Karlsruhe Institute of Technology

Institute for Program Structures
and Data Organization (IPD)

Carnegie Mellon University

institute for SOFTWARE RESEARCH

Gradual Program Verification with Implicit Dynamic Frames

Master's Thesis of

Johannes Bader

presented to the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Advisor: Prof. Jonathan Aldrich, Carnegie Mellon University - Pittsburgh, USA
Advisor: Prof. Éric Tanter, University of Chile - Santiago, Chile
Co-Advisor: Prof. Dr.-Ing. Gregor Snelting, Karlsruhe Institute of Technology - Karlsruhe, Germany

Duration: 2016-05-10 – 2016-10-04

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text, and have followed the rules of the KIT for upholding good scientific practice.

Karlsruhe, 2016-10-04

. . . . . .

(**Johannes Bader**)

Abstract

Both static and dynamic program verification approaches have disadvantages potentially disqualifying them as a single methodology to rely on. Motivated by gradual type systems which solve a very similar dilemma in the world of type systems, we propose *gradual verification*, an approach that seamlessly combines static and dynamic verification. Drawing on principles from abstract interpretation and recent work on *abstracting gradual typing* by Garcia, Clark and Tanter, we formalize steps to obtain a gradual verification system in terms of a static one.

This approach yields *by construction* a verification system that is compatible with the original static system, but overcomes its rigidity by resorting to methods of dynamic verification if necessary. In a case study, we show the flexibility of our approach by applying it to a statically verified language that uses implicit dynamic frames to enable safe reasoning about shared mutable state.

Acknowledgments

I wish to thank my advisors Prof. Aldrich and Prof. Tanter for offering me this topic and for their patient assistance throughout the past few months. In moments of uncertainty, their remarks and thoughts guided me in the right direction.

Furthermore I want to thank Prof. Aldrich and Prof. Snelting for giving me the opportunity to participate in the interACT exchange program, which enabled me to write this thesis at Carnegie Mellon University.

I am grateful to the Baden-Württemberg Stiftung for financially supporting my stay in Pittsburgh.

Also I am very grateful to all my family and friends who encouraged and supported me throughout my years of study.

Contents

1 Introduction

Program verification aims to check a computer program against its specification. Automated methods require this specification to be formalized, e.g. using annotations in the source code. Common examples are method contracts, loop invariants and assertions.

Approaches to check whether program behavior complies with given annotations can be divided into two categories:

Static verification
: The program is not executed. Instead *formal methods* (like Hoare logic or separation logic) are used, trying to derive a proof for the given annotations.

  Drawbacks
  The syntax available for static verification is naturally limited by the underlying formal logic. Complex properties (e.g. including database lookups or library calls) might thus not be expressible, resulting in inability to prove subsequent goals. Furthermore, the logic itself might fail proving certain goals due to code complexity and undecidability in general. Using static verification usually requires rigorous annotation of the entire source code, as otherwise there might be too little information to find a proof. While fully annotating own code can be tedious (there are supporting tools), using unannotated libraries can become a problem: Even if it is possible to annotate the API afterwards, lacking the source code the verifier is unable to prove those annotations. In case the annotations are wrong this results in inconsistent proofs.

Dynamic verification
: The specification is turned into *runtime checks*, making sure that the program adheres to its specification during execution. Violations cause a runtime exception to be thrown, effectively preventing the program from entering a state that contradicts its specification. Note that in practice this approach is often combined with control flow based testing techniques to detect misbehavior as early as possible.

  Drawbacks
  Violations are only detected at runtime, with the risk of going unnoticed before software is released. To minimize this risk, testing methods are required, i.e. more time has to be spent after compilation. The usage of runtime checks naturally imposes a runtime overhead which is not always acceptable.

The goal of this work is to formalize "gradual verification", an approach that seamlessly combines static and dynamic verification in order to weaken or even avoid the above drawbacks by introducing an "unknown" formula `?`. The resulting system provides a con-

tinuum between traditional static and dynamic verification, meaning that both extremes are compatible with, but only special cases of the gradual verification system. Section 1.1 gives example scenarios of both static and dynamic verification suffering from their drawbacks, illustrating how gradual verification could avoid them.

1.1 Motivational Examples

1.1.1 Parameter Validation

The following Java example motivates the use of verification for parameter validation.

```
// spec: callable only if (this.balance >= amount)
void withdrawCoins(int amount)
{
    // business logic
    this.balance -= amount;
}
```

It is supposed to be impossible for `withdrawCoins` to create a negative balance, hence the restriction. To enforce this specification in Java, one may validate the parameter `amount` before entering the business logic.

```
void withdrawCoins(int amount)
{
    if (!(this.balance >= amount))
        throw new IllegalArgEx("expected this.balance >= amount");

    // business logic
    this.balance -= amount;
}
```

Note that this runtime check dynamically verifies a method contract that has `this.balance >= amount` as precondition. Naturally, the drawbacks of dynamic verification apply: Violations of the method contract are only detected at runtime, possibly go unnoticed for a long time and impose a runtime overhead.

Java even has a dedicated assertion syntax, simplifying dynamic verification:

```
void withdrawCoins(int amount)
{
    assert this.balance >= amount;

    // business logic
    this.balance -= amount;
}
```

Using additional tools, a more declarative approach is possible using JML syntax:

```
//@ requires this.balance >= amount;
void withdrawCoins(int amount)
{
    // business logic
    this.balance -= amount;
}
```

Using JML4c (see [27]) or similar tools, this annotation is compiled back into equivalent runtime assertions. However, the declarative approach also enables the use of static verification tools like ESC/Java (see [16]) to ensure at compile time that every call to `hasLegalDriver` adheres to the method contract. Static verification will only succeed if the precondition is provable at all call sites.

Consider the following example code:

```
acc.balance = 100;
acc.withdrawCoins(50);
acc.withdrawCoins(30);
```

Using dynamic verification methods, the validity of both calls is only established at runtime, although the calls clearly adhere to the contract. However, static verification most likely fails due to a lack of knowledge: The method contract does not state how `this.balance` is changed during a call to `withdrawCoins`, meaning that the precondition for the second call cannot be proved (we assume that the static verifier exclusively relies on the contract while reasoning about a method call). Annotating a postcondition

```
this.balance = old(this.balance) - amount
```

would solve that problem, resulting in overall success of the verification.

Gradual verification would realize a combined, best-effort approach without the need for full annotation: Static verification is used where possible (here: the first method call), dynamic verification where needed (here: the second method call). Where static verification proves a violation of the method contract, the code is *statically rejected*. Where static verification proves compliance with the method contract, the code is *statically accepted*, i.e. no runtime checks are emitted in order to enforce the contract. In contrast to purely static verification, code that is provable neither to violate nor to comply with the contract will be *optimistically accepted* and enhanced with runtime checks to guarantee compliance at runtime.

In above example, this behavior would be realized by treating the postcondition of `withdrawCoins` as unknown ("`?`"). This will allow the gradual verifier to assume any plausible formula for subsequent reasoning, e.g. one stating that the balance is still high enough for the second method call. Such treatment of `?` is backed with a runtime check, making sure that all assumptions made by the gradual verifier were valid. Bottom line, the example would pass gradual verification and a single runtime check (for `acc.balance >= 30`) prior to the second call would be inserted.

1.1.2 Limitations of Static Verification

The following example is written in a Java-like language with dedicated syntax for method contracts (similar to Eiffel [19] and Spec# [21]). We assume that this language is statically verified, i.e. successful static verification is a requirement for compilation.

The example shows the limitations of static verification using the Collatz sequence as an algorithm too complex to describe and decide concisely in a method contract:

```
int collatzIterations(int iter, int start)
    requires 1 <= start;
    ensures  1 <= result;
{
    // ...
}

int myRandom(int seed)
    requires 1 <= seed   && seed   <= 10000;
    ensures  1 <= result && result <= 3;     // not provable
{
    int result = collatzIterations(300, seed);
    // we know:        result ∈ { 1, 2, 4 }
    // verifier knows: 1 <= result

    if (result == 4) result = 3;
    return result;
}
```

The first method `collatzIterations` iterates the Collatz sequence a given number of times `iter`, starting with given value `start`. We assume that the only provable contract is that a positive start value results in a positive result.

The second method `myRandom` uses the Collatz sequence to generate a pseudo random number from given seed. It is known to the programmer that start values up to 10000 result in convergence of the Collatz sequence after less than 300 iterations. After mapping 4 to 3, we are thus given a number between 1 and 3, as described in the postcondition.

Unfortunately, the verifier cannot deduce this fact since the postcondition of `collatzIterations` only guarantees positive results, but no specific range of values. We could resort to dynamic methods to aid verification, realizing a “cast” of knowledge:

```
...
{
    int result = collatzIterations(300, seed);
    // we know:        result ∈ { 1, 2, 4 }
    // verifier knows: 1 <= result

    // knowledge "cast"
    if (!(result <= 4))
        throw new IllegalStateException("expected result <= 4");

    // verifier knows: 1 <= result && result <= 4

    if (result == 4) result = 3;
    return result;
}
```

This solution is not satisfying as it required additional work by the programmer to convince the static verifier. Furthermore, the solution is in an unintuitive location: The problem is caused by the weak postcondition of `collatzIterations`, yet it is addressed in `myRandom`.

Gradual verification allows enhancing the postcondition with "unknown" knowledge that can be interpreted by the verifier as an arbitrary plausible static formula. Again, appropriate runtime checks will be injected to guarantee that the static treatment of `?` is in fact valid:

```
int collatzIterations(int iter, int start)
    requires 1 <= start;
    ensures  1 <= result && ?;
{
    // ...
}

int myRandom(int seed)
    requires 1 <= seed   && seed   <= 10000;
    ensures  1 <= result && result <= 3;
{
    int result = collatzIterations(300, seed);
    // we know: result ∈ { 1, 2, 4 }

    // verifier allowed to
    //  assume 1 <= result && result <= 4
    //  from   1 <= result && ?
    // (adding runtime check)

    if (result == 4) result = 3;
    return result;
}
```

Note the `?` in the postcondition of `collatzIterations`. As indicated in the comments, the verifier may now make assumptions in order to prove the postcondition of `myRandom`.

1.2 Overview

Our approach is based on recent advances in formalizing gradual typing, specifically "Abstracting Gradual Typing" [9] by Garcia, Clark and Tanter. They describe a process called "gradualization" that uses the concept of abstract interpretation to define a gradual system in terms of a static one. Gradual typing emerged from drawbacks of static and dynamic type systems that are very similar to the drawbacks of verification systems outlined and illustrated above. This is no surprise from a theoretical perspective as type systems are a special case of program verification. These similarities motivated our idea of reinterpreting and adapting the gradual typing approach to the verification setting.

Chapter 2 introduces the concepts motivating and driving our approach. Furthermore it categorizes existing work that goes in a similar direction, pointing out how it differs from our work. In chapter 3 we describe our approach of gradualization in a generic way, meant to be used as a procedure or template for designing gradual verification systems. We follow that procedure in form of a case study in chapter 4, applying the approach to a statically verified language that uses implicit dynamic frames in order to enable safe static reasoning about shared mutable state. We conclude with an evaluation of our approach (chapter 5) and a conclusion (chapter 6). The appendix contains additional information like proofs for lemmas and theorems stated throughout this work. We have also implemented the gradually verified language developed in chapter 4 as an interactive web page (see section 5.2).

2 Background

Design-by-Contract, a term coined by Bertrand Meyer [20], is a paradigm aiming for verifiable source code, e.g. by adding method contracts and tightly integrating them with the compiler and runtime. Meyer implemented this concept in his programming language Eiffel [19], providing compiler support for generating runtime checks required for dynamic verification (also called runtime verification). Combining design-by-contract with static verification techniques led to the concept of "verified design-by-contract" [8].

Similar developments took place regarding Java and JML specifications. Static verification using theorem provers was investigated by Jacobs and Poll [13] and is implemented as part of ESC/Java [24]. Turning JML specifications into runtime assertion checks (RAC) to drive dynamic verification was described by Cheon and Leavens [6] and led up to the development of JML4c [27].

A more recent programming language with built-in support to express specifications, coming with both static and dynamic verification tools is Spec# [21]. Its compiler facilitates theorem provers for static verification and emits runtime checks for dynamic verification. It was developed further to cope with the challenges of concurrent object-orientation [4]. The concepts found their way to mainstream programming in the form of "Code Contracts" [17], a toolset deeply integrated with the Microsoft .NET framework and thus available in a variety of programming languages.

The limitations of both static and dynamic verification motivated a recent trend of using both approaches at the same time (as observable in the above programming languages). Automated theorem provers are used to perform static verification as a best effort service rather than a requirement for successful compilation. Their goal is to detect as many inconsistencies or contract violations as possible, but not necessarily find a proof for correctness. So even if compliance with annotations cannot be proved programs are "optimistically accepted". Additionally, dynamic verification is used to restore the guarantee that static verification no longer provides.

Recent research focused on combining both approaches in a more meaningful and complementary way by focusing dynamic verification and testing efforts specifically to code areas where static verification had less success. Christakis, Müller and Wüstholz [7] describe how programs can be annotated during static verification in order to prioritize certain tests over others or even prune the search space by aborting tests that lead to fully verified code.

Still, static and dynamic verification concepts are treated as independent concepts for the most part. The same was once true for static and dynamic type systems, before advances in formalizing gradual type systems seamlessly bridged the gap. Our goal is to achieve

the same for program verification, i.e. static and dynamic verification are no longer to be treated as independent concepts (that are then combined as intelligently as possible) but instead treated as one concept with different manifestations.

Note that Arlt et al. [1] mention "gradual verification", yet it is meant as the process of gradually increasing the coverage of static verification. The work describes a metric for estimating this coverage, giving the developer feedback while annotating programs. A similar metric arises automatically from our notion of gradual verification: The number of dynamic checks injected to guarantee compliance with annotations is a direct indication of where static verification has failed so far.

2.1 Gradual Typing

As this work is based on the advances in gradual typing, it is helpful to understand the developments in that area.

Gradual type systems originated from efforts to overcome limitations and drawbacks of purely static or dynamic type systems. Corresponding extensions were proposed for .NET by Meijer and Drayton [18], for Java by Gray et al. [11] and for Scheme by Bres et al. [5]. Siek and Taha provided a type-theoretic foundation, formalizing gradual typing for functional programming [29]. They describe a λ-calculus with optional type annotations, which is sound w.r.t. the simply-typed λ-calculus for fully annotated terms. Static and dynamic type checking are seamlessly combined by automatically inserting runtime checks (casts) where necessary. They later adapted their approach to object-based languages [28].

Based on their work, Wolff et al. introduced "gradual typestate" [34], circumventing the rigidity of static typestate checking. Schwerter, Garcia and Tanter developed a theory of gradual effect systems [3], making it possible to incrementally annotate and statically check effects by adding a notion of unknown effects. An implementation for gradual effects in Scala was later given by Toro and Tanter [33].

Siek et al. recently formalized refined criteria for gradual typing, called the "gradual guarantee" [30]. The gradual guarantee states that well typed programs will stay well typed when removing type annotations (the static part of the guarantee). It furthermore states that well typed programs evaluating to a value will evaluate to the same value (in lockstep) when removing type annotations (the dynamic part of the guarantee).

With "Abstracting Gradual Typing" (AGT) [9] Garcia, Clark and Tanter propose a new formal foundation for gradual typing. Their approach draws on the principles of abstract interpretation, defining a gradual type system in terms of an existing static one. The resulting system satisfies the gradual guarantee by construction. Subsequent work by Garcia and Tanter demonstrates the flexibility of AGT by applying the concept to a security-typed language, yielding a gradual security language [10], which in contrast to prior work does not require explicit security casts. Furthermore Lehmann and Tanter [14] applied the approach to refinement types, resulting in a gradual language that is able to deal with imprecise logical information and dependent function types.

2.2 Hoare Logic

We use Hoare logic [12] as the formal logic used for static verification. We assume that source code annotations can be translated into a corresponding Hoare logic.

Example 2.1 (Hoare Logic for Contract Verification)**.**
Consider a programming language with a built-in syntax for method contracts.

```
int getArea(int w, int h)
    requires 0 <= w && 0 <= h;
    ensures  result == w * h;
{
    return w * h;
}
```

This method contract can be translated into a Hoare triple.

$$\{\texttt{0 <= w \&\& 0 <= h}\}\ \texttt{return w * h;}\ \{\texttt{result == w * h}\}$$

The validity of this triple may then be verified using a sound Hoare logic for given programming language.

2.3 Implicit Dynamic Frames

Reasoning about programs using shared mutable data structures (the default in object oriented programming languages) is not possible using traditional Hoare logic.

Example 2.2 (Limitations of Hoare Logic)**.**
The following Hoare triple is not verifiable using a sound Hoare logic due to potential aliasing.

$$\{\texttt{(p1.age = 19)} \wedge \texttt{(p2.age = 19)}\}\ \texttt{p1.age++}\ \{\texttt{(p1.age = 20)} \wedge \texttt{(p2.age = 19)}\}$$

The problem is that `p1` and `p2` might be aliases, meaning that they reference the same memory. The increment operation could thus also affect `p2.age`, rendering the postcondition invalid.

We want to demonstrate gradual verification on a Java-like language in chapter 4, so we need a logic that is capable of dealing with mutable data structures.

Separation logic [26] is an extension of Hoare logic that explicitly tracks mutable data structures (i.e. heap references) and adds a "separating conjunction" to the formula syntax. In contrast to ordinary conjunction ($\wedge$), separating conjunction ($*$) ensures that both sides of the conjunction reference disjoint areas of the heap.

Example 2.3 (Power of Separation Logic)**.**
The following Hoare triple is verifiable using separation logic.

$\{(\texttt{p1.age} \mapsto 19) * (\texttt{p2.age} \mapsto 19)\}\ \texttt{p1.age++}\ \{(\texttt{p1.age} \mapsto 20) * (\texttt{p2.age} \mapsto 19)\}$

The separating conjunction in the precondition guarantees that `p1.age` and `p2.age` reference disjoint memory locations. The operation is therefore guaranteed to leave `p2.age` untouched.

Note that proving the "more powerful" precondition $(\texttt{p1.age} \mapsto 19) * (\texttt{p2.age} \mapsto 19)$ challenges the system to establish that `p1` and `p2` do not alias. In other words, more sophisticated reasoning and possibly stronger contracts annotated by the user are required to derive formulas containing the separating conjunction.

A drawback of separation logic is that formulas cannot contain heap-dependent expressions (e.g. `p1.age <= 19`) as they are not directly expressible using the explicit syntax (see above: values of memory locations are explicitly tracked). The concept of implicit dynamic frames (IDF) [31] addresses this issue by decoupling the permissions to access a certain heap location from assertions about its value. It introduces an "accessibility predicate" `acc(`*`loc`*`)` that represents the permission to access heap location *loc*.

Parkinson and Summers worked out the formal relationship between separation logic and IDF [25]. Specifically, they define a logic that syntactically subsumes and semantically agrees with both separation logic and implicit dynamic frames. Finally, they show that the separation logic fragment of their logic can be encoded into the implicit dynamic frames fragment while preserving semantics.

Example 2.4 (Power of Implicit Dynamic Frames)**.**
The following Hoare triple is verifiable using implicit dynamic frames.

```
{acc(p1.age) * acc(p2.age) * (p1.age = 19) * (p2.age <= 19)}
p1.age++
{acc(p1.age) * acc(p2.age) * (p1.age = 20) * (p2.age <= 19)}
```

The separating conjunction makes sure that the accessibility predicates `acc(p1.age)` and `acc(p2.age)` mention disjoint memory locations, whereas it has no further meaning for "traditional" predicates. Note how more complex predicates like `<=` are now expressible.

As formulas can mention heap locations in arbitrary predicates, the verifier must ensure that a formula contains accessibility predicates to all heap locations mentioned. This property of formulas is called "self-framing". The pre- and postcondition of example 2.4 are self-framed whereas the sub-formula `(p1.age = 20)` would not be.

Implicit dynamic frames was implemented as part of the Chalice verifier [15]. Chalice

is also the name of the underlying simple imperative programming language that has constructs for thread creation and thus relies on IDF for sound race-free reasoning. Chalice was also implemented as a front-end of the Viper toolset [23].

The static semantics of our example language in chapter 4 is based on the Hoare logic for Chalice given by Summers and Drossopoulou [32].

3 Gradualization of an Imperative Statically Verified Language

As motivated in chapter 1 gradual verification can be seen as an extension of both static and dynamic verification. Yet, the approach of "gradualization" (adapted from AGT) derives the gradual semantics in terms of static semantics. In this chapter we will thus describe our approach of deriving an imperative gradually verified language "**GVL**" starting with a generic imperative statically verified language "**SVL**". An informal description of how to tackle the opposite direction can be found in section 5.3.

Section 3.1 contains the description of "**SVL**" or rather the assumptions we make about it. In section 3.2 we describe the syntax extensions necessary to give programmers the opportunity to deviate from purely static annotations. We immediately give a meaning to the new "gradual" syntax, driven by the concepts of abstract interpretation. In section 3.5 we explain "lifting", a procedure adapting predicates and functions in order for them to deal with gradual parameters. To guide the following efforts to determine gradual semantics of **GVL**, we present gradual soundness in section 3.6 and point out the associated challenges. With the necessary tools for gradualization available, we apply them to the static semantics of **SVL** in section 3.7. Finally, we develop gradual dynamic semantics in section 3.8 and show how gradual soundness is achievable.

3.1 A Generic Imperative Statically Verified Language SVL

While aiming to give a general procedure for deriving imperative gradually verified languages, we have to make certain assumptions about **SVL** in order to concisely describe our approach and reason about its correctness. We believe that most imperative statically verified programming languages satisfy the following assumptions and thus qualify as starting point for our procedure.

Syntax

We assume the existence of the following two syntactic categories

$$s \in \textsc{Stmt}$$
$$\phi \in \textsc{Formula}$$

for statements and formulas.

We assume that there is a sequence operator ; such that

$$\forall s_1, s_2 \in \textsc{Stmt}.\ \ s_1 ; s_2 \in \textsc{Stmt}$$

Program State

Dynamic semantics (see below) are formalized as discrete transitions between program states. Therefore a program state contains all information necessary to evaluate expressions and determine the next program state.

We assume that ProgramState is the set of all possible program states in **SVL**.

Example 3.1 (Program State: Primitive language with integer variables)**.**

$$\textsc{ProgramState} = \underbrace{(\textsc{Var} \rightharpoonup \mathbb{Z})}_{\textit{variable memory}} \times \underbrace{\textsc{Stmt}}_{\textit{continuation}}$$

Example 3.2 (Program State: Language with stack)**.**

$$\textsc{ProgramState} = \bigcup_{i \in \mathbb{N}_+} \underbrace{\Big((\textsc{Var} \rightharpoonup \mathbb{Z}) \times \textsc{Stmt}\Big)^i}_{\textit{stack frame}}$$

Note how these examples use statements to represent continuations (the "remaining work"), necessary for the operational semantics to deduce a state transition. In general, a different representation may be used to encode this continuation (e.g. a lower-level machine language emitted by the compiler).

Dynamic Semantics

We assume that **SVL** has a structural operational semantics or small-step semantics. This semantics is formalized as $\cdot \longrightarrow \cdot : \textsc{ProgramState} \rightharpoonup \textsc{ProgramState}$, describing precisely how program state is updated. (Non-deterministic semantics would also be a potential area to investigate in the future, see section 6.1.4.)

As usual, taking a finite amount of steps is abbreviated as $\cdot \longrightarrow^* \cdot$. Also, we write $\pi \not\longrightarrow$ to state that π is a "stuck" state, i.e. that π is not in the domain of the partial small-step function.

Furthermore we define $\cdot \xrightarrow{s} \cdot \subseteq \textsc{ProgramState} \times \textsc{ProgramState}$ as a predicate indicating whether a program state is reachable from another program state by executing statement s. This is useful for abstracting from the precise implementation of both ProgramState (see examples 3.1 and 3.2) and the small-step semantics.

Example 3.3 (Execution of Given Statement)**.** Assume that program state is defined as in example 3.1. If $\pi_1 \in \textsc{ProgramState}$ is defined as

$$\pi_1 = \langle [x \mapsto 4, y \mapsto 2],\ \texttt{x := 8; y := x} \rangle$$

then $\pi_1 \xrightarrow{\texttt{x := 8}} \pi_2$ is supposed to hold for

$$\pi_2 = \langle [x \mapsto 8, y \mapsto 2],\ \texttt{y := x} \rangle$$

Note how the notation hides away implementation details like the number of steps taken. While the judgment $\pi_1 \longrightarrow^* \pi_2$ is also independent from the number of steps taken, it is unable to encode what is supposed to happen during those steps:

For π_1 defined as above

$$\pi_2 = \langle [x \mapsto 8, y \mapsto 8],\ \texttt{skip} \rangle$$

or

$$\pi_2 = \pi_1$$

would be valid instantiations.

We assume that there is a designated non-empty set

$$\textsc{ProgramStateFin} \subseteq \textsc{ProgramState}$$

of "final" states indicating regular termination of the program. W.l.o.g. we assume $\forall \pi \in \textsc{ProgramStateFin}.\ \pi \not\longrightarrow$, i.e. final states are stuck.

We assume that dynamic semantics of a sequence $s_1\texttt{;}\ s_2$ is implemented such that

$$\pi_1 \xrightarrow{s_1\texttt{;}\ s_2} \pi_3 \iff \exists \pi_2 \in \textsc{ProgramState}.\ \pi_1 \xrightarrow{s_1} \pi_2 \wedge \pi_2 \xrightarrow{s_2} \pi_3 \tag{3.1}$$

for all $\pi_1, \pi_3 \in \textsc{ProgramState}$.

Formula Semantics

While types restrict which values or expressions are valid for a certain variable, formulas restrict which program states are valid for a certain point during execution.

They are used for annotations like method contracts or invariants. For example, a method contract stating `arg > 4` as precondition is supposed to make sure that the method is only entered if `arg` evaluates to a value larger than 4. This is a restriction of program states corresponding to the very beginning of the method call.

We assume that we are given a computable predicate

$$\cdot \vDash \cdot \ \subseteq\ \textsc{ProgramState} \times \textsc{Formula}$$

that decides, whether a formula is satisfied given a concrete program state.

From this predicate we can derive a number of concepts:

Definition 3.4 (Denotational Formula Semantics $[\![\cdot]\!]$)**.**
Let $[\![\cdot]\!] : \textsc{Formula} \to \mathcal{P}^{\textsc{ProgramState}}$ *be defined as*

$$[\![\phi]\!] \stackrel{\mathsf{def}}{=} \{\ \pi \in \textsc{ProgramState} \mid \pi \vDash \phi\ \}$$

Definition 3.5 (Formula Satisfiability)**.**
A formula ϕ *is* ***satisfiable*** *iff*

$$[\![\phi]\!] \neq \emptyset$$

Let $\textsc{SatFormula} \subseteq \textsc{Formula}$ *be the set of satisfiable formulas.*

Definition 3.6 (Formula Implication)**.**
A formula ϕ_1 ***implies*** *formula* ϕ_2 *(written* $\phi_1 \Rightarrow \phi_2$*) iff*

$$[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$$

We will consistently use $\Longrightarrow$ *to denote logical implication, whereas* $\Rightarrow$ *exclusively denotes formula implication as defined here. (Note the different lengths of the implication arrows.)*

Definition 3.7 (Formula Equality)**.**
*Two formulas ϕ_1 and ϕ_2 are **equal** iff*

$$[\![\phi_1]\!] = [\![\phi_2]\!]$$

Lemma 3.8 (Partial Order of Formulas)**.**
The implication predicate is a partial order on Formula.

We assume that there is a largest element $\texttt{true} \in$ Formula with $[\![\texttt{true}]\!] =$ ProgramState.

With this semantics we can formalize the notion of semantically valid Hoare triples.
Definition 3.9 (Semantical Validity of Hoare Triples)**.**
*A Hoare triple $\{\phi_{pre}\}\ s\ \{\phi_{post}\}$ is **valid**, written $\vDash \{\phi_{pre}\}\ s\ \{\phi_{post}\}$ iff*

$$\forall \pi_{pre}, \pi_{post} \in \text{ProgramState}.\ \pi_{pre} \xrightarrow{s} \pi_{post} \implies (\pi_{pre} \vDash \phi_{pre} \implies \pi_{post} \vDash \phi_{post})$$

Lemma 3.10 (Compositional Validity of Hoare Triples)**.**

$$\forall \phi_1, \phi_2, \phi_3 \in \text{Formula},\ s_1, s_2 \in \text{Stmt}.\ \vDash \{\phi_1\}\ s_1\ \{\phi_2\} \ \wedge\ \vDash \{\phi_2\}\ s_2\ \{\phi_3\} \implies \vDash \{\phi_1\}\ s_1;\ s_2\ \{\phi_3\}$$

We assume that the set $(\!|\pi|\!) \stackrel{\text{def}}{=} \{\ \phi \in \text{Formula} \mid \pi \vDash \phi\ \}$ is a filter, i.e.

1. It is non-empty. This ensures that every program state is describable by at least one formula. This is the case as we demanded that `true` describes every program state.

2. If $\pi \vDash \phi_a$ and $\pi \vDash \phi_b$, then there exists $\phi \in$ Formula with

   $$\pi \vDash \phi \quad \wedge \quad \phi \Rightarrow \phi_a \quad \wedge \quad \phi \Rightarrow \phi_b$$

   Intuitively, this states that multiple formulas about the same program state can be combined. In case the formula syntax contains a logical conjunction operator, this criterion is met.

3. If $\pi \vDash \phi_a$ and $\phi_a \Rightarrow \phi_b$ then $\pi \vDash \phi_b$. This is true by definition of $\cdot \Rightarrow \cdot$ (see 3.6).

