

Chapter 6

Viper Encoding

As seen in section 5.2, a Voila program consists of structure, region, and method declarations. A Voila program is correct if it is well-defined and if for each contained method the implementation satisfies its specification. We have developed the Voila verifier to decide whether or not a Voila program is correct. Moreover, if the program is incorrect the tool tries to return a useful error message. The verification approach taken by the Voila verifier goes as follows: the source Voila program is encoded to the Viper verification language. One of the Viper verifiers then verifies the generated Viper program. The Voila verifier returns a verification success if the Viper verifier successfully runs on the generated Program. Otherwise, if the Viper verifier does not run successfully, i.e. the generated Viper program is incorrect, the Viper error message is translated back to a Voila error message. In this chapter, we first introduce the basic encoding techniques showcased on the `addEven` method. Afterwards, we show the complete encoding.

6.1 Verification Approach

In figure 6.1 we give the encoding of the cell structure, as well as part of the encoding of the region. For the cell structure, a single field of type int is generated. More interesting are the predicate and the heap-dependent function generated for the **ECounter** region declaration. The region interpretation, in this case the points-to predicate `x.val ↦ ?s` as well as the invariant `even(s)`, is encoded as the `ECounter_interp` predicate. In the encoding permission to the predicate $R\texttt{\_interp}(a, \lambda, \vec{e})$ correspond to the knowledge of the existence of the region instance $R_a^\lambda(\vec{e})$. The state definition of **ECounter**, i.e. the value of the `x.val` field being the state of the region, is encoded by the `ECounter_state` heap-dependent function.

Next, is the encoding of the `addEven` method given in figure 6.2. The pre- and postcondition of the Voila method are directly encoded into the pre- and

```
field val: Int

predicate ECounter_interp(r: Ref, lvl: Int, x: Ref)
{ x.val ↦ ?s && even(s) }

function ECounter_state(: Ref, lvl: Int, x: Ref): Int
  requires acc(ECounter_interp(r, lvl, x))
{ unfolding acc(ECounter_interp(r, lvl, x) in x.val }
```

Figure 6.1: Encoding of the `cell` structure as well as the encoding of the interpretation and state clause of the **ECounter** region declaration.

postconditions of the generated Viper method: the previously introduced `ECounter_interp(r,lvl,x)` asserts the existence of an **ECounter** region instance, and `ECounter_state(r,lvl,x)` asserts constrains on the state of the region instance. Because levels are encoded using integer, the pre- and postcondition also contain `lvl >= 0`. The held **G** guard is encoded as held permission to an abstract predicate with the same name. The interference clause of the Voila method is encoded inside of the Viper method body: the for `ECounter(r,lvl,x)` specified interference set `Nat` is assumed to be the value of `ECounter_ictxt(r,lvl,x)`. In general, the abstract function *R*`_ictxt` is used to encode the Voila interference rely-guarantees. We refer to the set of values in *R*`_ictxt` also as the *interference context*, i.e. the interference rely-guarantee on the Viper level. The abstract function has the abstract predicate `ECounter_ifp` as its footprint, hence permission to that predicate is inhaled before the function is used. In the encoding we inhale the permission to `ECounter_ifp` for all region instances because in Viper this code has a better performance than inhaling the permission only if permission to `ECounter_interp` is held. Lastly, the level of the method is stored in the variable `level`. At the beginning of the Viper method body `level` is assumed to be larger than the region level `lvl` occurring in the Voila precondition because, as defined in section 5.2, the level of a Voila method is one plus the highest region level from the precondition. After the initialization of the interference context and the level, the declared Voila variables are declared in Viper and the encoding of the Voila `addEven` method body is generated.

A sketch of the `addEven` body encoding is given in figure 6.3. The encoding of the assignment as well as the encoding of the compare-and-swap call are straightforward. For the while loop, the invariant and the loop condition are encoded. Furthermore, some initialization code is generated inside the while body. Each Voila rule application is split into a prelude and postlude. For now, we do not show explicitly how the inferred atomic statements are encoded. As mentioned before, some Voila assertions have

```
predicate ECounter_ifp(r: Ref, lvl: Int, x: Ref)

function ECounter_ictxt(r: Ref, lvl: Int, x: Ref): Set[Int]
  requires acc(ECounter_ifp(r,lvl,x))

predicate G(r: Ref)

method addEven(r: Ref, lvl: Int, x: Ref)
  requires acc(ECounter_interp(r,lvl,x)) && acc(G(r))
  requires lvl >= 0
  ensures acc(ECounter_interp(r,lvl,x)) && acc(G(r))
  ensures lvl >= 0
  ensures ECounter_state(r,lvl,x) ==
          old(ECounter_state(r,lvl,x)) + 2
  {
    inhale forall r: Ref, lvl: Int, x: Ref ::
           acc(ECounter_ifp(r,lvl,x))
    assume ECounter_ictxt(r,lvl,x) == Nat

    var level: Int
    assume level > lvl

    var z: Int
    var b: Bool
    // encoding of Voila addEven method body
  }
```

Figure 6.2: Encoding of the `addEven` method without the method body.

to be stable. Analogously, the *knowledge of the verifier*, in other words, the set of all assertions that are derivable at some position in the encoding, has to be stabilized as well. Otherwise, the verifier could prove unsound properties, such as the value of `x.val` at the compare-and-swap call being always equal to `z`, which is false because the environment might change the field value in between. Therefore, in the encoding, code to stabilize the verifier knowledge is generated after the open_region and update_region application. In the rest of this section, we show how different components of a TaDA proof appear in the Voila encoding.

Level Checks In the method encoding of `addEven` given in figure 6.2, we saw how the method level is initialized. Afterwards in the method body, the `level` variable is used as the level of the current judgment, i.e. as the level with which the current atomic or non-atomic triple is proven. In figure

```
// make_atomic pre
do
  invariant // encoded invariant
{
  // while pre
  // open_region pre
  z := x.val
  // open_region post
  // stabilize verifier knowledge
  // update_region pre
  b := CAS_val(x, z, z+2)
  // update_region post
  // stabilize verifier knowledge
} while (b);
// make_atomic post
```

Figure 6.3: Encoding sketch of the `addEven` body.

6.4 we extended the previously shown encoding sketch with the generated checks on the judgment level. As mentioned in 2, rule applications that open a region require that the judgment level is larger than the opened region level. Furthermore, when the region is opened, the judgment level is set to the used region level. Corresponding checks and assignments are generated in the example for open_region and update_region. The variable `store_level` is used to temporarily store the value of `level` so that the judgment level can be restored after the region is closed again.

Atomicity Context Recall that the atomicity context is another part of a TaDA judgment. The atomicity context is only extended by MakeAtomic, and loses the added entry at the use of UpdateRegion. Figure 6.5 shows the proof sketch extended with atomicity context checks. In the prelude of the make_atomic statement, the atomicity context for the used region, in this case, the **ECounter** instance, is first checked to not already be defined and is then stored. Similarly to the interference context, the atomicity context is encoded trough an abstract function *R*`_actxt`. However, we do not encode the atomicity context as a mapping, but instead encode the domain of each mapping with *R*`_actxt`. We refer to those encoded domains also as *rely context*. For our example the rely context of the region instance is stored by assuming it to be the value of `ECounter_actxt(r,lvl,x)`. Again, the abstract function has a single abstract predicate `ECounter_afp` as its footprint. Therefore, before using the function the permission to the predicate has to be inhaled first. The check that the atomicity context was not defined beforehand is done by asserting that no permission to the `ECounter_afp(r,lvl,x)` is held. After-

```
// make_atomic pre
do
  invariant // encoded invariant
{
  // while pre
  // open_region pre
  assert level > lvl
  store_level := level
  level := lvl
  z := x.val
  // open_region post
  level := store_level
  // stabilize verifier knowledge
  // update_region pre
  assert level > lvl
  store_level := level
  level := lvl
  b := CAS_val(x, z, z+2)
  // update_region post
  level := store_level
  // stabilize verifier knowledge
} while (b);
// make_atomic post
```

Figure 6.4: Encoding sketch of the `addEven` body with level checks included.

wards, in the postlude of the make_atomic statement, the atomicity context is brought back to its former state by exhaling the predicate permission. In accordance to the aforementioned rules, the inverse, i.e. first exhaling, then inhaling the permission to the predicate, is done at the update_region statement. Inside of the while-loop, the knowledge of the verifier regarding the rely context is framed inside the loop from before the loop.

Guards and Tracking Resources As shown in the encoding of the `addEven` specification, guards are encoded as permissions to abstract predicates. Not yet introduced is the encoding of tracking resources: for the diamond resource a field `diamond` is generated, and for the transition resource fields `from` and `to` are generated. In general, because the type of the `from` and `to` fields depend on the region state type, versions of both transition resource fields are generated for each declared region. Figure 6.6 shows the proof sketch extended with guard checks and tracking resources. In our example the make_atomic statement is the only statement that enforces checks on guards: first, before make_atomic the **ECounter** guard specified in the

```
predicate ECounter_afp(r: Ref, lvl: Int, x: Ref)

function ECounter_actxt(r: Ref, lvl: Int, x: Ref): Set[Int]
  requires acc(ECounter_afp(r,lvl,x))

...
// make_atomic pre
assert perm(ECounter_afp(r,lvl,x)) == none
inhale acc(ECounter_afp(r,lvl,x))
assume ECounter_actxt(r,lvl,x) == ECounter_ictxt(r,lvl,x)
label preLoop
do
  invariant // encoded invariant
{
  inhale acc(ECounter_afp(r,lvl,x))
  assume ECounter_actxt(r,lvl,x) ==
        old[preLoop](ECounter_actxt(r,lvl,x))
  // open_region pre
  z := x.val
  // open_region post
  // stabilize verifier knowledge
  // update_region pre
  exhale acc(ECounter_afp(r,lvl,x))
  b := CAS_val(x, z, z+2)
  // update_region post
  inhale acc(ECounter_afp(r,lvl,x))
  // stabilize verifier knowledge
} while (b);
// make_atomic post
exhale acc(ECounter_afp(r,lvl,x))
```

Figure 6.5: Encoding sketch of the addEven body with the atomicity context encoding included.

`with` clause has to be owned. Second, the state transition performed in the make_atomic body has to be permitted by the specified guard. Recall that in MakeAtomic the performed state transition is tracked by the transition resource, hence the transition on **ECounter** is permitted if `r.to` is even as well as larger than `r.from`. Besides the guard checks, in the make_atomic prelude, the guard is exhaled and replaced with the diamond resource. In the postlude, the new region state is set as the target of the transition resource, the transition resource is removed, and finally, the guard is inhaled again. With the encoding of the transition resources, we can now encode the invariant of the while loop straightforwardly. Lastly, the `update_region` encoding requires the diamond resource being held in its prelude, and in the postlude either inhales the transition resources and assigns them accordingly if a state change occurred, or otherwise inhales the diamond resource again.

6.2 Encoding

In the encoding, we use sub-routines, written in italics, generating Viper code. Expressions and statements are encoded with an encoding context ϑ. The context maps variables to viper expressions. We write ϑ_0 for the initial context that respects all bindings. More formally, for each expression e occurring in the encoding at a Viper label `l` the initial context is defined as follows:

$$\vartheta_0(x) = \begin{cases} \textbf{old}\texttt{[l]}(R\texttt{\_state}(\vec{e}_{in})), & \text{if } R(\vec{e}_{in}; \vec{q}_{out}; ?x) = e \\ \textbf{old}\texttt{[l]}(R\texttt{\_state}(\vec{e}_{in})), & \text{if } (?x \text{ in } e_{set} \text{ on } R(\vec{e}_{in})) = e \\ \textbf{old}\texttt{[l]}(R\texttt{\_out\_}i(\vec{e}_{in})), & \text{if } R(\vec{e}_{in}; q_1, \ldots, q_{i-1}, ?x, \ldots; q_{state}) = e \\ \textbf{old}\texttt{[l]}(r.R\texttt{\_from}), & \text{if } (r \mapsto\!\!\!\Rightarrow (?x, q_{to})) = e \\ & \quad \text{and } r \text{ belongs to } R \\ \textbf{old}\texttt{[l]}(r.R\texttt{\_to}), & \text{if } \left(r \mapsto\!\!\!\Rightarrow (q_{from}, ?x)\right) = e \\ & \quad \text{and } r \text{ belongs to } R \\ \textbf{old}\texttt{[l]}(y.F), & \text{if } (y.F \mapsto ?x) = e \\ x, & \text{otherwise} \end{cases}$$

In the real implementation we simply assign the corresponding function value to the variable when it is bound. This trivially, guarantees that heap-dependent expressions are evaluated in the same state in which they are bound. The R`_out_`i function returns the i-th region out-argument for region type R (See section 5.2). As a reminder, we make the simplifying assumption that each binding variable is globally unique, i.e. it is only bound once. Otherwise, the context has to be initialized differently for each scope. Furthermore, to identify which region type belongs to which region identifier, the Voila tool tracks the region identifier used inside region assertions.

```
field diamond: Bool
field from: Int
field to: Int

// make_atomic pre
exhale acc(G(r))
inhale acc(r.diamond)
do
  invariant acc(ECounter_interp(r,lvl,x))
  invariant b ? acc(r.diamond) :
    acc(r.from) && r.from == ECounter_state(r,lvl,x) &&
    acc(r.to) && r.to == ECounter_state(r,lvl,x) + a
{
  // while pre
  // open_region pre
  z := x.val
  // open_region post
  // stabilize verifier knowledge
  // update_region pre
  exhale acc(r.diamond)
  lable preUpdate
  b := CAS_val(x, z, z+2)
  // update_region post
  if (ECounter_state(r,lvl,x) ==
      old[preUpdate](ECounter_state(r,lvl,x))) {
    inhale acc(r.diamond)
  } else {
    inhale acc(r.from) && acc(r.to)
    r.from := old[preUpdate](ECounter_state(r,lvl,x))
    r.to := ECounter_state(r,lvl,x)
  }
  // stabilize verifier knowledge
} while (b);
// make_atomic post
assert r.from < r.to && even(r.to)
assume ECounter_state(r,lvl,x) == r.to
exhale acc(r.from) && acc(r.to)
inhale acc(G(r))
```

Figure 6.6: Encoding sketch of the `addEven` body with guard checks and tracking resources included.

In case the tool cannot resolve a region identifier, an error is returned. Lastly, in the encoding we treat Voila variables and Viper variables synonymously. This property is sometimes exploited by using Viper variables inside Voila expressions.

Intuitively, to guarantee soundness of our verification approach, the encoding satisfies that for each Voila statement s, if Viper verifies on the encoding of s for some encoded pre- and postcondition $[\![P]\!]\vartheta_0$, $[\![Q]\!]\vartheta_0$, then the same non-encoded pre- and postcondition P, Q are a correct specification for s in Voila. The encoding satisfies a similar property for method declarations. For a more detailed soundness discussion, see chapter 7.

Regarding terminology, we refer to the assertions that would be pre- and postcondition of a Viper judgment for an encoded statement as the pre- and postcondition of the encoding.

6.2.1 Declarations

The encoding of structures and regions given in figure 6.7 is a straightforward generalization of the shown **ECounter** encoding: for each field declared in a structure, a Viper field declaration is generated. For a region type R, the `interp` and `state` clauses are encoded as predicate R`_interp` and function R`_state` with return type T_R, respectively, where T_R is the state type of R. For all guard declarations $k\ G(\vec{x})$ an abstract predicate $G(r, \vec{x})$ is generated where the first argument is the region identifier of a regioned guard. The guard algebra k does not influence the generated predicate. As shown in the previous section, to encode the interference rely-guarantee and atomicity context of a region instance, the function-predicate pairs R`_ictxt` and R`_ifp`, as well as R`_actxt` and R`_afp` are used. Lastly, the diamond tracking resource is encoded as a `diamond` field, and the transition tracking resource is encoded as the pair of fields R`_from` and R`_to` for each region. The transition resource fields are generated for each region because they hold values of the region state type. Another option would be to generate them for each occurring region state type.

Figure 6.8 shows the encoding of abstract atomic methods. Similar to regions, the method encoding is a straightforward generalization of the shown `addEven` encoding: the pre- and postcondition of the Voila method are encoded as the pre- and postcondition of the generated Viper method. The interference clause is encoded inside the Viper method body by first inhaling permissions to R`_ifp` and then storing the interference clauses with R`_ictxt`. The current judgment level is encoded as the `level` variable. Hence, at the beginning, the level is initialized as the method level by assuming `level` to be larger than all region levels occurring in the method precondition. This level initialization is sound because the exact method level, i.e. one plus the maximum occurring region level, is included as a

```
field diamond: Bool

⟦struct  S{$\vec{t}$  $\vec{F}$}⟧  ⇝
  foreach  ($t$  $F$) ∈ ($\vec{t}$  $\vec{F}$):
    field $F$: ⟦$t$⟧
  end

⟦region  $R(\vec{t}_{in}\ \ \vec{x}_{in};\ \vec{t}_{out}\ \ \vec{x}_{out})\{\vec{gdef}\}\{w_{interp}\}\{e_{state}\}\{\vec{action}\}$⟧  ⇝
  predicate $R$_interp($\vec{x}_{in}$ : ⟦$\vec{t}_{in}$⟧) { ⟦$w_{interp}$⟧$\vartheta_0$ }

  function $R$_state($\vec{x}_{in}$ : ⟦$\vec{t}_{in}$⟧): ⟦$T_R$⟧
    requires acc($R$_interp($\vec{x}_{in}$))
  { unfolding acc($R$_interp($\vec{x}_{in}$)) in ⟦$e_{state}$⟧$\vartheta_0$ }

  foreach  ($t_i$  $x_i$) ∈ ($\vec{t}_{out}$  $\vec{x}_{out}$):
    function $R$_out_$i$($\vec{x}_{in}$ : ⟦$\vec{t}_{in}$⟧): ⟦$t_i$⟧
      requires acc($R$_interp($\vec{x}_{in}$))
    { unfolding acc($R$_interp($\vec{x}_{in}$)) in ⟦$x_i$⟧$\vartheta_0$ }
  end

  field $R$_from: ⟦$T_R$⟧
  field $R$_to: ⟦$T_R$⟧

  predicate $R$_afp($\vec{x}_{in}$ : ⟦$\vec{t}_{in}$⟧)

  function $R$_actxt($\vec{x}_{in}$ : ⟦$\vec{t}_{in}$⟧): Set[⟦$T_R$⟧]
    requires acc($R$_afp($\vec{x}_{in}$))

  predicate $R$_ifp($\vec{x}_{in}$ : ⟦$\vec{t}_{in}$⟧)

  function $R$_ictxt($\vec{x}_{in}$ : ⟦$\vec{t}_{in}$⟧): Set[⟦$T_R$⟧]
    requires acc($R$_ifp($\vec{x}_{in}$))

  foreach  $k$  $G$($\vec{t}$  $\vec{x}$) ∈ $\vec{gdef}$:
    predicate $G$($r$ : ⟦id⟧, $\vec{x}$ : ⟦$\vec{t}$⟧)
  end
```

Figure 6.7: Encoding of structures and regions. Foreach loops are statically expanded in the translation.

```
⟦abstract_atomic procedure $\{\vec{rely}\}\{w_{pre}\}\{w_{post}\}\{\vec{t}_{var}\ \vec{x}_{var};\ a\}$⟧ $\rightsquigarrow$
  method $M(\vec{x}_{arg} : [\![\vec{t}_{arg}]\!])$ returns $(\vec{x}_{ret} : [\![\vec{t}_{ret}]\!])$
    requires $[\![w_{pre}]\!]\vartheta$
    ensures $[\![w_{post}]\!]\vartheta$
  {
    var level: Int
    var atomicity_level: Int

    foreach $l \in$ LEVELS$(w_{pre})$:
      assume level > $[\![l]\!]\vartheta$
    end
    atomicity_level := level

    foreach $R(\vec{t}\ \vec{x}) \in$ FORMAL$_{in}$(REGIONS):
      inhale forall $\vec{x}$: $[\![\vec{t}]\!]$ :: acc($R$_ifp($\vec{x}$))
    end

    foreach $(y$ in $e_{set}$ on $R(\vec{e}_{arg})) \in \vec{rely}$:
      assume $R$_ictxt($[\![\vec{e}_{arg}]\!]\vartheta$) == $[\![e_{set}]\!]\vartheta$ &&
             $R$_state($[\![\vec{e}_{arg}]\!]\vartheta$) in $R$_ictxt($[\![\vec{e}_{arg}]\!]\vartheta$)
    end

    var $\vec{x}_{var}$: $[\![\vec{t}_{var}]\!]$
    $[\![a]\!]\vartheta$

    foreach $R(\vec{t}\ \vec{x}) \in$ FORMAL$_{in}$(REGIONS):
      exhale forall $\vec{x}$: $[\![\vec{t}]\!]$ :: acc($R$_ifp($\vec{x}$))
    end
  }
```

Figure 6.8: Encoding of abstract atomic methods. For non-atomic methods the encoding is the same, but without inhaling permissions for R_ifp and without constraining R_ictxt.

potential value of `level`. The variable `atomicity_level` is used to store a lower bound of the region levels whose region is in the atomicity context of the current judgment. As mentioned in section 5.2, in Voila the atomicity context of a method is any mapping that has no entry for a region with a level lower than the method level. Hence, a lower bound of the region levels whose region is in the method atomicity context is the method level. The encoding for non-atomic methods is equal, except the interference clause is not present, hence not encoded, i.e. no permissions to R_ifp are inhaled and R_ictxt is not constraint.