Static Semantics

We assume that there is a Hoare logic

$$\vdash \{\cdot\} \cdot \{\cdot\} \quad \subseteq \quad \text{Formula} \times \text{Stmt} \times \text{Formula}$$

describing which programs (together with pre- and postconditions about the program state) are accepted.

In practice, this predicate might also have further parameters. For instance, a statically typed language might require a type context to safely deduce

$$\texttt{x : int} \vdash \{\texttt{true}\}\ \texttt{x := 3}\ \{\texttt{(x = 3)}\}$$

As we will see later, further parameters are generally irrelevant for and immune to gradualization, so it is reasonable to omit them for now.

Lemma 3.11 (Hoare Logic Monotonicity)**.**
We assume that the Hoare predicate is monotonic in the precondition w.r.t. implication:

$$\begin{aligned} &\forall s \in \textsc{Stmt}. \\ &\forall \phi_1, \phi_2 \in \textsc{Formula}. \\ &\quad \forall \phi_1' \in \textsc{Formula}.\ (\phi_1 \Rightarrow \phi_2) \quad \wedge \quad \vdash \{\phi_1\}\ s\ \{\phi_1'\} \\ \Longrightarrow\ &\exists \phi_2' \in \textsc{Formula}.\ (\phi_1' \Rightarrow \phi_2') \quad \wedge \quad \vdash \{\phi_2\}\ s\ \{\phi_2'\} \end{aligned}$$

Intuitively, this means that more knowledge about the initial program state cannot result in a loss of information about the final state.

Soundness

We expect that given static semantics are sound w.r.t. given dynamic semantics. This means that Hoare triples judged valid by the Hoare logic are also valid semantically:
Definition 3.12 (Soundness)**.**
Let $\phi_1, \phi_2 \in \textsc{Formula}$ *and* $s \in \textsc{Stmt}$.
If $\vdash \{\phi_1\}\ s\ \{\phi_2\}$ *then* $\vDash \{\phi_1\}\ s\ \{\phi_2\}$

3.2 Gradual Formulas

The fundamental concept of gradual verification is the introduction of a wildcard formula `?` into the formula syntax. Therefore, the first difference between **GVL** and **SVL** is an extension of the set of formulas Formula, resulting in a superset of gradual formulas $\widetilde{\textsc{Formula}} \supset \textsc{Formula}$ with `?` $\in \widetilde{\textsc{Formula}}$ but `?` $\notin \textsc{Formula}$. The gradual verifier is supposed to succeed in the presence of the wildcard, if it is plausible that there exists a static formula that would make a static verifier succeed.

Example 3.13 (Wildcard as Precondition)**.**

```
int getFour(int i)
    requires ?;
    ensures  result = 4;
{
    i := i + 1;
    return i;
}
```

The gradual verifier is expected to successfully verify this method contract since there exists a static formula (`i == 3`), that would let static verification succeed.

On the other hand, a gradual verifier should reject the following method contract as there is no plausible (i.e. satisfiable) instantiation of `?` that would make static verification succeed.

Example 3.14 (Implausible Wildcard as Precondition)**.**

```
int nonSense(int i)
    requires ?;
    ensures  result == result + 1;
{
    return i;
}
```

This intuition about `?` is formalized in the following sections. We decorate gradual formulas $\widetilde{\phi} \in \widetilde{\text{Formula}}$ to distinguish them from formulas drawn from Formula.

Using the concepts of abstract interpretation, we want to give meaning to gradual formulas by mapping them back to a set of static formulas (called "concretization"). This way we can reason about a gradual formula by applying preexisting static reasoning to the formula's concretization. For example, we want a program state π to satisfy a gradual formula $\widetilde{\phi}$ iff π satisfies (at least) one of the formulas drawn from the concretization of $\widetilde{\phi}$ (this example is formalized in lemma 3.29).

Definition 3.15 (Concretization)**.**
Let $\gamma : \widetilde{\text{Formula}} \to \mathcal{P}(\text{Formula})$ *be defined as*

$$\gamma(\phi) = \{\ \phi\ \}$$
$$\gamma(?) = \text{SatFormula}$$

Static formulas are mapped to a singleton set containing just them. This reflects our goal to preserve the meaning of static formulas in the gradual setting. The wildcard is mapped to the set of all satisfiable formulas, reflecting the idea of treating it as any plausible static formula.

Note that this definition of γ assumes that `?` is the only addition to $\widetilde{\text{Formula}}$. In fact, this is only one possible way to realize $\widetilde{\text{Formula}}$. In the following two sections we will further analyze both this and a more powerful alternative.

3.2.1 Total Unknown

As motivated previously, the most straight forward way to extend the syntax is by simply adding `?` as a dedicated formula:

$$\widetilde{\phi} ::= \phi \mid ?$$

This is analogous to how some gradually typed languages are realized (e.g. `dynamic`-type in C# 4.0 and upward).

We call this approach "total unknown" as programmers cannot express any additional static knowledge they might have in the presence of `?`. This poses a limitation that can be overcome using the "bounded unknown" approach shown in the following section.

3.2.2 Bounded Unknown

To allow combining wildcards with static knowledge, we might view `?` merely as an unknown conjunctive term within a formula (as motivated and defined by Lehmann and Tanter in their work on gradual refinement types [14]):

$$\widetilde{\phi} ::= \phi \mid \phi \ \wedge \ \texttt{?}$$

We pose $\texttt{?} \stackrel{\text{def}}{=} \texttt{true} \ \wedge \ \texttt{?}$.

Intuitively, we expect $\phi \ \wedge \ \texttt{?}$ to be a placeholder for a formula that implies ϕ. This intuition is formalized using concretization.

Definition 3.16 (Concretization)**.**
Let $\gamma : \widetilde{\text{Formula}} \to \mathcal{P}(\text{SatFormula})$ *be defined as*

$$\gamma(\phi) = \{ \ \phi \ \}$$
$$\gamma(\phi \ \wedge \ \texttt{?}) = \{ \ \phi' \in \text{SatFormula} \mid \phi' \Rightarrow \phi \ \}$$

Note that $\gamma(\texttt{?}) = \gamma(\texttt{true} \ \wedge \ \texttt{?}) = \{ \ \phi' \in \text{SatFormula} \mid \phi' \Rightarrow \texttt{true} \ \} = \text{SatFormula}$. The approach is thus compatible with and strictly superior to the previous one (see section 3.2.1).

3.2.3 Precision

Comparing gradual formulas (e.g. `x = 3`, `x = 3` $\wedge$ `?`, `?`) gives rise to a notion of "precision". Intuitively, `x = 3` is more precise than `x = 3` $\wedge$ `?` which is more precise than `?`. Using concretization, we can formalize this intuition.

Definition 3.17 (Formula Precision)**.**

$$\widetilde{\phi_a} \sqsubseteq \widetilde{\phi_b} \quad \iff \quad \gamma(\widetilde{\phi_a}) \subseteq \gamma(\widetilde{\phi_b})$$

Read: Formula $\widetilde{\phi_a}$ *is "at least as precise as"* $\widetilde{\phi_b}$*.*

The strict version $\sqsubset$ is defined accordingly.

3.2.4 Generalization

In general, we will make the following assumptions:

1. $\text{Formula} \subset \widetilde{\text{Formula}}$
2. $\texttt{?} \notin \text{Formula}$

3. $\texttt{?} \in \widetilde{\text{Formula}}$

4. $\gamma(\texttt{?}) = \text{SatFormula}$

5. $\gamma(\phi) = \{\ \phi\ \}$ for all $\phi \in \text{Formula}$

6. If $\widetilde{\phi} \in \widetilde{\text{Formula}}$ is partially unknown, then $\gamma(\widetilde{\phi})$ is closed over implication, i.e.

$$\forall \phi \in \gamma(\widetilde{\phi}), \phi' \in \text{SatFormula}.\ (\phi' \Rightarrow \phi) \implies \phi' \in \gamma(\widetilde{\phi})$$

The gradual syntax extensions introduced in sections 3.2.1 and 3.2.2 comply with the above assumptions.

3.3 Gradual Statements

Formulas play a role in some statements, so extending their syntax may also affect the syntax of statements. A common example are assertion statements `assert` ϕ, which can now be extended to `assert` $\widetilde{\phi}$. Note however, that having a gradual formula syntax available does not necessarily mean that all statements *have* to adopt it.

A more complex example affected by gradualization of formulas is a call statement m`()` in presence of method contracts.

Example 3.18 (Gradual Call Statement)**.**
Although not directly visible, the (static and dynamic) semantics of a call statement is affected by the contract of m, consisting of pre- and postcondition. One can think of m as a reference to or syntactic sugar for some method definition including a method contract. Note that in practice such method definitions usually reside in a "program context" that the static and dynamic semantics are parameterized with.

As the full meaning of such a statement is unknown without context, it is hard to reason about it abstractly. W.l.o.g. we will thus think of m as syntactic sugar for

`assert` $\phi_{m_{pre}}$`;`
`// body of` m
`assume` $\phi_{m_{post}}$`;`

As one of the motivations for gradual verification is allowing gradual method contracts, it makes sense to extend the syntax accordingly. This means that the syntax of the desugared call statement is affected:

`assert` $\widetilde{\phi_{m_{pre}}}$`;`
`// body of` m
`assume` $\widetilde{\phi_{m_{post}}}$`;`

In general, the statement syntax is extended, resulting in a superset $\widetilde{\text{STMT}} \supseteq \text{STMT}$ of gradual statements. Note that the superset is induced merely by allowing $\widetilde{\text{FORMULA}}$ instead of FORMULA in certain places (chosen by the gradual language designer). We give meaning to gradual statements using a concretization function.

Definition 3.19 (Concretization of Gradual Statements)**.**
Let $\gamma_s : \widetilde{\text{STMT}} \to \mathcal{P}(\text{STMT})$ *be defined as*

$$\gamma_s(\widetilde{s}) = \{\ s \in \text{STMT} \mid s \text{ is } \widetilde{s} \text{ with all gradual formulas replaced by some concretizations}\ \}$$

Definition 3.20 (Precision of Gradual Statement)**.**
Let $\cdot \sqsubseteq_s \cdot \subseteq \widetilde{\text{STMT}} \times \widetilde{\text{STMT}}$ *be a predicate defined as*

$$\widetilde{s_a} \sqsubseteq_s \widetilde{s_b} \quad \iff \quad \gamma_s(\widetilde{s_a}) \subseteq \gamma_s(\widetilde{s_b})$$

The notion of gradual statements will become important for the gradualized semantics of **GVL**.

3.4 Gradual Program State

Recall that program state has a notion of continuation, see section 3.1 for examples. As the set of possible statements has been augmented from STMT to $\widetilde{\text{STMT}}$, this notion might have to be augmented as well in order to allow encoding the additional statements.

This augmentation leads to a superset $\widetilde{\text{PROGRAMSTATE}} \supseteq \text{PROGRAMSTATE}$ of gradual program states.

> **Example 3.21** (Gradual Program State)**.**
>
> $$\text{PROGRAMSTATE} \;=\; (\text{VAR} \rightharpoonup \mathbb{Z}) \;\times\; \text{STMT}$$
>
> is extended to
>
> $$\widetilde{\text{PROGRAMSTATE}} \;=\; (\text{VAR} \rightharpoonup \mathbb{Z}) \;\times\; \widetilde{\text{STMT}}$$

Concretization and precision are defined accordingly, drawing on concretization of gradual statements.

Lemma 3.22 (Gradual Program State Does Not Affect Formula Semantics)**.**
We demand that formula semantics are not affected by gradualization of the program state:

$$\forall \phi \in \text{FORMULA}, \widetilde{\pi} \in \widetilde{\text{PROGRAMSTATE}}, \pi \in \gamma_\pi(\widetilde{\pi}).\ \ \widetilde{\pi} \vDash \phi \iff \pi \vDash \phi$$

This is trivially the case if evaluation does not depend on the (now gradual) continuation in the first place.

3.5 Lifting Predicates and Functions

The Hoare logic of **SVL** is a ternary predicate

$$\vdash \{\cdot\} \cdot \{\cdot\} \subseteq \textsc{Formula} \times \textsc{Stmt} \times \textsc{Formula}$$

Since **GVL** operates on gradual formulas and gradual statements, the gradualized Hoare logic is expected to have signature

$$\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\} \subseteq \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Stmt}} \times \widetilde{\textsc{Formula}}$$

Similarly, the small-step semantics

$$\cdot \longrightarrow \cdot : \textsc{ProgramState} \rightharpoonup \textsc{ProgramState}$$

has to be extended to deal with gradual program state:

$$\cdot \mathrel{\widetilde{\longrightarrow}} \cdot : \widetilde{\textsc{ProgramState}} \rightharpoonup \widetilde{\textsc{ProgramState}}$$

Formal semantics are sometimes defined inductively, meaning that they are defined in terms of further predicates or functions (e.g. implication between formulas). These functions will have new signatures as well in order to deal with the extended syntax of **GVL**. This section will present a procedure called "gradual lifting", which formalizes this adaptation of predicates and functions.

Our rules for gradual lifting rely merely on the existence of a concretization function and a notion of precision. We will thus restrict our formalizations and explanations to (gradual) formulas, the same ideas are directly applicable to other gradualized sets like statements and program states.

3.5.1 Gradual Guarantee of Verification

Since lifted predicates and functions directly affect the gradual semantics of **GVL**, they must adhere to certain rules in order to be sound. What soundness means is a direct consequence of the gradual guarantee for gradual verification systems, which we derive from the gradual guarantee for gradual type systems by Siek et al. [30].

For simplicity we will simply call programs "acceptable" if they are successfully verifiable by the gradual verifier.

Definition 3.23 (Gradual Guarantee (Static Semantics))**.**
Acceptable programs remain acceptable when reducing precision of any formula.

Definition 3.24 (Gradual Guarantee (Dynamic Semantics))**.**
Acceptable programs with a particular observational behavior (termination, values of variables, output, etc.) will have the same observational behavior after reducing precision of any formula.

3.5.2 Lifting Predicates

In this section, we assume that we are dealing with a binary predicate $P \subseteq \textsc{Formula} \times \textsc{Formula}$ and want to obtain a lifted predicate $\widetilde{P} \subseteq \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}}$. The concepts are directly applicable to predicates with different arity or with additional non-formula parameters.

We identify the following rules:

Introduction

We demand compatibility of the semantics of **GVL** with the semantics of **SVL**. In other words, switching to the gradual system may never "break the code". A predicate $\widetilde{P}$ that is part of the gradual semantics must thus satisfy:

If $P(\phi_1, \phi_2)$ then $\widetilde{P}(\phi_1, \phi_2)$ for all $\phi_1, \phi_2 \in \textsc{Formula}$.

Or equivalently, using set notation: $P \subseteq \widetilde{P}$

Monotonicity

In order to satisfy the gradual guarantee, the semantics of **GVL** must be immune to reduction of precision. A predicate $\widetilde{P}$ that is part of the gradual semantics must thus remain satisfied when reducing the precision of arguments

If $\widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2})$ then $\widetilde{P}(\widetilde{\phi'_1}, \widetilde{\phi'_2})$ for all $\widetilde{\phi_1}, \widetilde{\phi_2}, \widetilde{\phi'_1}, \widetilde{\phi'_2} \in \widetilde{\textsc{Formula}}$ with $\widetilde{\phi_1} \sqsubseteq \widetilde{\phi'_1}$ and $\widetilde{\phi_2} \sqsubseteq \widetilde{\phi'_2}$.

Definition 3.25 (Soundness of Predicate Lifting)**.**
*A lifted predicate is **sound** if it is closed under the above rules.*

Note that soundness only gives a lower bound for the predicate:

$$\widetilde{P} = \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}}$$

is a sound predicate lifting of any binary predicate

$$P \subseteq \textsc{Formula} \times \textsc{Formula}$$

This observation motivates an additional notion of optimality.

Definition 3.26 (Optimality of Predicate Lifting)**.**
*A sound lifted predicate is **optimal** if it is the smallest set closed under the above rules.*

The definition of optimal predicate lifting coincides with the definition of "consistent predicate lifting" given by AGT [9].

Lemma 3.27 (Equivalence with Consistent Predicate Lifting (AGT))**.**
Let $\widetilde{P} \subseteq \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}}$ be defined as

$$\widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2}) \quad \overset{\mathsf{def}}{\iff} \quad \exists \phi_1 \in \gamma(\widetilde{\phi_1}), \phi_2 \in \gamma(\widetilde{\phi_2}).\ P(\phi_1, \phi_2)$$

Then $\widetilde{P}$ is an optimal lifting of P.

This is an intriguing observation since different approaches were used to end up with the same definition: AGT immediately defines consistent predicate lifting as above, arguing that it reflects the intuition behind gradual formulas (as placeholders for plausible static formulas). Therefore $\widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2})$ is supposed to hold if it is plausible that there exists an instantiation satisfying the static predicate. This intuition is directly formalized, using concretization to map from gradual formulas to plausible static formulas.

Realizing that consistent lifting is actually not necessary for ending up with a sound gradual verification system, we took a different, more general approach to define lifting. Identifying the bare minimum of requirements (dictated by the gradual guarantee and compatibility with the static system) we ended up with our definition of sound lifting. The optional notion of optimality bridges the gap between both approaches.

It is worth noting that optimality may not only be hard to achieve sometimes, but might not even be desirable under all circumstances as outlined in section 6.1.

3.5.2.1 Examples

Lemma 3.28 (Optimal Lifting of Implication)**.**
Let $\widetilde{\text{Formula}}$ *be extended using the "total unknown" approach (see section 3.2.1).*
Let $\cdot \mathrel{\widetilde{\Rightarrow}} \cdot \subseteq \widetilde{\text{Formula}} \times \widetilde{\text{Formula}}$ *be defined inductively as*

$$\frac{\phi_1 \Rightarrow \phi_2}{\phi_1 \mathrel{\widetilde{\Rightarrow}} \phi_2}\ \widetilde{\text{ImplStatic}}$$

$$\frac{\phi \in \text{SatFormula}}{? \mathrel{\widetilde{\Rightarrow}} \phi}\ \widetilde{\text{ImplGrad1}}$$

$$\frac{}{\widetilde{\phi} \mathrel{\widetilde{\Rightarrow}} ?}\ \widetilde{\text{ImplGrad2}}$$

Then $\cdot \mathrel{\widetilde{\Rightarrow}} \cdot$ *is an optimal lifting of* $\cdot \Rightarrow \cdot$ *.*

Lemma 3.29 (Optimal Lifting of Evaluation)**.**
Let $\widetilde{\text{Formula}}$ *be extended using the "bounded unknown" approach (see section 3.2.2).*
Let $\cdot \mathrel{\widetilde{\vDash}} \cdot \subseteq \text{ProgramState} \times \widetilde{\text{Formula}}$ *be defined inductively as*

$$\frac{\pi \vDash \phi}{\pi \mathrel{\widetilde{\vDash}} \phi}\ \widetilde{\text{EvalStatic}}$$

$$\frac{\pi \vDash \phi}{\pi \mathrel{\widetilde{\vDash}} \phi \wedge ?}\ \widetilde{\text{EvalGrad}}$$

Then $\cdot \mathrel{\widetilde{\vDash}} \cdot$ *is an optimal lifting of* $\cdot \vDash \cdot$ *.*

Note that the definition of lifted evaluation was lifted only w.r.t. the second parameter. There is no point in lifting evaluation w.r.t. the program state since gradual program state has no impact on evaluation (see lemma 3.22).

Lemma 3.30 (Sound Lifting of Composite Predicate)**.**
Let $P, Q \subseteq \textsc{Formula} \times \textsc{Formula}$ *be arbitrary binary predicates.*
Let $(P \circ Q) \subseteq \textsc{Formula} \times \textsc{Formula}$ *be defined as*

$$(P \circ Q)(\phi_1, \phi_3) \iff^{\mathsf{def}} \exists \phi_2 \in \textsc{Formula}.\ P(\phi_1, \phi_2) \wedge Q(\phi_2, \phi_3)$$

Let $\widetilde{(P \circ Q)} \subseteq \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}}$ *be defined as*

$$\widetilde{(P \circ Q)} \stackrel{\mathsf{def}}{=} \widetilde{P} \circ \widetilde{Q}$$

with sound liftings $\widetilde{P}$ *and* $\widetilde{Q}$*.*

Then $\widetilde{(P \circ Q)}$ *is a sound lifting of* $(P \circ Q)$*, i.e. "piecewise" lifting of composite predicates is allowed. Optimality of* $\widetilde{P}$ *and* $\widetilde{Q}$ *does not imply optimality of* $\widetilde{(P \circ Q)}$*.*

Lemma 3.31 (Sound Lifting of Conjunctive Predicate)**.**
Let $P, Q \subseteq \textsc{Formula}$ *be arbitrary binary predicates.*
Let $(P \wedge Q) \subseteq \textsc{Formula}$ *be defined as*

$$(P \wedge Q)(\phi) \iff^{\mathsf{def}} P(\phi) \wedge Q(\phi)$$

Let $\widetilde{(P \wedge Q)} \subseteq \widetilde{\textsc{Formula}}$ *be defined as*

$$\widetilde{(P \wedge Q)} \stackrel{\mathsf{def}}{=} \widetilde{P} \wedge \widetilde{Q}$$

with sound liftings $\widetilde{P}$ *and* $\widetilde{Q}$*.*

Then $\widetilde{(P \wedge Q)}$ *is a sound lifting of* $(P \wedge Q)$*, i.e. term-wise lifting of conjunctive predicates is allowed. Optimality of* $\widetilde{P}$ *and* $\widetilde{Q}$ *does not imply optimality of* $\widetilde{(P \wedge Q)}$*.*

Lemma 3.32 (Optimal Lifting of Disjunctive Predicate)**.**
Let $P, Q \subseteq \textsc{Formula}$ *be arbitrary binary predicates.*
Let $(P \vee Q) \subseteq \textsc{Formula}$ *be defined as*

$$(P \vee Q)(\phi) \iff^{\mathsf{def}} P(\phi) \vee Q(\phi)$$

Let $\widetilde{(P \vee Q)} \subseteq \widetilde{\textsc{Formula}}$ *be defined as*

$$\widetilde{(P \vee Q)} \stackrel{\mathsf{def}}{=} \widetilde{P} \vee \widetilde{Q}$$

with sound liftings $\widetilde{P}$ *and* $\widetilde{Q}$*.*

Then $\widetilde{(P \vee Q)}$ *is a sound lifting of* $(P \vee Q)$*, i.e. term-wise lifting of disjunctive predicates is allowed. Optimality of* $\widetilde{P}$ *and* $\widetilde{Q}$ *implies optimality of* $\widetilde{(P \vee Q)}$*.*

3.5.3 Lifting Functions

In this section, we assume that we are dealing with a *total* function $f : \textsc{Formula} \to \textsc{Formula}$. Partial functions are dealt with in section 3.5.3.2.

The following concepts are directly applicable to functions with higher arity.

Introduction

We assume that f plays a role in the semantics of **SVL**, i.e. interacts with other functions or predicates. With predicate lifting we made sure to design a gradual verification system that is "immune" to reduction of precision. Therefore, when replacing function f with its gradual lifting $\widetilde{f}$, we expect the result to be the same or less precise.

$$\forall \phi \in \textsc{Formula}.\ f(\phi) \sqsubseteq \widetilde{f}(\phi)$$

Monotonicity

Reducing precision of a parameter may only result in a loss of precision of the result. In other words, the function must be monotonic w.r.t. $\sqsubseteq$.

$$\forall \widetilde{\phi_1}, \widetilde{\phi_2} \in \widetilde{\textsc{Formula}}.\ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_2} \implies \widetilde{f}(\widetilde{\phi_1}) \sqsubseteq \widetilde{f}(\widetilde{\phi_2})$$

Definition 3.33 (Sound Function Lifting)**.**
*A lifted function is **sound** if it adheres to the above rules.*

Note that the rules for sound lifting only give a lower bound for the gradual return values. Thus a function $\widetilde{f} : \widetilde{\textsc{Formula}} \to \widetilde{\textsc{Formula}}$ constantly returning `?` is a sound lifting of any function $f : \textsc{Formula} \to \textsc{Formula}$. This observation motivates an additional notion of optimality.

Definition 3.34 (Optimal Function Lifting)**.**
*A sound lifted function is **optimal** if its return values are at least as precise as the return values of any other sound lifted function.*

Again, the definition of optimal function lifting coincides with the definition of "consistent function lifting" given by AGT.

Lemma 3.35 (Equivalence with Consistent Function Lifting (AGT))**.**
Let $\alpha : \mathcal{P}(\textsc{Formula}) \rightharpoonup \widetilde{\textsc{Formula}}$ be a partial function such that $\langle \gamma, \alpha \rangle$ is a $\{\overline{f}\}$-partial Galois connection (see appendix A.1, definition A.1).
Let $\widetilde{f} : \widetilde{\textsc{Formula}} \to \widetilde{\textsc{Formula}}$ be defined as

$$\widetilde{f}(\widetilde{\phi}) \stackrel{\mathsf{def}}{=} \alpha(\overline{f}(\gamma(\widetilde{\phi})))$$

where $\overline{f}$ means that f is applied to every element of the set. Then $\widetilde{f}$ is an optimal lifting of f.

3.5.3.1 Examples

Lemma 3.36 (Sound Lifting of Composed Function)**.**
Let $g, f : \textsc{Formula} \to \textsc{Formula}$ *be arbitrary functions.*

Let $\widetilde{(g \circ f)} : \widetilde{\textsc{Formula}} \to \widetilde{\textsc{Formula}}$ *be defined as*

$$\widetilde{(g \circ f)} \stackrel{\mathsf{def}}{=} \widetilde{g} \circ \widetilde{f}$$

with sound liftings $\widetilde{g}$ *and* $\widetilde{f}$*.*

Then $\widetilde{(g \circ f)}$ *is a sound lifting of* $(g \circ f)$*, i.e. "piecewise" lifting of composed functions is allowed. Optimality of* $\widetilde{g}$ *and* $\widetilde{f}$ *does not imply optimality of* $\widetilde{(g \circ f)}$*.*

3.5.3.2 Lifting Partial Functions

Semantics can be defined in terms of partial functions, the small-step semantics of **SVL** even is a partial function itself. We derive rules for lifting partial functions by decomposing them into a total function (lifted as per 3.5.3) and a predicate indicating the domain of the partial function (lifted as per 3.5.2).

Definition 3.37 (Partial Function Decomposition)**.**
Let $f : \textsc{Formula} \rightharpoonup \textsc{Formula}$ *be a partial function. Then we define a decomposition* $\langle F, f' \rangle \in \mathcal{P}^{\textsc{Formula}} \times (\textsc{Formula} \to \textsc{Formula})$ *where*

$$\begin{aligned} F &= \mathsf{dom}(f) \\ f'(\phi) &= f(\phi) \quad \textit{if } F(\phi) \\ f'(\phi) &= \texttt{true} \quad \textit{otherwise} \end{aligned}$$

Note that we treat F *as a predicate. Then* f *can be defined in terms of* F *and* f'*:*

$$\begin{aligned} f(\phi) &= f'(\phi) \quad \textit{if } F(\phi) \\ f &\quad \textit{undefined otherwise} \end{aligned}$$

We define lifting of partial functions as decomposing the function, lifting the parts and then recomposing them. This process is equivalent to the following rules.

Introduction

$$\forall \phi \in \textsc{Formula} \cap \mathsf{dom}(f).\ f(\phi) \sqsubseteq \widetilde{f}(\phi)$$

We write $\mathsf{dom}(f)$ to denote the domain of function f.

Monotonicity

$$\forall \widetilde{\phi_1}, \widetilde{\phi_2} \in \widetilde{\textsc{Formula}}.\ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_2} \wedge \widetilde{\phi_1} \in \mathsf{dom}(\widetilde{f}) \implies \widetilde{f}(\widetilde{\phi_1}) \sqsubseteq \widetilde{f}(\widetilde{\phi_2})$$

Soundness and optimality are defined as in sections 3.5.2 and 3.5.3.

3.5.4 Generalized Lifting

The previous sections describe how lifting is performed in order to deal with $\widetilde{\text{Formula}}$ instead of Formula. In general, the same rules apply to any gradual extension of an existing set that comes with a concretization function, e.g. $\widetilde{\text{Stmt}}$ or $\widetilde{\text{ProgramState}}$.

For instance, the signature of Hoare rules contains Stmt and can therefore be lifted w.r.t. this parameter using the definitions in section 3.3.

3.6 Gradual Soundness vs Gradual Guarantee

With the notion of sound gradual lifting we have the tools to gradualize the semantics of **SVL**, resulting in gradual semantics of **GVL**. More specifically, predicate lifting is applied to the Hoare logic of **SVL**, resulting in a gradual Hoare logic

$$\widetilde{\vdash}\ \{\cdot\}\cdot\{\cdot\}\ \subseteq\ \widetilde{\text{Formula}} \times \widetilde{\text{Stmt}} \times \widetilde{\text{Formula}}$$

(see section 3.7). Furthermore, function lifting is applied to the small-step semantics of **SVL**, resulting in gradual small-step semantics

$$\cdot\ \widetilde{\longrightarrow}\ \cdot : \widetilde{\text{ProgramState}} \rightharpoonup \widetilde{\text{ProgramState}}$$

(see section 3.8). These semantics will by construction be compatible with the semantics of **SVL** and comply with the gradual guarantee.

Note however that there is an additional requirement concerning the correct interplay between Hoare logic and small-step semantics, namely soundness. The following definitions attempt to define soundness of **GVL** analogous to soundness of **SVL** (see definition 3.9 and 3.12).