$[\![\text{id}]\!] \rightsquigarrow \mathtt{Ref}$	
$[\![\text{bool}]\!] \rightsquigarrow \mathtt{Bool}$	
$[\![\text{int}]\!] \rightsquigarrow \mathtt{Int}$	$[\![x]\!]\vartheta \rightsquigarrow \vartheta(x)$
$[\![\text{frac}]\!] \rightsquigarrow \mathtt{Perm}$	$[\![?x]\!]\vartheta \rightsquigarrow \vartheta(x)$
$[\![S]\!] \rightsquigarrow \mathtt{Ref}$	$[\![n]\!]\vartheta \rightsquigarrow n$
$[\![\text{set}\langle t\rangle]\!] \rightsquigarrow \mathtt{Set}[[\![t]\!]]$	$[\![e_1 + e_2]\!]\vartheta \rightsquigarrow [\![e_1]\!]\vartheta + [\![e_2]\!]\vartheta$
$[\![\text{tuple}\langle\vec{t}\rangle]\!] \rightsquigarrow \mathtt{Tuple}[[\![\vec{t}]\!]]$	...

Figure 6.9: Encoding of types and expressions.

6.2.2 Types and Expressions

The encoding for types and some of the expressions are given in figure 6.9. Regarding types, id, bool, int, frac, set types, and tuples types are mapped to the corresponding primitive types of Viper. Furthermore, structure types are encoded as the Viper reference type `Ref`. Regarding expressions, variables and binders are encoded through the context. The remaining expressions are encoded straightforwardly since each Voila operation has a corresponding Viper operation.

For assertions, we only discuss the encoding of the added TaDA assertions given in figure 6.10. A region assertion $R(\vec{e}_{in}; \vec{q}_{out}; q_{state})$ is encoded as access to its interpretation R`_interp`$(\vec{e}_{in})$. Furthermore, conditions similar to well-formedness checks are generated guaranteeing that the region level is positive and that the out-arguments as well as the state are equal to their specified values. Points-to predicates are encoded as access permissions to the provided field as well as an equality constraint on the value of the field. The tracking resources are encoded similarly but instead operate on the fields `diamond`, R`_from`, and R`_to`.

6.2.3 Non-Atomic Statements

As mentioned in section 5.2, non-atomic triple pre- and postconditions, as well as preconditions of atomic triples are required to be stable. As a consequence, the encoding of non-atomic statements has to guarantee that the encoding postcondition is stable as well. As mentioned before, the verifier knowledge is stabilized by explicitly generating stabilization code which is done by the *stabilize* subroutine.

Figure 6.11 shows the encoding of sequential composition, pure assignments, if-else branches, and while loops. None of these encodings requires stabilization because the assigned value of the pure assignment, the if-else condition, and the while condition, respectively, do not change the knowledge of the heap. Furthermore, the invariant of the while loop is stable according to the well-definedness condition shown in 5.2. The encoding of sequential com-

```
⟦$R(\vec{e}_{in}; \vec{q}_{out}; q_{state})$⟧$\vartheta$  $\rightsquigarrow$
  acc($R$_interp(⟦$\vec{e}_{in}$⟧$\vartheta$)) && ⟦lvl($\vec{e}_{in}$)⟧$\vartheta$ >= 0 &&
  ⟦$q_{state}$⟧$\vartheta$ == $R$_state(⟦$\vec{e}_{in}$⟧$\vartheta$)
  foreach $q_i \in \vec{q}_{out}$:
    && ⟦$q_i$⟧$\vartheta$ == $R$_out_$i$(⟦$\vec{e}_{in}$⟧$\vartheta$)
  end

⟦$x.F \mapsto q$⟧$\vartheta$  $\rightsquigarrow$ acc(⟦$x$⟧$\vartheta$.F) && ⟦$x$⟧$\vartheta$.F == ⟦$q$⟧$\vartheta$
⟦$r \Mapsto \blacklozenge$⟧$\vartheta$  $\rightsquigarrow$ acc(⟦$r$⟧$\vartheta$.diamond)
⟦$r \Mapsto (q_{from}, q_{to})$⟧$\vartheta$  $\rightsquigarrow$
  acc(⟦$r$⟧$\vartheta$.$R$_from) && ⟦$r$⟧$\vartheta$.$R$_from == ⟦$q_{from}$⟧$\vartheta$ &&
  acc(⟦$r$⟧$\vartheta$.$R$_to) && ⟦$r$⟧$\vartheta$.$R$_to == ⟦$q_{to}$⟧$\vartheta$

  where the identifier $r$ belongs to region $R$
```

Figure 6.10: Encoding of assertions. *lvl*($\vec{e}_{in}$) returns the second entry of $\vec{e}_{in}$ which corresponds to the region level of the region assertion.

position, pure assignments, and if-else branches is straightforward because the same statements exist in Viper. For while loops, the knowledge about the atomicity context has to be framed inside the loop. This framing is performed in two steps: first, inhaling all the permissions to the rely context footprint R_afp that were held before the loop. Second, if permission was held, then the value of R_actxt is constrained to be the same as before. Regarding the notation, $\text{Formal}_{\text{in}}$(Regions) is the set of all regions together with their respective formal in-arguments. Furthermore, • indicates that variables or labels are fresh, i.e. do not occur elsewhere in the encoding.

The encoding of non-atomic method calls is given in figure 6.12. Intuitively, a call to a non-atomic method consists of four steps: first, the current judgment level is checked to be at least the method level. As mentioned in section 5.2, in Voila the method level is equal to the highest occurring level from the precondition plus 1. Hence, it suffices if the judgment level is larger than every level from the callee precondition. Second, the atomicity context of the current judgment has to be the same as one of the callee atomicity contexts. Recall, the atomicity context of a method is any mapping with no entry for a region with a level lower than the method level. Hence, the atomicity context of the current judgment has to guarantee that it does not contain a region with a level lower than the callee method level. Furthermore, recall that atomicity_level stores a lower bound on the levels of regions in the atomicity context of the current judgment. Thus, it is enough to check that atomicity_level is larger or equal to the callee method level. Third, the callee precondition has to hold. Lastly, the postcondition is assumed to hold. As we will discuss in chapter 7, the TaDA logic requires to stabilize

```
$[\![h_1; h_2]\!]\vartheta \rightsquigarrow [\![h_1]\!]\vartheta$; $[\![h_2]\!]\vartheta$
$[\![x := e]\!]\vartheta \rightsquigarrow [\![x]\!]\vartheta$ := $[\![e]\!]\vartheta$
$[\![\text{if}(b)\ \{h_1\}\ \text{else}\ \{h_2\}]\!]\vartheta \rightsquigarrow$ if($[\![b]\!]\vartheta$) {$[\![h_1]\!]\vartheta$} else {$[\![h_2]\!]\vartheta$}

$[\![\text{while}(b)\ \text{invariant}\ w_{inv}\ \{h\}]\!]\vartheta \rightsquigarrow$
  label $\bullet preLoop$
  while ($[\![b]\!]\vartheta$) invariant $[\![w_{inv}]\!]\vartheta$ {
    foreach $R(\vec{t}\ \vec{x}) \in$ FORMAL_in(REGIONS):
      inhale forall $\vec{x}$: $[\![\vec{t}]\!]$ :: acc[$preLoop$]($R$_afp($\vec{x}$))
      assume forall $\vec{x}$: $[\![\vec{t}]\!]$ ::
              perm[$preLoop$]($R$_afp($\vec{x}$)) > none ==>
                $R$_actxt($\vec{x}$) == old[$preLoop$]($R$_actxt($\vec{x}$))
    end
    $[\![h]\!]\vartheta$
  }
```

Figure 6.11: Encoding of sequential composition, if-else branches, while-loops, and assignments.

```
$[\![\vec{x} := M_H(\vec{e})]\!]\vartheta \rightsquigarrow$ call($\vec{x}$, $M$, $[\![\vec{e}]\!]\vartheta$)

call($\vec{x}$, $M$, $\vec{v}$) $\rightsquigarrow$
  label $\bullet preCall$
  foreach $l \in$ LEVELS(PRE($M$)):
    assert level > $[\![l]\!]\vartheta'$ && atomicity_level > $[\![l]\!]\vartheta'$
  end
  exhale $[\![\text{PRE}(M)]\!]\vartheta'$
  stabilize
  havoc($\vec{x}$)
  inhale $[\![\text{POST}(M)]\!]\vartheta''$

  where ($\vec{t}_{in}\ \vec{y}_{in}$) = FORMAL_in($M$) and $\vartheta' = \vartheta[\vec{y}_{in} \mapsto \vec{v}]$
        ($\vec{t}_{out}\ \vec{y}_{out}$) = FORMAL_out($M$) and $\vartheta'' = \vartheta'[\vec{y}_{out} \mapsto \vec{x}]$
```

Figure 6.12: Encoding of calls to non-atomic methods.

assertions that are not in the premise of a method call. This is achieved by stabilizing in between the precondition exhale and postcondition inhale because at his point, in a TaDA sense, only the assertions not required by the call remain. The postcondition of the encoding does not have to be stabilized because, as mention before, method postconditions are stable, and the separating conjunction of two stable assertions is stable itself.

As mentioned in section 5.2, the statement atomic$\{a\}$ switches for a from a non-atomic triple to an atomic triple and switches back afterwards. Recall

```
⟦atomic{$a$}⟧$\vartheta$ ⇝
  infer-interference-context
  ⟦$a$⟧$\vartheta$
  foreach $R(\vec{t}\ \vec{x}) \in \textsc{Formal}_{\text{in}}(\textsc{Regions})$:
    exhale forall $\vec{x}$: ⟦$\vec{t}$⟧ :: acc($R$_ifp($\vec{x}$))
  end
  stabilize
```

Figure 6.13: Encoding of the switch from non-atomic triples to atomic triples.

that atomic triples in TaDA specify an interference rely-guarantee expressing an upper bound on the interference caused by the environment under which the specification has to hold. The *infer-interference-context* subroutine generates Viper code that infers for each region instance an upper bound on the interference caused by the environment and stores it with R_ictxt. The generated code of the *infer-interference-context* subroutine is given in section 6.2.9. The encoding of the atomic$\{a\}$ statement is shown in figure 6.13: first *infer-interference-context* is called to infer the interference context. Afterwards, the interference context is forgotten by exhaling all the footprint permissions. Finally, because postconditions of atomic statement encodings do not have to be stable, the *stabilize* sub-routine is called.

6.2.4 Abstract Atomic Statements

We split the atomic statements into three groups: Basic statements, rule statements that open and close a region interpretation, and the make_atomic statement. The encoding of the basic statements, namely heap read and write, compare-and-swap calls, and calls to abstract atomic methods are given in figure 6.14. Heap read and writes are directly encoded to Viper. For the compare-and-swap call, a generated compare-and-swap Viper method is called. Lastly, calls to abstract atomic methods are encoded similar to calls to non-atomic methods, except an additional check that the interference clauses of the callee are satisfied. An interference clause is satisfied if the tracked interference context of a region, i.e. the upper bound on the interference caused by the environment on a region instance, is contained in the interference set specified by the callee interference clause. Intuitively, then the callee does not have to deal with more interference than the callee is specified to tolerate.

6.2.5 Inspection Rules

The encoding of all rule statements that open and close a region, namely use_atomic, open_region, and update_region, is given in figure 6.15. The general encoding pattern for those rule statements goes as follows: first, we store

```
$[\![x.F \ := \ e]\!]\vartheta \ \rightsquigarrow \ [\![x]\!]\vartheta$.F := $[\![e]\!]\vartheta$
$[\![x \ := \ y.F]\!]\vartheta \ \rightsquigarrow \ [\![x]\!]\vartheta$ := $[\![y]\!]\vartheta.F$
$[\![x \ := \ \mathsf{CAS}_F(e_1, e_2, e_3)]\!]\vartheta \ \rightsquigarrow$ x := CAS_F($[\![e_1]\!]\vartheta$, $[\![e_2]\!]\vartheta$, $[\![e_3]\!]\vartheta$)

$[\![\vec{x} \ := \ M_A(\vec{e})]\!]\vartheta \ \rightsquigarrow$
  foreach (?$y$ in $e_{set}$ on $R(\vec{e}_{in})) \in$ INTER($M$):
    assert $R$_ictxt($[\![\vec{e}_{in}]\!]$)$\vartheta'$ subset $[\![e_{set}]\!]\vartheta'$
  end
  call($\vec{x}$, $M$, $[\![\vec{e}]\!]\vartheta$)

  where ($\vec{t}_{in}$ $\vec{z}_{in}$) = FORMAL$_{\mathsf{in}}$($M$) and $\vartheta' = \vartheta[\vec{z}_{in} \mapsto [\![\vec{e}]\!]\vartheta]$
```

Figure 6.14: Encoding of heap mutations, heap lookups, and calls to abstract-atomic methods.

all the arguments of the region instance and guard instance specified by the `using` and `with` clause, respectively. For guards, ARGS(g) returns a sequence $\vec{e}$ of the guard arguments. Moreover, a sequence of arguments $\vec{e}$ can be applied to a guard with $g(\vec{e})$ where the existing guard arguments of g are replaced with the new arguments $\vec{e}$. The region and guard arguments are stored to trivially guarantee that the region and guard instance is always the same. We store the arguments by writing their values to fresh variables, annotated with $\bullet$. Next, if required, some checks before opening the region are generated. Then, the region is opened and closed with a call to the *inspect* subroutine. Lastly, checks after closing the region are generated. We first discuss the code generated by *inspect* and then go through the additional checks for each rule statement.

The `inspect` subroutine handles the following tasks: first, the current judgment level is checked to be larger than the region level of the opened region. Afterwards, the judgment level is set to the region level. Next, the region is opened by unfolding the interpretation predicate R`_interp`. Furthermore, the R`_state` function for the opened region is havoced to prevent Viper from framing information about the region state around the fold and unfold. The footnote of the *havoc-state* subroutine indicates the Viper label relative to which the region state is havoced. The label *now* notates the current Viper label. The *link-interference-context* subroutine infers and stores the interference contexts of the region instances inside the region interpretation. Lastly, after the body of the statement rule is emitted, the region is closed again by folding the interpretation predicate. Furthermore, the judgment level is restored.

Now we go through the additional checks for each rule statement. The use_atomic statement requires in the prelude that the guard specified in the `with` clause is held and that the modified region instance is not contained in the atomicity context of the current judgment. The access function *rid*$(\cdot)$ returns the region identifier of a region argument sequence. In the postlude, the *action-permitted* subroutines is called to check that the performed transition of the region state is permitted by the specified guard. For the open_region statement, we only need to check at the end that the region state was not changed. Lastly, the update_region statement: in the prelude, the diamond resource is checked to be held and removed. Furthermore, the region instance has to been added to the rely context by make_atomic and the entry is removed. Both those tasks are performed by exhaling the permission to R_afp. In the postlude, the if condition checks whether or not the region state was changed. In case the state was changed the encoding of the transition resource is inhaled and constrained to be the old and new region state accordingly. Otherwise, the encoded diamond resource is inhaled again. Lastly, the rely context entry that was removed in the prelude is restored again.

Regarding the encoding of update_region, the transition resource is only obtained if a change on the region state occurred. This encoding is weaker than the UpdateRegion rule in TaDA because there the transition resource can also be obtained if no state change occurred. It is difficult to encode a stronger version because Viper does not support angelic choice of permissions, i.e. we cannot inhale both tracking resources conditioned on properties we do not know yet. Hence, we have to choose a condition deciding when which resource is inhaled which in our case is whether or not the old state is equal to the new region state.

6.2.6 Make-Atomic Rule

Recall the MakeAtomic rule given again in 6.16 and its attributes: first, in the precondition of the premise, the specified held guard is replaced with a diamond resource and in the postcondition the guard is returned back. Second, the atomicity context is extended with an entry for the modified region. Third, the atomic triple is changed to a non-atomic triple. Fourth, the region state change tracked by the transition resource has to be permitted by the specified guard. Lastly, the rule has strict constraints on the form of the pre- and postcondition.

The same behavior is encoded in the encoding of the make_atomic statement shown in figure 6.17: first, the guard is checked and removed in the beginning by exhaling the guard. In return the diamond resource is inhaled. Then in the postlude, the specified guard is inhaled again. Second, the rely context is extended by inhaling the permission to R_afp and assuming the value of R_actxt to be the same as the interference context stored with R_ictxt. Furthermore, the atomicity_level variable that stores a lower bound on the region levels whose regions are in the atomicity context of

```
$[\![$use_atomic using $R(\vec{e})$ with $g$ $\{a\}]\!]\vartheta$  $\leadsto$
  var $\bullet\vec{rz}$ := $[\![\vec{e}]\!]\vartheta$
  var $\bullet\vec{gz}$ := $[\![\text{ARGS}(g)]\!]\vartheta$
  label $\bullet preUse$
  assert $[\![g(\vec{gz})@rid(\vec{rz})]\!]\vartheta$
  assert perm($R$_afp($\vec{rz}$)) == none
  inspect($R$_interp($\vec{rz}$), $[\![a]\!]\vartheta$)
  action-permitted(
     $R$, $g(\vec{gz})$, old[$preUse$]($R$_state($\vec{rz}$)), $R$_state($\vec{rz}$)
  )

$[\![$open_region using $R(\vec{e})$ $\{a\}]\!]\vartheta$  $\leadsto$
  var $\bullet\vec{rz}$ := $[\![\vec{e}]\!]\vartheta$
  label $\bullet preOpen$
  inspect($R$_interp($\vec{rz}$), $[\![a]\!]\vartheta$)
  assert $R$_state($\vec{rz}$) == old[$preOpen$]($R$_state($\vec{rz}$))

$[\![$update_region using $R(\vec{e})$ $\{a\}]\!]\vartheta$  $\leadsto$
  var $\bullet\vec{rz}$ := $[\![\vec{e}]\!]\vartheta$
  label $\bullet preUpdate$
  exhale acc($rid(\vec{rz})$.diamond)
  exhale acc($R$_afp($\vec{rz}$))
  inspect($R$_interp($\vec{rz}$), $[\![a]\!]\vartheta$)
  if ( $R$_state($\vec{rz}$) != old[$preOpen$]($R$_state($\vec{rz}$))) {
    inhale acc($rid(\vec{rz})$.from) && acc($rid(\vec{rz})$.to)
    $rid(\vec{rz})$.from := old[$preUpdate$]($R$_state($\vec{rz}$))
    $rid(\vec{rz})$.to := $R$_state($\vec{rz}$)
  } else {
    inhale acc($rid(\vec{rz})$.diamond)
  }
  inhale acc($R$_afp($\vec{rz}$))
  assume $R$_actxt($\vec{rz}$) == old[$preUpdate$]($R$_actxt($\vec{rz}$))

inspect($R$_interp($r,lvl,\vec{v}$), $c$)  $\leadsto$
  assert level > $lvl$
  var $\bullet level\_store$ := level
  level := $lvl$
  unfold $R$_interp($r,lvl,\vec{v}$)
  havoc-state$_{now}$($R$_state($r,lvl,\vec{v}$))
  link-interference-context($R$_ictxt($r,lvl,\vec{v}$))
  c
  fold $R$_interp($r,lvl,\vec{v}$)
  level := $level\_store$
```

Figure 6.15: Encoding of use_atomic, open_region, and update_region.

$$\dfrac{a \notin \mathcal{A} \qquad \{(s,y) \mid s \notin S, y \in Y\} \subseteq \mathbb{T}^{\star}_{\mathbf{R}}(G) \qquad \Gamma; \lambda; a : s \in S \rightsquigarrow Y, \mathcal{A} \vdash \begin{array}{c} \{\exists s \in S.\ \mathbf{R}^{\lambda}_{a}(\vec{e}, s) * a \Mapsto \blacklozenge\ \} \\ \mathbb{C} \\ \{\exists s \in S.\ \exists y \in Y.\ a \Mapsto (s,y)\} \end{array}}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{V}s \in S.\ \langle\ \mathbf{R}^{\lambda}_{a}(\vec{e}, s) * [G]_a\ \rangle\ \mathbb{C}\ \langle\ \exists y \in Y.\ \mathbf{R}^{\lambda}_{a}(\vec{e}, y) * [G]_a\ \rangle}\ \textsc{MakeAtomic}$$

Figure 6.16: The MAKEATOMIC TaDA rule.

the current judgment is updated accordingly. In the postlude, the atomicity context as well as the value of `atomicity_level` is restored. Third, the triple is changed to a non-atomic triple by exhaling all the permissions to R`_ifp`, i.e. forgetting the interference context. In the postlude, the triple is changed back by inhaling the permissions again. Fourth, the region state change tracked by the transition resource is checked to be permitted by the specified guard through a call to the *action-permitted* subroutine in the postlude. Furthermore, the region state is constrained according to the transition resource. Afterwards, the transition resource is exhaled. Lastly, to satisfy the strict constraints on the form of the pre- and postcondition, explicit frames are inserted where necessary by calling the *explicit-frame-pre* and *explicit-frame-post* subroutines.

6.2.7 Guards and Actions

So far we have not discussed the different predefined guard algebras of Voila. The syntax for guard expressions is shown in figure 6.18. We differentiate between primitive guards and normal guards. The primitive guard expressions $G_u(\vec{e})$ and $G_d(\vec{e})$ notate unique and duplicable guards, respectively, as defined in section 2. A set of each those guards is written as $G_i\langle e\rangle$ for some $i \in \{u, d\}$ where e is the set of all guard arguments, i.e. $G_i\langle e\rangle$ is the same as $\forall \vec{x} \in e.\ G_i(\vec{x})$. Next, G[p] is a divisible guard defined by $G[p_1] \bullet G[p_2] = G[p_1 + p_2]$ where $p_1 \geq 0$, $p_2 \geq 0$, and $p_1 + p_2 \leq 1$ has to hold. We refer to $G[p]$ as a p fraction of G. Lastly, the product guard g_1 `&&` g_2 combines two guards and is defined by $(g_1$ `&&` $g_2) \bullet (g'_1$ `&&` $g'_2) = (g_1 \bullet g'_1)$ `&&` $(g_2 \bullet g'_2)$. For simplicity we assume that all guard expressions are normalized, i.e. under the product operator `&&` guards are aggregated and ordered in a global order: for example $A[p_1]$ `&&` $B_u(x)$ `&&` $B_u\langle e\rangle$ `&&` $A[p_2]$ is normalized to $A[p_1 + p_2]$ `&&` $B_u\langle e \text{ union } \mathsf{Set}(x)\rangle$.

Figure 6.19 shows the encoding of guards. We have two different kinds of encoded assertions: first, $[\![g@r]\!]\vartheta$ asserts that the regioned guard is held. The encoding of those assertions for each guard type is straightforward: a held guard is encoded as held permission to an abstract guard predicate. For divisible guards, the permission amount corresponds to the permission

```
$[\![$make_atomic using $R(\vec{e})$ with $g$ $\{a\}]\!]\vartheta$ $\rightsquigarrow$
  var $\bullet\vec{rz}$ := $[\![\vec{e}]\!]\vartheta$
  var $\bullet\vec{gz}$ := $[\![\textsc{Args}(g)]\!]\vartheta$

  exhale $[\![g(\bullet\vec{gz})@rid(\vec{rz})]\!]\vartheta$
  exhale acc($R$_interp($\vec{rz}$))

  label $\bullet$preFrame
  explicit-frame-pre

  var $\bullet$atomicity_level_store := atomicity_level
  atomicity_level := min(atomicity_level, lvl($\vec{rz}$))

  assert perm($R$_afp($\vec{rz}$)) == none
  inhale acc($R$_afp($\vec{rz}$)) && $R$_actxt($\vec{rz}$) == $R$_ictxt($\vec{rz}$)
  inhale acc(rid($\vec{rz}$).diamond)
  inhale acc($R$_interp($\vec{rz}$)) && $R$_state($\vec{rz}$) in $R$_actxt($\vec{rz}$)

  foreach $R(\vec{t}\ \vec{x}) \in \textsc{Formal}_{\text{in}}(\textsc{Regions})$:
    exhale forall $\vec{x}$: $[\![\vec{t}]\!]$ :: acc($R$_ifp($\vec{x}$))
  end

  $[\![a]\!]\vartheta$

  action-permitted($R$, $g(\vec{gz})$, rid($\vec{rz}$).from, rid($\vec{rz}$).to)

  foreach $R(\vec{t}\ \vec{x}) \in \textsc{Formal}_{\text{in}}(\textsc{Regions})$:
    inhale forall $\vec{x}$: $[\![\vec{t}]\!]$ :: acc($R$_ifp($\vec{x}$))
  end

  explicit-frame-pre
  inhale acc($R$_interp($\vec{rz}$))
  inhale $R$_state($\vec{rz}$) == rid($\vec{rz}$).to
  exhale acc(rid($\vec{rz}$).from) && acc(rid($\vec{rz}$).to)
  inhale $[\![g(\vec{gz})@rid(\vec{rz})]\!]\vartheta$
  exhale acc($R$_afp($\vec{rz}$))

  atomicity_level := atomicity_level_store

  explicit-frame-post$_{preFrame}$
```

Figure 6.17: Encoding of the make_atomic statement.

$$
\begin{array}{c}
pg ::= G_i(\vec{e}) \mid G_i\langle e\rangle \mid G[p] \text{ where } i \in \{u, d\} \\
g ::= pg \mid g_1 \;\texttt{\&\&}\; g_2
\end{array}
$$

Figure 6.18: Guard syntax. We differentiate between primitive guards pg and guards g.

$$
\begin{array}{l}
[\![G_i(\vec{e})@r]\!]\vartheta \rightsquigarrow \textbf{acc}\texttt{(}G([\![r]\!]\vartheta\texttt{, } [\![\vec{e}]\!]\vartheta\texttt{))} \\
[\![G[p]@r]\!]\vartheta \rightsquigarrow \textbf{acc}\texttt{(}G([\![r]\!]\vartheta\texttt{), } [\![p]\!]\vartheta\texttt{)} \\
[\![G_i\langle e\rangle@r]\!]\vartheta \rightsquigarrow \texttt{forall } \vec{x}\texttt{: } T_G \texttt{ :: } \vec{x} \texttt{ in } [\![e]\!]\vartheta \texttt{ ==> } [\![G_i(\vec{x})@r]\!]\vartheta \\
[\![(g_1 \texttt{ \&\& } g_2)@r]\!]\vartheta \rightsquigarrow [\![g_1@r]\!]\vartheta \texttt{ \&\& } [\![g_2@r]\!]\vartheta \\
\\
[\![\overline{G_d(\vec{e})}@r]\!]\vartheta \rightsquigarrow \textbf{true} \\
[\![\overline{G_u(\vec{e})}@r]\!]\vartheta \rightsquigarrow \textbf{perm}\texttt{(}G([\![r]\!]\vartheta\texttt{, } [\![\vec{e}]\!]\vartheta\texttt{)) == } \textbf{none} \\
[\![\overline{G[p]}@r]\!]\vartheta \rightsquigarrow \textbf{perm}\texttt{(}G([\![r]\!]\vartheta\texttt{)) <= 1 - } [\![p]\!]\vartheta \\
[\![\overline{G_i\langle e\rangle}@r]\!]\vartheta \rightsquigarrow \texttt{forall } \vec{x}\texttt{: } T_G \texttt{ :: } \vec{x} \texttt{ in } [\![e]\!]\vartheta \texttt{ ==> } [\![\overline{G_i(\vec{x})}@r]\!]\vartheta \\
[\![\overline{(g_1 \texttt{ \&\& } g_2)}@r]\!]\vartheta \rightsquigarrow [\![\overline{g_1}@r]\!]\vartheta \texttt{ \&\& } [\![\overline{g_2}@r]\!]\vartheta
\end{array}
$$

Figure 6.19: Encoding of regioned guard expressions.

argument of the Voila guard. The type of the guard arguments for a guard G is notated as T_G. Second, $[\![\overline{g}@r]\!]\vartheta$ asserts that the environment might hold the regioned guard. Duplicable guards can always be held by the environment. On the other side, unique guards can only be held by the environment if the guard is not held by oneself. Similar for divisible guards, the environment can only hold a p fraction of a guard if oneself does not hold more than a $1 - p$ fraction of the same guard.

For the encoding of the make_atomic and use_atomic statements we used *action-permitted*$(R,\ g,\ a,\ b)$ to check whether or not the transition on region type R from a to b is permitted by guard g. The code generated by that subroutine is given in figure 6.20. As mentioned in section 5.2, a transition from a to b is permitted by g if the following condition holds:

$$
\begin{array}{c}
a = b \;\vee\; \exists A \in \textsc{Actions}(R).\ (a, b) \in \mathbb{T}_A(g) \\
\text{where } (a, b) \in \mathbb{T}_A(g) \Leftrightarrow \exists \vec{x}_A.\ a = \alpha_A \;\wedge\; b = \beta_A \;\wedge\; c_A \;\wedge\; g_A \leq g
\end{array}
$$

The *action-permitted* subroutine generates this condition as an assert directly where an expression checking $g' \leq g$ is generated by *less-guard*$(g',\ g)$. The code generated by the sub-routine *less-guard* follows directly from the definition of the respective guard algebras: for a product of guards, *less-guard* is split into calls to *less-guard* on primitive guards where the operands belong to the same guard type. For unique and duplicable guards, a guards G is less than or equal to another guard G' if the set of guard arguments for G is

```
action-permitted($R$, $g$, $v_{from}$, $v_{to}$) $\rightsquigarrow$
  assert $v_{from}$ == $v_{to}$
    foreach ( $\vec{t}$ $\vec{x}$ | $b$ | $g' : \alpha$ $\rightarrow$ $\beta$ ) $\in$ Actions($R$):
      || exists $\vec{x}$: $[\![\vec{t}]\!]$ :: $[\![\alpha]\!]\vartheta_0$ == $v_{from}$ && $[\![\beta]\!]\vartheta_0$ == $v_{to}$ &&
           $[\![b]\!]\vartheta_0$ && less-guard($g'$, $g$)
    end

less-guard($pg_{req}$ && $g_{req}$, $pg_{own}$ && $g_{own}$) $\rightsquigarrow$
   less-guard($pg_{req}$, $pg_{own}$) && less-guard($g_{req}$, $g_{own}$)
   where $pg_{req}$ and $pg_{own}$ belong to the same guard types

less-guard($pg_{req}$ && $g_{req}$, $pg_{own}$ && $g_{own}$) $\rightsquigarrow$
  less-guard($pg_{req}$ && $g_{req}$, $g_{own}$)
   where $pg_{req}$ and $pg_{own}$ belong to different guard types

less-guard($pg_{req}$ && $g_{req}$, $pg_{own}$) $\rightsquigarrow$ false
   where $pg_{req}$ and $pg_{own}$ belong to different guard types

less-guard($G_i(\vec{e}_{req})$, $G_i(\vec{e}_{own})$) $\rightsquigarrow$ $[\![\vec{e}_{req}]\!]\vartheta_0$ == $[\![\vec{e}_{own}]\!]\vartheta_0$
less-guard($G_i[p_{req}]$, $G_i[p_{own}]$) $\rightsquigarrow$ $[\![p_{req}]\!]\vartheta_0$ <= $[\![p_{own}]\!]\vartheta_0$
less-guard($G_i(\vec{e}_{req})$, $G_i\langle\vec{e}_{own}\rangle$) $\rightsquigarrow$ $[\![\vec{e}_{req}]\!]\vartheta_0$ in $[\![\vec{e}_{own}]\!]\vartheta_0$
less-guard($G_i\langle\vec{e}_{req}\rangle$, $G_i(\vec{e}_{own})$) $\rightsquigarrow$ $[\![\vec{e}_{req}]\!]\vartheta_0$ subset Set($[\![\vec{e}_{own}]\!]\vartheta_0$)
less-guard($G_i\langle\vec{e}_{req}\rangle$, $G_i\langle\vec{e}_{own}\rangle$) $\rightsquigarrow$ $[\![\vec{e}_{req}]\!]\vartheta_0$ subset $[\![\vec{e}_{own}]\!]\vartheta_0$

less-guard($pg_{req}$, $pg_{own}$) $\rightsquigarrow$ false
   where $pg_{req}$ and $pg_{own}$ belong to different guard types
```

Figure 6.20: Encoding of action checks.

contained in the set of guard arguments for G'. Lastly, for divisible guards, a guard is less or equal if its argument is less or equal.

6.2.8 Stabilization

Recall the TaDA stability definition shown again in figure 6.21. As mentioned in section 2, intuitively, an assertion is stable if the assertion still holds even if the environment performs any allowed state transition. Hence, in our encoding the verifier knowledge can be stabilized by encoding the environment performing potentially any possible state transition. The corresponding Viper code generated by the subroutine *stabilize* is shown in figure 6.22. The basic layout of the code is that for every region type, first the state of all region instances is havoced, then, for each region instance, the

$$\left.\begin{array}{l} \varphi \models P \\ \varphi \models \mathbf{t}_a^\lambda(\vec{e},\, s) \\ \varphi \sim [g]_a \\ (s, s') \in \mathbb{T}_{\mathbf{t}}^\star(g) \\ \varphi \models a \Mapsto \blacklozenge \Rightarrow s' \in dom(\mathcal{A}(a)) \end{array}\right\} \Rightarrow P \sim \mathbf{t}_a^\lambda(\vec{e},\, s')$$

Figure 6.21: Formal definition of stability of an assertion P. Regarding the notation, $\varphi \models Q$ is defined as $\varphi \in [\![Q]\!]$, $\varphi \sim Q$ is defined as $\exists \varphi' \in [\![Q]\!].\ \varphi \bullet \varphi'$, and $Q_1 \sim Q_2$ is defined accordingly where $[\![\cdot]\!]$ is the assertion semantics of TaDA.

new state is constrained so that only the stable knowledge about the state remains. Next, we discuss in more detail how the new state of each region instance is constrained.

First of all, we can only constrain the new state of a region instance if we are know of the region instance's existence in the first place. The corresponding condition in the encoding is that permission to R`_interp` is held. If the environment has performed potentially any state transition, then from the stability definition we still know two properties: first, if the diamond resource is held, then the environment can only change the state to a value from the domain of the atomicity context entry. In the encoding, this corresponds to the new state being in the rely context R`_actxt` if permission to the `diamond` field is held. Second, the new state must be reachable from the old state with an action that the environment is permitted to perform. Recall that the environment is permitted to perform an action ($\vec{t}\ \vec{x}\ \mid\ c\ \mid\ g : \alpha\ \rightarrow\ \beta$) if there exists $\vec{x}$ such that the new state is equal to α, the old state is equal to β, the condition c holds, and the environment might hold guard g for the respective region instance. The trivial action where the new state is equal to the old state is always permitted. The encoding expresses this property straightforwardly where, as mentioned before, the check whether or not the environment may hold a guard g for a region instance with identifier r is generated by $[\![\overline{g}@r]\!]\vartheta_0$.

6.2.9 Interference Inference

As shown in section 6.2.3, the encoding of the atomic$\{a\}$ statement uses the *infer-interference-context* subroutine to infer an upper bound on the interference caused by the environment. Inferring this upper bound is closely related to our encoding of stabilization because both deal with the set of all allowed state transitions performed by the environment. The difference from the stabilization encoding is that all possible new states are stored in a set, more specifically R`_ictxt` which represents the interference context.

```
$havoc\text{-}state\text{-}all_l(R\_\texttt{state})$ $\leadsto$
  exhale forall $\vec{x}$: $[\![\vec{t}]\!]$ :: acc[$l$]($R$_interp($\vec{x}$))
  inhale forall $\vec{x}$: $[\![\vec{t}]\!]$ :: acc[$l$]($R$_interp($\vec{x}$))

  where $(\vec{t}\ \vec{x}) = \textsc{Formal}_{\textsf{in}}(R)$

$havoc\text{-}state_l(R\_\texttt{state}(\vec{v}))$ $\leadsto$
  exhale acc[$l$]($R$_interp($\vec{v}$))
  inhale acc[$l$]($R$_interp($\vec{v}$))

stabilize $\leadsto$
  label $\bullet preStabilize$
  foreach $R(\vec{t}\ \vec{x}) \in \textsc{Formal}_{\textsf{in}}(\textsc{Regions})$:
    $havoc\text{-}state\text{-}all_{preStabilize}(R\_\texttt{state})$
    assume forall $\vec{x}$: $[\![\vec{t}]\!]$ ::
      none < perm[$l$]($R$_interp($\vec{x}$)) ==>
      (
       none < perm($rid(\vec{x})$.diamond) ==>
       $R$_state($\vec{x}$) in $R$_actxt($\vec{x}$)
      ) && (
       $R$_state($\vec{x}$) == old[$l$]($R$_state($\vec{x}$))
       foreach $(\ \vec{t}\ \vec{x}\ \mid\ b\ \mid\ g:\alpha\ \rightarrow\ \beta\ ) \in \textsc{Actions}(R)$:
         || exists $\vec{x}$: $[\![\vec{t}]\!]$ :: $[\![\alpha]\!]\vartheta_0$ == old[$l$]($R$_state($\vec{x}$)) &&
              $[\![\beta]\!]\vartheta_0$ == $R$_state($\vec{x}$) && $[\![b]\!]\vartheta_0$ && $[\![g@rid(\vec{x})]\!]\vartheta_0$
       end
      )
  end
```

Figure 6.22: Encoding of state stabilization.

Figure 6.23 shows the encoding of *infer-interference-context*: first, the permissions to R_`ifp` are inhaled. Next, the stabilization constraint from figure 6.22 is emitted where the new state R_`state` is replaces with a variable m that is contained in R_`ictxt` if and only if the modified stabilization constraint holds. Afterwards, every state is havoced and assumed to be in its corresponding interference context.