Definition 3.38 (Semantical Validity of Gradual Hoare Triples)**.**
A Hoare triple $\{\widetilde{\phi_{pre}}\}\ \widetilde{s}\ \{\widetilde{\phi_{post}}\}$ *is* ***valid****, written* $\widetilde{\models}\ \{\widetilde{\phi_{pre}}\}\ \widetilde{s}\ \{\widetilde{\phi_{post}}\}$ *iff*

$$\forall \widetilde{\pi_{pre}}, \widetilde{\pi_{post}} \in \widetilde{\text{ProgramState}}.\ \widetilde{\pi_{pre}} \overset{\widetilde{s}}{\widetilde{\longrightarrow}} \widetilde{\pi_{post}} \implies (\widetilde{\pi_{pre}} \mathrel{\widetilde{\models}} \widetilde{\phi_{pre}} \implies \widetilde{\pi_{post}} \mathrel{\widetilde{\models}} \widetilde{\phi_{post}})$$

Lemma 3.39 (Compositional Validity of Gradual Hoare Triples)**.**

$$\forall \widetilde{\phi_1}, \widetilde{\phi_2}, \widetilde{\phi_3} \in \widetilde{\text{Formula}},\ \widetilde{s_1}, \widetilde{s_2} \in \widetilde{\text{Stmt}}.\ \widetilde{\models}\ \{\widetilde{\phi_1}\}\ \widetilde{s_1}\ \{\widetilde{\phi_2}\}\ \wedge\ \widetilde{\models}\ \{\widetilde{\phi_2}\}\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}$$
$$\implies \models \{\widetilde{\phi_1}\}\ \widetilde{s_1}\texttt{;}\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}$$

Definition 3.40 ($\widetilde{\text{Soundness}}$ (First Iteration))**.**
Let $\widetilde{\phi_1}, \widetilde{\phi_2} \in \widetilde{\text{Formula}}$ *and* $\widetilde{s} \in \widetilde{\text{Stmt}}$.
If $\widetilde{\vdash}\ \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}$ *then* $\widetilde{\models}\ \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}$

Note that $\widetilde{\models}\ \{\cdot\}\cdot\{\cdot\}$ is not a sound gradual lifting of $\models\ \{\cdot\}\cdot\{\cdot\}$. A gradual lifting would declare the Hoare triple $\{\texttt{?}\}\ \texttt{x := 3}\ \{\texttt{(y = 4)}\}$ valid due to the existence of a valid concretization, e.g. $\{\texttt{(y = 4)}\}\ \texttt{x := 3}\ \{\texttt{(y = 4)}\}$). However, this triple is clearly not

valid semantically since the postcondition is not guaranteed for all executions satisfying the precondition (`?` is always satisfied). Recall that gradual lifting was introduced in order to comply with the expectations a programmer would have when using a gradual verification system. The validity predicate $\widetilde{\vDash} \{\cdot\} \cdot \{\cdot\}$ on the other hand is defined semantically and therefore not affected by the reasoning behind gradual lifting.

Unfortunately, the different concepts collide in gradual soundness as defined in definition 3.40. On the one hand gradual Hoare logic must comply with the gradual guarantee and thus verify $\widetilde{\vdash} \{?\}\ \texttt{x := 3}\ \{\texttt{(y = 4)}\}$. On the other hand $\widetilde{\vDash} \{?\}\ \texttt{x := 3}\ \{\texttt{(y = 4)}\}$ does not hold since the Hoare triple is not semantically valid. Gradual soundness is therefore unsatisfiable if formalized as above.

This conflict is nothing but a reminder that gradualization is not for free, but may require runtime checks in order to be sound. With this in mind we can revise our definition of soundness.

We extend the set of final states $\widetilde{\text{ProgramStateFin}}$ with a designated exceptional state π_{EX}, representing failure of a runtime check. Furthermore we introduce an assertion statement $\texttt{assert}\ \widetilde{\phi}$ (if not already available) that throws an exception, i.e. steps to π_{EX} should the condition not hold.

> **Example 3.41** (Gradual Small-Step Semantics for Runtime Assertion)**.**
> We assume that $\widetilde{\text{ProgramState}} = (\text{Var} \rightharpoonup \mathbb{Z}) \times \widetilde{\text{Stmt}}$ as introduced in example 3.21. Furthermore we assume that there exists a no-operation statement `skip`.
>
> $$\frac{\langle \sigma, \texttt{assert}\ \widetilde{\phi}\rangle \widetilde{\vDash} \widetilde{\phi}}{\langle \sigma, \texttt{assert}\ \widetilde{\phi}\rangle \widetilde{\longrightarrow} \langle \sigma, \texttt{skip}\rangle}\ \widetilde{\text{SsAssert}} \qquad \frac{\neg\ \langle \sigma, \texttt{assert}\ \widetilde{\phi}\rangle \widetilde{\vDash} \widetilde{\phi}}{\langle \sigma, \texttt{assert}\ \widetilde{\phi}\rangle \widetilde{\longrightarrow} \pi_{EX}}\ \widetilde{\text{SsAssertEx}}$$
>
> See lemma 3.29 for definition of $\cdot \widetilde{\vDash} \cdot$.

Soundness can then be redefined as

Definition 3.42 ($\widetilde{\text{Soundness}}$)**.**
Let $\widetilde{\phi_1}, \widetilde{\phi_2} \in \widetilde{\text{Formula}}$ *and* $\widetilde{s} \in \widetilde{\text{Stmt}}$.
If $\widetilde{\vdash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}$ *then* $\widetilde{\vDash} \{\widetilde{\phi_1}\}\ \widetilde{s}\texttt{; assert}\ \widetilde{\phi_2}\ \{\widetilde{\phi_2}\}$

Intuitively, this definition states that an assertion is inserted during verification, whenever gradual Hoare logic is used. Such a system can be viewed as a "safe baseline" and corresponds to a strong dynamic type system: Program states are only checked against their expectations as lately/lazily as possible (like Scheme or Python check values for their runtime type if a corresponding function is called). It may be desirable to check for inconsistencies more eagerly in order to terminate programs with unjustifiable behavior earlier or even as early as possible, like the "evidence" approach introduced by AGT [9]. In section 3.7.2 we will derive an approach that ultimately allows us to move towards more eager checking.

Lemma 3.43 ($\widetilde{\text{Soundness}}$ Tautology)**.**
$\widetilde{\text{Soundness}}$ *is a tautology.*

That $\widetilde{\text{Soundness}}$ is a tautology makes sense, realizing that (without demanding any degree of optimality) $\widetilde{\vdash}\ \{\cdot\} \cdot \{\cdot\}$ carries zero information. We will address this issue later (section 3.7.2), also rethinking the way we obtain $\widetilde{\vdash}\ \{\cdot\} \cdot \{\cdot\}$ in the first place.

Note that there is room for an optimized implementation of the injected assertions.

Example 3.44 (Non-Optimal Runtime Check Insertion)**.**
Deducing

$$\widetilde{\vdash}\ \{?\}\ \texttt{y := 2}\ \{\texttt{(x = 3)} \wedge \texttt{(y = 2)}\}$$

would lead to the injection of

$$\texttt{assert (x = 3)} \wedge \texttt{(y = 2)}$$

in order to make

$$\widetilde{\vdash}\ \{?\}\ \texttt{y := 2; assert (x = 3)} \wedge \texttt{(y = 2)}\ \{\texttt{(x = 3)} \wedge \texttt{(y = 2)}\}$$

hold. However,

$$\widetilde{\vdash}\ \{?\}\ \texttt{y := 2; assert (x = 3)}\ \{\texttt{(x = 3)} \wedge \texttt{(y = 2)}\}$$

holds as well since `(y = 2)` is implied by the assignment.

Sometimes it is possible to reduce (or even remove) the assertion and therefore the runtime overhead associated with it. We will address this observation later.

3.7 Abstracting Static Semantics

Lifting

$$\vdash \{\cdot\} \cdot \{\cdot\} \quad \subseteq \quad \text{Formula} \times \text{Stmt} \times \text{Formula}$$

w.r.t. all parameters yields

$$\widetilde{\vdash}\ \{\cdot\} \cdot \{\cdot\} \quad \subseteq \quad \widetilde{\text{Formula}} \times \widetilde{\text{Stmt}} \times \widetilde{\text{Formula}}$$

3.7.1 The Problem with Composite Predicate Lifting

In case the Hoare rules of **SVL** are given inductively, we can make use of the rules for composite, disjunctive and conjunctive predicate lifting (see section 3.5.2.1). The rules allow us to soundly lift each individual inductive rule in order to end up with a sound lifting of the overall predicate.

Example 3.45 (Rule-wise Hoare Logic Lifting)**.**
Assume that we are given a Hoare logic including rules for sequences and assignments:

$$\frac{\phi_{q1} \Rightarrow \phi_{q2} \quad \vdash \{\phi_p\}\ s_1\ \{\phi_{q1}\} \qquad \vdash \{\phi_{q2}\}\ s_2\ \{\phi_r\}}{\vdash \{\phi_p\}\ s_1\texttt{;}\ s_2\ \{\phi_r\}}\ \textsc{HSeq} \qquad \frac{}{\vdash \{\phi[e/x]\}\ x\ \texttt{:=}\ e\ \{\phi\}}\ \textsc{HAssign}$$

Then using rule-wise lifting we can obtain:

$$\frac{\widetilde{\phi_{q1}} \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi_{q2}} \quad \widetilde{\vdash} \{\widetilde{\phi_p}\}\ s_1\ \{\widetilde{\phi_{q1}}\} \qquad \widetilde{\vdash} \{\widetilde{\phi_{q2}}\}\ s_2\ \{\widetilde{\phi_r}\}}{\widetilde{\vdash} \{\widetilde{\phi_p}\}\ s_1\texttt{;}\ s_2\ \{\widetilde{\phi_r}\}}\ \widetilde{\textsc{H}}\textsc{Seq} \qquad \frac{}{\widetilde{\vdash} \{\phi[e/x]\}\ x\ \texttt{:=}\ e\ \{\phi\}}\ \widetilde{\textsc{H}}\textsc{Assign1}$$

$$\frac{}{\widetilde{\vdash} \{\texttt{?}\}\ x\ \texttt{:=}\ e\ \{\widetilde{\phi}\}}\ \widetilde{\textsc{H}}\textsc{Assign2} \qquad \frac{}{\widetilde{\vdash} \{\widetilde{\phi}\}\ x\ \texttt{:=}\ e\ \{\texttt{?}\}}\ \widetilde{\textsc{H}}\textsc{Assign3}$$

Note the usage of composite predicate lifting (lemma 3.30) for $\widetilde{\textsc{H}}\textsc{Seq}$.

Unfortunately, a gradual verifier using $\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\}$ gets into a practical dilemma. Consider the Hoare triple

$$\{\texttt{?}\}\ \texttt{y := 2; x := 3}\ \{\texttt{(x = 3)} \wedge \texttt{(y = 2)}\}$$

It is the job of the gradual verifier to prove the triple using above gradual inductive rules. Using rule inversion (rule $\widetilde{\textsc{H}}\textsc{Seq}$) it can deduce that

$$\widetilde{\phi_{q1}} \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi_{q2}}$$
$$\widetilde{\vdash} \{\texttt{?}\}\ \texttt{y := 2}\ \{\widetilde{\phi_{q1}}\}$$
$$\widetilde{\vdash} \{\widetilde{\phi_{q2}}\}\ \texttt{x := 3}\ \{\texttt{(x = 3)} \wedge \texttt{(y = 2)}\}$$

has to hold for some $\widetilde{\phi_{q1}}, \widetilde{\phi_{q2}} \in \widetilde{\textsc{Formula}}$. There are a variety of valid instantiations for both variables:

Good: $\widetilde{\phi_{q1}} = \texttt{(y = 2)}$, $\widetilde{\phi_{q2}} = \texttt{(y = 2)}$
This instantiation makes no assumptions and would even work if the precondition `?` was replaced by `true`.

Too weak: $\widetilde{\phi_{q1}} = \texttt{?}$, $\widetilde{\phi_{q2}} = \texttt{?}$
Whenever there exists a valid instantiation at all, the wildcard is also a valid one (due to the gradual guarantee), so always choosing it would be an easy way of implementing a gradual verifier. Note however, that the knowledge about the first statement is lost which results in the necessity of dynamic checks to ensure `(x = 3)` after `y := 2`. Before, this check could have been optimized away (using sufficiently sophisticated reasoning).

In general, choosing `?` as intermediate gradual formulas allows verifying arbitrary inconsistent judgments (a manifestation of the lack of optimality of rule $\widetilde{\textsc{H}}\textsc{Seq}$).

Example 3.46 (Inconsistent Gradual Verification)**.**

$$\widetilde{\vdash} \{\texttt{true}\}\ \texttt{y := 2; x := 3}\ \{\texttt{(y = 100)}\}$$

is verifiable if the gradual verifier chooses `?` as intermediate gradual formulas.

Consequently, the verifier may try to be "as precise as possible" in order to detect inconsistencies at compile time.

Too strict: $\widetilde{\phi_{q1}} = \texttt{(x = 42) } \wedge \texttt{ (y = 2)}$, $\widetilde{\phi_{q2}} = \texttt{(y = 2)}$
This instantiation is fully static (i.e. precise) and valid according to the rules: The implication holds (the knowledge about `x` is dropped), and

$$\widetilde{\vdash} \{\texttt{?}\}\ \texttt{y := 2}\ \{\texttt{(x = 42) } \wedge \texttt{ (y = 2)}\}$$

holds since

$$\vdash \{\texttt{(x = 42) } \wedge \texttt{ (2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(x = 42) } \wedge \texttt{ (y = 2)}\}$$

does.

However, the instantiation is stricter than necessary, meaning that the Hoare triple

$$\{\texttt{?}\}\ \texttt{y := 2}\ \{\texttt{(x = 42) } \wedge \texttt{ (y = 2)}\}$$

is not semantically valid.

While in general it is possible to derive semantically invalid Hoare triples (this is the motivation for runtime checks, see section 3.6), a gradual verifier should not deliberately produce any. Invalid Hoare triples which are based on annotations represent the programmers *intentions* and are thus ensured using runtime checks (see definition 3.42). On the other hand, invalid Hoare triples that are merely caused by an instantiation by the gradual verifier cannot be backed using runtime checks, as this would alter runtime behavior (in this example, a runtime check for `(x = 42)` to dynamically ensure the inferred postcondition is clearly unjustified).

As a result, such instantiations are not "credible" (neither semantically correct nor ensured using checks) and thus cannot be used for further reasoning (like branch prediction or simplifications based on the supposed value of `x`).

Note that above observations do not apply to a static verifier as the former two cases rely on the existence of the wildcard: The weak instantiation is syntactically impossible, the too strict instantiation relied on the fact that the precondition (of the overall judgment) is the wildcard.

How the gradual verifier instantiates variables influences whether obvious inconsistencies go unnoticed or formulas have any credibility. We propose a different approach that is not affected by above observations.

3.7.2 The Deterministic Approach

In the previous section we built an intuition about intermediate gradual formulas that are neither too weak (increasing the chance of hiding inconsistencies) nor too strict (introducing semantically invalid Hoare triples). While this problem could certainly be solved by designing a sophisticated inference algorithm for gradual formulas, we propose solving the problem at the level of the gradual Hoare logic.

In this section we derive a "deterministic gradual Hoare logic"

$$\vec{\vdash} : \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Stmt}} \rightharpoonup \widetilde{\textsc{Formula}}$$

from the static Hoare logic $\vdash \{\cdot\} \cdot \{\cdot\} \subseteq \textsc{Formula} \times \textsc{Stmt} \times \textsc{Formula}$ in a process we call "deterministic lifting". Note the difference to the signature of the gradual Hoare logic predicate

$$\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\} : \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Stmt}} \times \widetilde{\textsc{Formula}}$$

The deterministic gradual Hoare logic has the following desirable properties (formalized later): A gradual lifting $\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\}$ can be obtained from $\vec{\vdash}$, which is required for the gradual verifier. This gradual lifting is not defined inductively and does not rely on variables being instantiated. Instead it is syntax-directed, making the verification process deterministic. Furthermore $\vec{\vdash}$ allows for a stronger notion of soundness that does not rely on runtime checks.

We may use the familiar notation $\vec{\vdash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}$ as an alias for $\vec{\vdash}(\widetilde{\phi_1}, \widetilde{s}) = \widetilde{\phi_2}$

Our approach is based on the idea to treat the Hoare predicate as a (multivalued) function, mapping preconditions to the set of possible/verifiable postconditions. From this multivalued function we can obtain a lifted version, following similar rules to the ones for lifted partial functions (see 3.5.3.2).

Definition 3.47 (Deterministic Lifting)**.**
Given a binary predicate $P \subseteq \textsc{Formula} \times \textsc{Formula}$ *we call a partial function*

$$\vec{P} : \textsc{Formula} \rightharpoonup \textsc{Formula}$$

deterministic lifting *of* P *if the following three conditions are met.*
(As before, the rules can be adapted to different parameter types and higher arity. We indicate an adaptation to Hoare logic in blue.)

Introduction

The deterministic lifting should be defined whenever the underlying predicate is.

$$\forall \langle \phi_1, \phi_2 \rangle \in P.\ \phi_1 \in \mathsf{dom}(\vec{P})$$

$$\forall \langle \phi_1, s, \phi_2 \rangle \in\ \vdash \{\cdot\} \cdot \{\cdot\}.\ \langle \phi_1, s \rangle \in \mathsf{dom}(\vec{\vdash})$$

Strength

A return value of the deterministic lifting should agree with all instantiations of the underlying predicate.

$$\forall \widetilde{\phi_1}, \widetilde{\phi_2} \in \widetilde{\textsc{Formula}}.\ \vec{P}(\widetilde{\phi_1}) = \widetilde{\phi_2}$$
$$\Longrightarrow$$
$$\forall \phi_1 \in \gamma(\widetilde{\phi_1}),\ \phi \in \textsc{Formula}.\ P(\phi_1, \phi) \implies$$
$$\exists \phi_2 \in \gamma(\widetilde{\phi_2}).\ P(\phi_1, \phi_2)\ \wedge\ (\phi_2 \Rightarrow \phi)$$

$$\forall \widetilde{\phi_1}, \widetilde{\phi_2} \in \widetilde{\textsc{Formula}},\ \widetilde{s} \in \widetilde{\textsc{Stmt}}.\ \vec{\vdash}(\widetilde{\phi_1}, \widetilde{s}) = \widetilde{\phi_2}$$
$$\Longrightarrow$$
$$\forall \phi_1 \in \gamma(\widetilde{\phi_1}),\ s \in \gamma(\widetilde{s}),\ \phi \in \textsc{Formula}.\ \vdash \{\phi_1\}\ s\ \{\phi\} \implies$$
$$\exists \phi_2 \in \gamma(\widetilde{\phi_2}).\ \vdash \{\phi_1\}\ s\ \{\phi_2\}\ \wedge\ (\phi_2 \Rightarrow \phi)$$

For Hoare rules this means that the postcondition returned must be at least as strong as every postcondition returned by static Hoare logic (it might be less precise, though). The following example illustrates the effects of the rule:

Example 3.48 (Deterministic Lifting Strength)**.** Assume that the following list contains all instantiations of $\vdash \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\cdot\}$.

$$\vdash \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(2 = 2)}\}$$
$$\vdash \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(y = 2)}\}$$
$$\vdash \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(2 = y)}\}$$
$$\vdash \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(y = y)}\}$$

Then valid return values for the deterministic lifting are

$$\vec{\vdash} \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(y = 2)}\}$$
$$\vec{\vdash} \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(2 = y)}\}$$
$$\vec{\vdash} \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{?}\}$$
$$\vec{\vdash} \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(y = 2)} \wedge \texttt{?}\}$$
$$\vec{\vdash} \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(2 = y)} \wedge \texttt{?}\}$$

Not valid are weaker static values like

$$\vec{\vdash} \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(2 = 2)}\}$$
$$\vec{\vdash} \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(y = y)}\}$$

or stronger values like

$$\vec{\vdash} \{\texttt{(2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(x = 3)} \wedge \texttt{(y = 2)}\}$$

Likewise, with precondition `?` (motivated by case "too strict" in section 3.7.1) valid return values include

$$\vec{\vdash} \{?\}\ \texttt{y := 2}\ \{?\}$$
$$\vec{\vdash} \{?\}\ \texttt{y := 2}\ \{\texttt{(y = 2)} \wedge ?\}$$

bot not

$$\vec{\vdash} \{?\}\ \texttt{y := 2}\ \{\texttt{(x = 3)} \wedge \texttt{(y = 2)}\}$$

which corresponds to the instantiation criticized in section 3.7.1.

Monotonicity

This rule is identical to the monotonicity condition of lifted partial functions (see section 3.5.3.2).

$$\forall \widetilde{\phi_1}, \widetilde{\phi_2} \in \widetilde{\text{Formula}}.\ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_2} \quad \wedge \quad \widetilde{\phi_1} \in \mathsf{dom}(\vec{P}) \implies \vec{P}(\widetilde{\phi_1}) \sqsubseteq \vec{P}(\widetilde{\phi_2})$$

$$\forall \widetilde{\phi_1}, \widetilde{\phi_2} \in \widetilde{\text{Formula}},\ \widetilde{s_1}, \widetilde{s_2} \in \widetilde{\text{Stmt}}.\ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_2} \quad \wedge \quad \widetilde{s_1} \sqsubseteq \widetilde{s_2} \quad \wedge \quad \langle \widetilde{\phi_1}, \widetilde{s_1} \rangle \in \mathsf{dom}(\vec{\vdash}) \implies \vec{\vdash}(\widetilde{\phi_1}, \widetilde{s_1}) \sqsubseteq \vec{\vdash}(\widetilde{\phi_2}, \widetilde{s_2})$$

Soundness and optimality are defined as usual (see sections 3.5.2 or 3.5.3).

Assume we have obtained the deterministic lifting $\vec{\vdash} \{\cdot\} \cdot \{\cdot\}$ of our Hoare logic. As mentioned before, this gradual partial function has desirable properties:

(a) Obtaining a Sound Gradual Lifting

Lemma 3.49 (Deterministic Lifting as Sound Lifting)**.**
Let $\vec{P}$ be a deterministic lifting of P. Then

$$\widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2}) \quad \overset{\mathsf{def}}{\iff} \quad \exists \widetilde{\phi_2'}.\ \vec{P}(\widetilde{\phi_1}) = \widetilde{\phi_2'} \wedge \widetilde{\phi_2'} \mathrel{\widetilde{\Rrightarrow}} \widetilde{\phi_2}$$

is a sound gradual lifting of P.
See lemma 3.28 for definition of $\cdot \mathrel{\widetilde{\Rrightarrow}} \cdot$.

This observation bridges the gap between $\vec{\vdash} \{\cdot\} \cdot \{\cdot\}$ and the gradual verifier which is supposed to implement $\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\}$.

Optimality of the deterministic lifting does not imply optimality of the obtained gradual lifting.

Example 3.50 (Counterexample of Optimality Induced by Deterministic Lifting)**.**
Assume that one can verify

$$\vdash \{\texttt{(y = 4)}\}\ \texttt{x := 3}\ \{\texttt{(y = 4)} \wedge \texttt{(x = 3)}\} \tag{3.2}$$

and

$$\vdash \{\texttt{(y = 5)}\}\ \texttt{x := 3}\ \{\texttt{(y = 5)} \wedge \texttt{(x = 3)}\} \tag{3.3}$$

using Hoare logic.

Now, let $\widetilde{\text{Formula}}$ be extended using the "total unknown" approach (see section 3.2.1). The deterministic lifting of the Hoare logic will verify

$$\vec{\vdash} \{\texttt{?}\}\ \texttt{x := 3}\ \{\texttt{?}\}$$

The wildcard as postcondition is necessary, as anything else (e.g. `(x = 3)`) would break the strength condition (take judgments 3.2 and 3.3, both of their postconditions cannot be implied by any satisfiable static formula).

Using 3.49 we can deduce

$$\widetilde{\vdash} \{\texttt{?}\}\ \texttt{x := 3}\ \{\texttt{(x = 1)}\} \tag{3.4}$$

since `?` $\widetilde{\Rrightarrow}$ `(x = 1)` holds. An optimal gradual lifting would not be able to deduce this judgment, as it requires the existing of some $\phi \in \gamma(\texttt{?}) = \text{SatFormula}$ such that

$$\vdash \{\phi\}\ \texttt{x := 3}\ \{\texttt{(x = 1)}\}$$

holds. However, the only working instantiation according to Hoare logic is $\phi =$ `(3 = 1)` which is not satisfiable. It follows that the obtained lifting is not optimal, even though the deterministic lifting is.

Note that the "bounded unknown" approach (see 3.2.2) would allow specifying `(x = 3)` $\wedge$ `?` as postcondition. With this more precise postcondition, one cannot deduce judgment 3.4 (since `(x = 3)` $\wedge$ `?` $\widetilde{\Rrightarrow}$ `(x = 1)` does not hold). This example motivates the use of a more powerful gradual syntax, as they enable more optimal gradual liftings.

We conjecture that optimality of the deterministic lifting does imply optimality of the obtained gradual lifting if the gradual formula syntax is expressive enough to encode all knowledge about the dynamic semantics of a statement.

(b) Determinism of Verifier

As the name suggests, deterministic liftings leave no room for choice, but are instead syntax-directed. For the gradual verifier this means that there is no need to infer intermediate formulas, averting the risk of choosing bad instantiations (as illustrated in section 3.7.1).

(c) Soundness

Deterministic liftings are designed in a way that allows for a stronger notion of soundness that does not rely on runtime checks.

Definition 3.51 ($\vec{\text{S}}$oundness)**.**
Let $\widetilde{\phi_1}, \widetilde{\phi_2} \in \widetilde{\text{Formula}}$ *and* $\widetilde{s} \in \widetilde{\text{Stmt}}$.
If $\vec{\vdash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}$ *then* $\widetilde{\vDash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}$

Note that this rule is no longer a tautology, compliance also depends on the gradual dynamic semantics of **GVL**.

Example 3.52 ($\vec{\text{S}}$OUNDNESS Counterexample)**.**
Assume that **SVL** contains an assertion statement with corresponding Hoare rule.

$$\frac{\phi \Rightarrow \phi_a}{\vdash \{\phi\}\ \texttt{assert}\ \ \phi_a\ \{\phi\}}\ \text{HASSERT}$$

Soundness of the Hoare logic implies that assertions are guaranteed to hold at runtime. It is therefore reasonable for **SVL** to implement assertions as no-operations. It follows from the rules of gradual function lifting (applied to the small-step semantics) that **GVL** implements them the same way.

A valid deterministic lifting of HASSERT is able to verify

$$\vec{\vdash}\ \{\texttt{?}\}\ \texttt{assert}\ \ \phi_a\ \{\phi_a\ \wedge\ \texttt{?}\}$$

However $\widetilde{\vDash}\ \{\texttt{?}\}\ \texttt{assert}\ \ \phi_a\ \{\phi_a\ \wedge\ \texttt{?}\}$ does not hold for non-trivial ϕ_a:

Choose $\phi_a = \texttt{(x = 2)}$. Let $\widetilde{\pi}, \widetilde{\pi'} \in \widetilde{\text{P}}$ROGRAMSTATE such that $\widetilde{\pi} \overset{\widetilde{\texttt{assert}\ \phi_a}}{\longrightarrow} \widetilde{\pi'}$ and $\widetilde{\pi} \widetilde{\vDash} \texttt{(x = 3)}$ holds (both assumptions are compatible since assertions are implemented as no-operations, i.e. especially not runtime check $\widetilde{\pi} \widetilde{\vDash} \phi_a$ is performed). It follows that $\widetilde{\pi'} \widetilde{\vDash} \texttt{(x = 3)}$. According to definition 3.38, we can deduce from $\widetilde{\vDash}\ \{\texttt{?}\}\ \texttt{assert}\ \ \phi_a\ \{\phi_a\ \wedge\ \texttt{?}\}$ that $\widetilde{\pi'} \widetilde{\vDash} \texttt{(x = 2)}\ \wedge\ \texttt{?}$. No such $\widetilde{\pi'}$ exists.

On the other hand, a small-step semantics that is stuck or throws an exception if the assertion is violated would restore soundness. In general, small-step semantics that rely on guarantees given by static semantics may turn out too weak after gradualization of the static semantics.

Lemma 3.53 (Reduced Runtime Checks)**.**
Assume that $\vec{\vdash}\ \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2'}\}$ *holds for some* $\widetilde{\phi_1}, \widetilde{s}, \widetilde{\phi_2'}$*. If* $\vec{\text{S}}$OUNDNESS *holds, the runtime check* $\texttt{assert}\ \widetilde{\phi_2}$ *associated with a judgment* $\widetilde{\vdash}\ \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}$ *(as postulated by definition 3.42) can be simplified or omitted.*

Specifically, it is sufficient to insert a runtime check (predicate) $P \subseteq \widetilde{\text{P}}$ROGRAMSTATE *if*

$$[\![\widetilde{\phi_2'}]\!] \cap [\![\widetilde{\phi_2}]\!] \subseteq [\![\widetilde{\phi_2'}]\!] \cap P \subseteq [\![\widetilde{\phi_2}]\!]$$

holds where $[\![\cdot]\!] : \widetilde{\text{F}}$ORMULA $\to \mathcal{P}^{\widetilde{\text{P}}\text{ROGRAMSTATE}}$ *is defined analogous to definition 3.4:*

$$[\![\widetilde{\phi}]\!] \overset{\text{def}}{=} \{\ \widetilde{\pi} \in \widetilde{\text{P}}\text{ROGRAMSTATE} \mid \widetilde{\pi} \widetilde{\vDash} \widetilde{\phi}\ \}$$

Example 3.54 (Reduced Runtime Check)**.**
Let $\widetilde{\phi_2'} = \texttt{(x = 3)}\ \wedge\ \texttt{?}$ and $\widetilde{\phi_2} = \texttt{(x = 3)}\ \wedge\ \texttt{(y = 5)}$.

Then $P(\pi) \overset{\text{def}}{=} \pi \widetilde{\vDash} \texttt{(y = 5)}$ is a sufficient runtime check.