Besides inferring the interference context in atomic$\{a\}$, we also need to infer the interference context of regions that come from an opened region interpretation. Let the state of a region be expressed as $\varphi(\vec{s})$ where the arguments $\vec{s}$ are the states of all regions occurring in the opened region interpretation. Then, because we know $\varphi(m_1, \ldots, m_n) \in S'$ where S' is the interference context of the opened region, the interference contexts $S_1, \ldots, S_n$

```
infer-interference-context ⇝
 label •preInfer
 foreach $R(\vec{t}\ \vec{x}) \in$ FORMAL$_{\text{in}}$(REGIONS):
   inhale forall $\vec{x}$: $[\![\vec{t}]\!]$ :: acc($R$_ifp($\vec{x}$))
   assume forall $m$: $[\![T_R]\!]$, $\vec{x}$: $[\![\vec{t}]\!]$ ::
     none < perm[$l$]($R$_interp($\vec{x}$)) ==>
     $m$ in $R$_ictxt($\vec{x}$) ==
     (
      none < perm($rid(\vec{x})$.diamond) ==>
      $m$ in $R$_actxt($\vec{x}$)
     ) && (
      $R$_state($\vec{x}$) == old[$l$]($R$_state($\vec{x}$))
      foreach ( $\vec{t}\ \vec{x}$ | $b$ | $g:\alpha \rightarrow \beta$ ) $\in$ ACTIONS($R$):
        || exists $\vec{x}$: $[\![\vec{t}]\!]$ :: $[\![\alpha]\!]\vartheta_0$ == old[$l$]($R$_state($\vec{x}$)) &&
             $[\![\beta]\!]\vartheta_0$ == $m$ && $[\![b]\!]\vartheta_0$ && $[\![\overline{g}@rid(\vec{x})]\!]\vartheta_0$
      end
     )
   havoc-state-all$_{preInfer}$($R$_state)
   assume forall $\vec{x}$: $[\![\vec{t}]\!]$ ::
     none < perm[$l$]($R$_interp($\vec{x}$)) ==>
     $R$_state($\vec{x}$) in $R$_ictxt($\vec{x}$)
 end
```

Figure 6.23: Encoding of interference inference.

of all regions occurring in the region interpretation can be defined by $m_1 \in S_1 \wedge \ldots \wedge m_n \in S_n \Leftrightarrow \varphi(m_1, \ldots, m_n) \in S'$. This condition is encoded directly by the *link-interference-context* subroutine as shown in figure 6.24: first, footprints R_ifp are havoced for the regions inside the region interpretation. Then the interference context is constrained accordingly. The function REGIONS(·) returns all region assertions from a Voila assertion.

6.2.10 Well-definedness Checks

Recall that in a well-defined Voila program loop invariants, region interpretations, as well as pre- and postconditions have to be stable. Furthermore, action declarations have to be transitively closed. Figure 6.25 shows how we check whether or not assertions are stable. The check is straightforward: first, inhale the assertion, then stabilize, and finally assert the assertion. For loop invariants as well as pre- and postconditions the check depends on the constrained atomicity context, hence, for example for preconditions, the check is generated instead of inhaling the precondition. Region interpretations have to be stable under any atomicity context, so the check is emitted

```
havoc-ictxt$_l$($R$_ictxt($\vec{v}$))  $\rightsquigarrow$
  exhale acc[l]($R$_ifp($\vec{v}$))
  inhale acc[l]($R$_ifp($\vec{v}$))

link-interference-context($R$_ictxt($\vec{v}$))  $\rightsquigarrow$
  label $\bullet$preLink
  foreach $R'(\vec{e}) \in$ REGIONS(INTERP($R$)):
    havoc-ictxt$_{preLink}$($R'$_ictxt($[\![e]\!]\vartheta'$))
  end
  assume forall
    foreach $R'(\vec{e}) \in$ REGIONS(INTERP($R$)):
      $m_{R'}$: $T_{R'}$
    end
    ::
    $[\![$STATE($R$)$]\!]\vartheta'[y \mapsto m_{R'} \mid \vartheta'(y) = R'$_state($\vec{e}$)$]$ in $R$_ictxt($\vec{v}$) ==
    foreach $R'(\vec{e}) \in$ REGIONS(INTERP($R$)):
      && $m_{R'}$ in $R'$_ictxt($[\![e]\!]\vartheta'$)
    end

  where $(\vec{t}\ \ \vec{x})$ = FORMAL$_{in}$($R$) and $\vartheta' = \vartheta_0[\vec{x} \mapsto \vec{e}]$
```

Figure 6.24: Encoding of interference inference for hidden regions.

```
stability-check($w$)  $\rightsquigarrow$
  inhale $w$
  stabilize
  assert $w$
```

Figure 6.25: Encoding of the stability check.

into an extra method with an empty atomicity context. Being stable with an empty atomicity context implies being stable with any atomicity context since the atomicity context only provides restrictions on the interference.

Regarding the transitivity of action, recall from section 5.2 that the state transition system $\mathbb{T}^{\star}_{R}(g)$ in Voila is defined as follows:

$$(a, b) \in \mathbb{T}^{\star}_{R}(g) :\Longleftrightarrow a = b \ \lor\ \exists A \in \textsc{Actions}(R).\ (a, b) \in \mathbb{T}_{A}(g)$$

The benefit of this definition is that the verifier knowledge can be stabilized without having to infer the transitively closed state transition system. Instead, the action state transition systems can be used directly. However, for

```
transitivity-check($R$) $\leadsto$
  var $\vec{gz}$ := ARGS($g$)
  var $x$, $y$, $z$: $T_R$

  assume action-permitted($R$, $g$, $x$, $y$)
  assume action-permitted($R$, $g$, $y$, $z$)
  assert action-permitted($R$, $g$, $x$, $z$)

   where $g$ is
    foreach $m$ $G(\vec{t}\ \vec{x}) \in$ GUARDS($R$):
      && as-guard($m, G, \vec{t}$)
    end

as-guard(unique, $G$, $\vec{t}$) $\leadsto$
  $G_u\langle \bullet x \rangle$ where $x$ is of type Set[$\vec{t}$]
as-guard(duplicable, $G$, $\vec{t}$) $\leadsto$
  $G_u\langle \bullet x \rangle$ where $x$ is of type Set[$\vec{t}$]
as-guard(divisible, $G$, $\vec{t}$) $\leadsto$
  $G[\bullet x]$ where $x$ is of type Perm
```

Figure 6.26: Encoding of the transitivity check.

this definition to be sound, we require the following condition to hold:

$$(a,b) \in \mathbb{T}_{A_1}(g) \ \wedge\ (b,c) \in \mathbb{T}_{A_2}(g) \\ \implies a = c \ \vee\ \exists A \in \textsc{Actions}(R).\ (a,c) \in \mathbb{T}_A(g)$$

Intuitively, if some state b can be reached from some state a with an action A_1, and if then some state c can be reached from b with an action A_2, then there has to be an action that permits to reach c from a with the same amount of guards. We refer to this condition as action transitivity. Figure 6.26 shows the code generated to check action transitivity for a region. An arbitrary guard expression is encoded by taking every guard declared for a region and instantiating it with fresh variables. The rest is a straightforward encoding of the transitively closed actions condition.

6.3 Differences from the Prototype

As mentioned in the introduction, our thesis was build upon a Voila prototype. In our work, we added support for interference rely-guarantees, atomicity contexts, and region levels. Accordingly, the Voila assertion language for sets and tuples was improved to aid specifications. Furthermore, we added the encoding for unique and divisible guards, as well as products and sets of guards. Moreover, the well-definedness checks were developed and added to the verifier as well. All of the rule statements encodings were unsound, such as MakeAtomic whose Voila encoding did not follow the strict pre- and postcondition limitations of the TaDA rule. Those encodings were revised too. Lastly, we reduced the amount of generated stabilizations. Regarding the implementation of the prototype, we restructured the implementation to increase modularity.

Chapter 7

Soundness

In this chapter we give a detailed proof sketch for the soundness of a subset of our encoding. Furthermore, we discuss how the proof can be extended to cover the complete Voila encoding.

7.1 Voila Subset

The Voila subset contains the encoding of all statements with the exception of make_atomic, open_region, and update_region, but with the encoding of use_atomic. This selection allows us to focus on Voila components such as stabilization and interference rely-guarantee inference, without worrying about tracking resources or the use of R_actxt. Furthermore, we only cover unique and duplicable guards with no arguments, and assume that interference clauses do not reference themselves. The set of used TaDA rules is listed in appendix A

7.2 Proof Sketch

We split the proof sketch into fife parts. First, we determine invariants on the Viper programs generated by our encoding. Second, For Viper programs satisfying those invariants we define a mapping from Viper judgments on encoded Voila statements to TaDA judgments on a statement corresponding to the source Voila statement. Intuitively, this mapping expresses the current position in a TaDA proof. Third, we show that the mapping applied to the encoding of Voila statements results in derivable TaDA triples provided that the generated Viper program successfully verifies. Forth, we show for Voila methods that if the mapping is applied to the encoded method body, the resulting TaDA triple corresponds to the specification of the Voila method. Lastly, we show that an entailment automatically deduced by Viper corre-

$$\begin{aligned}
\mathbb{E} &::= \mathsf{field}(x.F;l) \mid R\_\texttt{state}(a,\lambda,\vec{x};l) \mid R\_\texttt{ictxt}(a,\lambda,\vec{x};l) \mid \ldots \\
\mathbb{A} &::= \mathbb{B} \mid \mathbb{A}_1 \;\texttt{\&\&}\; \mathbb{A}_2 \mid \mathbb{B} \Rightarrow \mathsf{acc}(\mathbb{H}) \mid \mathsf{acc}(\mathbb{P}) \\
\mathbb{H} &::= \mathsf{field}(x.F;l) \mid R\_\texttt{interp}(a,\lambda,\vec{x};l) \mid G(a;l) \\
\mathbb{P} &::= R\_\texttt{ifp}(a,\lambda,\vec{x};l)
\end{aligned}$$

Figure 7.1: Normalized Viper assertions.

sponds to an application of the Consequence rule in TaDA. This part is necessary to guarantee that Viper does not deduce unsound knowledge.

7.2.1 Invariants on generated Viper Programs

We represent the verifier knowledge, which includes pre- and postconditions of encoded statements, as normalized Viper assertions. The syntax for normalized Viper assertions is given in Figure 7.1: different from normal Viper assertions, we notate fields, predicates, and functions together with their Viper label at which they are evaluated. For example, the assertion $\mathsf{old[l]}(x.f) = 42$ is represented as $\mathsf{field}(x.f;l) = 42$. Those labels do not have to be explicitly defined in the Viper program. Furthermore, we write the current label of an assertion P as $\mathsf{label}(P)$.

Some invariant on the generated Viper programs are already defined by the normalized assertion syntax: first, access predicate always have write permission because without divisible guards the encoding never generates other permission amounts. Second, permissions to $R\_\texttt{ifp}$ are never held conditionally since the same holds for the encoding. The remaining required invariant will be defined in section 7.2.3 in the form of lemmas.

7.2.2 Viper Specification Mapping

Before we show the mapping of Viper judgments on encoded Voila statements, we first discuss the mapping of normalized Viper assertions to TaDA assertions as given in figure 7.2. The mapping of a Viper assertion P depends on the current label $\mathsf{label}(P)$ of that Viper assertion. Intuitively, the label indicates if a Viper resource constrains the TaDA assertions of the current judgment or just resources from previous judgments. For fields, an access to a field $\mathsf{field}(x.F;l')$ is encoded as a variable $F_x^{l'}$. The meaning of that variable is then provided by the field permission: if the field permission is held in the current state then the variable mapped for the field access is bound by the points-to predicate in the TaDA assertion. The $F_x^l = F(x,l)$ constraint is added to account for aliasing of the reference. Otherwise, if the field permission is hold in an older state, then only the aliasing constraint is mapped. Similar for $R\_\texttt{state}$ and $R\_\texttt{interp}$: the variable mapped for $R\_\texttt{state}$ is only the state of a region assertion if permission to R`_interp` is held in the current state. The same for guards. On the other side, R`_ictxt` and R`_ifp` are not mapped to a TaDA assertion but to the interference rely-guarantee of the TaDA judgment. Finally, the mapping for normal expressions $\mathbb{E}$ is straightforward and not shown.

Next, the mapping for the complete Viper judgment on encoded Voila statements is shown in figure 7.3. We distinguish between judgments on encoded atomic and non-atomic statements. Note that we omit the encoding context since for statements the encoding context is always the initial encoding context as defined in section 6.2. For non-atomic statements the pre- and postcondition of the TaDA judgment are the mapped pre- and postcondition of the Viper judgment. The statement of the judgment is $\lfloor\ h\ \rfloor$, a non-atomic Voila statement h mapped to a TaDA statement straightforwardly, i.e. ignoring rule statements and atomic, removing invariants from while loops, and mapping all occurring Voila expressions to their direct TaDA counterpart. More interesting are the level, atomicity context, and functional environment of the judgment. The judgment level is minimum value the variable `level` can have in the Viper assertion P. Regarding the notation, $\mathbb{D}_P(e)$ returns all values the expression e might have in assertion P. For example, for $Q \equiv 3 < x$ `&&` $x < 7$ we have $\mathbb{D}_Q(x) = \{4, 5, 6\}$. The ∂ variable represents the lower bound of the region levels occurring in the atomicity context. Accordingly, $\mathcal{A}_\partial$ is defined as the atomicity contexts that contains entries on all regions with a level of at least ∂. The TaDA judgment is then proven forall atomicity contexts that are a subset of $\mathcal{A}_\partial$. Note, that because we do not deal with make_atomic and update_region, we can define the domain and image of each atomicity context entry as the universe, i.e. the set containing all possible values. Lastly, the functional environment is the set of all Voila method specifications mapped to TaDA. The mapping of Voila method specifications to TaDA specifications follows directly from the Voila language syntax defined in section 5.2. For simplicity we annotate the interference clause of an atomic Voila method equivalent to the interference rely-guarantee of a TaDA method specification.

For an encoded atomic statement the mapping $(\!|\ \cdot\ |\!)$ is similar to the mapping for an encoded non-atomic statement. However, the interference rely-guarantee of the atomic TaDA triple also has to be specified. For simplicity, we assume that we have a list notated as $\textsc{RegionInstances}_P$. of all region instances for whose region interpretation we have a positive amount of permission in P. Note that due to our invariants on the generated Viper programs we know that a positive permission amount to a region interpretation is always at least write permission. The interference rely-guarantee for the TaDA judgment is then defined as the sequence of $s_i \in S_i$ where s_i is the variable representing the region state of the i-th region according to $\textsc{RegionInstances}_P$, and S_i is the set of all possible values in R`_ictxt` for

$$\begin{array}{lll}
\llparenthesis\ P\ \rrparenthesis & \rightsquigarrow & \llparenthesis\ P\ \rrparenthesis_{\mathsf{label}(P)} \\
 & & \\
\llparenthesis\ \mathsf{field}(x.F;l')\ \rrparenthesis_l & \rightsquigarrow & F_x^{l'} \\
\llparenthesis\ \mathsf{acc}(\mathsf{field}(x.F;l))\ \rrparenthesis_l & \rightsquigarrow & x.F \mapsto F_x^l \ * \ F_x^l = F(x,l) \\
\llparenthesis\ \mathsf{acc}(\mathsf{field}(x.F;l'))\ \rrparenthesis_l & \rightsquigarrow & F_x^{l'} = F(x,l') \quad \texttt{where } l \neq l' \\
 & & \\
\llparenthesis\ R\_\texttt{state}(a,\lambda,\vec{x};l')\ \rrparenthesis_l & \rightsquigarrow & R^{l'}_{a,\lambda,\vec{x}} \\
\llparenthesis\ \mathsf{acc}(R\_\texttt{interp}(a,\lambda,\vec{x};l))\ \rrparenthesis_l & \rightsquigarrow & R_a^\lambda(\vec{x}, R^l_{\vec{x}}) \ * \ R^l_{a,\lambda,\vec{x}} = R(a,\lambda,\vec{x},l) \\
\llparenthesis\ \mathsf{acc}(R\_\texttt{interp}(a,\lambda,\vec{x};l'))\ \rrparenthesis_l & \rightsquigarrow & R^{l'}_{a,\lambda,\vec{x}} = R(a,\lambda,\vec{x},l') \quad \texttt{where } l \neq l' \\
 & & \\
\llparenthesis\ G(a;l)\ \rrparenthesis_l & \rightsquigarrow & [G]_a
\end{array}$$

Figure 7.2: Mapping from normalized Viper assertions to TaDA assertions.

the same region instance.

7.2.3 Lemmas

Before we continue with the soundness of the statement encodings we first show a set a lemmas that we will require in the following steps.

Lemma 7.1 *$\vdash_V \{P\}\ [\![\ s\ ]\!]\ \{Q\}$ implies that $\mathbb{D}_P(\texttt{level}) = \mathbb{D}_Q(\texttt{level})$ and $\mathbb{D}_P(\texttt{atomcitiy\_level}) = \mathbb{D}_Q(\texttt{atomicity\_level})$ hold.*

Proof Proof by induction over s: if any of those two variables is changed in the encoding, then the value is written back at the end of the encoding. □

Lemma 7.2 *$\llparenthesis\ P(v)\ \rrparenthesis \preceq \exists s \in S.\ \llparenthesis\ P(s)\ \rrparenthesis$ and $\exists s \in S.\ \llparenthesis\ P(s)\ \rrparenthesis \preceq \llparenthesis\ P(v)\ \rrparenthesis$ where $S = \mathbb{D}_{P(v)}(v)$.*

Proof Proof by induction over P, follows directly from soundness of Viper.□

Lemma 7.3 *$\llparenthesis\ P[[\![\ e\ ]\!]/x]\ \rrparenthesis \preceq \llparenthesis\ P\ \rrparenthesis[\lfloor\ e\ \rfloor/x]$ and $\llparenthesis\ P\ \rrparenthesis[\lfloor\ e\ \rfloor/x] \preceq \llparenthesis\ P[[\![\ e\ ]\!]/x]\ \rrparenthesis$.*

Proof Can be proven in two steps. First, show $\llparenthesis\ P[t/x]\ \rrparenthesis = \llparenthesis\ P\ \rrparenthesis[\llparenthesis\ t\ \rrparenthesis/x]$ via induction on P. Afterwards, show $\llparenthesis\ [\![\ e\ ]\!]\ \rrparenthesis = \lfloor\ e\ \rfloor$ modulo variable renaming by induction on e. □

Lemma 7.4 *$\vdash_V \{P\}stabilize\{Q\}$ implies $\llparenthesis\ Q\ \rrparenthesis$ is stable and $\llparenthesis\ P\ \rrparenthesis \preceq \llparenthesis\ Q\ \rrparenthesis$ holds.*

Proof The second statement follows directly from the reflexivity of the generated assertion, i.e. that the old state is always a possible new state. In other words, the generated code only changes the regions states and for every region instance $\mathbb{D}_P(R\_\texttt{state}(\vec{x})) \subseteq \mathbb{D}_Q(R\_\texttt{state}(\vec{x}))$ holds.