Corollary 3.55 (Superfluous Runtime Checks)**.**
$P(\pi) = \textsc{ProgramState}$ *is a sufficient runtime check if* $[\![\widetilde{\phi_2'}]\!] \subseteq [\![\widetilde{\phi_2}]\!]$.
In other words, no runtime check is necessary in this case.

3.7.2.1 Examples

Lemma 3.56 (Composability of Deterministic Lifting)**.**
Let $\vec{P_1}, \vec{P_2}$ *be sound deterministic liftings of predicates* P_1, P_2.
Let P_2 *be monotonic w.r.t. implication in its first parameter. Then*

$$\vec{P_3} \stackrel{\mathsf{def}}{=} \vec{P_2} \circ \vec{P_1}$$

is a sound deterministic lifting of $P_3(\phi_1, \phi_3) = \exists \phi_2.\ P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi_3)$.

In other words, deterministic liftings of composite predicates can be obtained by composing piecewise deterministic liftings. Optimality of the piecewise liftings does not guarantee optimality of the overall lifting.

We further demonstrate deterministic lifting by means of the following Hoare rules:

$$\frac{\begin{array}{c}\phi_{q1} \Rightarrow \phi_{q2} \\ \vdash \{\phi_p\}\ s_1\ \{\phi_{q1}\} \qquad \vdash \{\phi_{q2}\}\ s_2\ \{\phi_r\}\end{array}}{\vdash \{\phi_p\}\ s_1\texttt{;}\ s_2\ \{\phi_r\}}\ \textsc{HSeq} \qquad \frac{}{\vdash \{\phi[e/x]\}\ x\ \texttt{:=}\ e\ \{\phi\}}\ \textsc{HAssign}$$

Note that gradual liftings of these rules can be found in section 3.7.1, where we demonstrated the problems that ultimately lead to deterministic liftings.

We start by lifting implication, which is used in HSeq.

Lemma 3.57 (Optimal Deterministic Lifting of Implication)**.**
Let $\mathsf{id} : \widetilde{\textsc{Formula}} \to \widetilde{\textsc{Formula}}$ *be the identity function.*
Then id *is an optimal deterministic lifting of* $\cdot \Rightarrow \cdot \subseteq \textsc{Formula} \times \textsc{Formula}$.

Using lemma 3.56 and lemma 3.57, the deterministic lifting of HSeq reduces to

$$\frac{\vec{\vdash}\ \{\widetilde{\phi_p}\}\ \widetilde{s_1}\ \{\widetilde{\phi_q}\} \qquad \vec{\vdash}\ \{\widetilde{\phi_q}\}\ \widetilde{s_2}\ \{\widetilde{\phi_r}\}}{\vec{\vdash}\ \{\widetilde{\phi_p}\}\ \widetilde{s_1}\texttt{;}\ \widetilde{s_2}\ \{\widetilde{\phi_r}\}}\ \vec{\textsc{H}}\textsc{Seq}$$

Note that we used inductive notation to define a function – there are no free variables, i.e. the rule is syntax-directed.

For the assignment rule we can derive

$$\frac{x \notin \mathsf{FV}(\phi) \qquad x \notin \mathsf{FV}(e)}{\vec{\vdash}\ \{\phi\}\ x\ \texttt{:=}\ e\ \{\phi\ \wedge\ (x\ \texttt{=}\ e)\}}\ \vec{\textsc{H}}\textsc{Assign1} \qquad \frac{\vec{\textsc{H}}\textsc{Assign1}\ \textit{does not apply}}{\vec{\vdash}\ \{\widetilde{\phi}\}\ x\ \texttt{:=}\ e\ \{\texttt{?}\}}\ \vec{\textsc{H}}\textsc{Assign2}$$

as a sound deterministic lifting. Note that this lifting is far from optimal, but rather a heuristic. A more optimal lifting requires more sophisticated reasoning.

3.7.2.2 Gradual Verification Illustrated

Having introduced a lot of concepts that all play together, it is worth going through a gradual verification process in its entirety. We assume that **GVL** knows the Hoare rules lifted in section 3.7.2.1.

Example **GVL** program:

```
int getFourA(int i)
    requires true;
    ensures  result = 4;
{
    i := 4;
    return i;
}

int getFourB(int i)
    requires ?; // too lazy to figure that one out, yet
    ensures  result = 4;
{
    i := i + 1;
    return i;
}
```

We assume that `return i` is syntactic sugar for an assignment `result := i` to the reserved variable `result`.

During compilation, the gradual verifier is expected to verify the method contracts which is equivalent to deducing

$$\widetilde{\vdash} \{\texttt{true}\}\ \texttt{i := 4; result := i}\ \{(\texttt{result = 4})\}$$

and

$$\widetilde{\vdash} \{?\}\ \texttt{i := i + 1; result := i}\ \{(\texttt{result = 4})\}$$

It does so by applying the deterministic lifting (see section 3.7.2.1) and then using lemma 3.49 to deduce the gradual lifting.

Deducing $\widetilde{\vdash} \{\texttt{true}\}\ \texttt{i := 4; result := i}\ \{(\texttt{result = 4})\}$:

$$\dfrac{\dfrac{\dfrac{\texttt{i} \notin \mathsf{FV}(\texttt{true})}{\texttt{i} \notin \mathsf{FV}(4)}\ \vec{\mathrm{H}}\textsc{Assign1}}{\mathcal{D}_1} \qquad \dfrac{\dfrac{\texttt{result} \notin \mathsf{FV}(\texttt{true} \wedge (\texttt{i = 4}))}{\texttt{result} \notin \mathsf{FV}(\texttt{i})}\ \vec{\mathrm{H}}\textsc{Assign1}}{\mathcal{D}_2}}{\dfrac{\mathcal{D}}{\widetilde{\vdash} \{\texttt{true}\}\ \texttt{i := 4; result := i}\ \{(\texttt{result = 4})\}}\ \mathcal{I}}\ \vec{\mathrm{H}}\textsc{Seq}\ \textsc{Lemma 3.49}$$

where

$$\mathcal{D}_1 \equiv \vec{\vdash} \{\texttt{true}\}\ \texttt{i := 4}\ \{\texttt{true} \wedge \texttt{(i = 4)}\}$$
$$\mathcal{D}_2 \equiv \vec{\vdash} \{\texttt{true} \wedge \texttt{(i = 4)}\}\ \texttt{result := i}\ \{\texttt{true} \wedge \texttt{(i = 4)} \wedge \texttt{(result = i)}\}$$
$$\mathcal{D} \equiv \vec{\vdash} \{\texttt{true}\}\ \texttt{i := 4; result := i}\ \{\texttt{true} \wedge \texttt{(i = 4)} \wedge \texttt{(result = i)}\}$$
$$\mathcal{I} \equiv \texttt{true} \wedge \texttt{(i = 4)} \wedge \texttt{(result = i)} \mathrel{\widetilde{\Rightarrow}} \texttt{(result = 4)}$$

Deducing $\widetilde{\vdash} \{?\}\ \texttt{i := i + 1; result := i}\ \{\texttt{(result = 4)}\}$:

$$\dfrac{\dfrac{\dfrac{}{\mathcal{D}_1}\,\vec{\text{H}}\textsc{Assign2} \qquad \dfrac{}{\mathcal{D}_2}\,\vec{\text{H}}\textsc{Assign2}}{\mathcal{D}}\,\vec{\text{H}}\textsc{Seq} \qquad \mathcal{I}}{\widetilde{\vdash} \{?\}\ \texttt{i := i + 1; result := i}\ \{\texttt{(result = 4)}\}}\ \textsc{Lemma}\ 3.49$$

where

$$\mathcal{D}_1 \equiv \vec{\vdash} \{?\}\ \texttt{i := i + 1}\ \{?\}$$
$$\mathcal{D}_2 \equiv \vec{\vdash} \{?\}\ \texttt{result := i}\ \{?\}$$
$$\mathcal{D} \equiv \vec{\vdash} \{?\}\ \texttt{i := i + 1; result := i}\ \{?\}$$
$$\mathcal{I} \equiv ? \mathrel{\widetilde{\Rightarrow}} \texttt{(result = 4)}$$

As established in section 3.6, runtime checks have to be injected whenever such a judgment is made, in order to guarantee soundness. Definition 3.42 of $\widetilde{\text{S}}$OUNDNESS suggests inserting a runtime assertion for the entire postcondition. The compiled program (as executed by the small-step semantics) would correspond to the following source code:

```
int getFourA(int i)
{
    i := 4;
    result := i;
    assert (result = 4);
}

int getFourB(int i)
{
    i := i + 1;
    result := i;
    assert (result = 4);
}
```

The runtime check in method `getFourA` looks redundant. As motivated in section 3.7.2, the gradual verifier may formally reason about optimized runtime checks using the return

values of the deterministic gradual Hoare logic in case $\vec{\text{Soundness}}$ holds. Lemma 3.53 and corollary 3.55 give refined criteria for sufficient runtime checks.

Assuming that $\vec{\text{Soundness}}$ does indeed hold, one can use corollary 3.55 to deduce that no runtime check is necessary for method `getFourA` since

$$[\![\texttt{true} \wedge \texttt{(i = 4)} \wedge \texttt{(result = i)}]\!] \subseteq [\![\texttt{(result = 4)}]\!]$$

holds.

For method `getFourB` it is valid to check for $P(\widetilde{\pi}) = \widetilde{\pi} \mathrel{\widetilde{\models}} \texttt{(result = 4)}$ (according to lemma 3.53), which again corresponds to executing a runtime assertion `assert (result = 4)`.

As a result, the compiled program corresponds to the following source code.

```
int getFourA(int i)
{
    i := 4;
    result := i;
}

int getFourB(int i)
{
    i := i + 1;
    result := i;
    assert (result = 4);
}
```

3.8 Abstracting Dynamic Semantics

Let

$$\cdot \mathrel{\widetilde{\Longrightarrow}} \cdot : \widetilde{\text{ProgramState}} \rightharpoonup \widetilde{\text{ProgramState}}$$

be a sound gradual lifting of

$$\cdot \longrightarrow \cdot : \text{ProgramState} \rightharpoonup \text{ProgramState}$$

As shown in section 3.6, $\widetilde{\text{Soundness}}$ is a tautology, i.e. the gradual system we have derived is sound.

However, in case the deterministic lifting approach is used to derive a gradual Hoare logic it is desirable to satisfy the stronger notion of soundness "$\vec{\text{Soundness}}$" introduced in section 3.7.2 (definition 3.51). This stronger notion allows considering return values of the deterministic lifting as static knowledge. As illustrated in section 3.7.2.2, such static knowledge can be used for optimizations.

In the remainder of this section we will thus determine sufficient criteria for $\vec{\text{Soundness}}$.

3.8.1 Perfect Knowledge

When first introducing $\vec{\text{S}}$OUNDNESS in section 3.7.2, we also gave an example (3.52) of a language violating $\vec{\text{S}}$OUNDNESS. The root of the problem seemed to be the existence of small-step derivations that are not verifiable using static Hoare logic. A Hoare logic that can prove all small-step derivations correct is called "complete". However, as illustrated in example 3.52, incompleteness of the Hoare logic may not necessarily indicate weakness of the Hoare logic, but may also be caused by optimized small-step semantics.

As completeness seems to be a property of the entire semantics (instead of just the Hoare logic), we will define it as such.

Definition 3.58 (Completeness)**.**
A semantics is ***complete*** *if every execution of a statement s can be supported with a matching Hoare logic derivation:*

$$\begin{aligned}\forall s \in \text{STMT},\, \pi_1, \pi_2 \in \text{PROGRAMSTATE}.\ \pi_1 \xrightarrow{s} \pi_2 \\ \implies \exists \phi_1, \phi_2 \in \text{FORMULA}.\ \vdash \{\phi_1\}\ s\ \{\phi_2\} \land \pi_1 \vDash \phi_1\end{aligned}$$

(Note that $\pi_2 \vDash \phi_2$ is then implied by soundness of the static language and thus not stated in above definition.)

As indicated before, complete semantics may be derived by enhancing the existing Hoare logic but also by restricting the domain of the existing small-step semantics. As long as soundness is not broken, those changes do not have observable effects on **SVL** programs, i.e. valid programs will behave identically before and after respective adjustments.

Lemma 3.59 (Completion of Semantics)**.**
Deriving complete semantics by restricting the domain of the small-step semantics or extending the Hoare logic of ***SVL*** *(without breaking soundness) does not have observable effects.*

Based on a complete semantics of **SVL**, the following requirements for **GVL** are then sufficient to satisfy $\vec{\text{S}}$OUNDNESS.

Definition 3.60 (Semi-Optimal Gradual Deterministic Hoare Logic)**.**
We call a deterministically lifted Hoare logic $\vec{\vdash}\ \{\cdot\}\ \cdot\ \{\cdot\}$ ***semi-optimal*** *iff*

$$\begin{aligned}\forall \phi_1 \in \text{FORMULA},\, \widetilde{s} \in \widetilde{\text{S}}\text{TMT},\, \widetilde{\phi_2} \in \widetilde{\text{F}}\text{ORMULA}.\ \vec{\vdash}\ \{\phi_1\}\ \widetilde{s}\ \{\widetilde{\phi_2}\} \\ \implies \exists s \in \gamma(\widetilde{s}),\, \phi_2 \in \text{FORMULA}.\ \vdash \{\phi_1\}\ s\ \{\phi_2\}\end{aligned}$$

Intuitively, the deterministic lifting is only supposed to be defined for static preconditions if there exists some corresponding derivation of the static Hoare logic.

As expected, optimal deterministic liftings are semi-optimal.

Definition 3.61 (Semi-Optimal Gradual Small-Step Semantics)**.**
We call a total, lifted small-step semantics $\cdot \xrightarrow{\sim} \cdot$ ***semi-optimal*** *iff*

$$\begin{aligned}\forall \widetilde{\pi_1}, \widetilde{\pi_2} \in \widetilde{\text{P}}\text{ROGRAMSTATE},\, \widetilde{s} \in \widetilde{\text{S}}\text{TMT}.\ \widetilde{\pi_1} \xrightarrow{\widetilde{s}} \widetilde{\pi_2} \\ \implies \exists \pi_1 \in \gamma(\widetilde{\pi_1}),\, s \in \gamma(\widetilde{s}), \pi_2 \in \text{PROGRAMSTATE}.\ \pi_1 \xrightarrow{s} \pi_2\end{aligned}$$

Intuitively, the gradual small-step semantics is not supposed to define derivations if all concretizations would be stuck in the original small-step semantics. Note that the rules of gradual lifting (of partial functions) allow arbitrary behavior in case the small-step semantics are stuck (i.e. the function is undefined).

We call the semantics of **GVL** semi-optimal if both Hoare logic and small-step semantics are semi-optimal as defined above.

Theorem 3.62 (Completeness and Semi-Optimality imply $\vec{\text{S}}$OUNDNESS)**.**
*If the semantics of **SVL** are complete and the semantics of **GVL** are semi-optimal, then* $\vec{\text{S}}$OUNDNESS *is satisfied.*

3.8.2 Partial Knowledge

Completeness as defined in definition 3.58 tightly couples Hoare logic and small-step semantics. In practice this can be impossible to achieve due to decidability, i.e. not all valid small-step derivations can be modeled using Hoare logic. Similarly, semi-optimality of the gradual semantics might be hard to ensure. Semi-optimality of the gradual Hoare logic requires deciding the existence of a Hoare logic derivation which involves first order logic for composite statements like sequences (which was a problem we originally wanted to avoid with deterministic liftings). In this section we motivate that a weaker notion of completeness and semi-optimality may be sufficient to prove $\vec{\text{S}}$OUNDNESS.

The sequence operator ; is key to defining composite statements in most programming languages. Fortunately, $\vec{\text{S}}$OUNDNESS can be proved for sequences inductively.

Lemma 3.63 ($\vec{\text{S}}$OUNDNESS for Sequences)**.**
If $\vec{\text{S}}$OUNDNESS *holds for statements s_1 and s_2 then it holds for $s_1; s_2$*

In the case study we will use this lemma to prove $\vec{\text{S}}$OUNDNESS of a gradually verified language, but also show that this inductive approach can be applied to other composite statements like method calls (section 4.7). As a result, it is sufficient to prove $\vec{\text{S}}$OUNDNESS for "primitive" statements, e.g. by using the approach introduced in the previous section: Note that the definitions of completeness and semi-optimality are universally quantified over the set of all (gradual) statements. Instead, they can be weakened to quantify only over a limited set of primitive statements. The resulting proof of $\vec{\text{S}}$OUNDNESS (lemma 3.62) will apply only to this set of statements.

Again, we want to point out that we only give examples of sufficient criteria to prove $\vec{\text{S}}$OUNDNESS. It is possible that the approaches do not work for a certain programming language, or even that it is entirely impossible to satisfy $\vec{\text{S}}$OUNDNESS. However, recall that $\vec{\text{S}}$OUNDNESS is not necessary for **GVL** to be sound (i.e. satisfy $\widetilde{\text{S}}$OUNDNESS).

4 Case Study: Implicit Dynamic Frames

To show the flexibility of our approach, we apply it to a simple statically verified Java-like language $\mathbf{SVL_{IDF}}$ that uses implicit dynamic frames to enable safe reasoning about mutable state (see section 2.3 for introduction and examples). The usage of implicit dynamic frames poses a challenge as it introduces elements of linear logic into formula semantics. However, despite the fact that we used classical logic for the examples throughout chapter 3, our approach never made an assumption about it. The logic at hand is abstracted away behind formula semantics $\pi \vDash \phi$.

This chapter roughly follows the structure of chapter 3, obtaining a gradually verified language $\mathbf{GVL_{IDF}}$ from $\mathbf{SVL_{IDF}}$. It starts with a full definition of $\mathbf{SVL_{IDF}}$ in section 4.1, instantiating the elements postulated in section 3.1. Section 4.2 describes our decisions regarding the syntax of $\mathbf{GVL_{IDF}}$, defining $\widetilde{\text{Formula}}$, $\widetilde{\text{Stmt}}$ and $\widetilde{\text{ProgramState}}$. Using the concept of deterministic lifting introduced in section 3.7.2 we will obtain gradual Hoare logic of $\mathbf{GVL_{IDF}}$ in section 4.4. In section 4.5 we derive gradual small-step semantics in a way that makes the gradual verification system sound and also complies with the stronger notion of soundness "$\vec{\text{Soundness}}$" postulated in section 3.6.

4.1 The Statically Verified Language $\mathrm{SVL_{IDF}}$

We now introduce a simplified Java-like statically verified language $\mathbf{SVL_{IDF}}$ that uses Chalice-like sub-syntax to express method contracts.

4.1.1 Syntax

Figure 4.1 shows the full syntax of $\mathbf{SVL_{IDF}}$.

$$
\begin{array}{rll}
\mathit{program} \in \textsc{Program} & ::= & \overline{\mathit{cls}}\ \ s \\
\mathit{cls} \in \textsc{Class} & ::= & \texttt{class}\ C\ \texttt{\{}\ \overline{\mathit{field}}\ \ \overline{\mathit{method}}\ \texttt{\}} \\
\mathit{field} \in \textsc{Field} & ::= & T\ \ f\texttt{;} \\
\mathit{method} \in \textsc{Method} & ::= & T\ \ m\texttt{(}T\ \ x\texttt{)}\ \ \mathit{contract}\ \texttt{\{}\ s\ \texttt{\}} \\
\mathit{contract} \in \textsc{Contract} & ::= & \texttt{requires}\ \phi\texttt{;}\ \texttt{ensures}\ \phi\texttt{;} \\
T \in \textsc{Type} & ::= & \texttt{int} \mid C \\
s \in \textsc{Stmt} & ::= & \texttt{skip} \mid T\ \ x \mid x\texttt{.}f\ \ \texttt{:=}\ y \mid x\ \ \texttt{:=}\ e \mid x\ \ \texttt{:=}\ \texttt{new}\ C \mid x\ \ \texttt{:=}\ y\texttt{.}m\texttt{(}z\texttt{)} \\
 & & \quad \mid \texttt{return}\ x \mid \texttt{assert}\ \phi \mid \texttt{release}\ \phi \mid \texttt{hold}\ \phi\ \texttt{\{}\ s\ \texttt{\}} \mid s_1\texttt{;}\ \ s_2 \\
\phi \in \textsc{Formula} & ::= & \texttt{true} \mid \texttt{(}e\ \texttt{=}\ e\texttt{)} \mid \texttt{(}e\ \neq\ e\texttt{)} \mid \texttt{acc(}e\texttt{.}f\texttt{)} \mid \phi\ \texttt{*}\ \phi \\
e \in \textsc{Expr} & ::= & v \mid x \mid e\texttt{.}f \\
x, y, z \in \textsc{Var} & ::= & \texttt{this} \mid \texttt{result} \mid \mathit{identifier} \\
v \in \textsc{Val} & ::= & o \mid n \mid \texttt{null} \\
o \in \textsc{Loc} & & \text{(infinite set of memory locations)} \\
n \in \mathbb{Z} & & \\
C \in \textsc{ClassName} & ::= & \mathit{identifier} \\
f \in \textsc{FieldName} & ::= & \mathit{identifier} \\
m \in \textsc{MethodName} & ::= & \mathit{identifier}
\end{array}
$$

We pose $\texttt{false} \stackrel{\mathsf{def}}{=} \texttt{(null}\ \neq\ \texttt{null)}$.

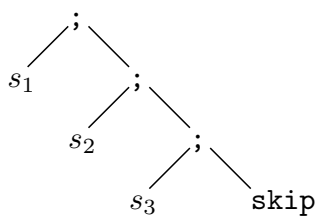
Figure 4.1. $\mathbf{SVL_{IDF}}$: Syntax

We define ; to be right-associative and assume that parsing a sequence of statements (e.g. method body) operates analogously, obviating the need for parenthesis. Furthermore we assume that the parser terminates every sequence with `skip`.

Example 4.1.

```
...
{
    s1;
    s2;
    s3;
}
```

is parsed as



These assumptions highly simplify reasoning about statements.

4.1.1.1 Helper Methods

We define the following helper methods:

Extraction

To extract elements from a given program $p \in \textsc{Program}$ we define the following functions:

$\mathsf{fields}_p : \textsc{ClassName} \rightharpoonup \mathcal{P}^{\textsc{Field}}$
$\mathsf{fields}_p(C) =$ field declarations of class C in p

$\mathsf{fieldType}_p : \textsc{ClassName} \times \textsc{FieldName} \rightharpoonup \textsc{Type}$
$\mathsf{fieldType}_p(C, f) =$ type of field f in class C in p

$\mathsf{method}_p : \textsc{ClassName} \times \textsc{MethodName} \rightharpoonup \textsc{Method}$
$\mathsf{method}_p(C, m) =$ declaration of method m in class C in p

$\mathsf{mpre}_p : \textsc{ClassName} \times \textsc{MethodName} \rightharpoonup \textsc{Formula}$
$\mathsf{mpre}_p(C, m) =$ precondition of method m in class C in p

$\mathsf{mpost}_p : \textsc{ClassName} \times \textsc{MethodName} \rightharpoonup \textsc{Formula}$
$\mathsf{mpost}_p(C, m) =$ postcondition of method m in class C in p

Free Variables

The semantics of $\mathbf{SVL_{IDF}}$ will sometimes reason about the free variables of expressions or formulas.

We define $\mathsf{FV} : (\textsc{Expr} \cup \textsc{Formula}) \to \mathcal{P}^{\textsc{Var}}$ as

$$
\begin{aligned}
\mathsf{FV}(v) &= \emptyset \\
\mathsf{FV}(x) &= \{x\} \\
\mathsf{FV}(e\,.\,f) &= \mathsf{FV}(e)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{FV}(\texttt{true}) &= \emptyset \\
\mathsf{FV}(\texttt{(}e_1 \texttt{ = } e_2\texttt{)}) &= \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\
\mathsf{FV}(\texttt{(}e_1 \texttt{ } \neq \texttt{ } e_2\texttt{)}) &= \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\
\mathsf{FV}(\texttt{acc(}e\texttt{.}f\texttt{)}) &= \mathsf{FV}(e) \\
\mathsf{FV}(\phi_1 \texttt{ * } \phi_1) &= \mathsf{FV}(\phi_1) \cup \mathsf{FV}(\phi_2)
\end{aligned}
$$

Default Value of Given Type

$\mathbf{SVL_{IDF}}$ assigns the following default values to variables when they are declared and to fields when classes are instantiated.

We define $\mathsf{defaultValue} : \textsc{Type} \to \textsc{Val}$ as

$$\begin{aligned} \mathsf{defaultValue}(\texttt{int}) &= 0 \\ \mathsf{defaultValue}(C) &= \texttt{null} \end{aligned}$$

Required Access

Expressions mentioning fields are heap-dependent and thus require access. To enable treating all expressions (including bare variables and values) in a uniform fashion, we define a pseudo accessibility-predicate which is also defined for expressions that do not mention fields.

We define $\mathsf{acc} : \textsc{Expr} \to \textsc{Formula}$ as

$$\begin{aligned} \mathsf{acc}(v) &= \texttt{true} \\ \mathsf{acc}(x) &= \texttt{true} \\ \mathsf{acc}(e\texttt{.}f) &= \texttt{acc(}e\texttt{.}f\texttt{)} \end{aligned}$$

Preventing Writes

Under some circumstances (`hold`-statements, method bodies), overwriting a certain variable is not allowed in $\mathbf{SVL_{IDF}}$. To reliably check which variables are written to by a statement, we define the following function.

We define $\mathsf{mod} : \textsc{Stmt} \to \mathcal{P}^{\textsc{Var}}$ as

$$\begin{aligned} \mathsf{mod}(x \texttt{ := } e) &= \{\ x\ \} \\ \mathsf{mod}(x \texttt{ := new } C) &= \{\ x\ \} \\ \mathsf{mod}(x \texttt{ := } y\texttt{.}m\texttt{(}z\texttt{)}) &= \{\ x\ \} \\ \mathsf{mod}(\texttt{return } x) &= \{\ \texttt{result}\ \} \\ \mathsf{mod}(T\ x) &= \{\ x\ \} \\ \mathsf{mod}(\texttt{hold } p \texttt{ \{ } s \texttt{ \}}) &= \mathsf{mod}(s) \\ \mathsf{mod}(s_1\texttt{;}\ s_2) &= \mathsf{mod}(s_1) \cup \mathsf{mod}(s_2) \\ \mathsf{mod}(s) &= \emptyset \quad \textit{otherwise} \end{aligned}$$

4.1.2 Expression Evaluation

This section gives the rules for evaluating expressions in $\mathbf{SVL_{IDF}}$. Evaluating expressions is not only a fundamental part of subsequent definitions (like formula semantics), but also gives an idea about how memory is abstracted in $\mathbf{SVL_{IDF}}$.

In figure 4.2 we define a ternary partial evaluation function

$$\cdot, \cdot \vdash \cdot \Downarrow \cdot\ :\ \textsc{Heap} \times \textsc{VarEnv} \times \textsc{Expr} \rightharpoonup \textsc{Val}$$

$\boxed{H, \rho \vdash e \Downarrow v}$

$$\frac{}{H, \rho \vdash x \Downarrow \rho(x)}\ \text{EEVar} \qquad \frac{}{H, \rho \vdash v \Downarrow v}\ \text{EEValue} \qquad \frac{H, \rho \vdash e \Downarrow o \qquad H(o) = \langle C, l \rangle}{H, \rho \vdash e.f \Downarrow l(f)}\ \text{EEAcc}$$

Figure 4.2. $\mathbf{SVL_{IDF}}$: Evaluating Expressions

that uses the following definitions of heaps H and local variable environments ρ:

$$\begin{aligned} H \in \textsc{Heap} &= \textsc{Loc} \rightharpoonup (\textsc{ClassName} \times (\textsc{FieldName} \rightharpoonup \textsc{Val})) \\ \rho \in \textsc{VarEnv} &= \textsc{Var} \rightharpoonup \textsc{Val} \end{aligned}$$

4.1.3 Footprints and Framing

The integral advantage of IDF is the ability to use heap-dependent predicates in formulas. Accessibility-predicates are explicitly tracked as part of formulas and represent exclusive access to a certain field. As illustrated in section 2.3 sound reasoning is only possible if a formula contains access for all the fields it mentions. This property is called self-framing and will be formalized in this section, drawing on the definitions given by Summers and Drossopoulou [32].

A related concept, necessary for formalizing self-framing but also for the semantics of $\mathbf{SVL_{IDF}}$ in general, is the notion of footprints. The footprint of a formula is the set of fields it has access to, which can be interpreted in two ways.

Static Footprint

Figure 4.3 defines $\lfloor \cdot \rfloor : \textsc{Formula} \to \textsc{StaticFootprint}$, a function for obtaining static footprints where

$$\textsc{StaticFootprint} \stackrel{\text{def}}{=} \mathcal{P}^{\textsc{Expr} \times \textsc{FieldName}}$$

Static footprints are required for static reasoning, mainly for determining whether or not a formula is self-framing. Later, they will be helpful for formalizing the gradual Hoare logic of $\mathbf{GVL_{IDF}}$.

Example 4.2 (Static Footprints of Formulas)**.**

$$\begin{aligned} \lfloor \texttt{(x = 3) * (p.age} \neq \texttt{24)} \rfloor &= \emptyset \\ \lfloor \texttt{(x = 3) * acc(p.age) * (p.age} \neq \texttt{24)} \rfloor &= \{\langle \texttt{p}, \texttt{age} \rangle\} \\ \lfloor \texttt{acc(p.age) * acc(p.name) * acc(x.f)} \rfloor &= \{\langle \texttt{p}, \texttt{age} \rangle, \langle \texttt{p}, \texttt{name} \rangle, \langle \texttt{x}, \texttt{f} \rangle\} \end{aligned}$$

Dynamic Footprint

Figure 4.4 defines $\lfloor \cdot \rfloor_{\cdot,\cdot} : \textsc{Heap} \times \textsc{VarEnv} \times \textsc{Formula} \rightharpoonup \textsc{DynamicFootprint}$, a function for obtaining dynamic footprints where

$$\textsc{DynamicFootprint} \stackrel{\text{def}}{=} \mathcal{P}^{\textsc{Loc} \times \textsc{FieldName}}$$

Dynamic footprints are required for the small-step semantics of $\mathbf{SVL_{IDF}}$. Note that the subtle difference of evaluating the expression (compared to static footprints) can have two effects: First, evaluation might fail (e.g. due to accessing non-existing fields or dereferencing null but also due to not returning a memory location o) which results in the partiality of the function. Second, multiple syntactically distinct accessibility-predicates might result in one tuple in the dynamic footprint.

Example 4.3 (Static and Dynamic Footprints)**.**
Let H and ρ be defined such that $a, b, c \in$ Expr all evaluate to the same memory location $o \in$ Loc. Furthermore, $d \in$ Expr evaluates to `null`.

Then the following footprints are calculated:

$$\lfloor \texttt{acc(a.f) * acc(b.f) * acc(c.g) * acc(d.h)} \rfloor = \{\langle a, f\rangle, \langle b, f\rangle, \langle c, g\rangle, \langle d, h\rangle\}$$
$$\lfloor \texttt{acc(a.f) * acc(b.f) * acc(c.g) * acc(d.h)} \rfloor_{H,\rho} = \{\langle o, f\rangle, \langle o, g\rangle\}$$

Note that the scenario in example 4.3 violates the intuition behind the separating conjunction (introduced in section 2.3), which is supposed to guarantee that both "sides" of it have access to disjoint memory locations. This principle is violated by above formula because of `acc(a.f)` and `acc(b.f)`. The formula semantics introduced in section 4.1.5 will make sure that the separating conjunction is not violated, rendering above formulas unsatisfiable in any program state where `a` and `b` alias.

$\boxed{\lfloor \phi \rfloor = A_s}$

$$\begin{aligned}
\lfloor \texttt{true} \rfloor &= \emptyset \\
\lfloor (e_1 \texttt{ = } e_2) \rfloor &= \emptyset \\
\lfloor (e_1 \texttt{ } \neq \texttt{ } e_2) \rfloor &= \emptyset \\
\lfloor \texttt{acc(}e\texttt{.}f\texttt{)} \rfloor &= \{\langle e, f\rangle\} \\
\lfloor \phi_1 \texttt{ * } \phi_2 \rfloor &= \lfloor \phi_1 \rfloor \cup \lfloor \phi_2 \rfloor
\end{aligned}$$

Figure 4.3. $\mathbf{SVL_{IDF}}$: Static Footprint

$\boxed{\lfloor \phi \rfloor_{H,\rho} = A_d}$

$$\begin{aligned}
\lfloor \texttt{true} \rfloor_{H,\rho} &= \emptyset \\
\lfloor (e_1 \texttt{ = } e_2) \rfloor_{H,\rho} &= \emptyset \\
\lfloor (e_1 \texttt{ } \neq \texttt{ } e_2) \rfloor_{H,\rho} &= \emptyset \\
\lfloor \texttt{acc(}e\texttt{.}f\texttt{)} \rfloor_{H,\rho} &= \{\langle o, f\rangle\} \text{ where } H, \rho \vdash e \Downarrow o \\
\lfloor \phi_1 \texttt{ * } \phi_2 \rfloor_{H,\rho} &= \lfloor \phi_1 \rfloor_{H,\rho} \cup \lfloor \phi_2 \rfloor_{H,\rho}
\end{aligned}$$

Figure 4.4. $\mathbf{SVL_{IDF}}$: Dynamic Footprint

With the notion of footprints, we can determine whether an expression is "framed" by a given footprint, i.e. whether the footprint contains all the fields the expression contains. We formalize this check using a predicate $\vdash_{\texttt{frm}} \subseteq$ StaticFootprint $\times$ Expr, defined

$\boxed{A_s \vdash_{\texttt{frm}} e}$

$$\frac{}{A_s \vdash_{\texttt{frm}} x}\ \textsc{FrmVar} \qquad \frac{}{A_s \vdash_{\texttt{frm}} v}\ \textsc{FrmVal} \qquad \frac{(e, f) \in A \qquad A_s \vdash_{\texttt{frm}} e}{A_s \vdash_{\texttt{frm}} e.f}\ \textsc{FrmField}$$

Figure 4.5. $\mathbf{SVL_{IDF}}$: Framing Expressions

$\boxed{A_s \vdash_{\texttt{sfrm}} \phi}$

$$\frac{}{A_s \vdash_{\texttt{sfrm}} \texttt{true}}\ \textsc{SfrmTrue} \qquad \frac{A_s \vdash_{\texttt{frm}} e_1 \qquad A_s \vdash_{\texttt{frm}} e_2}{A_s \vdash_{\texttt{sfrm}} (e_1 \texttt{ = } e_2)}\ \textsc{SfrmEqual}$$

$$\frac{A_s \vdash_{\texttt{frm}} e_1 \qquad A_s \vdash_{\texttt{frm}} e_2}{A_s \vdash_{\texttt{sfrm}} (e_1 \neq e_2)}\ \textsc{SfrmNEqual} \qquad \frac{A_s \vdash_{\texttt{frm}} e}{A_s \vdash_{\texttt{sfrm}} \texttt{acc}(e.f)}\ \textsc{SfrmAcc}$$

$$\frac{A_s \vdash_{\texttt{sfrm}} \phi_1 \qquad A_s \cup \lfloor \phi_1 \rfloor \vdash_{\texttt{sfrm}} \phi_2}{A_s \vdash_{\texttt{sfrm}} \phi_1 * \phi_2}\ \textsc{SfrmSepOp}$$

Figure 4.6. $\mathbf{SVL_{IDF}}$: Framing Formulas

in figure 4.5.

At last, we can define a predicate $\vdash_{\texttt{sfrm}} \subseteq \textsc{StaticFootprint} \times \textsc{Formula}$ that checks whether given footprint is sufficient to frame an entire formula, i.e. all the expressions the formula contains. Figure 4.6 formalizes the predicate. Note how SfrmSepOp augments the footprint the right sub-formula is checked against using the footprint of the left sub-formula. This way, access predicates within a formula are able to frame expressions in the same formula, if mentioned in the right order.

Example 4.4 (Framing Formulas)**.**

$\emptyset \vdash_{\texttt{sfrm}} \texttt{(p.age} \neq \texttt{24)}$	does not hold
$\{\langle \texttt{p}, \texttt{age} \rangle\} \vdash_{\texttt{sfrm}} \texttt{(p.age} \neq \texttt{24)}$	holds
$\emptyset \vdash_{\texttt{sfrm}} \texttt{acc(p.age)} * \texttt{(p.age} \neq \texttt{24)}$	holds
$\emptyset \vdash_{\texttt{sfrm}} \texttt{(p.age} \neq \texttt{24)} * \texttt{acc(p.age)}$	does not hold

We can now define self-framing as a special case of framing, namely framing without relying on an "external" footprint.

Definition 4.5 (Self-Framed Formula)**.** *A formula ϕ is **self-framed** iff*

$$\emptyset \vdash_{sfrm} \phi$$

For the remainder of this work we omit empty footprints:

$$\vdash_{sfrm} \phi$$

Let SfrmFormula $\subseteq$ SatFormula *be the set of* ***self-framed and satisfiable*** *formulas.*

4.1.4 Program State

The set of program states is defined as ProgramState = Heap $\times$ Stack where

$$\begin{aligned} S \in \textsc{Stack} &::= E \cdot S \mid \mathsf{nil} \\ E \in \textsc{StackFrame} &= \textsc{VarEnv} \times \textsc{DynamicFootprint} \times \textsc{Stmt} \end{aligned}$$

Each stack frame has an environment VarEnv for local variables, tracks a set of accessible fields DynamicFootprint and stores a continuation Stmt. Stack frames will be introduced by method calls, but also by dedicated scopes as used by the `hold` statement.

4.1.5 Formula Semantics

Formula semantics of $\mathbf{SVL_{IDF}}$ only depend on the heap and on the top most stack frame of a program state (more specifically: the local variable environment and the accessible fields, but not the continuation statement). To drastically simplify notation, we will therefore define $\cdot \vDash \cdot \subseteq$ ProgramState $\times$ Formula (as postulated in section 3.1) indirectly:

$$\frac{H, \rho, A \vDash \phi}{\langle H, \langle \rho, A, s\rangle \cdot S\rangle \vDash \phi}\ \textsc{EvalFrm}$$

Figure 4.7 defines the actual semantics as a predicate

$$\vDash \subseteq \textsc{Heap} \times \textsc{VarEnv} \times \textsc{DynamicFootprint} \times \textsc{Formula}$$

We assume that denotational semantics $[\![\cdot]\!]$, implication and satisfiability are defined as described in section 3.1.

A number of rules arise naturally from this semantical definition of implication.

Example 4.6 ($\mathbf{SVL_{IDF}}$: Admissible Rules)**.**

Projection

$$\frac{}{\phi_1 * \phi_2 \Rightarrow \phi_1}\ \textsc{FrmProjL}$$

$$\frac{}{\phi_1 * \phi_2 \Rightarrow \phi_2}\ \textsc{FrmProjR}$$

Transitivity

$$\frac{\phi_a \Rightarrow \phi_b \qquad \phi_b \Rightarrow \phi_c}{\phi_a \Rightarrow \phi_c}\ \textsc{FrmTrans}$$

$\boxed{H, \rho, A \vDash \phi}$

$$\frac{}{H, \rho, A \vDash \texttt{true}}\ \textsc{EATrue} \qquad \frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad H, \rho \vdash e_2 \Downarrow v_2 \qquad v_1 = v_2}{H, \rho, A \vDash (e_1\ \texttt{=}\ e_2)}\ \textsc{EAEqual}$$

$$\frac{H, \rho \vdash e_1 \Downarrow v_1 \qquad H, \rho \vdash e_2 \Downarrow v_2 \qquad v_1 \neq v_2}{H, \rho, A \vDash (e_1\ \texttt{≠}\ e_2)}\ \textsc{EANEqual}$$

$$\frac{H, \rho \vdash e \Downarrow o \qquad H, \rho \vdash e.f \Downarrow v \qquad \langle o, f \rangle \in A}{H, \rho, A \vDash \texttt{acc}(e.f)}\ \textsc{EAAcc}$$

$$\frac{A_1 = A \backslash A_2 \qquad H, \rho, A_1 \vDash \phi_1 \qquad H, \rho, A_2 \vDash \phi_2}{H, \rho, A \vDash \phi_1 * \phi_2}\ \textsc{EASepOp}$$

Figure 4.7. $\mathbf{SVL_{IDF}}$: Evaluating Formulas

$$\frac{}{(a\ \texttt{=}\ b) * (b\ \texttt{=}\ c) \Rightarrow (a\ \texttt{=}\ c)}\ \textsc{FrmTransEq}$$

Aliasing Prevention

$$\frac{}{\texttt{acc}(e_1.f) * \texttt{acc}(e_2.f) \Rightarrow (e_1\ \texttt{≠}\ e_2)}\ \textsc{FrmAlias}$$

Conjunction

$$\frac{\begin{array}{c}\phi \Rightarrow \phi_a \\ \phi \Rightarrow \phi_b \qquad \forall H \in \textsc{Heap},\ \rho \in \textsc{VarEnv}.\ \lfloor \phi_a \rfloor_{H,\rho} \cap \lfloor \phi_b \rfloor_{H,\rho} = \emptyset\end{array}}{\phi \Rightarrow \phi_a * \phi_b}\ \textsc{FrmComp}$$

4.1.5.1 Regular Conjunction

The syntax of $\mathbf{SVL_{IDF}}$ does not include a regular (non-separating) conjunction operator $\cdot \wedge \cdot$.

It would prove useful under some circumstances (e.g. to simply combine the knowledge of several formulas into one formula), but would complicate formula semantics and subsequent reasoning significantly. In this section we will show that there is also no way of defining the operator in terms of existing syntax, i.e. as syntactic sugar.

Intuitively, we expect a conjunction to be satisfied by a program environment iff both of its operands are satisfied by the same program environment. This intuition can be

formalized as

$$[\![\phi_1 \ \wedge \ \phi_2]\!] = [\![\phi_1]\!] \cap [\![\phi_2]\!]$$

Example 4.7 (Tricky Access)**.** Consider $\phi \stackrel{\mathsf{def}}{=} \texttt{acc(a.f)} \wedge \texttt{acc(b.f)}$. According to above equation, ϕ is satisfied by an environment π, if it contains access to `a.f` and `b.f`. However, whether `a` and `b` alias or not is irrelevant for satisfying ϕ, i.e. both

$$\phi_1 = \texttt{acc(a.f)} * \texttt{acc(b.f)}$$

and

$$\phi_2 = \texttt{(a = b)} * \texttt{acc(b.f)}$$

imply ϕ.

As above example shows, regular conjunction can be indifferent about aliasing in the presence of access. Unfortunately, there is no way to be express such indifference using the syntax of $\mathbf{SVL_{IDF}}$: Access to a field can be given either by inserting an accessibility predicate (implying that the field references a *different* heap location than all other access referenced far) or by inserting an equation, relating it to existing access (explicitly stating that some fields reference the *same* memory). The fundamental problem is the linear nature of accessibility-predicates, which makes explicit whether certain memory locations alias or not. Therefore, any formula giving access to fields `a.f` and `b.f` will always imply either `(a = b)` or `(a ≠ b)`.

4.1.6 Static Semantics

The static semantics of $\mathbf{SVL_{IDF}}$ consist of typing rules and a Hoare logic making use of those typing rules. Both are implicitly parameterized over some program $p \in \textsc{Program}$, necessary for example to extract the type of a field or retrieve the contract of a called method.

Figure 4.8 inductively defines a typing predicate $\cdot \vdash \cdot : \cdot \subseteq \textsc{TypeEnv} \times \textsc{Expr} \times \textsc{Type}$ where

$$\textsc{TypeEnv} \stackrel{\mathsf{def}}{=} \textsc{Var} \rightharpoonup \textsc{Type}$$

$\boxed{\Gamma \vdash e : T}$

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \textsc{StValNum} \qquad \frac{}{\Gamma \vdash \texttt{null} : C}\ \textsc{StValNull} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \textsc{StVar}$$

$$\frac{\Gamma \vdash e : C \qquad \mathsf{fieldType}_p(C, f) = T}{\Gamma \vdash e.f : T}\ \textsc{StField}$$

Figure 4.8. $\mathbf{SVL_{IDF}}$: Static Typing of Expressions

Figure 4.9 inductively defines a Hoare logic for $\mathbf{SVL_{IDF}}$.

$\boxed{\Gamma \vdash \{\phi_{pre}\}\ s\ \{\phi_{post}\}}$

$$\frac{\phi \Rightarrow \phi'}{\Gamma \vdash \{\phi\}\ \texttt{skip}\ \{\phi'\}}\ \textsc{HSkip}$$

$$\frac{\phi \Rightarrow \phi' \qquad \vdash_{\texttt{sfrm}} \phi' \qquad x \notin \mathsf{FV}(\phi') \qquad \Gamma \vdash x : C \qquad \mathsf{fields}_p(C) = \overline{T\ f;}}{\Gamma \vdash \{\phi\}\ x\ \texttt{:= new}\ C\ \{\phi' * (x\ \texttt{!=}\ \texttt{null}) * \overline{\texttt{acc}(x.f_i) * (x.f_i\ \texttt{=}\ \mathsf{defaultValue}(T_i))}\}}\ \textsc{HAlloc}$$

$$\frac{\begin{array}{c}\phi \Rightarrow \texttt{acc}(x.f) * \phi' \\ \vdash_{\texttt{sfrm}} \phi' \qquad \Gamma \vdash x : C \qquad \Gamma \vdash y : T \qquad \vdash C.f : T\end{array}}{\Gamma \vdash \{\phi\}\ x.f\ \texttt{:=}\ y\ \{\phi' * \texttt{acc}(x.f) * (x\ \texttt{!=}\ \texttt{null}) * (x.f\ \texttt{=}\ y)\}}\ \textsc{HFieldAssign}$$

$$\frac{\begin{array}{c}\phi \Rightarrow \phi' \qquad \phi \Rightarrow \mathsf{acc}(e) \\ \vdash_{\texttt{sfrm}} \phi' \qquad x \notin \mathsf{FV}(\phi') \qquad x \notin \mathsf{FV}(e) \qquad \Gamma \vdash x : T \qquad \Gamma \vdash e : T\end{array}}{\Gamma \vdash \{\phi\}\ x\ \texttt{:=}\ e\ \{\phi' * (x\ \texttt{=}\ e)\}}\ \textsc{HVarAssign}$$

$$\frac{\phi \Rightarrow \phi' \qquad \vdash_{\texttt{sfrm}} \phi' \qquad \texttt{result} \notin \mathsf{FV}(\phi') \qquad \Gamma \vdash x : T \qquad \Gamma \vdash \texttt{result} : T}{\Gamma \vdash \{\phi\}\ \texttt{return}\ x\ \{\phi' * (\texttt{result}\ \texttt{=}\ x)\}}\ \textsc{HReturn}$$

$$\frac{\begin{array}{c}\Gamma \vdash y : C \\ \mathsf{method}_p(C, m) = T_r\ \ m(T_p\ \ z)\ \texttt{requires}\ \phi_{pre};\ \texttt{ensures}\ \phi_{post};\ \{\ \_\ \} \\ \Gamma \vdash x : T_r \qquad \Gamma \vdash z' : T_p \\ \phi \Rightarrow (y\ \texttt{!=}\ \texttt{null}) * \phi_p * \phi' \qquad \vdash_{\texttt{sfrm}} \phi' \qquad x \notin \mathsf{FV}(\phi') \qquad x \neq y \wedge x \neq z' \\ \phi_p = \phi_{pre}[y, z'/\texttt{this}, z] \qquad \phi_q = \phi_{post}[y, z', x/\texttt{this}, z, \texttt{result}]\end{array}}{\Gamma \vdash \{\phi\}\ x\ \texttt{:=}\ y.m(z')\ \{\phi' * \phi_q\}}\ \textsc{HCall}$$

$$\frac{\phi \Rightarrow \phi_a}{\Gamma \vdash \{\phi\}\ \texttt{assert}\ \phi_a\ \{\phi\}}\ \textsc{HAssert} \qquad \frac{\phi \Rightarrow \phi_r * \phi' \qquad \vdash_{\texttt{sfrm}} \phi'}{\Gamma \vdash \{\phi\}\ \texttt{release}\ \phi_r\ \{\phi'\}}\ \textsc{HRelease}$$

$$\frac{\begin{array}{c}x \notin \mathsf{dom}(\Gamma) \\ x \notin \mathsf{FV}(\phi) \qquad \Gamma, x : T \vdash \{(x\ \texttt{=}\ \mathsf{defaultValue}(T)) * \phi\}\ s\ \{\phi'\}\end{array}}{\Gamma \vdash \{\phi\}\ T\ \ x;\ \ \mathtt{s}\ \{\phi'\}}\ \textsc{HDeclare}$$

$$\frac{\begin{array}{c}\vdash_{\texttt{sfrm}} \phi \qquad \phi_f \Rightarrow \phi_r * \phi' \\ \phi' \Rightarrow \phi \qquad \mathsf{FV}(\phi') = \mathsf{FV}(\phi) \qquad \mathsf{mod}(s) \cap \mathsf{FV}(\phi) = \emptyset \qquad \Gamma \vdash \{\phi_r\}\ s\ \{\phi'_r\}\end{array}}{\Gamma \vdash \{\phi_f\}\ \texttt{hold}\ \phi\ \{\ s\ \}\ \{\phi'_r * \phi'\}}\ \textsc{HHold}$$

$$\frac{\Gamma \vdash \{\phi_p\}\ s_1\ \{\phi_q\} \qquad \Gamma \vdash \{\phi_q\}\ s_2\ \{\phi_r\}}{\Gamma \vdash \{\phi_p\}\ s_1;\ \ s_2\ \{\phi_r\}}\ \textsc{HSeq}$$

Figure 4.9. SVL$_{\mathbf{IDF}}$: Hoare Logic

4.1.7 Dynamic Semantics

The small-step semantics $\cdot \longrightarrow \cdot : \textsc{ProgramState} \rightharpoonup \textsc{ProgramState}$ of $\mathbf{SVL_{IDF}}$ are defined inductively in figure 4.10. Note that right-associativity of `;` and termination of sequences with `skip` (see section 4.1.1) obviate the need for dedicated sequence rules.

Using inductive rules to define a partial function, we have to make sure that at most one result is deducible for every input.

Lemma 4.8 ($\cdot \longrightarrow \cdot$ Well-Defined)**.**
The small-step semantics of $\boldsymbol{SVL_{IDF}}$ is well-defined.

4.1.8 Well-Formedness

Apart from checking method contracts, a verifier or compiler may enforce further rules before accepting a program as "well-formed". For $\mathbf{SVL_{IDF}}$ we give the rules formalized in figure 4.11.

The premises of OkMethod make sure that reasoning about calls is sound. As expected, the method contract is checked, while also making sure that it contains self-framing formulas. Furthermore, the free variables are restricted to those occurring in the method signature. The following example illustrates why this is necessary.

Example 4.9 (Leaking Postcondition)**.**

```
int identity(int a)
    requires true;
    ensures  (b = 3);
{
    int b;
    b = 3;
    return a;
}
```

While the method itself passes static verification, it could lead to unsound proofs when called. Note how HCall forwards the postcondition after replacing known variables with their counterparts. `(b = 3)` is unaffected by this replacement, ending up in the postcondition of the call statement.

Should the call site also know a variable `b`, then `(b = 3)` will most likely not reflect the true state of that variable.

For similar reasons, we prevent writes to the method's parameter and `this`. Changes to those variables would otherwise end up in the postcondition which is forwarded to the call site. However, since writes to the parameter or `this` do not affect variables at the call site (due to call-by-value semantics), this information would be false.

$\boxed{\pi \longrightarrow \pi}$

$$\frac{}{\langle H, \langle \rho, A, \texttt{skip;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, A, s\rangle \cdot S\rangle}\ \textsc{SsSkip}$$

$$\frac{H, \rho \vdash x \Downarrow o \qquad H, \rho \vdash y \Downarrow v_y \qquad \langle o, f\rangle \in A \qquad H' = H[o \mapsto [f \mapsto v_y]]}{\langle H, \langle \rho, A, x\,.\,f\ \texttt{:=}\ y\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H', \langle \rho, A, s\rangle \cdot S\rangle}\ \textsc{SsFieldAssign}$$

$$\frac{x \notin \mathsf{FV}(e) \qquad H, \rho, A \vDash \mathsf{acc}(e) \qquad H, \rho \vdash e \Downarrow v \qquad \rho' = \rho[x \mapsto v]}{\langle H, \langle \rho, A, x\ \texttt{:=}\ e\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', A, s\rangle \cdot S\rangle}\ \textsc{SsVarAssign}$$

$$\frac{\begin{array}{c} o \notin \mathsf{dom}(H) \qquad \mathsf{fields}_p(C) = \overline{T\ \ f\texttt{;}} \\ \rho' = \rho[x \mapsto o] \qquad A' = A \cup \overline{\langle o, f_i\rangle} \qquad H' = H[o \mapsto [\overline{f_i \mapsto \mathsf{defaultValue}(T_i)}]] \end{array}}{\langle H, \langle \rho, A, x\ \texttt{:=}\ \texttt{new}\ C\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H', \langle \rho', A', s\rangle \cdot S\rangle}\ \textsc{SsAlloc}$$

$$\frac{H, \rho \vdash x \Downarrow v_x \qquad \rho' = \rho[\texttt{result} \mapsto v_x]}{\langle H, \langle \rho, A, \texttt{return}\ x\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', A, s\rangle \cdot S\rangle}\ \textsc{SsReturn}$$

$$\frac{\begin{array}{c} H, \rho \vdash y \Downarrow o \qquad H, \rho \vdash z \Downarrow v \\ H(o) = \langle C, \_\rangle \qquad \mathsf{method}_p(C, m) = T_r\ \ m\texttt{(}T\ \ w\texttt{)}\ \texttt{requires}\ \phi\texttt{;}\ \texttt{ensures}\ \_\texttt{;}\ \texttt{\{}\ r\ \texttt{\}} \\ \rho' = [\texttt{result} \mapsto \mathsf{defaultValue}(T_r), \texttt{this} \mapsto o, w \mapsto v] \qquad H, \rho', A \vDash \phi \qquad A' = \lfloor \phi \rfloor_{H,\rho'} \end{array}}{\langle H, \langle \rho, A, x\ \texttt{:=}\ y\texttt{.}m\texttt{(}z\texttt{);}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', A', r\rangle \cdot \langle \rho, A \backslash A', x\ \texttt{:=}\ y\texttt{.}m\texttt{(}z\texttt{);}\ s\rangle \cdot S\rangle}\ \textsc{SsCall}$$

$$\frac{\begin{array}{c} H, \rho \vdash y \Downarrow o \\ H(o) = \langle C, \_\rangle \qquad \mathsf{mpost}_p(C, m) = \phi \qquad H, \rho', A' \vDash \phi \qquad H, \rho' \vdash \texttt{result} \Downarrow v_r \end{array}}{\langle H, \langle \rho', A', \texttt{skip}\rangle \cdot \langle \rho, A, x\ \texttt{:=}\ y\texttt{.}m\texttt{(}z\texttt{);}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho[x \mapsto v_r], A \cup A', s\rangle \cdot S\rangle}\ \textsc{SsCallFinish}$$

$$\frac{H, \rho, A \vDash \phi}{\langle H, \langle \rho, A, \texttt{assert}\ \phi\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, A, s\rangle \cdot S\rangle}\ \textsc{SsAssert}$$

$$\frac{H, \rho, A \vDash \phi \qquad A' = A \backslash \lfloor \phi \rfloor_{H,\rho}}{\langle H, \langle \rho, A, \texttt{release}\ \phi\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, A', s\rangle \cdot S\rangle}\ \textsc{SsRelease}$$

$$\frac{\rho' = \rho[x \mapsto \mathsf{defaultValue}(T)]}{\langle H, \langle \rho, A, T\ \ x\texttt{;}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', A, s\rangle \cdot S\rangle}\ \textsc{SsDeclare}$$

$$\frac{H, \rho, A \vDash \phi \qquad A' = \lfloor \phi \rfloor_{H,\rho}}{\langle H, \langle \rho, A, \texttt{hold}\ \phi\ \texttt{\{}\ s'\ \texttt{\};}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho, A \backslash A', \overline{s'}\rangle \cdot \langle \rho, A', \texttt{hold}\ \phi\ \texttt{\{}\ s'\ \texttt{\};}\ s\rangle \cdot S\rangle}\ \textsc{SsHold}$$

$$\frac{}{\langle H, \langle \rho', A', \texttt{skip}\rangle \cdot \langle \rho, A, \texttt{hold}\ \phi\ \texttt{\{}\ s'\ \texttt{\};}\ s\rangle \cdot S\rangle \longrightarrow \langle H, \langle \rho', A \cup A', s\rangle \cdot S\rangle}\ \textsc{SsHoldFinish}$$

Figure 4.10. $\mathbf{SVL_{IDF}}$: Small-Step Semantics

$$\frac{\overline{cls_i\ \mathsf{OK}} \qquad \vdash \{\mathtt{true}\}\ s\ \{\phi\}}{(\overline{cls_i}\ s)\ \mathsf{OK}}\ \textsc{OkProgram}$$

$$\frac{\text{unique } field\text{-names} \qquad \text{unique } method\text{-names} \qquad \overline{method_i\ \mathsf{OK\ in}\ C}}{(\mathtt{class}\ C\ \mathtt{\{}\ \overline{field_i}\ \overline{method_i}\ \mathtt{\}})\ \mathsf{OK}}\ \textsc{OkClass}$$

$$\frac{\begin{array}{c} x : T_x, \mathtt{this} : C, \mathtt{result} : T_m \vdash \{\phi_1\}\ s\ \{\phi_2\} \\ \mathsf{FV}(\phi_1) \subseteq \{x, \mathtt{this}\} \qquad \mathsf{FV}(\phi_2) \subseteq \{x, \mathtt{this}, \mathtt{result}\} \\ \vdash_{\mathtt{sfrm}} \phi_1 \qquad \vdash_{\mathtt{sfrm}} \phi_2 \qquad x, \mathtt{this} \notin \mathsf{mod}(s) \end{array}}{(T_m\ \ m\mathtt{(}T_x\ \ x\mathtt{)}\ \mathtt{requires}\ \phi_1\mathtt{;}\ \mathtt{ensures}\ \phi_2\mathtt{;}\ \mathtt{\{}\ s\ \mathtt{\}})\ \mathsf{OK\ in}\ C}\ \textsc{OkMethod}$$

Figure 4.11. $\mathbf{SVL_{IDF}}$: Well-Formedness

4.1.9 Soundness

Lemma 4.10 (Soundness of $\mathbf{SVL_{IDF}}$)**.**
Given some well-formed program p, the Hoare logic is sound w.r.t. the small-step semantics (both parameterized over p).

4.2 Gradual Syntax

The first step of deriving $\mathbf{GVL_{IDF}}$ is defining its modified syntax. As motivated in section 3.2 we first extend the formula syntax, afterwards everything that depends on it.

In our design of gradual formulas we aim for "bounded unknown" formulas as introduced in section 3.2.2, however with separating conjunction as a connective:

Syntax

$$\widetilde{\phi} \quad ::= \quad \phi \mid \mathtt{?} * \phi$$

We pose $\mathtt{?} \stackrel{\mathsf{def}}{=} \mathtt{?} * \mathtt{true}$.