The proof of $\vdash_V \{P\}stabilize\{Q\}$ implying $\llparenthesis\ Q\ \rrparenthesis$ being stable is a bit more involved. We first prove three corollaries. We will omit the mapping of

$$\mathcal{A}_\partial := \{\ r : \mathbb{U}\ \rightarrow\ \mathbb{U} \mid \texttt{level of } r \geq \partial\ \}$$

$$\begin{array}{l}
(\!|\ \vdash_V \{P\}\ [\![\ h\ ]\!]\ \{Q\}\ |\!) \rightsquigarrow \\
\quad \forall \mathcal{A} \subseteq \mathcal{A}_\partial.\ \Gamma;\lambda;\mathcal{A} \vdash \{\ (\!|\ P\ |\!)\ \}\ \lfloor\ h\ \rfloor\ \{\ (\!|\ Q\ |\!)\ \} \\
\quad\ \texttt{where} \\
\quad\quad \lambda = \mathsf{min}\ \mathbb{D}_P(\texttt{level}) \\
\quad\quad \partial = \mathsf{min}\ \mathbb{D}_P(\texttt{atomicity\_level}) \\
\quad\quad \Gamma = \{\ \lfloor\ T\ \rfloor \mid T\ \texttt{is a Voila specification}\ \}
\end{array}$$

$$\begin{array}{l}
(\!|\ \vdash_V \{P\}\ [\![\ a\ ]\!]\ \{Q\}\ |\!) \rightsquigarrow \\
\quad \forall \mathcal{A} \subseteq \mathcal{A}_\partial.\ \Gamma;\lambda;\mathcal{A} \vdash \mathbb{W}\vec{s} \in \vec{S}.\ \langle\ (\!|\ P\ |\!)\ \rangle\ \lfloor\ h\ \rfloor\ \langle\ (\!|\ Q\ |\!)\ \rangle \\
\quad\ \texttt{where} \\
\quad\quad \lambda = \mathsf{min}\ \mathbb{D}_P(\texttt{level}) \\
\quad\quad \partial = \mathsf{min}\ \mathbb{D}_P(\texttt{atomicity\_level}) \\
\quad\quad \Gamma = \{\ \lfloor\ T\ \rfloor \mid T\ \texttt{is a Voila specification}\ \} \\
\quad\quad s_i = R_{\vec{x}}^{\mathsf{label}(P)} \\
\quad\quad S_i = \bigcup \mathbb{D}_P(R\texttt{\_ictxt}(\vec{x};\mathsf{label}(P))) \\
\quad\quad\quad \texttt{where}\quad (R,\vec{x}) = \textsc{RegionInstances}_P(i) \\
\quad\quad\quad\quad\quad\quad\ \mathsf{true} \in \mathbb{D}_P(\mathsf{perm}(R\texttt{\_interp}(\vec{x})) > \mathsf{none})
\end{array}$$

$$\begin{array}{l}
\lfloor\ \{P\}\ M_H(\vec{z})\ \mathsf{returns}\ (\vec{r})\ \{Q\}\ \rfloor \rightsquigarrow \\
\quad \forall \mathcal{A} \subseteq \mathcal{A}_\lambda.\ \lambda;\mathcal{A} \vdash \{\ \lfloor\ P\ \rfloor\ \}\ M(\vec{z})\ \mathsf{returns}\ \{\ \lfloor\ Q\ \rfloor\ \} \\
\quad\quad \texttt{where}\quad \lambda = 1 + \mathsf{max}\ \textsc{Levels}(P)
\end{array}$$

$$\begin{array}{l}
\lfloor\ \mathbb{W}\vec{s} \in \vec{S}.\ \langle P\rangle\ M_A(\vec{z})\ \mathsf{returns}\ (\vec{r})\ \langle Q\rangle\ \rfloor \rightsquigarrow \\
\quad \forall \mathcal{A} \subseteq \mathcal{A}_\lambda.\ \lambda;\mathcal{A} \vdash \mathbb{W}\vec{s} \in \lfloor\ \vec{S}\ \rfloor.\ \langle\ \lfloor\ P\ \rfloor\ \rangle\ M(\vec{z})\ \mathsf{returns}\ \langle\ \lfloor\ Q\ \rfloor\ \rangle \\
\quad\quad \texttt{where}\quad \lambda = 1 + \mathsf{max}\ \textsc{Levels}(P)
\end{array}$$

Figure 7.3: Mapping from Viper judgments on encoded Voila statements for normalize Viper pre- and postconditions to TaDA judgments.

expressions between Voila and TaDA since they behave isomorph to each other.

Corollary 7.5 *A TaDA assertion P is stable if* $\forall s' \in \Phi_r^A(P).\ P \sim R_r^\lambda(\vec{z}, s')$ *where* $\Phi_r^A(P) := \{\ s' \mid \exists \vec{x}_A.\ (P\ \wedge\ R_r^\lambda(\vec{z}, s)\ \sim\ [g_A]_r),\ s = \alpha_A,\ s' = \beta_A,\ c_A\ \}$

Proof For our Voila subset a TaDA assertion P is stable if $(P\ \wedge\ R_r^\lambda(\vec{z}, s)\ \sim\ [g]_r)$ and $(s, s') \in \mathbb{T}_R^\star(g)$ imply $P \sim R_r^\lambda(\vec{z}, s')$. In other words, if a guard $[g]_a$ might be held by the environment when the state of some region is s, and if furthermore a transition from s to s' with g is permitted by the state transition system, then P is not allowed to contradict with the same region being in state s'. The corollary follows directly from this stability definition and the state transition system definition given as follows:

$$(s,s') \in \mathbb{T}^{\star}_{R}(g) \Leftrightarrow s = s' \ \vee\ \exists A.\ \exists \vec{x}_A.\ s = \alpha_A \ \wedge\ s' = \beta_A \ \wedge\ c_A \ \wedge\ g_A \leq g$$

where g_A, α_A, β_A, and c_A are the corresponding action guard, action source, action target, and action condition, respectively, that can depend on the action variables $\vec{x}_A$. □

Corollary 7.6 *For a Viper assertion P, $s \in \mathbb{D}_P(R\_\mathtt{state}(r,\lambda,\vec{z};\mathsf{label}(P)))$ implies $(\!|\ P\ |\!) \sim R^{\lambda}_{r}(\vec{z},s)$*

Proof Follows directly from the mapping shown in figure 7.3. □

Corollary 7.7 *For a Viper assertion P the property $(\!|\ P\ |\!) \sim [g]_r$ implies* $\mathsf{true} \in \mathbb{D}_P([\![\ \overline{g}@r\ ]\!])$

Proof Proof by contradiction: $\mathsf{true} \notin \mathbb{D}_P([\![\ \overline{g}@r\ ]\!])$ implies that Viper can deduce from P that $[\![\ \overline{g}@r\ ]\!]$ does not hold. Next, we proceed with a case distinction on the type of the guard g. If g is a duplicable guard then $[\![\ \overline{g}@r\ ]\!]$ always holds, hence a contradiction. Otherwise, if g is a unique guard then the unsatisfiability of $[\![\ \overline{g}@r\ ]\!]$ contradicts with the assumption $(\!|\ P\ |\!) \sim [g]_r$.□

From the first two corollaries we know that $(\!|\ Q\ |\!)$ is stable if $\Phi^{A}_{r}((\!|\ Q\ |\!)) \subseteq \mathbb{D}_Q(R\_\mathtt{state}(r,\lambda,\vec{z};\mathsf{label}(Q)))$ holds. We show this property by proving $\Phi^{A}_{r}((\!|\ Q\ |\!)) \subseteq \Phi^{A}_{r}((\!|\ P\ |\!))$ and $\Phi^{A}_{r}((\!|\ P\ |\!)) \subseteq \mathbb{D}_Q(R\_\mathtt{state}(r,\lambda,\vec{z};\mathsf{label}(Q)))$ separately.

First, we prove $\Phi^{A}_{r}((\!|\ Q\ |\!)) \subseteq \Phi^{A}_{r}((\!|\ P\ |\!))$ through the definition of Φ^{A}_{r} by assuming $\exists \vec{x}_A.\ ((\!|\ Q\ |\!) \ \wedge\ R^{\lambda}_{r}(\vec{z},s) \ \sim\ [g_A]_r)$, $s = \alpha_A$, $s' = \beta_A$, c_A and then showing $\exists s'', \vec{x}_A.\ ((\!|\ P\ |\!) \ \wedge\ R^{\lambda}_{r}(\vec{z},s'') \ \sim\ [g_A]_r)$, $s'' = \alpha_A$, $s' = \beta_A$, c_A. The right side follows directly by choosing s'' as s. Note that $((\!|\ Q\ |\!) \ \wedge\ R^{\lambda}_{r}(\vec{z},s) \ \sim\ [g_A]_r)$ implies $((\!|\ P\ |\!) \ \wedge\ R^{\lambda}_{r}(\vec{z},s) \ \sim\ [g_A]_r)$ because held guards do not change when stabilizing.

Next, we show $\Phi^{A}_{r}((\!|\ P\ |\!)) \subseteq \mathbb{D}_{(\!|\ Q\ |\!)}(R\_\mathtt{state}(r,\lambda,\vec{z};\mathsf{label}(Q)))$ through the definition of Φ^{A}_{r} by assuming $\exists \vec{x}_A.\ ((\!|\ P\ |\!) \ \wedge\ R^{\lambda}_{r}(\vec{z},s) \ \sim\ [g_A]_r)$, $s = \alpha_A$, $s' = \beta_A$, c_A and then showing $s' \in \mathbb{D}_{(\!|\ Q\ |\!)}(R\_\mathtt{state}(r,\lambda,\vec{z};\mathsf{label}(Q)))$. Let l be $\mathsf{label}(Q)$. From $((\!|\ P\ |\!) \ \wedge\ R^{\lambda}_{r}(\vec{z},s) \ \sim\ [g_A]_r)$ we can derive $(\!|\ P\ \&\&\ R^{l}_{r,\lambda,\vec{z}}\ |\!) \sim [g_A]_r$. Together with corollary 7.7 this implies $\mathsf{true} \in \mathbb{D}_{P\ \&\&\ R^{l}_{r,\lambda,\vec{z}}}([\![\ \overline{g_A}@r\ ]\!])$. Combined with the previous assumption we now have that the assertion $\exists \vec{x}_A.\ s = [\![\ \alpha_A\ ]\!] \ \wedge\ s' = [\![\ \beta_A\ ]\!] \ \wedge\ [\![\ c_A\ ]\!] \ \wedge\ [\![\ \overline{g_A}@r\ ]\!]$ holds. This assertion corresponds to a constrain generated by *stabilize*, hence s' is a possible new state and thus contained in $\mathbb{D}_{(\!|\ Q\ |\!)}(R\_\mathtt{state}(r,\lambda,\vec{z};\mathsf{label}(Q)))$. □

Lemma 7.8 $\vdash_V$ *$\{P\}$infer-interference-context$\{Q\}$ implies $(\!|\ Q\ |\!)$ is stable and $(\!|\ P\ |\!) \preceq (\!|\ Q\ |\!)$ holds . Furthermore, for every existing region instance in Q the property $\mathbb{D}_Q(R\_\mathtt{state}(\vec{x})) \subseteq \bigcup \mathbb{D}_Q(R\_\mathtt{ictxt}(\vec{x}))$ holds.*

$$\begin{aligned}W(s) := \forall P, Q.\ &\vdash_V \{P\}\ [\![\ s\ ]\!]\ \{Q\}\\ &\quad\wedge\ P, Q \text{ satisfy our invariants } \wedge\ (\!|\ P\ |\!) \text{ is stable}\\ &\quad\Longrightarrow (\!|\ \vdash_V \{P\}\ [\![\ s\ ]\!]\ \{Q\}\ |\!) \text{ is derivable in TaDA}\\ &\qquad\qquad\wedge\ (\!|\ Q\ |\!) \text{ is stable if } s \text{ is a non-atomic statement}\end{aligned}$$

Figure 7.4: Induction predicate.

Proof The lemma follows directly from the encoding of *infer-interference-context* and lemma 7.4. The new region state is constrained logically equivalent to the constrain generated by *stabilize*. Furthermore, the last assertion that $R\_\mathtt{state}(\vec{x})$ is contained in $R\_\mathtt{ictxt}(\vec{x})$ guarantees that $\mathbb{D}_Q(R\_\mathtt{state}(\vec{x})) \subseteq \bigcup \mathbb{D}_Q(R\_\mathtt{ictxt}(\vec{x}))$ holds. □

Lemma 7.9 $\vdash_V$ *$\{P\}$action-permitted$(R,\ g,\ a,\ b)\{Q\}$ implies $((\!|\ a\ |\!), (\!|\ b\ |\!)) \in \mathbb{T}^{\star}_{g}(R)$*

Proof This lemma follows directly from the definition of the state transition system, from the assumption that actions are transitively closed, from the encoding of *action-permitted*, and from the property *less-guard*$(g,\ g') \Rightarrow g \leq g'$ which trivially holds for unique and duplicable guards without arguments. □

7.2.4 Soundness of Statement Encoding

We show with induction over the Voila statement s that our induction predicate W shown in figure 7.4 holds for all statements. We proceed with a case distinction on the Voila statements s.

Case $s \equiv h_1; h_2$ Let $\mathcal{A}_\partial$, $\mathcal{A}$, Γ, and λ be defined as specified by the mapping in figure 7.3. From the viper semantics and from the induction hypothesis, we learn that there exists some R such that $\forall \mathcal{A} \subseteq \mathcal{A}_\partial.\ \Gamma; \lambda; \mathcal{A} \vdash \{\ (\!|\ P\ |\!)\ \}\ \lfloor\ h_1\ \rfloor\ \{\ (\!|\ R\ |\!)\ \}$ and $\forall \mathcal{A} \subseteq \mathcal{A}_{\partial'}.\ \Gamma; \lambda'; \mathcal{A} \vdash \{\ (\!|\ R\ |\!)\ \}\ \lfloor\ h_2\ \rfloor\ \{\ (\!|\ Q\ |\!)\ \}$ hold. Furthermore, from lemma 7.1, we know that $\lambda = \lambda'$ and $\partial = \partial'$. Finally, we know that P, Q, and R are all stable. Hence, the mapped triple can be directly derived through Seqas shown below.

$$\dfrac{\dfrac{}{\Gamma; \lambda; \mathcal{A} \vdash \{\ (\!|\ P\ |\!)\ \}\ \lfloor\ h_1\ \rfloor\ \{\ (\!|\ R\ |\!)\ \}}\,\text{IH} \qquad \dfrac{}{\Gamma; \lambda; \mathcal{A} \vdash \{\ (\!|\ R\ |\!)\ \}\ \lfloor\ h_2\ \rfloor\ \{\ (\!|\ Q\ |\!)\ \}}\,\text{IH}}{\Gamma; \lambda; \mathcal{A} \vdash \{\ (\!|\ P\ |\!)\ \}\ \lfloor\ h_1; h_2\ \rfloor\ \{\ (\!|\ Q\ |\!)\ \}}\,\textsc{Seq}$$

Case $s \equiv \mathsf{if}(b)\ \{h_1\}\ \mathsf{else}\ \{h_2\}$ Let $\mathcal{A}_\partial$, $\mathcal{A}$, Γ, and λ be defined as specified by the mapping in figure 7.3. From the viper semantics and from the induction hypothesis, we learn that $\forall \mathcal{A} \subseteq \mathcal{A}_\partial.\ \Gamma; \lambda; \mathcal{A} \vdash \{\ (\!|\ P\ \&\&\ [\![\ b\ ]\!]\ |\!)\ \}\ \lfloor\ h_1\ \rfloor\ \{\ (\!|\ Q\ |\!)\ \}$

and $\forall \mathcal{A} \subseteq \mathcal{A}_{\partial'}.\ \Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, P\, \rrparenthesis\ \&\&\ !\llbracket\, b\, \rrbracket\ \}\ \lfloor\, h_2\, \rfloor\ \{\ \llparenthesis\, Q\, \rrparenthesis\ \}$ hold. Furthermore, from lemma 7.1, we know that $\lambda = \lambda'$ and $\partial = \partial'$. Finally, we know that P, and Q are stable. Hence, the mapped triple can be directly derived through If and two application of Cons$_{\text{NA}}$justified by a variation of lemma 7.4 as shown below.

$$\dfrac{\dfrac{\dfrac{}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, P\ \&\&\ \llbracket\, b\, \rrbracket\, \rrparenthesis\ \}\ \lfloor\, h_1\, \rfloor\ \{\ \llparenthesis\, Q\, \rrparenthesis\ \}}\ \text{IH}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, P\, \rrparenthesis\ *\ \lfloor\, b\, \rfloor\ \}\ \lfloor\, h_1\, \rfloor\ \{\ \llparenthesis\, Q\, \rrparenthesis\ \}}\ \textsc{Cons}_{\text{NA}} \quad \dfrac{\dfrac{}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, P\, \rrparenthesis\ \&\&\ !\llbracket\, b\, \rrbracket\ \}\ \lfloor\, h_2\, \rfloor\ \{\ \llparenthesis\, Q\, \rrparenthesis\ \}}\ \text{IH}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, P\, \rrparenthesis\ *\ \neg\lfloor\, b\, \rfloor\ \}\ \lfloor\, h_2\, \rfloor\ \{\ \llparenthesis\, Q\, \rrparenthesis\ \}}\ \textsc{Cons}_{\text{NA}}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, P\, \rrparenthesis\ \}\ \mathsf{if}(\lfloor\, b\, \rfloor)\ \{\lfloor\, h_1\, \rfloor\}\ \mathsf{else}\ \{\lfloor\, h_2\, \rfloor\}\ \{\ \llparenthesis\, Q\, \rrparenthesis\ \}}\ \textsc{If}$$

Case $s \equiv \mathsf{while}(b)\ \mathsf{invariant}\ T\ \{h\}$ Let $\mathcal{A}_\partial$, $\mathcal{A}$, Γ, and λ be defined as specified by the mapping in figure 7.3. From the viper semantics and from the induction hypothesis, we learn that for the mapped TaDA judgment from the while body $\forall \mathcal{A} \subseteq \mathcal{A}_\partial.\ \Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, T\ \&\&\ \llbracket\, b\, \rrbracket\, \rrparenthesis\ \}\ \mathsf{while}(\lfloor\, b\, \rfloor)\ \{\lfloor\, h\, \rfloor\}\ \{\ \llparenthesis\, T\, \rrparenthesis\ \}$ holds. Furthermore, from lemma 7.1, we know that $\lambda = \lambda'$ and $\partial = \partial'$. Moreover, T can be derived by P and Q can be derived by $T\ \&\&\ !\llbracket\, b\, \rrbracket$. Finally, we know that P, T, and Q are all stable. Hence, the mapped triple can be directly derived through Loop and two application of Cons$_{\text{NA}}$justified by a variation of lemma 7.4 as shown below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, T\ \&\&\ \llbracket\, b\, \rrbracket\, \rrparenthesis\ \}\ \mathsf{while}(\lfloor\, b\, \rfloor)\ \{\lfloor\, h\, \rfloor\}\ \{\ \llparenthesis\, T\, \rrparenthesis\ \}}\ \text{IH}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, T\, \rrparenthesis\ *\ \lfloor\, b\, \rfloor\ \}\ \mathsf{while}(\lfloor\, b\, \rfloor)\ \{\lfloor\, h\, \rfloor\}\ \{\ \llparenthesis\, T\, \rrparenthesis\ \}}\ \textsc{Cons}_{\text{NA}}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, T\, \rrparenthesis\ \}\ \mathsf{while}(\lfloor\, b\, \rfloor)\ \{\lfloor\, h\, \rfloor\}\ \{\ \llparenthesis\, T\, \rrparenthesis\ *\ \neg\lfloor\, b\, \rfloor\ \}}\ \textsc{Loop}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, T\, \rrparenthesis\ \}\ \mathsf{while}(\lfloor\, b\, \rfloor)\ \{\lfloor\, h\, \rfloor\}\ \{\ \llparenthesis\, T\ \&\&\ !\llbracket\, b\, \rrbracket\, \rrparenthesis\ \}}\ \textsc{Cons}_{\text{NA}}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, P\, \rrparenthesis\ \}\ \mathsf{while}(\lfloor\, b\, \rfloor)\ \{\lfloor\, h\, \rfloor\}\ \{\ \llparenthesis\, Q\, \rrparenthesis\ \}}\ \textsc{Cons}_{\text{NA}}$$