Concretization

$$\gamma(\phi) = \{\ \phi\ \}$$
$$\gamma(\mathtt{?} * \phi) = \{\ \phi' \in \textsc{SfrmFormula} \mid \phi' \Rightarrow \phi\ \}$$

Note that we do not require the static part of $\mathtt{?} * \phi$ to be self-framing. On the contrary, we want concretizations to be able to provide framing for an otherwise unframed formula. (We decided to put ? in front of the static part to emphasize

that fact.) This allows programmers to resort to gradual formulas when being uncertain or indifferent about the concrete framing of a heap-dependent formula.

Example 4.11 (Unknown Framing)**.** The programmer may want to express `(a.name = b.name)`, but is indifferent about whether `a` and `b` alias or not. As shown in section 4.1.5.1, both options are not expressible at the same time using a static formula. Fortunately, `?` can be used to frame the formula, covering both alternatives:

$$\texttt{acc(a.name)} * \texttt{acc(b.name)} * \texttt{(a.name = b.name)} \in \gamma(\texttt{?} * \texttt{(a.name = b.name)})$$
$$\texttt{(a = b)} * \texttt{acc(b.name)} * \texttt{(b.name = b.name)} \in \gamma(\texttt{?} * \texttt{(a.name = b.name)})$$

Static Knowledge Extraction

We define a helper function $\mathsf{static} : \widetilde{\text{Formula}} \to \text{Formula}$ that extracts the static part of a gradual formula as follows:

$$\mathsf{static}(\phi) = \phi$$
$$\mathsf{static}(\texttt{?} * \phi) = \phi$$

We want to only allow gradual formulas in method contracts, resulting in the gradual syntax shown in figure 4.12. Note how the small change propagates throughout

$$\begin{aligned}
\widetilde{program} \in \widetilde{\text{Program}} &::= \overline{\widetilde{cls}}\ \ s \\
\widetilde{cls} \in \widetilde{\text{Class}} &::= \texttt{class}\ C\ \texttt{\{}\ \overline{field}\ \ \overline{\widetilde{method}}\ \texttt{\}} \\
\widetilde{method} \in \widetilde{\text{Method}} &::= T\ \ m\texttt{(}T\ \ x\texttt{)}\ \ \widetilde{contract}\ \texttt{\{}\ s\ \texttt{\}} \\
\widetilde{contract} \in \widetilde{\text{Contract}} &::= \texttt{requires}\ \widetilde{\phi}\texttt{;}\ \texttt{ensures}\ \widetilde{\phi}\texttt{;} \\
\widetilde{\phi} \in \widetilde{\text{Formula}} &::= \phi \mid \texttt{?} * \phi
\end{aligned}$$

Figure 4.12. $\mathbf{GVL_{IDF}}$: Syntax

other constructs, all the way up to gradual programs. However, the change has no direct impact on statement syntax, allowing us to define $\widetilde{\text{Stmt}} \overset{\mathsf{def}}{=} \text{Stmt}$ and therefore $\widetilde{\text{ProgramState}} \overset{\mathsf{def}}{=} \text{ProgramState}$. Note that we also have not decorated s in figure 4.12 although it is formally drawn from $\widetilde{\text{Stmt}}$.

However, note that static and dynamic semantics of the call statement x `:=` $y.m(z)$ will still have to be lifted as m now references a gradual method contract (see section 3.3 for detailed discussion).

4.3 Gradual Liftings

With a concretization γ at hand, we can use gradual lifting (see section 3.5) to derive gradual versions of some commonly used predicates. Furthermore, we deterministically

lift (see section 3.7.2) predicates that appear as components in the Hoare logic (see figure 4.9).

Implementation strategies for all of the following predicates can be found in the appendix A.2.

Gradual Formula Semantics

Let $\cdot \mathrel{\widetilde{\models}} \cdot \subseteq \textsc{ProgramState} \times \widetilde{\textsc{Formula}}$ be an optimal predicate lifting of $\cdot \models \cdot$
(defined in section 4.1.5)

Gradual Self-Framing

Let $\mathrel{\widetilde{\vdash}}_{\texttt{sfrm}} \cdot \subseteq \widetilde{\textsc{Formula}}$ be an optimal predicate lifting of $\vdash_{\texttt{sfrm}} \cdot$
(defined in section 4.1.3)

Gradual Implication of Static Formula

Let $\vec{P}_{\phi_t} \subseteq \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ be an optimal deterministic lifting of
$P_{\phi_t}(\phi_1, \phi_2) \overset{\mathsf{def}}{=} \phi_1 \Rightarrow \phi_t \wedge \phi_1 = \phi_2$
(shape of HAssert, found as component of HVarAssign and HHold)
We define a more intuitive notation that matches the original static predicate:

$$\widetilde{\phi_2} \vdash \widetilde{\phi_1} \mathrel{\widetilde{\Rrightarrow}} \phi_t \iff \vec{P}_{\phi_t}(\widetilde{\phi_1}) = \widetilde{\phi_2}$$

Gradual Formula Extraction

Let $\vec{P} \subseteq \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ be an optimal deterministic lifting of
$P(\phi_1, \phi_t, \phi_2) \overset{\mathsf{def}}{=} \phi_1 \Rightarrow \phi_t * \phi_2 \ \wedge \vdash_{\texttt{sfrm}} \phi_2$
(shape of HRelease, found as component of HFieldAssign, HCall and HHold)
We define a more intuitive notation that reflects the dual nature with "$*$":

$$\widetilde{\phi} \div \widetilde{\phi_t} \overset{\mathsf{def}}{=} \vec{P}(\widetilde{\phi}, \widetilde{\phi_t})$$

Gradual Variable Extraction

Let $\overline{x} \in \mathcal{P}^{\textsc{Var}}$ be a set of variables.
Let $\vec{P}_{\overline{x}} \subseteq \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ be an optimal deterministic lifting of
$P_{\overline{x}}(\phi_1, \phi_2) \overset{\mathsf{def}}{=} \phi_1 \Rightarrow \phi_2 \ \wedge \ \mathsf{FV}(\phi_2) \cap \overline{x} = \emptyset \ \wedge \ \vdash_{\texttt{sfrm}} \phi_2$
(found as component of HAlloc, HVarAssign, HReturn, HCall and in a sense HHold: the equality between the free variable sets can be reformulated as an application of $P_{\overline{x}}$)
We extend the domain of the extraction function $\div$ to deal with variables:

$$\widetilde{\phi} \div \overline{x} \overset{\mathsf{def}}{=} \vec{P}_{\overline{x}}(\widetilde{\phi})$$

4.4 Gradual Static Semantics

We choose the deterministic approach proposed in section 3.7.2. Figure 4.13 shows a deterministic lifting of the Hoare logic defined by figure 4.9. It uses a number of lifted components that we defined in section 4.3.

$\boxed{\Gamma \vec{\vdash} \{\widetilde{\phi_{pre}}\}\ s\ \{\widetilde{\phi_{post}}\}}$

$$\frac{}{\Gamma \vec{\vdash} \{\widetilde{\phi}\}\ \texttt{skip}\ \{\widetilde{\phi}\}}\ \vec{\text{H}}\textsc{Skip}$$

$$\frac{\widetilde{\phi} \div x = \widetilde{\phi'} \qquad \Gamma \vdash x : C \qquad \mathsf{fields}_p(C) = \overline{T\ \ f;}}{\Gamma \vec{\vdash} \{\widetilde{\phi}\}\ x\ \texttt{:= new}\ C\ \{\widetilde{\phi'} \mathbin{\tilde{*}} (x\ \texttt{≠ null}) \mathbin{\tilde{*}} \overline{\texttt{acc}(x.f_i) \mathbin{\tilde{*}} (x.f_i\ \texttt{= defaultValue}(T_i))}\ \}}\ \vec{\text{H}}\textsc{Alloc}$$

$$\frac{\widetilde{\phi} \div \texttt{acc}(x.f) = \widetilde{\phi'} \qquad \Gamma \vdash x : C \qquad \Gamma \vdash y : T \qquad \vdash C.f : T}{\Gamma \vec{\vdash} \{\widetilde{\phi}\}\ x.f\ \texttt{:=}\ y\ \{\widetilde{\phi'} \mathbin{\tilde{*}} \texttt{acc}(x.f) \mathbin{\tilde{*}} (x\ \texttt{≠ null}) \mathbin{\tilde{*}} (x.f\ \texttt{=}\ y)\}}\ \vec{\text{H}}\textsc{FieldAssign}$$

$$\frac{\widetilde{\phi'} \vdash \widetilde{\phi} \mathrel{\tilde{\Rrightarrow}} \mathsf{acc}(e) \qquad \widetilde{\phi'} \div x = \widetilde{\phi''} \qquad x \notin \mathsf{FV}(e) \qquad \Gamma \vdash x : T \qquad \Gamma \vdash e : T}{\Gamma \vec{\vdash} \{\widetilde{\phi}\}\ x\ \texttt{:=}\ e\ \{\widetilde{\phi''} \mathbin{\tilde{*}} (x\ \texttt{=}\ e)\}}\ \vec{\text{H}}\textsc{VarAssign}$$

$$\frac{\widetilde{\phi} \div \texttt{result} = \widetilde{\phi'} \qquad \Gamma \vdash x : T \qquad \Gamma \vdash \texttt{result} : T}{\Gamma \vec{\vdash} \{\widetilde{\phi}\}\ \texttt{return}\ x\ \{\widetilde{\phi'} \mathbin{\tilde{*}} (\texttt{result = } x)\}}\ \vec{\text{H}}\textsc{Return}$$

$$\frac{\begin{array}{c}\widetilde{\phi} \div x \div \widetilde{\phi_p} = \widetilde{\phi'} \\ \Gamma \vdash y : C \qquad \mathsf{method}_p(C, m) = T_r\ \ m(T_p\ \ z)\ \texttt{requires}\ \widetilde{\phi_{pre}};\ \texttt{ensures}\ \widetilde{\phi_{post}};\ \{\ \_\ \} \\ \Gamma \vdash x : T_r \qquad \Gamma \vdash z' : T_p \qquad \widetilde{\phi} \mathrel{\tilde{\Rrightarrow}} (y\ \texttt{≠ null}) \mathbin{\tilde{*}} \widetilde{\phi_p} \\ x \neq y \wedge x \neq z' \qquad \widetilde{\phi_p} = \widetilde{\phi_{pre}}[y, z'/\texttt{this}, z] \qquad \widetilde{\phi_q} = \widetilde{\phi_{post}}[y, z', x/\texttt{this}, z, \texttt{result}]\end{array}}{\Gamma \vec{\vdash} \{\widetilde{\phi}\}\ x\ \texttt{:=}\ y.m(z')\ \{\widetilde{\phi'} \mathbin{\tilde{*}} \widetilde{\phi_q}\}}\ \vec{\text{H}}\textsc{Call}$$

$$\frac{\widetilde{\phi'} \vdash \widetilde{\phi} \mathrel{\tilde{\Rrightarrow}} \phi_a}{\Gamma \vec{\vdash} \{\widetilde{\phi}\}\ \texttt{assert}\ \phi_a\ \{\widetilde{\phi'}\}}\ \vec{\text{H}}\textsc{Assert} \qquad \frac{\widetilde{\phi'} \div \phi_r = \widetilde{\phi''}}{\Gamma \vec{\vdash} \{\widetilde{\phi}\}\ \texttt{release}\ \phi_r\ \{\widetilde{\phi''}\}}\ \vec{\text{H}}\textsc{Release}$$

$$\frac{x \notin \mathsf{dom}(\Gamma) \qquad x \notin \mathsf{FV}(\widetilde{\phi}) \qquad \Gamma, x : T \vec{\vdash} \{(x\ \texttt{= defaultValue}(T)) \mathbin{\tilde{*}} \widetilde{\phi}\}\ s\ \{\widetilde{\phi'}\}}{\Gamma \vec{\vdash} \{\widetilde{\phi}\}\ T\ \ x;\ \ s\ \{\widetilde{\phi'}\}}\ \vec{\text{H}}\textsc{Declare}$$

$$\frac{\begin{array}{c}\vdash_{\texttt{sfrm}} \phi \qquad \widetilde{\phi'_f} \vdash \widetilde{\phi_f} \mathrel{\tilde{\Rrightarrow}} \phi \qquad \widetilde{\phi'_f} \div \phi = \widetilde{\phi_r} \\ \widetilde{\phi'_f} \div \mathsf{static}(\widetilde{\phi_r}) \div (\mathsf{FV}(\widetilde{\phi'_f}) \backslash \mathsf{FV}(\phi)) = \widetilde{\phi'} \qquad \mathsf{mod}(s) \cap \mathsf{FV}(\phi) = \emptyset \qquad \Gamma \vec{\vdash} \{\widetilde{\phi_r}\}\ s\ \{\widetilde{\phi'_r}\}\end{array}}{\Gamma \vec{\vdash} \{\widetilde{\phi_f}\}\ \texttt{hold}\ \phi\ \{\ s\ \}\ \{\widetilde{\phi'_r} \mathbin{\tilde{*}} \widetilde{\phi'}\}}\ \vec{\text{H}}\textsc{Hold}$$

$$\frac{\Gamma \vec{\vdash} \{\widetilde{\phi_p}\}\ s_1\ \{\widetilde{\phi_q}\} \qquad \Gamma \vec{\vdash} \{\widetilde{\phi_q}\}\ s_2\ \{\widetilde{\phi_r}\}}{\Gamma \vec{\vdash} \{\widetilde{\phi_p}\}\ s_1;\ \ s_2\ \{\widetilde{\phi_r}\}}\ \vec{\text{H}}\textsc{Seq}$$

Figure 4.13. GVL: Gradual Deterministic Hoare Logic

Lemma 4.12 (**GVL$_{\text{IDF}}$**: Sound Deterministic Lifting of Hoare Logic)**.**
*The inductive definition we give induces a well-defined partial function. This function is a sound deterministic lifting (see 3.7.2) of the Hoare logic of **SVL$_{IDF}$** defined in section 4.1.6.*

A sound predicate lifting $\cdot \mathrel{\widetilde{\vDash}} \{\cdot\} \cdot \{\cdot\}$ can be derived using lemma 3.49.

4.5 Gradual Dynamic Semantics

The call statement is the only statement that is affected by the introduction of gradual formulas. As we do not want the semantics of any of the other statements to change, we only have to adjust the rules for calls. Figure 4.14 shows those rules.

$\boxed{\pi \mathrel{\widetilde{\longrightarrow}} \pi}$

$$\frac{\begin{array}{c} H, \rho \vdash y \Downarrow o \qquad H, \rho \vdash z \Downarrow v \\ H(o) = \langle C, \_ \rangle \qquad \mathsf{method}_p(C, m) = T_r\ m(T\ w)\ \texttt{requires}\ \widetilde{\phi}\texttt{;}\ \texttt{ensures}\ \_\texttt{;}\ \{\ r\ \} \\ \rho' = [\texttt{result} \mapsto \mathsf{defaultValue}(T_r), \texttt{this} \mapsto o, w \mapsto v] \\ H, \rho', A \mathrel{\widetilde{\vDash}} \widetilde{\phi} \qquad A' = \begin{cases} \lfloor \widetilde{\phi} \rfloor_{H,\rho'} & \text{if } \widetilde{\phi} \in \textsc{Formula} \\ A & \text{otherwise} \end{cases} \end{array}}{\langle H, \langle \rho, A, x\ \texttt{:=}\ y\texttt{.}m\texttt{(}z\texttt{);}\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho', A', r \rangle \cdot \langle \rho, A \backslash A', x\ \texttt{:=}\ y\texttt{.}m\texttt{(}z\texttt{);}\ s \rangle \cdot S \rangle}\ \widetilde{\textsc{SsCall}}$$

$$\frac{\begin{array}{c} H, \rho \vdash y \Downarrow o \\ H(o) = \langle C, \_ \rangle \qquad \mathsf{mpost}_p(C, m) = \widetilde{\phi} \qquad H, \rho', A' \mathrel{\widetilde{\vDash}} \widetilde{\phi} \qquad H, \rho' \vdash \texttt{result} \Downarrow v_r \end{array}}{\langle H, \langle \rho', A', \texttt{skip} \rangle \cdot \langle \rho, A, x\ \texttt{:=}\ y\texttt{.}m\texttt{(}z\texttt{);}\ s \rangle \cdot S \rangle \longrightarrow \langle H, \langle \rho[x \mapsto v_r], A \cup A', s \rangle \cdot S \rangle}\ \widetilde{\textsc{SsCallFinish}}$$

Figure 4.14. $\mathbf{GVL_{IDF}}$: Small-Step Semantics for Method Call

For all other statements $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot$ is identical to $\cdot \longrightarrow \cdot$ (see figure 4.10). Furthermore, we design $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot$ to be total. We extend the set of final states $\widetilde{\textsc{ProgramStateFin}}$ with a designated exceptional state π_{EX}, representing dynamic verification failure. If no other derivation exists, we map to this state.

Lemma 4.13 ($\cdot \mathrel{\widetilde{\longrightarrow}} \cdot$ Well-Defined)**.**
The small-step semantics of $\boldsymbol{GVL_{IDF}}$ is well-defined.

Lemma 4.14 ($\mathbf{GVL_{IDF}}$: Sound Lifting of Small-Step-Semantics)**.**
The lifting we propose is sound as defined in section 3.5.3.2.

4.6 Gradual Well-Formedness

As well-formedness was a condition of soundness of $\mathbf{SVL_{IDF}}$, we give an adjusted version that is required for soundness of $\mathbf{GVL_{IDF}}$ (see figure 4.15). Note that the only changes are a more flexible contract for the main method, as well as the gradual method contract. We define $\mathsf{FV}(? * \phi) \stackrel{\mathsf{def}}{=} \mathsf{FV}(\phi)$.

$$\frac{\overline{cls_i\ \widetilde{\mathsf{OK}}} \qquad \widetilde{\vdash}\ \{?\}\ s\ \{?\}}{(\overline{cls_i}\ s)\ \widetilde{\mathsf{OK}}}\ \widetilde{\text{OkProgram}}$$

$$\frac{\text{unique } field\text{-names} \qquad \text{unique } \widetilde{method}\text{-names} \qquad \overline{\widetilde{method}_i\ \widetilde{\mathsf{OK}}\ \mathsf{in}\ C}}{(\texttt{class}\ C\ \texttt{\{}\ \overline{field_i}\ \overline{\widetilde{method}_i}\ \texttt{\}})\ \widetilde{\mathsf{OK}}}\ \widetilde{\text{OkClass}}$$

$$\frac{\begin{array}{c} x : T_x, \texttt{this} : C, \texttt{result} : T_m \ \widetilde{\vdash}\ \{\widetilde{\phi_1}\}\ s\ \{\widetilde{\phi_2}\} \\ \mathsf{FV}(\widetilde{\phi_1}) \subseteq \{x, \texttt{this}\} \qquad \mathsf{FV}(\widetilde{\phi_2}) \subseteq \{x, \texttt{this}, \texttt{result}\} \\ \widetilde{\vdash}_{\texttt{sfrm}}\ \widetilde{\phi_1} \qquad \widetilde{\vdash}_{\texttt{sfrm}}\ \widetilde{\phi_2} \qquad x, \texttt{this} \notin \mathsf{mod}(s) \end{array}}{(T_m\ \ m\texttt{(}T_x\ x\texttt{)}\ \texttt{requires}\ \widetilde{\phi_1}\texttt{;}\ \texttt{ensures}\ \widetilde{\phi_2}\texttt{;}\ \texttt{\{}\ s\ \texttt{\}})\ \widetilde{\mathsf{OK}}\ \mathsf{in}\ C}\ \widetilde{\text{OkMethod}}$$

Figure 4.15. $\mathbf{GVL_{IDF}}$: Well-Formedness

4.7 Gradual Soundness

Proposed gradual Hoare logic is sound w.r.t. the small-step semantics if runtime checks are injected as proposed in section 3.6 (see lemma 3.43 for proof).

However, $\mathbf{GVL_{IDF}}$ also complies with the stronger notion of soundness $\vec{\text{Soundness}}$ introduced in section 3.6. To show this, we follow the approach introduced in section 3.8.2.

Specifically, we show that the semantics are complete and semi-optimal w.r.t. all statements except declarations, sequences, method calls and hold. For those remaining statements we prove by induction that $\vec{\text{Soundness}}$ still holds. The overall proof is a structural induction on statement s.

Lemma 4.15 (Partial Completeness of $\mathbf{SVL_{IDF}}$)**.**
The semantics of $\boldsymbol{SVL_{IDF}}$ is complete (see definition 3.58) w.r.t.
$\texttt{skip}, x\texttt{.}f\ \texttt{:=}\ y, x\ \texttt{:=}\ e, x\ \texttt{:=}\ \texttt{new}\ C, \texttt{return}\ x, \texttt{assert}\ \phi, \texttt{release}\ \phi$

Lemma 4.16 (Partial Semi-Optimality of $\mathbf{GVL_{IDF}}$)**.**
The semantics of $\boldsymbol{GVL_{IDF}}$ is semi-optimal (see definitions 3.60 and 3.61) w.r.t.
$\texttt{skip}, x\texttt{.}f\ \texttt{:=}\ y, x\ \texttt{:=}\ e, x\ \texttt{:=}\ \texttt{new}\ C, \texttt{return}\ x, \texttt{assert}\ \phi, \texttt{release}\ \phi$

Lemma 4.17 ($\vec{\text{Soundness}}$ Induction Step for Declaration)**.**
Assume $\vec{\text{Soundness}}$ *holds for* $s \in \widetilde{\text{Stmt}}$.
Then $\vec{\text{Soundness}}$ *holds for* $T\ \ x\texttt{;}\ \ s$ *(for all* $T \in \text{Type}$, $x \in \text{Var}$*).*

Lemma 4.18 ($\vec{\text{Soundness}}$ Induction Step for Calls)**.**
Assume $\vec{\text{Soundness}}$ *holds for the method body of method* $y\texttt{.}m$.
Then $\vec{\text{Soundness}}$ *holds for* $x\ \texttt{:=}\ y\texttt{.}m\texttt{(}z\texttt{)}$ *(for all* $m \in \text{MethodName}$, $x, y, z \in \text{Var}$*).*

Lemma 4.19 ($\vec{\text{S}}$OUNDNESS Induction Step for Hold)**.**
Assume $\vec{\text{S}}$OUNDNESS *holds for* $s \in \widetilde{\text{S}}$TMT*.*
Then $\vec{\text{S}}$OUNDNESS *holds for* `hold` ϕ `{` s `}` *(for all* $\phi \in$ FORMULA*).*

Lemma 4.20 ($\vec{\text{S}}$OUNDNESS Induction Step for Sequences)**.**
Assume $\vec{\text{S}}$OUNDNESS *holds for* $s_1, s_2 \in \widetilde{\text{S}}$TMT*.*
Then $\vec{\text{S}}$OUNDNESS *holds for* s_1`;` s_2*.*

Theorem 4.21 ($\mathbf{GVL_{IDF}}$ satisfies $\vec{\text{S}}$OUNDNESS)**.**
The semantics of $\boldsymbol{GVL_{IDF}}$ *satisfy* $\vec{\text{S}}$OUNDNESS*.*

5 Evaluation

5.1 Runtime Overhead

It is expected that gradual verification can induce runtime overhead. Specifically, in the presence of unknown formulas runtime checks are necessary to guarantee soundness of the gradual system.

With the methodology we introduced in chapter 3 there are two potential causes for runtime checks.

1. In general, the gradual Hoare judgment requires injection of runtime checks to ensure the postcondition dynamically (see section 3.6). We have also discussed how those runtime checks can be reduced or even eliminated if $\vec{\text{Soundness}}$ holds (see lemma 3.53).

2. This cause is more subtle and originates in restricting the domain of the small-step semantics to achieve completeness as described in section 3.8.1. Restricting the domain usually means adding additional checks that determine whether the function is defined or not. Example 3.52 illustrates this effect using the example of an assertion statement. Note that a compiler may also simplify or omit these static checks using the knowledge provided by the deterministic lifting (given that $\vec{\text{Soundness}}$ holds).

The small-step semantics of $\mathbf{SVL_{IDF}}$ we gave in chapter 4 was designed to be (partially) complete in the first place (see lemma 4.15). Note however, that a lot of checks they perform could have been omitted for $\mathbf{SVL_{IDF}}$ (e.g. assertions could have been implemented as no-operations as motivated in example 3.52), yet they are necessary for $\vec{\text{Soundness}}$ of $\mathbf{GVL_{IDF}}$.

It is a desirable property of gradual systems to not impose any runtime overhead (compared to the static system) should all annotations be static. We give sufficient criteria for a gradual verification system with this property.

Lemma 5.1 (Overhead-Free Gradual Verification System)**.**
For fully statically annotated programs, a gradual verification system is overhead-free if $\vec{\text{Soundness}}$ *and*

$$\forall \phi_1 \in \text{Formula}, s \in \text{Stmt}, \widetilde{\phi_2} \in \widetilde{\text{Formula}}.\ \vec{\vdash} \{\phi_1\}\ s\ \{\widetilde{\phi_2}\} \implies \widetilde{\phi_2} \in \text{Formula} \tag{5.1}$$

holds. Intuitively, this condition states that the deterministic gradual Hoare logic is precise for all static input.

Note: $\mathbf{GVL_{IDF}}$ satisfies the criteria given by lemma 5.1 and can thus be implemented overhead-free for statically annotated programs. The implementation we mention in section 5.2 is overhead-free for statically annotated programs.

5.2 Implementation

In chapter 4 we instantiated the approach proposed in chapter 3, deriving a gradually verified language $\mathbf{GVL_{IDF}}$. We mainly modeled $\mathbf{GVL_{IDF}}$ abstractly, assuming that there exist implementations for the predicates and functions used in the process. In section A.2 we give hints towards an implementation of some of those theoretical constructs. We refrained from giving more detailed implementation instructions (with proofs that they behave as desired) as they would be very voluminous and have to deal with specifics completely unrelated to gradual typing per se.

However, we have implemented $\mathbf{GVL_{IDF}}$ as an interactive web page using TypeScript. The implementation can be found at `http://olydis.github.io/GradVer/impl/HTML5/` (part of [2]).

5.3 Enhancing a Dynamically Verified Language

The approach we presented constructs a gradually verified language in terms of a statically verified one. However, as the example in section 1.1.1 motivates, gradual verification can also be seen as an extension of the dynamically verified setting. The main drawbacks of dynamic verification (runtime overhead and potentially late error detection) would be counteracted by using static verification techniques where possible. A gradual verifier will attempt to prove compliance with annotations (making runtime checks unnecessary) or may detect an inevitable violation of an annotation (detecting an error before the program is executed). In this section, we will briefly describe how to turn a dynamically verified language into a gradually verified one.

To understand how to approach dynamically verified languages, we have to examine how they fit into the spectrum of a gradually verified one. The static end is (by construction) the one where all formulas are static and the unknown formula "`?`" is never used. In contrast, annotating `?` everywhere corresponds to a system that solely relies on runtime checks. One can further define that leaving out annotations (e.g. invariants, preconditions, postconditions, entire methods contracts) corresponds to an annotation of `?`. A dynamically verified language (say, Java with its mechanisms like runtime assertions) can be thought of as a gradually verified language with all annotations left out, defaulting to `?`. Such a gradually verified language can be derived as follows:

1. Identify what the goal of a static verification system would be for the language at hand. Common examples include race-condition avoidance or a no-throw guarantee.

2. Develop a sound Hoare logic that achieves that goal, including syntax extensions

to the language that allow leveraging that Hoare logic. The resulting system corresponds to a statically verified language.

3. Gradualization can be applied to that system, introducing `?` as a "default formula", which allows making all previously introduced syntax extensions optional again. The syntax of the resulting gradually verified language is therefore a superset of the original, dynamically verified one.

Contracts that were before explicitly realized as runtime checks can now be removed and encoded using the optional annotation syntax.

6 Conclusion

Gradual verification aims to overcome drawbacks of static or dynamic verification by seamlessly combining both approaches. As a result, programmers can freely choose the amount of static annotations to provide and thus have control over the degree of static verification used.

In this work we have developed a procedure of deriving an imperative gradually verified programming language from a statically verified one. We have done so by adapting the work on "Abstracting Gradual Typing" (AGT, [9]) by Garcia, Clark and Tanter to the setting of program verification. Therefore, we used the concepts of abstract interpretation to give a meaning to gradual formulas. Furthermore we generalized their notion of gradual lifting and applied it to the semantics of the statically verified language. We also developed the notion of deterministic lifting in order to overcome a number of practical issues related to predicate lifting and transitivity. Note that we did not make use of their way of providing gradual dynamic semantics using "evidence", see section 6.2 for a discussion.

Finally, we illustrated our approach by gradualizing an imperative statically verified language using implicit dynamic frames to enable safe reasoning about shared mutable data structures.

6.1 Limitations and Future Work

6.1.1 Optimality Revised

After realizing that optimality of gradual liftings (or "consistent lifting", as called in AGT [9]) is an unnecessary restriction in the verification setting, we focused mainly on soundness of liftings. Furthermore, we showed that our approach of obtaining a gradual Hoare logic using deterministic liftings does not result in an optimally lifted Hoare logic, even if the deterministic lifting was optimal (see example 3.50). We have also shown that a more expressive gradual syntax can reduce that problem, yet we have not formally proved that relationship.

Furthermore, it is unclear whether optimality/consistency as defined by AGT is even desirable under all circumstances:

> **Example 6.1** (Optimality and Decidability)**.** Consider a statement s which is very hard to reason about (for instance, think of 300 Collatz sequence iterations). With

the static Hoare logic, we assume that

$$\vdash \{\texttt{(x = 10000)}\}\ s\ \{\texttt{1 <= x} \land \texttt{x <= 4}\}$$

can be verified and that no stronger postcondition could be verified (due to limitations of the verifier). However, we know that `x` will have value 4 afterwards, i.e. the following Hoare triple is valid:

$$\{\texttt{(x = 10000)}\}\ s\ \{\texttt{(x = 4)}\}$$

Using gradual verification, we want to overcome the limitations imposed by decidability and be able to deduce

$$\widetilde{\vdash} \{\texttt{(x = 10000)} \land \texttt{?}\}\ s\ \{\texttt{(x = 4)} \land \texttt{?}\}$$

or similar. However, an optimal gradual lifting is not able to deduce this fact, since there exists no instantiation of the postcondition for which a corresponding static judgment holds (since $\texttt{1 <= x} \land \texttt{x <= 4} \notin \gamma(\texttt{(x = 4)} \land \texttt{?})$).

Note that our approach using a deterministic gradual Hoare logic would work in this case:

$$\vec{\vdash} \{\texttt{(x = 10000)} \land \texttt{?}\}\ s\ \{\texttt{1 <= x} \land \texttt{x <= 4} \land \texttt{?}\}$$

or a more imprecise postcondition must be deducible due to introduction, strength and monotonicity rules. Furthermore $\texttt{1 <= x} \land \texttt{x <= 4} \land \texttt{?} \mathrel{\widetilde{\Rightarrow}} \texttt{(x = 4)}$ holds (emitting a runtime check), such that

$$\widetilde{\vdash} \{\texttt{(x = 10000)} \land \texttt{?}\}\ s\ \{\texttt{(x = 4)}\}$$

is deducible.

We conjecture that a notion of consistency that relies on semantically valid Hoare triples instead of statically deducible ones (Hoare logic) would solve this problem, yet it is unclear what the implications of this definition would be. Most importantly, it would severely complicate optimality proofs, as they suddenly rely on dynamic instead of static semantics.

6.1.2 Gradual Non-Termination

The "plausibility" interpretation of unknown formulas motivated us to define $\gamma(\texttt{?})$ as the set of all *satisfiable* formulas SatFormula (see section 3.2). However, this definition implies that `?` may not be usable as a postcondition of non-terminating statements, for which proving `false` as a postcondition would make sense. As a consequence, programmers may not be able to be imprecise about termination of a statement. Note that $\mathbf{SVL_{IDF}}$ does not contain non-terminating constructs, so we did not deal with this problem in chapter 4.

Maybe the introduction of an additional wildcard $?_\bot$ with $\gamma(?_\bot) = \textsc{Formula}$ proves useful in order to be imprecise about termination.

6.1.3 Leveraging Implicit Dynamic Frames

In the case study, we have not made full use of the capabilities of implicit dynamic frames, yet. Tracking exclusive access to memory locations also allows race-free reasoning about concurrent programs (Summers and Drossopoulou [32] give a corresponding Hoare logic). It would be interesting to see gradual verification applied to such a setting, as it reflects more closely the reality of modern programming languages. Further potential extensions include the introduction of shared access or the addition of non-separating conjunction to the formula syntax.

6.1.4 Non-Deterministic Semantics

In section 3.1 we assumed that the dynamic semantics of **SVL** is deterministic. It may be worth investigating the implications of non-deterministic semantics on gradual lifting, soundness, and gradualization in general.

6.1.5 Evidence-Based Gradual Dynamic Semantics

Regarding gradual dynamic semantics (and soundness) of the gradual system, we have deviated from the approach presented by AGT.

AGT derives a direct runtime semantics for the gradual system that uses "evidence" to ensure that products of a runtime derivation (evaluating an application $t_1\ t_2$) are still type safe. Key to this approach is the observation that type safety proofs smoothly evolve as terms are reduced. More specifically, a type safety proof for the reduced term can be computed from a type safety proof of the unreduced term (function application, according to rule "Tapp").

In the static system, this process would always succeed (and is thus not necessary to ensure type safety). However, in the gradual system, deriving a proof for a reduced term may fail due to inconsistent judgments in the proofs of the unreduced term. For example, the unreduced term may judge $T_1 \mathrel{\widetilde{<}} ?$ and $? \mathrel{\widetilde{<}} T_2$, whereas the reduced term requires $T_1 \mathrel{\widetilde{<}} T_2$ to hold (a property called "consistent transitivity"). Evidence is the minimum information necessary to detect those inconsistencies. It is thus created while initially type checking a term ("initial evidence", side product of gradual typing rule "$\widetilde{\text{T}}$app") and evolved during reduction as part of the runtime semantics.

In our setting, there are two candidate judgments that one could try to enforce using evidence (like type safety is enforced in AGT).

Semantical

When searching for a direct counterpart for type judgments $\vdash e : T$ in simply typed λ-calculus, one may come up with the formula semantics $\pi \vDash \phi$. It can be seen as a way to "type" program states using formulas ("$\vdash \pi : \phi$"). Just like type safety states that statically postulated type judgments hold during reduction of terms and expressions, it has to be ensured that annotated formulas are complied with during evolution of program states (which is what soundness states).

However the formula semantics $\pi \vDash \phi$ is a semantical judgment, whereas $\vdash e : T$ is syntactical. We will illustrate in section 6.2 why the evidence approach is not feasible when choosing formula semantics as the judgment to ensure.

Syntactical

Instead, one may find that Hoare logic $\vdash \{\phi_1\}\ s\ \{\phi_2\}$ acts like the conceptual counterpart of $\vdash e : T$. While the signature slightly differs, Hoare logic can be regarded as a way to "type" statements as a function from program states satisfying ϕ_1 to program states satisfying ϕ_2 ("$\vdash s : \phi_1 \to \phi_2$").

During execution of the program, evidence would have to make sure that the "remaining work" s is consistently typeable using Hoare logic. In other words, there has to be a Hoare logic deduction justifying every intermediate state during execution.

Example 6.2.
Let

$$\phi_1 = \texttt{(x = 10)} \wedge \texttt{(a = 5)}$$
$$\phi_3 = \texttt{(x = 3)} \wedge \texttt{(y = 4)}$$

Assume that

$$\vdash \{\phi_1\}\ \texttt{x := 3; y := 4}\ \{\phi_3\}$$

was proved by the verifier. Starting execution of `x := 3; y := 4` with a program state π_1 that satisfies ϕ_1 one will reach a program state π_2 where `x := 3` has been executed but not yet `y := 4`. Type safety as motivated above would have to ensure that there exists a formula ϕ_2 with

$$\pi_2 \vDash \phi_2$$

and

$$\vdash \{\phi_2\}\ \texttt{y := 4}\ \{\phi_3\}$$

In this example $\phi_2 = \texttt{(x = 3)} \wedge \texttt{(a = 5)}$ would be suitable.

While such a formula always exists in the static setting (choose the one that was used as an intermediate formula when applying the sequence rule HSeq), it may not exist in the gradual setting. It would be the job of evidence to ensure the existence of such a formula at runtime.

This approach requires a slightly stronger notion of soundness than the one we demanded in section 3.1. Specifically, a statement has to be made about intermediate

execution steps (as illustrated in example 6.2) instead of merely the execution as a whole.

As mentioned in section 3.6 such an approach would detect inconsistencies more eagerly than the "safe baseline" we introduced with $\widetilde{\text{Soundness}}$ and later improved with $\vec{\text{Soundness}}$. Unfortunately, investigating this path was not in the scope of this work.

6.2 Evidence for Semantical Judgment Considered Bad

In section 6.1.5 we claimed that the evidence approach is not feasible to ensure that the semantical judgment $\pi \vDash \phi$ holds for given formula ϕ and program state π after executing some statement. This would be an alternative way to ensure soundness of the gradual system, in contrast to *evaluating* $\widetilde{\pi} \mathrel{\widetilde{\vDash}} \widetilde{\phi}$ as suggested by the baseline approach we present as $\widetilde{\text{Soundness}}$ in section 3.6.

Given an execution $\widetilde{\pi_1} \xrightarrow{\widetilde{s}} \widetilde{\pi_2}$ with $\widetilde{\pi_1} \mathrel{\widetilde{\vDash}} \widetilde{\phi_1}$, it would have to be the job of evidence to ensure that $\widetilde{\pi_2} \mathrel{\widetilde{\vDash}} \widetilde{\phi_2}$. One can imagine how complex evidence would have to be, as it effectively tracks all changes to $\widetilde{\pi_1}$ (on its way to become $\widetilde{\pi_2}$) in order to figure out whether these changes are consistent with the postulated "type" $\widetilde{\phi_2}$.

Ultimately, both the explicit check and the evidence approach are equivalent, yet one is more natural than the other depending on the setting. The following example tries to illustrate how soundness of a gradual assertion statement would be deducible using evidence.

> **Example 6.3** (Sound Gradual Assertion using Evidence)**.**
> Given a judgment $\widetilde{\vdash} \{?\}$ `assert (x = 3)` $\{$`(x = 3)`$\}$ and an execution $\widetilde{\pi_1} \xrightarrow{\texttt{assert (x = 3)}} \widetilde{\pi_2}$ with $\widetilde{\pi_1} \mathrel{\widetilde{\vDash}}$ `?`, we have to *derive* $\widetilde{\pi_2} \mathrel{\widetilde{\vDash}}$ `(x = 3)`.
>
> From runtime derivations lying in the past we are given a proof why $\widetilde{\pi_1} \mathrel{\widetilde{\vDash}}$ `?` holds. Ultimately, this proof may involve information about the value of x. For instance, there might have been an assignment to x in the past, i.e. $\widetilde{\pi_1}$ has evolved from a program state that had type `(x = ...)`. Now, this proof can be combined with the proof for $\widetilde{\vdash} \{?\}$ `assert (x = 3)` $\{$`(x = 3)`$\}$, say `?` $\widetilde{\Rightarrow}$ `(x = 3)`. In case the combined proof results in a transitive judgment $\phi \mathrel{\widetilde{\Rightarrow}}$ `?` $\mathrel{\widetilde{\Rightarrow}}$ `(x = 3)`, this judgment can be checked for consistency (i.e. whether $\phi \mathrel{\widetilde{\Rightarrow}}$ `(x = 3)` holds).
>
> Should combining both proofs succeed, we ultimately end up with a proof for $\widetilde{\pi_1} \mathrel{\widetilde{\vDash}}$ `(x = 3)` which is also a proof for $\widetilde{\pi_2} \mathrel{\widetilde{\vDash}}$ `(x = 3)` (due to the small-step semantics of assertions).
>
> Evaluating $\widetilde{\pi_2} \mathrel{\widetilde{\vDash}}$ `(x = 3)` is arguably less complex.

On the other hand, our checking approach adapted to AGT would correspond to literally

checking whether a reduced term has its postulated type – however, this is only possible with additional runtime support (track runtime type information, tagging) and is therefore not as straight-forward to implement as the evidence approach. Note that strong dynamic type systems provide this kind of runtime support.

The fundamental conceptual difference between type judgments (setting of AGT) and formula semantics (setting of this work) is that the former is syntax-driven, whereas the latter is defined semantically. Note also that the signature of Tapp corresponds precisely to the signature of Soundness. Both make a statement about the type of a term/program state *after* applying a function/executing a statement. However, in one system, the rule is given while in the other system the rule is derivable from the given static semantics (if they are sound). On the other hand, the rule HSeq is given in our work but would correspond to a function composition rule that is derivable in AGT using Tapp and Tλ. Note the duality of which rules are given and which are derivable in each setting.

A Appendix

A.1 Partial Galois Connection

In lemma 3.35 we mentioned partial Galois connections. In this section, we give a formal definition and motivate why we used them instead of regular Galois connections.

The following definition is drawn from Miné [22], but adapted to our concrete setting.

Definition A.1 (Partial Galois connection)**.**
Let $(\mathcal{P}^{\text{Formula}}, \subseteq)$ *and* $(\widetilde{\text{Formula}}, \sqsubseteq)$ *be two posets,* $\mathcal{F}$ *a set of operators on* $\mathcal{P}^{\text{Formula}}$*,* $\alpha : \mathcal{P}^{\text{Formula}} \rightharpoonup \widetilde{\text{Formula}}$ *a partial function and* $\gamma : \widetilde{\text{Formula}} \to \mathcal{P}^{\text{Formula}}$ *a total function.*

The pair $\langle \alpha, \gamma \rangle$ *is an* $\mathcal{F}$*-partial Galois connection if and only if:*

1. *If* $\alpha(\overline{\phi})$ *is defined, then* $\overline{\phi} \subseteq \gamma(\alpha(\overline{\phi}))$
2. *If* $\alpha(\overline{\phi})$ *is defined, then* $\overline{\phi} \subseteq \gamma(\widetilde{\phi})$ *implies* $\alpha(\overline{\phi}) \sqsubseteq \widetilde{\phi}$
3. *For all* $F \in \mathcal{F}$ *and* $\widetilde{\phi} \in \widetilde{\text{Formula}}$*,* $\alpha(F(\gamma(\widetilde{\phi})))$ *is defined*

This definition can be generalized for a set $\mathcal{F}$ *of arbitrary* n*-ary operators.*

In our work we use partial Galois connections because a traditional Galois connection between $\widetilde{\text{Formula}}$ and $\mathcal{P}^{\text{Formula}}$ may not always exist. Specifically, it does not exist in case of $\mathbf{GVL_{IDF}}$, as introduced in chapter 4.

Lemma A.2 (Non-Existence of Galois Connection in $\mathbf{GVL_{IDF}}$)**.**
Assume that Formula*,* $\widetilde{\text{Formula}}$ *and* γ *are defined as in* $\boldsymbol{SVL_{IDF}/GVL_{IDF}}$ *(see chapter 4). Then there exists no* α *such that* $\langle \alpha, \gamma \rangle$ *is a Galois connection.*

Partial Galois connections are more flexible by also giving a (sufficiently strong) meaning to partial abstraction functions α.

A.2 Implementation Strategies

In this section we give a number of implementation strategies, that can be used to implement the functions and predicates defined in section 4.3.

Gradual Formula Semantics

Lemma A.3 (Optimal Lifting of Formula Evaluation)**.**
Let $\cdot \mathrel{\widetilde{\vDash}} \cdot \subseteq \textsc{ProgramState} \times \widetilde{\textsc{Formula}}$ *be defined inductively as*

$$\frac{\pi \vDash \phi}{\pi \mathrel{\widetilde{\vDash}} \phi}\ \widetilde{\textsc{EvalStatic}}$$

$$\frac{\pi \vDash \phi \qquad A_s \vdash_{\mathit{sfrm}} \phi \qquad \forall \langle e, f\rangle \in A_s.\ \pi \vDash \mathtt{acc}(e.f)}{\pi \mathrel{\widetilde{\vDash}} \mathtt{?} * \phi}\ \widetilde{\textsc{EvalGrad}}$$

Then $\cdot \mathrel{\widetilde{\vDash}} \cdot$ *is an optimal lifting of* $\cdot \vDash \cdot$*.*

Note the additional effort necessary in rule $\widetilde{\textsc{EvalGrad}}$ compared to the non-IDF version in lemma 3.29. The additional premises make sure that there exists a way to frame ϕ with the access provided by π.

Gradual Self-Framing

Lemma A.4 (Optimal Lifting of Self-Framing)**.**
Let $\mathrel{\widetilde{\vdash}}_{\mathit{sfrm}} \cdot \subseteq \widetilde{\textsc{Formula}}$ *be defined inductively as*

$$\frac{\vdash_{\mathit{sfrm}} \phi}{\mathrel{\widetilde{\vdash}}_{\mathit{sfrm}} \phi}\ \widetilde{\textsc{SfrmStatic}}$$

$$\frac{\pi \vDash \phi}{\mathrel{\widetilde{\vdash}}_{\mathit{sfrm}} \mathtt{?} * \phi}\ \widetilde{\textsc{SfrmGrad}}$$

Then $\mathrel{\widetilde{\vdash}}_{\mathit{sfrm}} \cdot$ *is an optimal lifting of* $\vdash_{\mathit{sfrm}} \cdot$*.*

The premise of $\widetilde{\textsc{SfrmGrad}}$ makes sure that the concretization of $\mathtt{?} * \phi$ is not empty.

Gradual Implication of Static Formula

Lemma A.5.
Let $\vec{P}_{\phi_t} : \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ *be defined as*

$$\begin{aligned} \vec{P}_{\phi_t}(\phi) &= \phi \quad \textit{if } \phi \Rightarrow \phi_t \\ \vec{P}_{\phi_t}(\mathtt{?} * \phi) &= \mathtt{?} * |\phi| * |\phi_t| \end{aligned}$$

Then $\vec{P}_{\phi_t}$ *is an optimal deterministic lifting of* $P_{\phi_t} \subseteq \textsc{Formula} \times \textsc{Formula}$ *with*

$$P_{\phi_t}(\phi_1, \phi_2) \stackrel{\mathsf{def}}{=} \phi_1 \Rightarrow \phi_t \wedge \phi_1 = \phi_2$$

The definition makes use of $|\phi|$ which is a normal form of ϕ defined in section A.2.1.

Gradual Formula Extraction

Lemma A.6.
There exists a function $\vec{P} : \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ *such that* $\vec{P}$ *is an optimal deterministic lifting of* $P \subseteq \textsc{Formula} \times \textsc{Formula} \times \textsc{Formula}$ *with*

$$P(\phi_1, \phi_t, \phi_2) \stackrel{\mathsf{def}}{=} \phi_1 \Rightarrow \phi_t * \phi_2 \ \wedge \vdash_{\mathit{sfrm}} \phi_2$$

Example A.7 (Behavior of $\vec{P}$)**.**

$$\begin{aligned}
&\vec{P}(\widetilde{\phi}, \phi) && = \widetilde{\phi} \quad \text{if } \lfloor \phi \rfloor = \emptyset \\
&\vec{P}(\texttt{acc(x.f) * (x.f = 3)}, \texttt{acc(x.f)}) && = \texttt{true} \\
&\vec{P}(\texttt{? * (a = x.f) * (x.f = b)}, \texttt{acc(x.f)}) && = \texttt{(a = b)} \\
&\vec{P}(\texttt{acc(y.f)}, \texttt{acc(x.f)}) && \textit{undefined (extracted formula not implied)}
\end{aligned}$$

Gradual Variable Extraction

Lemma A.8.
There exists a function $\vec{P}_{\overline{x}} \subseteq \widetilde{\text{Formula}} \rightharpoonup \widetilde{\text{Formula}}$ *such that* $\vec{P}$ *is an optimal deterministic lifting of* $P_{\overline{x}} \subseteq \text{Formula} \times \text{Formula}$ *with*

$$P_{\overline{x}}(\phi_1, \phi_2) \stackrel{\text{def}}{=} \phi_1 \Rightarrow \phi_2 \quad \wedge \quad \mathsf{FV}(\phi_2) \cap \overline{x} = \emptyset \quad \wedge \quad \vdash_{\mathit{sfrm}} \phi_2$$

Example A.9 (Behavior of $\vec{P}_{\overline{x}}$)**.**

$$\begin{aligned}
&\vec{P}_{\emptyset}(\widetilde{\phi}) = \widetilde{\phi} \\
&\vec{P}_{X}(\texttt{? * } \phi) = \texttt{? * } \vec{P}_{X}(\phi) \\
&\vec{P}_{\{x\}}(\texttt{acc(x.f) * (x.f = 3)}) = \texttt{true} \\
&\vec{P}_{\{x\}}(\texttt{(a = x) * (x = b)}) = \texttt{(a = b)}
\end{aligned}$$

A.2.1 Access-Free Gradual Normal Form

In this section we will introduce a normal form for partially unknown formulas that proves useful for implementing a gradual verifier for $\mathbf{GVL_{IDF}}$. One example is the function postulated in lemma A.5, another example is gradual formula precision $\cdot \sqsubseteq \cdot$: Originally it is defined as comparing (possibly infinite) concretizations for inclusion (see definition 3.17), using the following theorem the problem is reducible to checking an implication.

Theorem A.10 (Gradual Normal Form)**.**
There exists a computable function $|\cdot| : \text{Formula} \to \text{Formula}$*, such that*

1. $||\texttt{? * } \phi|| \stackrel{\text{def}}{=} \texttt{? * } |\phi|$ *is equivalent to* $\texttt{? * } \phi$ *(for all* $\phi \in \text{Formula}$*)*
2. $\phi \implies |\phi|$
3. $|\phi|$ *contains no accessibility-predicates*
4. $\texttt{? * } \phi_1 \sqsubseteq \texttt{? * } \phi_2 \iff |\phi_1| \implies |\phi_2|$

A.3 Proofs

This section contains proofs for most lemmas and theorems mentioned in this work. Further automated proofs can be found in [2].

Proof of Lemma 3.8.
Follows from definitions 3.6, 3.7 and the fact that $\subseteq$ induces a partial order. □

Proof of Lemma 3.10.
We are given

$$\pi_1 \vDash \phi_1$$

and

$$\pi_1 \overset{s_1\,;\; s_2}{\longrightarrow} \pi_3$$

for some $\pi_1, \pi_3 \in \textsc{ProgramState}$.

According to assumption 3.1 (see section 3.1) this implies

$$\pi_1 \xrightarrow{s_1} \pi_2 \wedge \pi_2 \xrightarrow{s_2} \pi_3$$

for some $\pi_2 \in \textsc{ProgramState}$. Using the assumptions

$$\vDash \{\phi_1\}\ s_1\ \{\phi_2\}\ \wedge\ \ \vDash \{\phi_2\}\ s_2\ \{\phi_3\}$$

we can deduce $\pi_3 \vDash \phi_3$. □

Proof of Lemma 3.28.
Goal:

$$\widetilde{\phi_1} \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi_2} \iff \exists \phi_1 \in \gamma(\widetilde{\phi_1}), \phi_2 \in \gamma(\widetilde{\phi_2}).\ \phi_1 \Rightarrow \phi_2$$

Case $\Longrightarrow$

Case $\widetilde{\text{ImplStatic}}$

$$\begin{aligned} & \phi_1 \mathrel{\widetilde{\Rightarrow}} \phi_2 \\ \Longrightarrow\ & \phi_1 \Rightarrow \phi_2 \\ \Longrightarrow\ & (\exists \phi_1' \in \gamma(\phi_1), \phi_2' \in \gamma(\phi_2).\ \phi_1' \Rightarrow \phi_2') \end{aligned}$$

Case $\widetilde{\text{ImplGrad1}}$

$$\begin{aligned} & ? \mathrel{\widetilde{\Rightarrow}} \phi \\ \Longrightarrow\ & \phi \in \textsc{SatFormula} \\ \Longrightarrow\ & (\exists \phi_1 \in \textsc{SatFormula}.\ \phi_1 \Rightarrow \phi) \\ \Longrightarrow\ & (\exists \phi_1 \in \gamma(?), \phi_2 \in \gamma(\phi).\ \phi_1 \Rightarrow \phi_2) \end{aligned}$$

Case $\widetilde{\text{ImplGrad2}}$

$$\begin{aligned} & \gamma(\widetilde{\phi}) \neq \emptyset \ \wedge \ \texttt{true} \in \gamma(?) \\ \Longrightarrow\ & (\exists \phi_1 \in \gamma(\widetilde{\phi}).\ \phi_1 \Rightarrow \texttt{true}) \ \wedge \ \texttt{true} \in \gamma(?) \\ \Longrightarrow\ & (\exists \phi_1 \in \gamma(\widetilde{\phi}), \phi_2 \in \gamma(?).\ \phi_1 \Rightarrow \phi_2) \end{aligned}$$

Case $\Longleftarrow$

Given: $\phi_1 \in \gamma(\widetilde{\phi_1}) \wedge \phi_2 \in \gamma(\widetilde{\phi_2}) \wedge \phi_1 \Rightarrow \phi_2$

Case $\widetilde{\phi_2} = ?$

Apply $\widetilde{\text{ImplGrad2}}$.

Case $\widetilde{\phi_2} = \phi_2 \wedge \widetilde{\phi_1} = ?$

$$\begin{aligned} & \phi_1 \in \gamma(?) \wedge \phi_1 \Rightarrow \phi_2 \\ \Longrightarrow\ & \phi_1 \in \text{SatFormula} \wedge \phi_1 \Rightarrow \phi_2 \\ \Longrightarrow\ & \phi_2 \in \text{SatFormula} \end{aligned}$$

Apply $\widetilde{\text{ImplGrad1}}$.

Case $\widetilde{\phi_2} = \phi_2 \wedge \widetilde{\phi_1} = \phi_1$

Apply $\widetilde{\text{ImplStatic}}$.

□

Proof of Lemma 3.29.
Goal:

$$\pi \mathrel{\widetilde{\models}} \widetilde{\phi} \iff \exists \phi \in \gamma(\widetilde{\phi}).\ \pi \models \phi$$

Case $\Longrightarrow$

Case $\widetilde{\text{EvalStatic}}$

$$\begin{aligned} & \pi \mathrel{\widetilde{\models}} \phi \\ \Longrightarrow\ & \pi \models \phi \\ \Longrightarrow\ & (\exists \phi' \in \gamma(\phi).\ \pi \models \phi') \end{aligned}$$

Case $\widetilde{\text{EvalGrad}}$

$$\begin{aligned} & \pi \mathrel{\widetilde{\models}} \phi \ \wedge \ ? \\ \Longrightarrow\ & \pi \models \phi \\ \Longrightarrow\ & (\exists \phi' \in \gamma(\phi \ \wedge \ ?).\ \pi \models \phi') \end{aligned}$$

Case $\Longleftarrow$

Given: $\phi' \in \gamma(\widetilde{\phi}) \wedge \pi \models \phi'$

Case $\widetilde{\phi} = \phi \ \wedge \ ?$
It follows from definition 3.16 that $\phi' \Rightarrow \phi$ and thus $\pi \vDash \phi$. Apply $\widetilde{\text{EvalGrad}}$.

Case $\widetilde{\phi} = \phi$
It follows that $\phi = \phi'$. Apply $\widetilde{\text{EvalStatic}}$.

□

Proof of Lemma 3.30.

Introduction

$$
\begin{array}{ll}
 & (P \circ Q)(\phi_1, \phi_3) \\
\overset{Definition}{\Longrightarrow} & (\exists \phi_2 \in \text{Formula}.\ P(\phi_1, \phi_2) \wedge Q(\phi_2, \phi_3)) \\
\overset{Introduction}{\Longrightarrow} & (\exists \phi_2 \in \text{Formula}.\ \widetilde{P}(\phi_1, \phi_2) \wedge \widetilde{Q}(\phi_2, \phi_3)) \\
\overset{Definition}{\Longrightarrow} & (\widetilde{P} \circ \widetilde{Q})(\phi_1, \phi_3) \\
\overset{Definition}{\Longrightarrow} & \widetilde{(P \circ Q)}(\phi_1, \phi_3)
\end{array}
$$

Monotonicity

$$
\begin{array}{ll}
 & \widetilde{(P \circ Q)}(\widetilde{\phi_1}, \widetilde{\phi_3}) \ \wedge \ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_1'} \ \wedge \ \widetilde{\phi_3} \sqsubseteq \widetilde{\phi_3'} \\
\overset{Definition}{\Longrightarrow} & (\widetilde{P} \circ \widetilde{Q})(\widetilde{\phi_1}, \widetilde{\phi_3}) \ \wedge \ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_1'} \ \wedge \ \widetilde{\phi_3} \sqsubseteq \widetilde{\phi_3'} \\
\overset{Definition}{\Longrightarrow} & (\exists \widetilde{\phi_2} \in \widetilde{\text{Formula}}.\ \widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2}) \wedge \widetilde{Q}(\widetilde{\phi_2}, \widetilde{\phi_3})) \ \wedge \ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_1'} \ \wedge \ \widetilde{\phi_3} \sqsubseteq \widetilde{\phi_3'} \\
\overset{Monotonicity}{\Longrightarrow} & (\exists \widetilde{\phi_2} \in \widetilde{\text{Formula}}.\ \widetilde{P}(\widetilde{\phi_1'}, \widetilde{\phi_2}) \wedge \widetilde{Q}(\widetilde{\phi_2}, \widetilde{\phi_3'})) \\
\overset{Definition}{\Longrightarrow} & (\widetilde{P} \circ \widetilde{Q})(\widetilde{\phi_1'}, \widetilde{\phi_3'}) \\
\overset{Definition}{\Longrightarrow} & \widetilde{(P \circ Q)}(\widetilde{\phi_1'}, \widetilde{\phi_3'})
\end{array}
$$

□

Proof of Lemma 3.31. **Introduction**

$$
\begin{array}{ll}
 & (P \wedge Q)(\phi) \\
\overset{Definition}{\Longrightarrow} & P(\phi) \ \wedge \ Q(\phi) \\
\overset{Introduction}{\Longrightarrow} & \widetilde{P}(\phi) \ \wedge \ \widetilde{Q}(\phi) \\
\overset{Definition}{\Longrightarrow} & (\widetilde{P} \ \wedge \ \widetilde{Q})(\phi) \\
\overset{Definition}{\Longrightarrow} & \widetilde{(P \wedge Q)}(\phi)
\end{array}
$$

Monotonicity

$$\begin{array}{ll} & (\widetilde{P} \wedge \widetilde{Q})(\widetilde{\phi}) \ \wedge \ \widetilde{\phi} \sqsubseteq \widetilde{\phi'} \\ \overset{Definition}{\Longrightarrow} & \widetilde{P}(\widetilde{\phi}) \ \wedge \ \widetilde{Q}(\widetilde{\phi}) \ \wedge \ \widetilde{\phi} \sqsubseteq \widetilde{\phi'} \\ \overset{Monotonicity}{\Longrightarrow} & \widetilde{P}(\widetilde{\phi'}) \ \wedge \ \widetilde{Q}(\widetilde{\phi'}) \\ \overset{Definition}{\Longrightarrow} & (\widetilde{P} \wedge \widetilde{Q})(\widetilde{\phi'}) \\ \overset{Definition}{\Longrightarrow} & (\widetilde{P \wedge Q})(\widetilde{\phi'}) \end{array}$$

□

Proof of Lemma 3.32.

$$\begin{array}{ll} & (\widetilde{P \vee Q})(\widetilde{\phi}) \\ \overset{Definition}{\Longleftrightarrow} & (\widetilde{P} \vee \widetilde{Q})(\widetilde{\phi}) \\ \overset{Definition}{\Longleftrightarrow} & \widetilde{P}(\widetilde{\phi}) \vee \widetilde{Q}(\widetilde{\phi}) \\ \overset{AGTDef.}{\Longleftrightarrow} & (\exists \phi \in \gamma(\widetilde{\phi}).\ P(\phi)) \vee (\exists \phi \in \gamma(\widetilde{\phi}).\ Q(\phi)) \\ \Longleftrightarrow & (\exists \phi \in \gamma(\widetilde{\phi}).\ P(\phi) \ \vee \ Q(\phi)) \\ \overset{Definition}{\Longleftrightarrow} & (\exists \phi \in \gamma(\widetilde{\phi}).\ (P \vee Q)(\phi)) \end{array}$$

□

Proof of Lemma 3.35.