Case $s \equiv x\ :=\ e$ Let $\mathcal{A}_\partial$, $\mathcal{A}$, Γ, and λ be defined as specified by the mapping in figure 7.3. Because the generated Viper code verifies successfully, we know that there exists some R that Viper can frame around the variable assignment. Important is that this frame does not contain the assigned variable. Furthermore, from the assumption that $\llparenthesis\ P\ \rrparenthesis$ is stable we know that $\llparenthesis\ R\ \rrparenthesis$, and subsequently $\llparenthesis\ Q\ \rrparenthesis$, is stable. The derivation of the mapped triple is shown below. First, from our invariants we know that the mapped judgment level is positive, hence we can apply AWeakening3$_{\text{NA}}$to weaken the judgment level to 0. Next, we derive the frame R through Cons$_{\text{NA}}$and then frame it with Frame$_{\text{NA}}$. Lastly, Assignmentis applied to finish the derivation tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;0;\mathcal{A} \vdash \{\ x = v\ \}\ x\ :=\ \lfloor\, e\, \rfloor\ \{\ x = \lfloor\, e\, \rfloor[v/x]\ \}}\ \textsc{Assignment}}{\Gamma;0;\mathcal{A} \vdash \{\ \llparenthesis\, R\, \rrparenthesis\ *\ x = v\ \}\ x\ :=\ \lfloor\, e\, \rfloor\ \{\ \llparenthesis\, R\, \rrparenthesis\ *\ x = \lfloor\, e\, \rfloor[v/x]\ \}}\ \textsc{Frame}_{\text{NA}}}{\Gamma;0;\mathcal{A} \vdash \{\ \llparenthesis\, P\, \rrparenthesis\ \}\ x\ :=\ \lfloor\, e\, \rfloor\ \{\ \llparenthesis\, Q\, \rrparenthesis\ \}}\ \textsc{Cons}_{\text{NA}}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, P\, \rrparenthesis\ \}\ x\ :=\ \lfloor\, e\, \rfloor\ \{\ \llparenthesis\, Q\, \rrparenthesis\ \}}\ \textsc{AWeakening3}_{\text{NA}}$$

Case $s \equiv x.F\ :=\ e$ Let $\mathcal{A}_\partial$, $\mathcal{A}$, Γ, λ, $\vec{s}$, and $\vec{S}$ be defined as specified by the mapping in figure 7.3. Similar to the previous case, because the generated Viper code verifies successfully, we know that there exists some R that Viper can frame around the field assignment. The derivation tree for the mapped triple is given below. First, we weaken the judgment level as seen before. Next, we derive the frame R and the context $V = \mathbb{D}_P(v)$ of the field value v with rule $\textsc{Cons}_\textsc{A}$. This is step is justified by lemma 7.2 and the soundness of Viper implications. Next, $(\!|\ R\ |\!)$ is framed around with $\textsc{Frame}_\textsc{A}$. The stability of $(\!|\ R\ |\!)$ and subsequently $(\!|\ Q\ |\!)$ follows from the assumption that $(\!|\ Q\ |\!)$ is stable. Lastly, we apply AExiststo extends the interference context with the field value, and then apply Mutationto complete the derivation tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;0;\mathcal{A} \vdash \forall\!\!\!\forall\vec{s} \in \vec{S}, v \in V.\ \langle\ x.F \mapsto v\ \rangle\ x.F\ :=\ \lfloor\ e\ \rfloor\ \langle\ x.F \mapsto \lfloor\ e\ \rfloor\ \rangle}\textsc{Mutation}}{\Gamma;0;\mathcal{A} \vdash \forall\!\!\!\forall\vec{s} \in \vec{S}.\ \langle\ \exists v \in V.\ x.F \mapsto v\ \rangle\ x.F\ :=\ \lfloor\ e\ \rfloor\ \langle\ \exists v \in V.\ x.F \mapsto \lfloor\ e\ \rfloor\ \rangle}\textsc{AExists}}{\Gamma;0;\mathcal{A} \vdash \forall\!\!\!\forall\vec{s} \in \vec{S}.\ \langle\ (\!|\ R\ |\!)\ *\ \exists v \in V.\ x.F \mapsto v\ \rangle\ x.F\ :=\ \lfloor\ e\ \rfloor\ \langle\ (\!|\ R\ |\!)\ *\ \exists v \in V.\ x.F \mapsto \lfloor\ e\ \rfloor\ \rangle}\textsc{Frame}_\textsc{A}}{\Gamma;0;\mathcal{A} \vdash \forall\!\!\!\forall\vec{s} \in \vec{S}.\ \langle\ (\!|\ P\ |\!)\ \rangle\ x.F\ :=\ \lfloor\ e\ \rfloor\ \langle\ (\!|\ Q\ |\!)\ \rangle}\textsc{Cons}_\textsc{A}}{\Gamma;\lambda;\mathcal{A} \vdash \forall\!\!\!\forall\vec{s} \in \vec{S}.\ \langle\ (\!|\ P\ |\!)\ \rangle\ x.F\ :=\ \lfloor\ e\ \rfloor\ \langle\ (\!|\ Q\ |\!)\ \rangle}\textsc{AWeakening3}_\textsc{A}$$

Case $s \equiv y\ :=\ x.F$ The derivation tree is shown below, the proof goes analogous to the previous case.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;0;\mathcal{A} \vdash \forall\!\!\!\forall\vec{s} \in \vec{S}, v \in V.\ \langle\ y.F \mapsto v\ \rangle\ x\ :=\ y.F\ \langle\ y.F \mapsto v\ *\ x = v\ \rangle}\textsc{Lookup}}{\Gamma;0;\mathcal{A} \vdash \forall\!\!\!\forall\vec{s} \in \vec{S}.\ \langle\ \exists v \in V.\ y.F \mapsto v\ \rangle\ x\ :=\ y.F\ \langle\ \exists v \in V.\ y.F \mapsto v\ *\ x = v\ \rangle}\textsc{AExists}}{\Gamma;0;\mathcal{A} \vdash \forall\!\!\!\forall\vec{s} \in \vec{S}.\ \langle\ (\!|\ R\ |\!)\ *\ \exists v \in V.\ y.F \mapsto v\ \rangle\ x\ :=\ y.F\ \langle\ (\!|\ R\ |\!)\ *\ \exists v \in V.\ y.F \mapsto v\ *\ x = v\ \rangle}\textsc{Frame}_\textsc{A}}{\Gamma;0;\mathcal{A} \vdash \forall\!\!\!\forall\vec{s} \in \vec{S}.\ \langle\ (\!|\ P\ |\!)\ \rangle\ x\ :=\ y.F\ \langle\ (\!|\ Q\ |\!)\ \rangle}\textsc{Cons}_\textsc{A}}{\Gamma;\lambda;\mathcal{A} \vdash \forall\!\!\!\forall\vec{s} \in \vec{S}.\ \langle\ (\!|\ P\ |\!)\ \rangle\ x\ :=\ y.F\ \langle\ (\!|\ Q\ |\!)\ \rangle}\textsc{AWeakening3}_\textsc{A}$$

Case $s \equiv \vec{x}\ :=\ M_H(\vec{e})$ Let $\mathcal{A}_\partial$, $\mathcal{A}$, Γ, λ, and ∂ be defined as specified by the mapping in figure 7.3. Furthermore, let $\forall \mathcal{A}' \subseteq \mathcal{A}_{\lambda'}.\ \lambda';\mathcal{A}' \vdash \{\ \lfloor\ A\ \rfloor\ \}\ M(\vec{z})$ returns $\{\ \lfloor\ B\ \rfloor\ \}$ where $\lambda' = 1 + \max \textsc{Levels}(A)$ be the mapped TaDA specification of the call.

Because the encoded Viper program verifies successfully, we know four properties: first, from Q Viper can derive some R that Viper can frame around the call as well as a $A[[\![\ \vec{e}\ ]\!]/\vec{z}]$ which is the precondition of the call. Second, there exists some T such that $\vdash_V \{R\}\mathit{stabilize}\{T\}$ holds. Third, Q can be derived from T and $B[[\![\ \vec{e}\ ]\!]/\vec{z}][\vec{x}/\vec{r}]$. Fourth, with P the values of `level` and `atomic_context` are larger than all levels from the callee precondition. The second property together with lemma 7.3 implies that $(\!|\ R\ |\!) \preceq (\!|\ T\ |\!)$ as well as that $(\!|\ T\ |\!)$ is stable. The fourth property together with the mapping of Voila method specifications from figure 7.3 implies that $\lambda geq \lambda'$ and $\partial \geq \lambda'$, i.e. that the current judgment level as well as the value of `atomicity_context` is larger or equals to the method level. Moreover, from the definition of $\mathcal{A}_\partial$ we know that $\mathcal{A}_\partial \subseteq \mathcal{A}_{\lambda'}$, hence for every possible judgment atomicity context there exists an equivalent callee method atomicity context, i.e. $\mathcal{A} \in \mathcal{A}_{\lambda'}$.

With that we have all ingredients for the derivation tree given below. First, the judgment level is weakened through AWEAKENING3$_{\text{A}}$to λ'. Next, the $\llparenthesis\ P\ \rrparenthesis$ is split into frame and precondition with CONS$_{\text{NA}}$. Next, lemma 7.4 allows us to move the variable substitution outside of the mapping. Lastly, $\llparenthesis\ R\ \rrparenthesis$ is framed around as $\llparenthesis\ T\ \rrparenthesis$ and CALL$_{\text{NA}}$completes the proof. The postcondition $\llparenthesis\ Q\ \rrparenthesis$ is stable it is derived from two stable assertions. As a reminder, we assume that Voila method pre- and postconditions are stable.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\lambda';\mathcal{A} \vdash \{\ \llparenthesis\ A\ \rrparenthesis[\lfloor\ \vec{e}\ \rfloor/\vec{z}]\ \}\ \vec{x}\ :=\ M(\lfloor\ \vec{e}\ \rfloor)\ \{\ \llparenthesis\ B\ \rrparenthesis[\lfloor\ \vec{e}\ \rfloor/\vec{z}][\vec{x}/\vec{r}]\ \}}\ \textsc{Call}_{\text{NA}}}{\Gamma;\lambda';\mathcal{A} \vdash \{\ \llparenthesis\ R\ \rrparenthesis\ *\ \llparenthesis\ A\ \rrparenthesis[\lfloor\ \vec{e}\ \rfloor/\vec{z}]\ \}\ \vec{x}\ :=\ M(\lfloor\ \vec{e}\ \rfloor)\ \{\ \llparenthesis\ T\ \rrparenthesis\ *\ \llparenthesis\ B\ \rrparenthesis[\lfloor\ \vec{e}\ \rfloor/\vec{z}][\vec{x}/\vec{r}]\ \}}\ \textsc{Frame}_{\text{NA}}}{\Gamma;\lambda';\mathcal{A} \vdash \{\ \llparenthesis\ R\ \rrparenthesis\ *\ \llparenthesis\ A[\llbracket\ \vec{e}\ \rrbracket/\vec{z}]\ \rrparenthesis\ \}\ \vec{x}\ :=\ M(\lfloor\ \vec{e}\ \rfloor)\ \{\ \llparenthesis\ T\ \rrparenthesis\ *\ \llparenthesis\ B[\llbracket\ \vec{e}\ \rrbracket/\vec{z}][\vec{x}/\vec{r}]\ \rrparenthesis\ \}}\ \textsc{Cons}_{\text{NA}}}{\Gamma;\lambda';\mathcal{A} \vdash \{\ \llparenthesis\ P\ \rrparenthesis\ \}\ \vec{x}\ :=\ M(\lfloor\ \vec{e}\ \rfloor)\ \{\ \llparenthesis\ Q\ \rrparenthesis\ \}}\ \textsc{Cons}_{\text{NA}}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\ P\ \rrparenthesis\ \}\ \vec{x}\ :=\ M(\lfloor\ \vec{e}\ \rfloor)\ \{\ \llparenthesis\ Q\ \rrparenthesis\ \}}\ \text{A}\textsc{Weakening}3_{\text{A}}$$

Case $s \equiv \vec{x}\ :=\ M_A(\vec{e})$ Most parts of the derivation are performed analogously to the non-atomic case. The difference is that from the successful verification of the encoding we further know that the respective interference context of the current judgment $\vec{S}$ is a subset of the method interference context $\vec{S}'$. Hence, the SUBSTapplication before CALL$_{\text{A}}$completes the proof is justified. The derivation tree is given below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\lambda';\mathcal{A} \vdash \forall\!\!\!\forall \vec{s}' \in \vec{S}'.\ \langle\ \llparenthesis\ A\ \rrparenthesis[\lfloor\ \vec{e}\ \rfloor/\vec{z}]\ \rangle\ \vec{x}\ :=\ M(\lfloor\ \vec{e}\ \rfloor)\ \langle\ \llparenthesis\ B\ \rrparenthesis[\lfloor\ \vec{e}\ \rfloor/\vec{z}][\vec{x}/\vec{r}]\ \rangle}\ \textsc{Call}_{\text{A}}}{\Gamma;\lambda';\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\ \llparenthesis\ A\ \rrparenthesis[\lfloor\ \vec{e}\ \rfloor/\vec{z}]\ \rangle\ \vec{x}\ :=\ M(\lfloor\ \vec{e}\ \rfloor)\ \langle\ \llparenthesis\ B\ \rrparenthesis[\lfloor\ \vec{e}\ \rfloor/\vec{z}][\vec{x}/\vec{r}]\ \rangle}\ \textsc{Subst}}{\Gamma;\lambda';\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\ \llparenthesis\ R\ \rrparenthesis\ *\ \llparenthesis\ A\ \rrparenthesis[\lfloor\ \vec{e}\ \rfloor/\vec{z}]\ \rangle\ \vec{x}\ :=\ M(\lfloor\ \vec{e}\ \rfloor)\ \langle\ \llparenthesis\ T\ \rrparenthesis\ *\ \llparenthesis\ B\ \rrparenthesis[\lfloor\ \vec{e}\ \rfloor/\vec{z}][\vec{x}/\vec{r}]\ \rangle}\ \textsc{Frame}_{\text{NA}}}{\Gamma;\lambda';\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\ \llparenthesis\ R\ \rrparenthesis\ *\ \llparenthesis\ A[\llbracket\ \vec{e}\ \rrbracket/\vec{z}]\ \rrparenthesis\ \rangle\ \vec{x}\ :=\ M(\lfloor\ \vec{e}\ \rfloor)\ \langle\ \llparenthesis\ T\ \rrparenthesis\ *\ \llparenthesis\ B[\llbracket\ \vec{e}\ \rrbracket/\vec{z}][\vec{x}/\vec{r}]\ \rrparenthesis\ \rangle}\ \textsc{Cons}_{\text{NA}}}{\Gamma;\lambda';\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\ \llparenthesis\ P\ \rrparenthesis\ \rangle\ \vec{x}\ :=\ M(\lfloor\ \vec{e}\ \rfloor)\ \langle\ \llparenthesis\ Q\ \rrparenthesis\ \rangle}\ \textsc{Cons}_{\text{A}}}{\Gamma;\lambda;\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\ \llparenthesis\ P\ \rrparenthesis\ \rangle\ \vec{x}\ :=\ M(\lfloor\ \vec{e}\ \rfloor)\ \langle\ \llparenthesis\ Q\ \rrparenthesis\ \rangle}\ \text{A}\textsc{Weakening}3_{\text{A}}$$

Case $s\ \equiv\ \mathsf{atomic}\{a\}$ Let $\mathcal{A}_\partial$, $\mathcal{A}$, Γ, and λ be defined as specified by the mapping in figure 7.3.

Because the encoded Viper program verifies successfully, we know three properties: first, there exists some R such that $\vdash_V\ \{P\}$*infer-interference-context*$\{R\}$ holds. Second, there exists some T, $\vec{s}$, and $\vec{S}$ such that $\Gamma;\lambda;\mathcal{A}\ \vdash\ \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\ \llparenthesis\ R\ \rrparenthesis\ \rangle\ \lfloor\ a\ \rfloor\ \langle\ \llparenthesis\ T\ \rrparenthesis\ \rangle$ is derivable in TaDA where $\vec{s}$ and $\vec{S}$ are defined according to the mapping in figure 7.3. Finally, $\vdash_V\ \{T\}$*stabilize*$\{Q\}$ holds. Similar to the previous case, from lemma 7.3 and 7.8 we know that $\llparenthesis\ R\ \rrparenthesis$ and $\llparenthesis\ Q\ \rrparenthesis$ are stable, as well as that $\llparenthesis\ P\ \rrparenthesis \preceq \llparenthesis\ R\ \rrparenthesis$ and $\llparenthesis\ T\ \rrparenthesis \preceq \llparenthesis\ Q\ \rrparenthesis$ hold. The first property together with lemma 7.8 and 7.2 implies that $\llparenthesis\, R \,\rrparenthesis \preceq \exists \vec{s} \in \vec{S}.\ \llparenthesis\, R \,\rrparenthesis$ holds.

With that we have all ingredients for the derivation tree below. First, $\textsc{Cons}_{\text{NA}}$is applied twice to get triple into the right form. Next, the non-atomic triple is transformed into a atomic triple with AWeakening2and AExists. Then, $\textsc{Cons}_{\text{A}}$rule is applied to guarantee that the postcondition is stable before the atomic triple changes back to a non-atomic triple. Lastly, we complete the derivation tree with the induction hypothesis.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\lambda;\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\ \llparenthesis\, R \,\rrparenthesis\ \rangle\ \lfloor\, a \,\rfloor\ \langle\ \llparenthesis\, T \,\rrparenthesis\ \rangle}\ \text{IH}}{\Gamma;\lambda;\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\ \llparenthesis\, R \,\rrparenthesis\ \rangle\ \lfloor\, a \,\rfloor\ \langle\ \llparenthesis\, Q \,\rrparenthesis\ \rangle}\ \textsc{Cons}_{\text{A}}}{\Gamma;\lambda;\mathcal{A} \vdash \langle\ \exists \vec{s} \in \vec{S}.\ \llparenthesis\, R \,\rrparenthesis\ \rangle\ \lfloor\, a \,\rfloor\ \langle\ \exists \vec{s} \in \vec{S}.\ \llparenthesis\, Q \,\rrparenthesis\ \rangle}\ \textsc{AExists}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \exists \vec{s} \in \vec{S}.\ \llparenthesis\, R \,\rrparenthesis\ \}\ \lfloor\, a \,\rfloor\ \{\ \exists \vec{s} \in \vec{S}.\ \llparenthesis\, Q \,\rrparenthesis\ \}}\ \textsc{AWeakening2}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, R \,\rrparenthesis\ \}\ \lfloor\, a \,\rfloor\ \{\ \llparenthesis\, Q \,\rrparenthesis\ \}}\ \textsc{Cons}_{\text{NA}}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ \llparenthesis\, P \,\rrparenthesis\ \}\ \lfloor\, a \,\rfloor\ \{\ \llparenthesis\, Q \,\rrparenthesis\ \}}\ \textsc{Cons}_{\text{NA}}$$

Case $s \equiv$ use_atomic using $R(r, \lambda', \vec{e})$ with g $\{a\}$ Let $\mathcal{A}_\partial$, $\mathcal{A}$, Γ, λ, ∂, $\vec{s}$, and $\vec{S}$ be defined as specified by the mapping in figure 7.3.

As before, because the encoded Viper program verifies, we know fife properties: first, there exists some R such that $\mathsf{acc}(G(r))$, $\mathsf{acc}(R\_\mathtt{interp}(r, \lambda, \vec{e}))$, and R can be derived from P. Second, using the induction hypothesis on the body of s we know that there exists some T such that $\Gamma; \lambda'; \mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}, \vec{m} \in \vec{M}.\ \langle\ I\left(\mathbf{R}_r^{\lambda'}(\vec{e}, z)\right)\ *\ [G]_r\ *\ \llparenthesis\, R \,\rrparenthesis\ \rangle\ \lfloor\, a \,\rfloor\ \langle\ I\left(\mathbf{R}_r^{\lambda'}(\vec{e}, y)\right)\ *\ \llparenthesis\, T \,\rrparenthesis\ \rangle$ is derivable in TaDA. Here, $\vec{M}$ is the part of the interference context that was added by the *link-interference-context* call. Furthermore, we mapped the region interpretation and guard to TaDA. Third, Q is derivable from T and the constrained region assertion, i.e. the Viper assertion corresponding to $\exists y \in Y.\ \mathbf{R}_r^{\lambda'}(\vec{e}, y)$. Fourth, the the current judgment level is larger than the specified region level λ'. Fifth, the assertion generated by *action-permitted*$(R,\ G,\ a,\ b)$ where a and b are the old and new region state, respectively, is asserted successfully. Furthermore, for the derivation tree we use lemma 7.2 to get $I\left(\mathbf{R}_r^{\lambda'}(\vec{e}, y)\right)\ *\ \llparenthesis\, T \,\rrparenthesis \preceq \exists y \in Y.\ I\left(\mathbf{R}_r^{\lambda'}(\vec{e}, y)\right)\ *\ \llparenthesis\, T \,\rrparenthesis$ where Y is defined as $\mathbb{D}_Z(R\_\mathtt{state}(r, \lambda, \vec{e}))$ and Z is the viper assertion corresponding to $I\left(\mathbf{R}_r^{\lambda'}(\vec{e}, y)\right)\ *\ \llparenthesis\, T \,\rrparenthesis$, i.e. the postcondition of the encoding of the substatement h.

Below we give the derivation for the mapped triple. First, we use AWeakening$3_{\text{A}}$to weaken the judgment level to the region level plus one. Next,

$\textsc{Cons}_{\mathrm{A}}$brings the pre- and post-condition into the correct form. Then, the $\textsc{UseAtomic}$rule is applied. Here we have to satisfy the two side conditions: first, the specified region is not contained in the interference context. This condition can be derived from the atomicity context definition from figure 7.3 and the facts that the region level is smaller than the judgment level, a judgment level is smaller than the method level, and in our considered subset of Voila the atomicity context does not change. The second side condition is that state transition is permitted by the the held guard. This condition follows from the definition of Y and lemma 7.9. Next, the $\textsc{Cons}_{\mathrm{A}}$rule is used to infer the interference contexts $\vec{M}$ of the regions hidden in the region interpretation. From the encoding of *link-interference-context* we know that $\vec{M}$ is defined by $\vec{m} \in \vec{M} \leftrightarrow \varphi(\vec{m}) \in S$ where φ is the region interpretation expressed as a function from the regions states hidden in the interpretation and where S is the interference context of the opened region. Similar to the proof of lemma 7.2 this interference context definition is a sound choice because in Voila a corresponding TaDA region interpretation contains $z = \varphi(\vec{m})$. Hence, the $\textsc{Cons}_{\mathrm{A}}$application is justified. Next, the $\vec{M}$ is added to the judgment interference context trough $\textsc{AExists}$. The last $\textsc{Cons}_{\mathrm{A}}$application is only to infer Y and was already justified before. Lastly, the induction hypothesis is used to complete the derivation tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\lambda';\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}, \vec{m} \in \vec{M}.\ \langle\, I\left(\mathbf{R}_r^{\lambda'}(\vec{e},z)\right) \ *\ [G]_r \ *\ \llparenthesis R \rrparenthesis \,\rangle \lfloor a \rfloor \langle\, I\left(\mathbf{R}_r^{\lambda'}(\vec{e},y)\right) \ *\ \llparenthesis T \rrparenthesis \,\rangle}\ \mathrm{IH}}{\Gamma;\lambda';\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}, \vec{m} \in \vec{M}.\ \langle\, I\left(\mathbf{R}_r^{\lambda'}(\vec{e},z)\right) \ *\ [G]_r \ *\ \llparenthesis R \rrparenthesis \,\rangle \lfloor a \rfloor \langle\, \exists y \in Y.\ I\left(\mathbf{R}_r^{\lambda'}(\vec{e},y)\right) \ *\ \llparenthesis T \rrparenthesis \,\rangle}\ \textsc{Cons}_{\mathrm{A}}}{\Gamma;\lambda';\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\, \exists \vec{m} \in \vec{M}.\ I\left(\mathbf{R}_r^{\lambda'}(\vec{e},z)\right) \ *\ [G]_r \ *\ \llparenthesis R \rrparenthesis \,\rangle \lfloor a \rfloor \langle\, \exists \vec{m} \in \vec{M}.\ \exists y \in Y.\ I\left(\mathbf{R}_r^{\lambda'}(\vec{e},y)\right) \ *\ \llparenthesis T \rrparenthesis \,\rangle}\ \textsc{AExists}}{\Gamma;\lambda';\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\, I\left(\mathbf{R}_r^{\lambda'}(\vec{e},z)\right) \ *\ [G]_r \ *\ \llparenthesis R \rrparenthesis \,\rangle \lfloor a \rfloor \langle\, \exists y \in Y.\ I\left(\mathbf{R}_r^{\lambda'}(\vec{e},y)\right) \ *\ \llparenthesis T \rrparenthesis \,\rangle}\ \textsc{Cons}_{\mathrm{A}}}{\Gamma;\lambda'+1;\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\, \mathbf{R}_r^{\lambda'}(\vec{e},z) \ *\ [G]_r \ *\ \llparenthesis R \rrparenthesis \,\rangle \lfloor a \rfloor \langle\, \exists y \in Y.\ \mathbf{R}_r^{\lambda'}(\vec{e},y) \ *\ \llparenthesis T \rrparenthesis \,\rangle}\ \textsc{UseAtomic}}{\Gamma;\lambda'+1;\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\, \llparenthesis P \rrparenthesis \,\rangle \lfloor a \rfloor \langle\, \llparenthesis Q \rrparenthesis \,\rangle}\ \textsc{Cons}_{\mathrm{A}}}{\Gamma;\lambda;\mathcal{A} \vdash \forall\!\!\!\forall \vec{s} \in \vec{S}.\ \langle\, \llparenthesis P \rrparenthesis \,\rangle \lfloor a \rfloor \langle\, \llparenthesis Q \rrparenthesis \,\rangle}\ \textsc{AWeakening3}_{\mathrm{A}}$$

7.2.5 Soundness of the Method Encoding

Recall, we have to show for a Voila method if the mapping is applied to the encoded method body, then the resulting TaDA triple corresponds to the specification of the Voila method. The soundness of both, non-atomic and abstract atomic, method encodings follows directly from the mapping of method specifications shown in figure 7.3 as well as the mapping of atomic and non-atomic statements. For both method types, the judgment level and the judgment atomicity context are soundly initialized by assuming the value of `level` to be larger than all region levels occurring in the precondition and by assuming the value of `atomicity_context` to be the same as the value of `level`. Furthermore, for abstract atomic methods the interference context is correctly initialized by assuming the interference context R`_ictxt` which is mapped to the judgment interference rely-guarantee to be the same as the respective contexts specified in the interference clauses.

Soundness of Viper Implications

Lastly, we require that every implication deduced automatically by Viper can be justified on the mapped TaDA judgment by TaDA's rule of consequence. In other words we require that if Viper can deduce Q from P then $\llbracket P \rrbracket \preceq \llbracket Q \rrbracket$ and $\llbracket Q \rrbracket \preceq \llbracket P \rrbracket$ has to hold. Both directions are needed to be sound inside of mapped atomic TaDA judgments. Entailments that do not involve encoded assertions of Voila are trivially valid vie shifts in both directions. The relevant encoded assertions of TaDA are the current judgment level, atomicity context, interference rely-guarantee, as well as points-to predicates and region assertions. Deduction on the first three components are correct because of their definition shown in figure 7.3 and lemma 7.2. Points-to predicates are encoded using fields, hence Viper imposes the same restriction on them as TaDA [18] [21]. Lastly, region assertions are encoded with abstract predicates which in Viper are treated as black boxes, i.e. nothing is deduced from them [21].

7.2.6 Extension of the Proof Sketch

Our proof sketch can easily be extended to cover all predefined guard algebras by extended the proof of corollary 7.7 and lemma 7.9. We omitted them mainly to not clutter the proof. Furthermore, the proof sketch for the open_region statement is a variation of the proof sketch for use_atomic. More involved is the extension to also cover update_region and make_atomic since this requires a more precise mapping of the atomicity context. In particular, the mapping of the images of the atomicity is difficult because we do not encode them. In order to fix this issue we propose that one first proofs soundness of a modified TaDA rule set where the images of the atomicity context are omitted. Instead, the modified rule set should associate the images only with the MakeAtomic and corresponding UpdateRegion rule instance. That way one would have to map the images of the atomicity context only for the encoding of update_region and make_atomic, which is then feasible by defining the image to be the set of all states to which the transition is permitted by the specified guard.

Chapter 8

Evaluation

8.1 Proof aiding statements

Besides method specifications and loop invariants, the user can also provide additional specifications to aid the proof or to manually encode language features.

On one side, to aid the verifier we added assert statements, lemma rules, and view shift statements. Assertions can be used to help the verifier deduce intermediate steps. Lemma methods are abstract methods with only a pre- and postcondition. Those lemma methods can be applied to manually deduce assertions. For example, some unsupported guard algebras can be encoded in Voila by defining the guard algebra through lemma methods. Lastly, to aid the verifier we added two view shift statements that encode special instances of the TaDA rule of consequence: first, the statement peak-into-region$(R(\vec{e}))$ unfolds and folds the region interpretation predicate R_`interp`$(\vec{e})$ so that the verifier can gain knowledge about invariants inside the region interpretation. Second, unique-guard(g) compensates for an incompleteness in our encoding where for a unique guard g the verifier cannot deduce $g(x)$ `&&` $g(y) \Longrightarrow x \neq y$.

On the other side, to manually encode language features, we directly support some of the Viper statements such as `unfold`, `fold`, `inhale`, and `exhale`.

8.2 Performance

Our verifier was tested on a variety of concurrent programs from literature [2], [3], [4], [15]. Table 8.1 shows some interesting evaluated programs, together with their number of lines of program code, specification, generated code, as well as the median execution time in seconds. The five benchmark

Name	Loc	Spec	Viper Loc	Time (s)
Counter	9	17	630	6
Counter Client	19	69	1426	5
Ticket Lock 1	11	39	971	92
Ticket Lock 2	11	29	880	23
Treiber Stack	106	217	2302	55

Table 8.1: Benchmark Programs

programs are a counter implementation similar to our running example, a counter client where the counter is used by multiple threads, two variations of ticket locks with atomic and non-atomic specifications, respectively, and lastly, an implementation of the Treiber Stack [23] in Voila.

All measurements were carried out on an Aspirer A715-71G-78NZ with an Intel Core i7-7700HQ and 16GB of RAM, running Windows 10 Home (version 1803). The used JVM is Java HotSpot^{TM} . Sbt version is 0.13.15, Scala version is 2.11.11. All benchmarks were run without the optional well-definedness checks and with the silicon Viper backend verifier. The verifier was warmed up, with Voila programs from our collection, before the results were measured. We measured 20 iterations and then took the median. We do not take the results as a performance quantification, but instead use them to compare between different specification patterns, such as excessive use of lemma methods, atomic specifications, not-atomic specifications and so on.

The ratio of lines of code to lines specification is roughly 1 to 2.4, indicating that the developer effort is successfully kept reasonably low. The existing overhead is mainly caused by guard specifications and explicit encodings. In more detail, the counter as well as the two ticket lock variants require only the region declarations, method specifications, rule statements, and loop invariants. The counter client uses an unsupported guard algebra, hence requires explicit applications of lemma methods that specify the guard algebra. Lastly, the Treiber stack requires additional help from the user because of its complex region declaration.

Next, the number of generated lines of codes indicates the complexity of the encoding. Regarding the measured execution time, we can make the following observations: The ticket lock version with the atomic specification is considerably slower than the version with a non-atomic specification. We mainly, attribute this performance decrease to the explicit frames generated inside the make_atomic encoding in combination with the used complex guard algebra. Surprisingly, the counter client compares well compared to its size, even though it makes heavy use of explicitly applies lemma methods.

8.3 Optimizations

In our verifier, we employ two optimizations. The first optimization is that we replace multiple stabilizations with a single one. For example, when two atomic statements are sequentially composed, then we remove the stabilization in the postlude of the first atomic statement. This optimization is sound since further stabilizations directly after a first one do not change the verifier knowledge due to the reflexive transitivity of the state transition system.

The second optimization is that existentially bound variables are instantiated if possible. As a reminder, in the encoding of the action state transition system, existential quantification is used: for example, to check if the action $(\ x,y\ \mid\ x < y\ \mid\ G : x\ \rightarrow\ y\ )$ permits a transition from a to b with guard G, the assertion $\exists x,y.\ x = a\ \wedge\ y = b\ \wedge\ x < y\ \wedge\ G \leq G$ would be generated. If such bound variables can be uniquely determined by the arguments a, b, and G, then we try to instantiate them. For the previously shown assertion we would get the simplified assertion $a < b\ \wedge\ G \leq G$.

Quantified trigger [13] are another important part of the encoding that we do not cover in this report. A quantified trigger belonging to a Viper `forall` expression determines when the expression is instantiated. If a trigger is not precise enough then too many instantiates can decrease performance. On the other side, if a trigger is too conservative then the verifier might not be able to derive a valid assertion. Because triggers are especially important for the automation of guard algebra encodings, i.e. play a key role for a proof outline checker, we decided to trade off some performance to gain more automation.

8.4 Limitations

As mentioned in the introduction, Voila does not support all components of TaDA. In this section, we discuss such unsupported components and their impact on the set of programs Voila can verify.

Private and Public Assertion Parts Figure 8.1 shows a complete syntactic judgment in TaDA. In an atomic triple, an assertion is split into a private and public part. The public part is allowed to depend on variables bound by the pseudo quantifier whereas the private part is not. A non-atomic triple then is defined as an atomic triple without a public part, i.e. $\{P\}\ \mathbb{C}\ \{Q\}$ corresponds to $\langle P \mid \mathsf{true}\rangle\ \mathbb{C}\ \langle Q \mid \mathsf{true}\rangle$. In Voila, for method specifications we support either private assertions or public assertions, but not both. As a consequence, we cannot prove triples specifying both, atomic and non-atomic, behavior. For example, the triple given in figure 8.2 specifies that a call to $\mathsf{incTo}(x,y)$ atomically increases the value of a **ECounter** region by

$$\Gamma; \lambda; \mathcal{A} \vdash \forall\!\!\forall \vec{x} \in \vec{X}.\ \langle\ P_P \mid P(\vec{x})\ \rangle\ \mathbb{C}\ \exists\!\!\exists \vec{y} \in \vec{Y}.\ \langle\ Q_P(\vec{x},\vec{y}) \mid Q(\vec{x},\vec{y})\ \rangle$$

Figure 8.1: A complete syntactic judgments in TaDA. The assertions P_P and $Q_P(\vec{x},\vec{y})$ are referred to as private whereas P and $Q(\vec{x},\vec{y})$ are called public parts of the assertions.

$$\begin{array}{ll} & \forall\!\!\forall v.\ \langle y.val \mapsto z \mid \mathbf{ECounter}_a^{\lambda}(x,\ v)\rangle \\ \Gamma; \lambda; \mathcal{A} \vdash & \qquad\qquad \mathsf{incTo(x, y)} \\ & \langle y.val \mapsto v \mid \mathbf{ECounter}_a^{\lambda}(x,\ v+42)\rangle \end{array}$$

Figure 8.2: Example specification with private and public parts.

42 and at the same time non-atomically writes the old state value to a field *y.val*. At least to our knowledge, adding a separation between private and public assertions in Voila is feasible, but requires an extended encoding that tracks whether resources are private or public.

View Shifts As mentioned in chapter 7, in TaDA view shifts are used to update ghost state without changing the underlying program state. For example, view shifts can be used to learn an invariant that is hidden inside a region interpretation. Furthermore, view shifts can transform guards according to their guard algebras. Not all view shifts can be performed automatically by the Voila verifier: in general only surface level implications, e.g. without opening regions, or predefined operations such as predefined guard algebras are supported automatically. Some more complex view shifts can be manually achieved through lemma methods or the special proof statements mentioned in section 8.1. Further view shifts are not supported, hence some correct Voila programs are not verifiable with the Voila verifier. Even though such incompletenesses are an inherent flaw in our verification approach, we do not believe it to be a major obstacle for the verification of real-world programs since such verifications do not require arbitrary view shifts.

Angelic Choice As mentioned in section 6.2.5, Viper does not support angelic choice, hence some TaDA components cannot be directly encoded in Viper. For example, the UpdateRegion rule employs angelic choice to express that an update might or might not have happened. In such cases, we have to either employ approximations of similar behavior or push proof obligations to the user. In the case of UpdateRegion we approximated the behavior by defining that an update only happened if the region state changed. As a consequence, the user has to explicitly specify if an update occurred without the region state changing.

Heap Representation In Voila the heap is accessed through field reads and writes. On the other side, in TaDA the heap is accessed with addresses. As a consequence, TaDA allows address arithmetic whereas Voila does not. Hence, programs that employ complex pointer arithmetic might not be verifiable in Voila.

Artificial Cases In some cases, we decided to enforce limitations to increase the performance of the verifier. Often, those limitations only rule out artificial programs. For example, in Voila a method cannot be called inside a make_atomic statement if the method's level is higher than the level of the region updated in make_atomic. This restriction is weaker in TaDA where such a method can be called inside a UpdateRegion rule. However, UpdateRegion opens a region interpretation and hence the judgment level prevents such a method call if region interpretations are defined in a useful way, meaning regions contained in an interpretation have a lower level than the region they are contained in.

Chapter 9

Conclusion

9.1 Related Work

We group related work into three categories: other logics besides TaDA for the verification of fine-grained concurrent code, other verifiers for such logics, and finally other encodings to Viper.

Besides TaDA, FCSL [15] and Iris [10] are other logics for verification of fine-grained concurrent code. FCSL offers the subjective separating conjunction to reason about the distribution of resources among multiple threads, hence enable reasoning about interference caused by the environment. In order to specify fine-grained concurrency, histories [20] that abstractly describe atomic actions can be employed as a distributed resource. Iris [10] uses invariants and extended monoids to reason about fine-grained concurrency. Its generalized logic allows the implementation and verification of specialized programming logics inside the Iris framework. Interestingly, Iris and TaDA share deep commonalities. For example, even though shared regions are not part of the Iris logic, they can be easily expressed within the Iris framework by defining regions as Iris invariants. As a consequence, opening a region in Iris has similar limitations as in TaDA, namely a region interpretation is required to be stable before use, a region cannot be opened twice, and a region can only be opened an abstractly discrete instance in time. In TaDA functional correctness together with abstract atomicity are both proven in a single proof. Some other logics [24] [6] show functional correctness and abstract atomicity separately: first, the functional correctness of a coarse-grained implementation is proven. Afterwards, it is shown that a fine-grained implementation contextually refines the coarse-grained one.

Regarding other verifiers, there exists the Caper verifier [4] for TaDA or Coq for FCSL and Iris. Caper supports a smaller subset of TaDA core mechanics but shines with its higher degree of automation compared to Voila and its powerful guard support. Some of the design choices for Voila, such as the state transition system defined by actions or some of the guard algebras, were motivated by Caper. On the other side, Coq is a declarative proof checker. The versatility of declarative proof checker allows Coq to thoroughly verify a logic or programs. As a trade-off, Coq offers less automation and hence requires more developer effort.

Regarding other encodings to Viper, in recent years more verifiers have been build on top of the Viper infrastructure. Chalice [12], another verifier for concurrent code, got an implementation to Viper besides its older implementation to Boogie [11]. Nagini [5] is a powerful verification frontend for Python that can verify more complex properties such as memory safety, deadlock freedom, and input-output behavior. Nagini is currently used in the VerifiedScion project to verify the python implementation of scion. Lastly, [21] shows how weak-memory programs can be encoded and verified in Viper.

9.2 Future Work

Most of the future work follows directly from the limitations discussed in section 8.4. In general, we identify two different directions: increasing the set of verifiable programs inside the current logic or extending the supported Voila logic. Future work regarding Viper is covered in the next section.

The set of verifiable programs could be increased by first investigating the relation between programming patterns in fine-grained concurrent programming and the required guard algebras together with view shifts to prove these patterns. And then, developing and implementing guards and view shifts that aid programming patterns in Voila.

For example, in many concurrent programs reduction is used to join the results of different threads. Assume a program that concurrently counts some elements and then joins those partial counts to a total count by concurrently increasing some heap cell. For such a program, part of a shared region that contains the heap cell could be modeled with a guard $W(p, x)$ where p is the fraction of the result and x is the partial result. Combined with the algebra defined by $W(p_1, x_1) \bullet W(p_2, m_2) = W(p_1 + p_2, m_1 + m_2)$ where $p_1 + p_2 \leq 1$, we could reason about the relation of partial count to total count.

On the other side, the Voila logic could be broadened to support Total-TaDA [3], an extension of TaDA that enables total-correctness proofs. Another possibility would be to integrate weak-memory reasoning to enable verification with a more realistic concurrent memory model.

9.3 Encoding to Viper

Using Viper as an intermediate verification language has several advantages: foremost, the powerful support for permission-based reasoning integrated into Viper facilitates the encoding of other logics by offering higher-level abstractions. For example, predicates and heap-dependent functions are directly a good fit for the encoding region interpretations and region state, respectively. Furthermore, `inhale` and `exhale` statements provide good primitives to encode more complex logic components. Lastly, a strong tool support for Viper as well as existing developing tools such as the Viper IDE aid a faster development cycle.

However, during the development of the Voila verifier, we also encountered a range of limitations. We mentioned some of them, such as the absence of angelic choice, in section 8.4. In the rest of this section, we go into detail on some of the more low-level limitations we encountered during the development of the Voila verifier.

As mentioned before, quantified trigger are one of the core mechanics to enable automatic deductions. Intuitively, a trigger guards a quantified expression by only instantiating the expression if a corresponding ground term derived by the verifier matches the specified trigger. The general limitations of triggers make it difficult to enable more complex automations that are required, for example, for the automatic support of some guard algebras. One future work regarding Viper could be to support more complex automation strategies. An example of such strategy support can be found in Isabelle [19] where the declaration of introduction, elimination, and destruction rules can aid the automation process.

Havocing of permissions and values is one of the key technique employed in our encoding. However, havocing as a primitive is not supported in Viper and has to be performed through successive inhaling and exhaling of resources. As a consequence, particularly, havocing of quantified resources is made difficult because also the trigger support for such assertions is lacking. Here, we identify that investigating the potential of havocing as a primitive in Voila and a subsequent implementation is another suitable future work. In a similar direction, as we have seen in the evaluation, explicit frames as we employed them in the make_atomic encoding can cause performance issues. Hence, supporting explicit frames directly in Viper might fix such performance issues.

9.4 Final Conclusion

In our work, we have shown how to encode more complex logics such as TaDA into Viper. Furthermore, we outlined the soundness of a subset of our encoding. The Voila verifier supports a larger subset of TaDA core components than other existing verifier and poses a successful implementation of an outline checker. However, the current Voila verifier is lacking in its support for guards and more complex view shifts which limit the set of verifiable programs.

Nevertheless, we illustrated the potential of outline checker that require users to reason about key steps mainly and automatically deduce lower level proof obligations. We believe that outline checkers are a suitable choice to achieve a good trade-off between fully-automatic verification and declarative proof checking.

Appendix A

TaDA rules

List of the TaDA rules we used in this thesis. The rules were adapted so that points-to predicates have fields as receiver, that methods can have multiple return arguments, and that every judgment is either a completely non-atomic or atomic judgment.

$$\frac{\Gamma;\lambda;\mathcal{A} \vdash \{\ P\ \}\ \mathbb{C}_1\ \{\ R\ \} \quad \Gamma;\lambda;\mathcal{A} \vdash \{\ R\ \}\ \mathbb{C}_2\ \{\ Q\ \}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ P\ \}\ \mathbb{C}_1;\mathbb{C}_2\ \{\ Q\ \}}\ \textsc{Seq}$$

$$\frac{\Gamma;\lambda;\mathcal{A} \vdash \{\ P * b\ \}\ \mathbb{C}_1\ \{\ Q\ \} \quad \Gamma;\lambda;\mathcal{A} \vdash \{\ P * \neg b\ \}\ \mathbb{C}_2\ \{\ Q\ \}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ P\ \}\ \mathsf{if}(b)\ \{\mathbb{C}_1\}\ \mathsf{else}\ \{\mathbb{C}_2\}\ \{\ Q\ \}}\ \textsc{If}$$

$$\frac{\Gamma;\lambda;\mathcal{A} \vdash \{\ P * b\ \}\ \mathbb{C}\ \{\ P\ \}}{\Gamma;\lambda;\mathcal{A} \vdash \{\ P\ \}\ \mathsf{while}(b)\ \{\mathbb{C}\}\ \{\ P * \neg b\ \}}\ \textsc{Loop}$$

$$\frac{(\ \lambda;\mathcal{A} \vdash \{\ P(\vec{z})\ \}\ M(\vec{z})\ \mathsf{returns}\ (\vec{r})\ \{\ Q(\vec{z},\vec{r})\ \}\ ) \in \Gamma}{\Gamma;\lambda;\mathcal{A} \vdash \{\ P(\vec{e})\ \}\ \vec{x}\ :=\ M(e)\ \{\ Q(\vec{e},\vec{x})\ \}}\ \textsc{Call}_{\textsc{NA}}$$

$$\frac{(\ \lambda;\mathcal{A} \vdash \mathbb{\forall}\vec{s} \in \vec{S}.\ \langle\ P(\vec{s},\vec{z})\ \rangle\ M(\vec{z})\ \mathsf{returns}\ (\vec{r})\ \langle\ Q(\vec{s},\vec{z},\vec{r})\ \rangle\ ) \in \Gamma}{\Gamma;\lambda;\mathcal{A} \vdash \mathbb{\forall}\vec{s} \in \vec{S}.\ \langle\ P(\vec{s},\vec{e})\ \rangle\ \vec{x}\ :=\ M(e)\ \langle\ Q(\vec{s},\vec{e},\vec{x})\ \rangle}\ \textsc{Call}_{\textsc{A}}$$

$$\frac{}{\Gamma;\lambda;\mathcal{A} \vdash \{\ x = v\ \}\ x\ :=\ e\ \{\ x = e[v/x]\ \}}\ \textsc{Assignment}$$

$$\frac{}{\Gamma;0;\mathcal{A} \vdash \mathbb{\forall} v \in V.\ \langle\ x.F \mapsto v\ \rangle\ x.F\ :=\ e\ \langle\ x.F \mapsto e\ \rangle}\ \textsc{Mutation}$$

$$\frac{}{\Gamma;0;\mathcal{A} \vdash \mathbb{\forall} v \in V.\ \langle\ x.F \mapsto v\ \rangle\ x\ :=\ y.F\ \langle\ x.F \mapsto v * x = v\ \rangle}\ \textsc{Lookup}$$

$$\frac{}{\Gamma;\lambda;\mathcal{A}\vdash \begin{array}{c}\forall\!\!\!\forall v\in V.\ \langle\, y.F\mapsto v\,\rangle\\ x\ :=\ \mathsf{CAS}_F(y,e_{from},e_{to})\\ \langle\, v=e_{from}\ ?\ y.F\mapsto e_{to}\ :\ y.F\mapsto e_{from}\,\rangle\end{array}}\ \textsc{CompareAndSet}$$

$$\frac{\Gamma;\lambda;\mathcal{A}\vdash \forall\!\!\!\forall s\in S.\ \langle\, I\left(\mathbf{R}^{\lambda}_{a}(\vec{e},s)\right)*P(s)\,\rangle\ \mathbb{C}\ \langle\, I\left(\mathbf{R}^{\lambda}_{a}(\vec{e},s)\right)*Q(s)\,\rangle}{\Gamma;\lambda+1;\mathcal{A}\vdash \forall\!\!\!\forall s\in S.\ \langle\, \mathbf{R}^{\lambda}_{a}(\vec{e},s)*P(s)\,\rangle\ \mathbb{C}\ \langle\, \mathbf{R}^{\lambda}_{a}(\vec{e},s)\ *\ Q(s)\,\rangle}\ \textsc{OpenRegion}$$

$$\frac{\begin{array}{c}a\notin\mathcal{A}\qquad\{(s,y)\mid s\notin S, y\in Y\}\subseteq\mathbb{T}^{\star}_{\mathbf{R}}(G)\\ \Gamma;\lambda;\mathcal{A}\vdash\forall\!\!\!\forall s\in S.\ \begin{array}{c}\langle I\left(\mathbf{R}^{\lambda}_{a}(\vec{e},s)\right)*[G]_a*P(s)\rangle\\ \mathbb{C}\\ \langle\exists y\in Y.\ I\left(\mathbf{R}^{\lambda}_{a}(\vec{e},y)\right)*Q(s)\rangle\end{array}\end{array}}{\Gamma;\lambda+1;\mathcal{A}\vdash\forall\!\!\!\forall s\in S.\ \begin{array}{c}\langle\mathbf{R}^{\lambda}_{a}(\vec{e},s)*[G]_a*P(s)\rangle\\ \mathbb{C}\\ \langle\exists y\in Y.\ \mathbf{R}^{\lambda}_{a}(\vec{e},y)*Q(s)\rangle\end{array}}\ \textsc{UseAtomic}$$

$$\frac{\begin{array}{c}a\notin\mathcal{A}\qquad\{(s,y)\mid s\notin S, y\in Y\}\subseteq\mathbb{T}^{\star}_{\mathbf{R}}(G)\\ \Gamma;\lambda;a:s\in S\rightsquigarrow Y,\mathcal{A}\vdash\ \begin{array}{c}\{\exists s\in S.\ \mathbf{R}^{\lambda}_{a}(\vec{e},s)*a\ \Rrightarrow\ \blacklozenge\ \}\\ \mathbb{C}\\ \{\exists s\in S.\ \exists y\in Y.\ a\ \Rrightarrow\ (s,y)\}\end{array}\end{array}}{\Gamma;\lambda;\mathcal{A}\vdash\forall\!\!\!\forall s\in S.\ \langle\,\mathbf{R}^{\lambda}_{a}(\vec{e},s)*[G]_a\,\rangle\ \mathbb{C}\ \langle\,\exists y\in Y.\ \mathbf{R}^{\lambda}_{a}(\vec{e},y)*[G]_a\,\rangle}\ \textsc{MakeAtomic}$$

$$\frac{\begin{array}{c}a\notin\mathcal{A}\qquad\{(s,y)\mid s\notin S, y\in Y\}\subseteq\mathbb{T}^{\star}_{\mathbf{R}}(G)\\ \Gamma;\lambda;\mathcal{A}\vdash\forall\!\!\!\forall s\in S.\ \begin{array}{c}\langle I\left(\mathbf{R}^{\lambda}_{a}(\vec{e},s)\right)*P(s)\rangle\\ \mathbb{C}\\ \left\langle\begin{array}{c}\exists y\in Y.\ I\left(\mathbf{R}^{\lambda}_{a}(\vec{e},y)\right)*Q_1(s,y)\ \vee\\ I\left(\mathbf{R}^{\lambda}_{a}(\vec{e},s)\right)*Q_2(s)\end{array}\right\rangle\end{array}\end{array}}{\begin{array}{c}\Gamma;\lambda+1;a:s\in S\rightsquigarrow Y,\mathcal{A}\vdash\forall\!\!\!\forall s\in S.\ \langle\mathbf{R}^{\lambda}_{a}(\vec{e},s)*a\ \Rrightarrow\ \blacklozenge*P(s)\rangle\\ \mathbb{C}\\ \left\langle\begin{array}{c}\exists y\in Y.\ \mathbf{R}^{\lambda}_{a}(\vec{e},y)*Q_1(s,y)*a\ \Rrightarrow\ (s,y)\ \vee\\ \mathbf{R}^{\lambda}_{a}(\vec{e},s)*Q_2(s)*a\ \Rrightarrow\ \blacklozenge\end{array}\right\rangle\end{array}}\ \textsc{UpdateRegion}$$

$$\frac{\begin{array}{c}\lambda;\mathcal{A}\vdash R\preceq T\qquad\mathcal{A}\vDash T\ \mathsf{stable}\qquad\mathsf{mods}(\mathbb{C})\cap\mathsf{pvars}(\mathsf{R})=\emptyset\\ \Gamma;\lambda;\mathcal{A}\vdash\{\ P\ \}\ \mathbb{C}\ \{\ Q\ \}\end{array}}{\Gamma;\lambda;\mathcal{A}\vdash\{\ R*P\ \}\ \mathbb{C}\ \{\ T*Q\ \}}\ \textsc{Frame}_{\textsc{NA}}$$

$$\frac{\begin{array}{c}\lambda;\mathcal{A}\vdash\forall\vec{s}\in\vec{S}.\ R(\vec{s})\preceq T(\vec{s})\qquad\mathcal{A}\vDash T(\vec{s})\ \mathsf{stable}\qquad\mathsf{mods}(\mathbb{C})\cap\mathsf{pvars}(\mathsf{R}(\vec{s}))=\emptyset\\ \Gamma;\lambda;\mathcal{A}\vdash\forall\!\!\!\forall\vec{s}\in\vec{S}.\ \langle\ P(\vec{s})\ \rangle\ \mathbb{C}\ \langle\ Q(\vec{s})\ \rangle\end{array}}{\Gamma;\lambda;\mathcal{A}\vdash\forall\!\!\!\forall\vec{s}\in\vec{S}.\ \langle\ R(\vec{s})*P(\vec{s})\ \rangle\ \mathbb{C}\ \langle\ T(\vec{s})*Q(\vec{s})\ \rangle}\ \textsc{Frame}_{\textsc{A}}$$

$$\dfrac{\begin{array}{c} f : Z \to S \\ \Gamma; \lambda; \mathcal{A} \vdash \mathbb{\forall} s \in S.\ \langle\ P(s)\ \rangle\ \mathbb{C}\ \langle\ Q(s)\ \rangle \end{array}}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{\forall} z \in Z.\ \langle\ P(f(z))\ \rangle\ \mathbb{C}\ \langle\ Q(f(z))\ \rangle}\ \textsc{Subst}$$

$$\dfrac{\begin{array}{c} \lambda; \mathcal{A} \vdash P \preceq P' \qquad \lambda; \mathcal{A} \vdash Q' \preceq Q \\ \Gamma; \lambda; \mathcal{A} \vdash \{\ P'\ \}\ \mathbb{C}\ \{\ Q'\ \} \end{array}}{\Gamma; \lambda; \mathcal{A} \vdash \{\ P\ \}\ \mathbb{C}\ \{\ Q\ \}}\ \textsc{Cons}_{\mathrm{NA}}$$

$$\dfrac{\begin{array}{c} \lambda; \mathcal{A} \vdash \forall \vec{s} \in \vec{S}.\ Q'(\vec{s}) \preceq Q(\vec{s}) \\ \lambda; \mathcal{A} \vdash \forall \vec{s} \in \vec{S}.\ P(\vec{s}) \preceq P'(\vec{s}) \land P'(\vec{s}) \preceq P(\vec{s}) \\ \Gamma; \lambda; \mathcal{A} \vdash \mathbb{\forall} \vec{s} \in \vec{S}.\ \langle\ P'(\vec{s})\ \rangle\ \mathbb{C}\ \langle\ Q'(\vec{s})\ \rangle \end{array}}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{\forall} \vec{s} \in \vec{S}.\ \langle\ P(\vec{s})\ \rangle\ \mathbb{C}\ \langle\ Q(\vec{s})\ \rangle}\ \textsc{Cons}_{\mathrm{A}}$$

$$\dfrac{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{\forall} \vec{s} \in \vec{S}, \vec{z} \in \vec{Z}.\ \langle\ P(\vec{s}, \vec{z})\ \rangle\ \mathbb{C}\ \langle\ Q(\vec{s}, \vec{z})\ \rangle}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{\forall} \vec{s} \in \vec{S}.\ \langle\ \exists \vec{z} \in \vec{Z}.\ P(\vec{s}, \vec{z})\ \rangle\ \mathbb{C}\ \langle\ \exists \vec{z} \in \vec{Z}.\ Q(\vec{s}, \vec{z})\ \rangle}\ \textsc{AExists}$$

$$\dfrac{\Gamma; \lambda; \mathcal{A} \vdash \langle\ P\ \rangle\ \mathbb{C}\ \langle\ \mathbb{Q}\ \rangle}{\Gamma; \lambda; \mathcal{A} \vdash \{\ P\ \}\ \mathbb{C}\ \{\ \mathbb{Q}\ \}}\ \textsc{AWeakening2}$$

$$\dfrac{\lambda' \leq \lambda \qquad \Gamma; \lambda'; \mathcal{A} \vdash \mathbb{\forall} \vec{s} \in \vec{S}.\ \langle\ P(\vec{s})\ \rangle\ \mathbb{C}\ \langle\ \mathbb{Q}(\vec{s})\ \rangle}{\Gamma; \lambda; \mathcal{A} \vdash \mathbb{\forall} \vec{s} \in \vec{S}.\ \langle\ P(\vec{s})\ \rangle\ \mathbb{C}\ \langle\ \mathbb{Q}(\vec{s})\ \rangle}\ \textsc{AWeakening3}_{\mathrm{A}}$$

$$\dfrac{\lambda' \leq \lambda \qquad \Gamma; \lambda'; \mathcal{A} \vdash \{\ P(\vec{s})\ \}\ \mathbb{C}\ \{\ \mathbb{Q}(\vec{s})\ \}}{\Gamma; \lambda; \mathcal{A} \vdash \{\ P(\vec{s})\ \}\ \mathbb{C}\ \{\ \mathbb{Q}(\vec{s})\ \}}\ \textsc{AWeakening3}_{\mathrm{NA}}$$

Bibliography

[1] Pedro da Rocha Pinto. *Reasoning with time and data abstractions*. PhD thesis, Ph. D. thesis, Imperial College London, 2017.

[2] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In *European Conference on Object-Oriented Programming*, pages 207–231. Springer, 2014.

[3] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular termination verification for non-blocking concurrency. In *European Symposium on Programming Languages and Systems*, pages 176–201. Springer, 2016.

[4] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. Caper: Automatic verification for fine-grained concurrency. In *European Symposium on Programming*, pages 420–447. Springer, 2017.

[5] M. Eilers and P. Müller. Nagini: A static verifier for python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification (CAV)*, LNCS, pages 596–603. Springer International Publishing, 2018.

[6] Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc: A mechanised relational logic for fine-grained concurrency. 2018.

[7] Charles Antony Richard Hoare. Proof of correctness of data representations. In *Programming Methodology*, pages 269–281. Springer, 1978.

[8] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011.

[9] Cliff B Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

[10] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *ACM SIGPLAN Notices*, volume 50, pages 637–650. ACM, 2015.

[11] K Rustan M Leino. This is boogie 2. *Manuscript KRML*, 178(131), 2008.

[12] K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009.

[13] Michał Moskal. Programming with triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 20–29. ACM, 2009.

[14] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.

[15] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *European Symposium on Programming Languages and Systems*, pages 290–310. Springer, 2014.

[16] Peter W O'hearn. Resources, concurrency, and local reasoning. *Theoretical computer science*, 375(1-3):271–307, 2007.

[17] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *ACM SIGPLAN Notices*, volume 40, pages 247–258. ACM, 2005.

[18] Matthew J Parkinson and Alexander J Summers. The relationship between separation logic and implicit dynamic frames. *arXiv preprint arXiv:1203.6859*, 2012.

[19] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[20] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *European Symposium on Programming Languages and Systems*, pages 333–358. Springer, 2015.

[21] A. J. Summers and P. Müller. Automating deductive verification for weak-memory programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 190–209. Springer-Verlag, 2018.

[22] Alexander J Summers and Sophia Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In *European Conference on Object-Oriented Programming*, pages 129–153. Springer, 2013.

[23] R Kent Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center New York, 1986.

[24] Aaron J Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *ACM SIGPLAN Notices*, volume 48, pages 343–356. ACM, 2013.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Verifying Fine-Grained Concurrent Data Structures

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Wolf	Felix

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Luzern, 21.08.2018	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.