Adjoint Equation

$$\alpha(\overline{f}(\gamma(\phi))) = f(\phi)$$

Proof:

$\alpha(\overline{f}(\gamma(\phi)))$ defined, since $\{\overline{f}\}$-partial Galois connection, i.e.

$$\alpha(\overline{f}(\gamma(\phi))) = \alpha(\{f(\phi)\}) = \widetilde{\phi} \tag{A.1}$$

Applying rule 1 of partial Galois connections to A.1

$$\{\ f(\phi)\ \} \subseteq \gamma(\widetilde{\phi}) \tag{A.2}$$

(A.3)

Applying rule 2 of partial Galois connections to A.1, using $\{\ f(\phi)\ \} \subseteq \gamma(f(\phi))$

$$\widetilde{\phi} \sqsubseteq f(\phi) \tag{A.4}$$

Combining A.2 and A.4

$$\begin{aligned} & \{\ f(\phi)\ \} \subseteq \gamma(\widetilde{\phi}) \subseteq \gamma(f(\phi)) \\ \Longrightarrow\ & \gamma(\widetilde{\phi}) = \{\ f(\phi)\ \} \\ \Longrightarrow\ & \widetilde{\phi} = f(\phi) \end{aligned}$$

Soundness

Introduction

$$\begin{aligned} & \widetilde{f}(\phi) \\ =\ & \alpha(\overline{f}(\gamma(\phi))) \\ =\ & f(\phi) \end{aligned}$$

Monotonicity

We assume $\widetilde{\phi_1}, \widetilde{\phi_2} \in \widetilde{\text{Formula}}$ with $\widetilde{\phi_1} \sqsubseteq \widetilde{\phi_2}$

$$\begin{aligned} & \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_2} \\ \Longrightarrow\ & \gamma(\widetilde{\phi_1}) \subseteq \gamma(\widetilde{\phi_2}) \\ \Longrightarrow\ & \overline{f}(\gamma(\widetilde{\phi_1})) \subseteq \overline{f}(\gamma(\widetilde{\phi_2})) \\ \Longrightarrow\ & \overline{f}(\gamma(\widetilde{\phi_1})) \subseteq \gamma(\alpha(\overline{f}(\gamma(\widetilde{\phi_2})))) \\ \Longrightarrow\ & \alpha(\overline{f}(\gamma(\widetilde{\phi_1}))) \sqsubseteq \alpha(\overline{f}(\gamma(\widetilde{\phi_2}))) \end{aligned}$$

Optimality

Proof by contradiction. Assume there exists a sound lifting $\widetilde{f'}$ such that $\widetilde{f'}(\widetilde{\phi}) \sqsubset \widetilde{f}(\widetilde{\phi})$ for some $\widetilde{\phi} \in \widetilde{\text{Formula}}$. Using the introduction rule:

$$\forall \phi \in \text{Formula}.\ f(\phi) \sqsubseteq \widetilde{f'}(\phi)$$

Using the monotonicity rule:

$$\forall \phi \in \gamma(\widetilde{\phi}).\ \widetilde{f'}(\phi) \sqsubseteq \widetilde{f'}(\widetilde{\phi})$$

Transitivity:

$$\begin{aligned} & \forall \phi \in \gamma(\widetilde{\phi}).\ f(\phi) \sqsubseteq \widetilde{f'}(\widetilde{\phi}) \\ \Longrightarrow\ & \forall \phi \in \gamma(\widetilde{\phi}).\ f(\phi) \in \gamma(\widetilde{f'}(\widetilde{\phi})) \\ \Longrightarrow\ & \overline{f}(\gamma(\widetilde{\phi})) \subseteq \gamma(\widetilde{f'}(\widetilde{\phi})) \end{aligned}$$

Using rule 2 of partial Galois connections

$$\begin{aligned} & \overline{f}(\gamma(\widetilde{\phi})) \subseteq \gamma(\widetilde{f}'(\widetilde{\phi})) \\ \Longrightarrow\ & \alpha(\overline{f}(\gamma(\widetilde{\phi}))) \sqsubseteq \widetilde{f}'(\widetilde{\phi}) \\ \Longrightarrow\ & \widetilde{f}(\widetilde{\phi}) \sqsubseteq \widetilde{f}'(\widetilde{\phi}) \end{aligned}$$

Contradiction.

□

Proof of Lemma 3.36.

Introduction

$$\begin{aligned} & g(f(\phi)) \\ \overset{\textit{Introduction } \widetilde{g}}{\sqsubseteq}\ & \widetilde{g}(f(\phi)) \\ \overset{\substack{\textit{Introduction } \widetilde{f} \\ \& \\ \textit{Monotonicity } \widetilde{g}}}{\sqsubseteq}\ & \widetilde{g}(\widetilde{f}(\phi)) \\ =\ & \widetilde{g}(\widetilde{f}(\phi)) \\ =\ & (\widetilde{g} \circ \widetilde{f})(\phi) \\ =\ & \widetilde{(g \circ f)}(\phi) \end{aligned}$$

Monotonicity

$$\begin{aligned} & \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_2} \\ \overset{\textit{Monotonicity } \widetilde{f}}{\Longrightarrow}\ & \widetilde{f}(\widetilde{\phi_1}) \sqsubseteq \widetilde{f}(\widetilde{\phi_2}) \\ \overset{\textit{Monotonicity } \widetilde{g}}{\Longrightarrow}\ & \widetilde{g}(\widetilde{f}(\widetilde{\phi_1})) \sqsubseteq \widetilde{g}(\widetilde{f}(\widetilde{\phi_2})) \\ \overset{\textit{Definition}}{\Longrightarrow}\ & \widetilde{(g \circ f)}(\widetilde{\phi_1}) \sqsubseteq \widetilde{(g \circ f)}(\widetilde{\phi_2}) \end{aligned}$$

□

Proof of Lemma 3.39.
Analogous to proof of lemma 3.10. □

Proof of Lemma 3.43.
Goal: $\widetilde{\vDash} \{\widetilde{\phi_1}\}\ \widetilde{s}\texttt{; assert } \widetilde{\phi_2}\ \{\widetilde{\phi_2}\}$.

According to definition 3.38 we may assume

$$\widetilde{\pi_1} \overset{\widetilde{s}\texttt{; assert } \widetilde{\phi_2}}{\rightsquigarrow} \widetilde{\pi_2} \tag{A.5}$$

and

$$\widetilde{\pi_1} \mathrel{\widetilde{\models}} \widetilde{\phi_1} \tag{A.6}$$

for some $\widetilde{\pi_1}, \widetilde{\pi_2} \in \widetilde{\text{ProgramState}}$ and have to show

$$\widetilde{\pi_2} \mathrel{\widetilde{\models}} \widetilde{\phi_2} \tag{A.7}$$

It follows from A.5 and lemma 3.39 that there exists $\widetilde{\pi} \in \widetilde{\text{ProgramState}}$ such that

$$\widetilde{\pi} \xrightarrow{\texttt{assert}\ \widetilde{\phi_2}} \widetilde{\pi_2} \tag{A.8}$$

Then A.7 follows by definition of the assertion statement (see example 3.41 for an illustration). □

Proof of Lemma 3.49.

Introduction

$$\begin{aligned}
& P(\phi_1, \phi_2) \\
\overset{Introduction}{\Longrightarrow}\ & \exists \widetilde{\phi_2}.\ \widetilde{P}(\phi_1) = \widetilde{\phi_2} \\
\overset{Strength}{\Longrightarrow}\ & \exists \widetilde{\phi_2}.\ \widetilde{P}(\phi_1) = \widetilde{\phi_2} \ \wedge\ \exists \phi \in \gamma(\widetilde{\phi_2}).\ P(\phi_1, \phi) \wedge \phi \Rightarrow \phi_2 \\
\Longrightarrow\ & \exists \widetilde{\phi_2}.\ \widetilde{P}(\phi_1) = \widetilde{\phi_2} \ \wedge\ \exists \phi \in \gamma(\widetilde{\phi_2}).\ \phi \Rightarrow \phi_2 \\
\Longrightarrow\ & \exists \widetilde{\phi_2}.\ \widetilde{P}(\phi_1) = \widetilde{\phi_2} \ \wedge\ \widetilde{\phi_2} \mathrel{\widetilde{\Rightarrow}} \phi_2 \\
\overset{Definition}{\Longrightarrow}\ & \widetilde{P}(\phi_1, \phi_2)
\end{aligned}$$

Monotonicity

$$\begin{aligned}
& \widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2}) \ \wedge\ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_1'} \ \wedge\ \widetilde{\phi_2} \sqsubseteq \widetilde{\phi_2'} \\
\overset{Definition}{\Longrightarrow}\ & (\exists \widetilde{\phi}.\ \vec{P}(\widetilde{\phi_1}) = \widetilde{\phi} \wedge \widetilde{\phi} \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi_2}) \ \wedge\ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_1'} \ \wedge\ \widetilde{\phi_2} \sqsubseteq \widetilde{\phi_2'} \\
\overset{Monotonicity}{\Longrightarrow}\ & (\exists \widetilde{\phi}, \widetilde{\phi'}.\ \vec{P}(\widetilde{\phi_1'}) = \widetilde{\phi'} \wedge \widetilde{\phi} \sqsubseteq \widetilde{\phi'} \wedge \widetilde{\phi} \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi_2}) \ \wedge\ \widetilde{\phi_2} \sqsubseteq \widetilde{\phi_2'} \\
\Longrightarrow\ & (\exists \widetilde{\phi'}.\ \vec{P}(\widetilde{\phi_1'}) = \widetilde{\phi'} \wedge \widetilde{\phi'} \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi_2}) \ \wedge\ \widetilde{\phi_2} \sqsubseteq \widetilde{\phi_2'} \\
\Longrightarrow\ & (\exists \widetilde{\phi'}.\ \vec{P}(\widetilde{\phi_1'}) = \widetilde{\phi'} \wedge \widetilde{\phi'} \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi_2'}) \\
\overset{Definition}{\Longrightarrow}\ & \widetilde{P}(\phi_1', \phi_2')
\end{aligned}$$

□

Proof of Lemma 3.53.
The restriction $[\![\widetilde{\phi_2'}]\!] \cap P \subseteq [\![\widetilde{\phi_2}]\!]$ makes sure that only program states that satisfy $\widetilde{\phi_2}$ reach the corresponding point during execution. $\widetilde{\phi_2'}$ is ensured by $\vec{\text{Soundness}}$, P is explicitly checked.

On the other hand, $[\![\widetilde{\phi_2'}]\!] \cap [\![\widetilde{\phi_2}]\!] \subseteq [\![\widetilde{\phi_2'}]\!] \cap P$ ensures that P is not stricter than necessary. Assume there exists a program state $\widetilde{\pi} \in ([\![\widetilde{\phi_2'}]\!] \cap [\![\widetilde{\phi_2}]\!])$ with $\widetilde{\pi} \notin ([\![\widetilde{\phi_2'}]\!] \cap P)$. It follows that $\neg P(\widetilde{\pi})$, i.e. a runtime exception is thrown that indicates that a runtime check fails. However, $\widetilde{\pi}$ neither violates $\widetilde{\phi_2'}$ nor $\widetilde{\phi_2}$ and is thus not expected to violate any runtime checks. □

Proof of Corollary 3.55.
The condition of lemma 3.53 is directly implied. □

Proof of Lemma 3.56.

Introduction

$$
\begin{aligned}
& P_3(\phi_1, \phi_3) \\
\Longrightarrow \quad & \exists \phi_2.\ P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi_3) \\
\overset{\textit{Introduction}}{\Longrightarrow} \quad & \exists \phi_2.\ P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi_3) \wedge (\exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2}) \\
\overset{\textit{Strength}}{\Longrightarrow} \quad & \exists \phi_2.\ P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi_3) \wedge (\exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2} \\
& \qquad \wedge (\exists \phi_2' \in \gamma(\widetilde{\phi_2}).\ \phi_2' \Rightarrow \phi_2)) \\
\overset{\textit{Lemma 3.11}}{\Longrightarrow} \quad & \exists \phi_2.\ P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi_3) \wedge (\exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2} \\
& \qquad \wedge (\exists \phi_2' \in \gamma(\widetilde{\phi_2}), \phi_3'.\ \phi_2' \Rightarrow \phi_2 \wedge P_2(\phi_2', \phi_3'))) \\
\Longrightarrow \quad & \exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2} \wedge (\exists \phi_2' \in \gamma(\widetilde{\phi_2}), \phi_3'.\ P_2(\phi_2', \phi_3')) \\
\overset{\textit{Introduction}}{\Longrightarrow} \quad & \exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2} \wedge (\exists \phi_2' \in \gamma(\widetilde{\phi_2}), \widetilde{\phi_3'}.\ \vec{P_2}(\phi_2') = \widetilde{\phi_3'}) \\
\overset{\textit{Monotonicity}}{\Longrightarrow} \quad & \exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2} \wedge (\exists \widetilde{\phi_3}.\ \vec{P_2}(\widetilde{\phi_2}) = \widetilde{\phi_3}) \\
\Longrightarrow \quad & \exists \widetilde{\phi_3}.\ \vec{P_2}(\vec{P_1}(\phi_1)) = \widetilde{\phi_3} \\
\Longrightarrow \quad & \exists \widetilde{\phi_3}.\ \vec{P_3}(\phi_1) = \widetilde{\phi_3}
\end{aligned}
$$

Strength

$$
\begin{aligned}
& \vec{P}_3(\widetilde{\phi_1}) = \widetilde{\phi_3} \wedge \phi_1 \in \gamma(\widetilde{\phi_1}) \wedge P_3(\phi_1, \phi) \\
\overset{Defenitions}{\Longrightarrow} \quad & \exists \widetilde{\phi_2}, \phi'.\ \vec{P}_1(\widetilde{\phi_1}) = \widetilde{\phi_2} \wedge \vec{P}_2(\widetilde{\phi_2}) = \widetilde{\phi_3} \wedge \phi_1 \in \gamma(\widetilde{\phi_1}) \\
& \qquad \wedge P_1(\phi_1, \phi') \wedge P_2(\phi', \phi) \\
\overset{Strength}{\Longrightarrow} \quad & \exists \widetilde{\phi_2}, \phi'.\ \vec{P}_1(\widetilde{\phi_1}) = \widetilde{\phi_2} \wedge \vec{P}_2(\widetilde{\phi_2}) = \widetilde{\phi_3} \wedge \phi_1 \in \gamma(\widetilde{\phi_1}) \\
& \qquad \wedge P_1(\phi_1, \phi') \wedge P_2(\phi', \phi) \wedge (\exists \phi_2 \in \gamma(\widetilde{\phi_2}).\ P_1(\phi_1, \phi_2) \wedge \phi_2 \Rightarrow \phi') \\
\Longrightarrow \quad & \exists \widetilde{\phi_2}, \phi', \phi_2 \in \gamma(\widetilde{\phi_2}).\ \vec{P}_1(\widetilde{\phi_1}) = \widetilde{\phi_2} \wedge \vec{P}_2(\widetilde{\phi_2}) = \widetilde{\phi_3} \wedge \phi_1 \in \gamma(\widetilde{\phi_1}) \\
& \qquad \wedge P_1(\phi_1, \phi_2) \wedge P_2(\phi', \phi) \wedge \phi_2 \Rightarrow \phi' \\
\Longrightarrow \quad & \exists \widetilde{\phi_2}, \phi', \phi_2 \in \gamma(\widetilde{\phi_2}).\ \vec{P}_2(\widetilde{\phi_2}) = \widetilde{\phi_3} \\
& \qquad \wedge P_1(\phi_1, \phi_2) \wedge P_2(\phi', \phi) \wedge \phi_2 \Rightarrow \phi' \\
\overset{Lemma\ 3.11}{\Longrightarrow} \quad & \exists \widetilde{\phi_2}, \phi_2 \in \gamma(\widetilde{\phi_2}), \phi''.\ \vec{P}_2(\widetilde{\phi_2}) = \widetilde{\phi_3} \\
& \qquad \wedge P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi'') \wedge \phi'' \Rightarrow \phi \\
\overset{Strength}{\Longrightarrow} \quad & \exists \widetilde{\phi_2}, \phi_2 \in \gamma(\widetilde{\phi_2}), \phi''.\ \vec{P}_2(\widetilde{\phi_2}) = \widetilde{\phi_3} \\
& \qquad \wedge P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi'') \wedge \phi'' \Rightarrow \phi \\
& \qquad \wedge (\exists \phi_3 \in \gamma(\widetilde{\phi_3}).\ P_2(\phi_2, \phi_3) \wedge \phi_3 \Rightarrow \phi'') \\
\Longrightarrow \quad & \exists \phi_2 \in \gamma(\widetilde{\phi_2}), \phi_3 \in \gamma(\widetilde{\phi_3}).\ P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi_3) \wedge \phi_3 \Rightarrow \phi \\
\overset{Defenition}{\Longrightarrow} \quad & \exists \phi_3 \in \gamma(\widetilde{\phi_3}).\ P_3(\phi_1, \phi_3) \wedge \phi_3 \Rightarrow \phi
\end{aligned}
$$

Monotonicity

Compositions of monotonic functions are monotonic.

□

Proof of Lemma 3.57.

Soundness Introduction The identity function is total.

Strength

Known:

$$\mathsf{id}(\widetilde{\phi_1}) = \widetilde{\phi_2} \wedge \phi_1 \in \gamma(\widetilde{\phi_1}) \wedge \phi_1 \Rightarrow \phi$$

Goal:

$$\exists \phi_2 \in \gamma(\widetilde{\phi_2}).\ \phi_1 \Rightarrow \phi_2 \wedge \phi_2 \Rightarrow \phi$$

The goal is satisfied when choosing $\phi_2 = \phi_1$

Monotonicity Trivial.

Optimality

Proof by contradiction.

Assume there was another, more optimal deterministic lifting $\vec{f} : \widetilde{\text{Formula}} \to \widetilde{\text{Formula}}$. Note that $\vec{f}$ must be total as it would otherwise not satisfy the introduction rule. Let $\widetilde{\phi} \in \widetilde{\text{Formula}}$ be a chosen such that $\vec{f}(\widetilde{\phi}) \sqsubset \mathsf{id}(\widetilde{\phi})$ (exists, since $\vec{f}$ is more optimal).

Then there exists $\phi \in (\gamma(\mathsf{id}(\widetilde{\phi}))\backslash\gamma(\vec{f}(\widetilde{\phi}))$, i.e. $\phi \in (\gamma(\widetilde{\phi})\backslash\gamma(\vec{f}(\widetilde{\phi}))$. Applying the strength rule to $\vec{f}$ and $\phi \Rightarrow \phi$ we can deduce

$$\exists \phi' \in \gamma(\vec{f}(\widetilde{\phi})).\ \phi \Rightarrow \phi' \ \wedge\ \phi' \Rightarrow \phi$$

This means that $\phi \in \gamma(\vec{f}(\widetilde{\phi}))$. Contradiction.

□

Proof of Lemma 3.59.
SVL only ever executes a statement s if it was successfully verified. After extending the Hoare logic, derivable Hoare triples are still derivable. Successful verification implies that there exists some Hoare derivation $\vdash \{\phi_1\}\ s\ \{\phi_2\}$ for every statement s executed by the program. Soundness of the Hoare logic implies that the small-step semantics do not get stuck while executing s. Thus, the restricted domain of $\cdot \longrightarrow \cdot$ can never affect reachable executions. □

Proof of Theorem 3.62.
To prove $\vec{\text{Soundness}}$ we may assume:

$$\vec{\vdash}\ \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\} \tag{A.9}$$

$$\widetilde{\pi_1} \xrightarrow{\widetilde{s}} \widetilde{\pi_2} \tag{A.10}$$

$$\widetilde{\pi_1} \mathrel{\widetilde{\vDash}} \widetilde{\phi_1} \tag{A.11}$$

Goal:

$$\widetilde{\pi_2} \mathrel{\widetilde{\vDash}} \widetilde{\phi_2} \tag{A.12}$$

Case: $\widetilde{\phi_1} = \phi_1$ for some $\phi_1 \in \text{Formula}$
We can simplify A.11 as

$$\widetilde{\pi_1} \vDash \phi_1 \tag{A.13}$$

From semi-optimality of A.9 (see definition 3.61) we can derive

$$\vdash \{\phi_1\}\ s\ \{\phi\} \qquad \text{for some } s \in \gamma(\widetilde{s}),\ \phi_2 \in \text{Formula} \tag{A.14}$$

Applying the strength rule for deterministic liftings we can derive from A.14 and A.9 that

$$\vdash \{\phi_1\}\ s\ \{\phi_2\}\ \wedge\ \phi_2 \Rightarrow \phi \qquad \text{for some } \phi_2 \in \gamma(\widetilde{\phi_2}) \tag{A.15}$$

From soundness (progress) of the static Hoare logic we can deduce that executing s from a program state satisfying ϕ_1 (like $\widetilde{\pi_1}$) does not end up in a stuck state. Combining this knowledge with $A.10$ we can conclude that executing s must terminate (otherwise, $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot$, being the gradual lifting, could not terminate for $\widetilde{s} \sqsupseteq s$).

$$\pi_1 \xrightarrow{s} \pi_2 \qquad \text{for some } \pi_1 \in \gamma(\widetilde{\pi_1}),\ \pi_2 \in \textsc{ProgramState} \tag{A.16}$$

From soundness of the static Hoare logic we can deduce

$$\pi_2 \vDash \phi_2 \tag{A.17}$$

Applying the introduction rule of gradually lifted functions to A.16 we can derive

$$\pi_1 \mathrel{\widetilde{\xrightarrow{s}}} \widetilde{\pi} \ \wedge\ \pi_2 \sqsubseteq \widetilde{\pi} \qquad \text{for some } \widetilde{\pi} \in \widetilde{\textsc{ProgramState}} \tag{A.18}$$

Applying the monotonicity rule of gradually lifted functions to A.18 and A.10 we can derive that

$$\widetilde{\pi} \sqsubseteq \widetilde{\pi_2} \tag{A.19}$$

It follows from lemma 3.22 that

$$\widetilde{\pi_2} \vDash \phi_2 \tag{A.20}$$

We can generalize this using $\phi_2 \in \gamma(\widetilde{\phi_2})$ (see A.15)

$$\widetilde{\pi_2} \mathrel{\widetilde{\vDash}} \widetilde{\phi_2} \tag{A.21}$$

Case: $\widetilde{\phi_1}$ partially unknown

From semi-optimality of A.10 (see definition 3.61) we can derive

$$\pi_1 \xrightarrow{s} \pi_2 \qquad \text{for some } \pi_1 \in \gamma(\widetilde{\pi_1}),\ s \in \gamma(\widetilde{s}),\ \pi_2 \in \textsc{ProgramState} \tag{A.22}$$

Recall that formula evaluation is immune to concretization (REF), so from A.11 follows

$$\pi_1 \mathrel{\widetilde{\vDash}} \widetilde{\phi_1} \qquad \text{or equivalently} \qquad \pi_1 \vDash \phi_{1a} \qquad \text{for some } \phi_{1a} \in \gamma(\widetilde{\phi_1}) \tag{A.23}$$

Using the monotonicity of $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot$ we can deduce from A.10 and A.22 that

$$\pi_2 \in \gamma(\widetilde{\pi_2}) \tag{A.24}$$

From completeness of the static system (see definition 3.58) it follows from A.22 that

$$\vdash \{\phi_{1b}\}\ s\ \{\phi\} \ \wedge\ \pi_1 \vDash \phi_{1b} \qquad \text{for some } \phi_{1b}, \phi \in \textsc{Formula} \tag{A.25}$$

Due to $[\![\pi_1]\!]$ being a filter we can derive from $\pi_1 \vDash \phi_{1a}$ (A.23) and $\pi_1 \vDash \phi_{1b}$ (A.25) that

$$\pi_1 \vDash \phi_1 \ \wedge\ \phi_1 \Rightarrow \phi_{1a} \ \wedge\ \phi_1 \Rightarrow \phi_{1b} \qquad \text{for some } \phi_1 \in \textsc{Formula} \tag{A.26}$$

From monotonicity of $\vdash \{\cdot\} \cdot \{\cdot\}$ in its first argument (lemma 3.11) we can deduce

$$\vdash \{\phi_1\}\ s\ \{\phi'\}\ \wedge\ \phi' \Rightarrow \phi \qquad \text{for some } \phi_2 \in \textsc{Formula} \tag{A.27}$$

Applying the introduction rule for deterministic liftings we can deduce

$$\vec{\vdash} \{\phi_1\}\ s\ \{\widetilde{\phi}\} \qquad \text{for some } \widetilde{\phi} \in \widetilde{\textsc{Formula}} \tag{A.28}$$

Applying the strength rule for deterministic liftings we can derive from A.27 and A.28 that

$$\vdash \{\phi_1\}\ s\ \{\phi_2\}\ \wedge\ \phi_2 \Rightarrow \phi' \qquad \text{for some } \phi_2 \in \gamma(\widetilde{\phi}) \tag{A.29}$$

From soundness of the static system we can deduce (using A.28, A.22, A.26) that

$$\pi_2 \vDash \phi_2 \tag{A.30}$$

and therefore (using A.24 and lemma 3.22)

$$\widetilde{\pi_2} \mathrel{\widetilde{\vDash}} \phi_2 \tag{A.31}$$

and therefore (using A.29)

$$\widetilde{\pi_2} \mathrel{\widetilde{\vDash}} \widetilde{\phi} \tag{A.32}$$

Now, using $\phi_1 \in \gamma(\widetilde{\phi_1})$ (implied from the fact that $\phi_{1a} \in \gamma(\widetilde{\phi_1})$, $\phi_1 \Rightarrow \phi_{1a}$, see see section 3.2.4) we can apply monotonicity of deterministic liftings to A.28, we can derive

$$\widetilde{\phi} \sqsubseteq \widetilde{\phi_2} \tag{A.33}$$

and therefore

$$\widetilde{\pi_2} \mathrel{\widetilde{\vDash}} \widetilde{\phi_2} \tag{A.34}$$

□

Proof of Lemma 3.63.

$$\dfrac{\dfrac{\dfrac{\vec{\vdash} \{\widetilde{\phi_1}\}\ \widetilde{s_1};\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}{\vec{\vdash} \{\widetilde{\phi_1}\}\ \widetilde{s_1}\ \{\widetilde{\phi_2}\} \qquad \vec{\vdash} \{\widetilde{\phi_2}\}\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}\ \textsc{inversion}}{\widetilde{\vDash} \{\widetilde{\phi_1}\}\ \widetilde{s_1}\ \{\widetilde{\phi_2}\} \qquad \widetilde{\vDash} \{\widetilde{\phi_2}\}\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}\ \vec{\textsc{S}}\textsc{oundness}}{\widetilde{\vDash} \{\widetilde{\phi_1}\}\ \widetilde{s_1};\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}\ \textsc{seq}$$

□

Proof of Lemma 4.10. Omitted (proof in [2]). □

Proof of Lemma 4.8.
For $\cdot \longrightarrow \cdot$ to be well-defined, at most one result can be deducible per input. The rules in figure 4.10 are syntax directed, so we can focus on individual rules when checking for determinism. This can be done by looking at the source of all variables used to construct return values (i.e. the variables used on the right hand side of $\longrightarrow$ in the conclusion). All those variables must either be drawn directly from the input or be uniquely specified using premises.

$\widetilde{\text{S}}$sSkip H, ρ, A, s, S are directly drawn from the input.

$\widetilde{\text{S}}$sFieldAssign ρ, A, s, S are directly drawn from the input, H' is defined using premises and depends on the uniqueness of H, o, f, v_y. These are drawn from input or are result of expression evaluation which is deterministic.

The same approach can be used for all remaining rules. □

Proof of Lemma 4.12.

Well-definedness The rules are syntax directed and only can deduce at most one result per input.

Deterministic lifting Rule-wise.

HSkip: See lemma 3.57.

HAlloc, HFieldAssign, HReturn, HAssert: Composition of lifted components as defined in 4.3.

Remaining rules analogous.

□

Proof of Lemma 4.13.
For $\cdot \widetilde{\longrightarrow} \cdot$ to be well-defined, the inductive rules may allow deducing at most one return value for every input. For the most part, this is the case due to $\cdot \longrightarrow \cdot$ being well-defined (see lemma 4.8). We show that the same is true for adjustments made in figure 4.14. Note the adjustments do not change the fact that the inductive rules are syntax-directed.

Rule $\widetilde{\text{S}}$sCall is deterministic: $H, \rho, A, x, y, m, z, s, S$ are forwarded from the input, ρ', A', r are uniquely determined by the premises.

Rule $\widetilde{\text{S}}$sCallFinish is deterministic: H, ρ, x, A, A', s, S are forwarded from the input, v_r is uniquely determined by the premises. □

Proof of Lemma 4.14. For the rules copied from $\cdot \longrightarrow \cdot$, the rules are trivially satisfied as there is no difference between gradual and non-gradual statements. Precision is only meaningful for call statements:

$\widetilde{\text{SsCall}}$

Introduction

For static precondition, $\widetilde{\text{SsCall}}$ is identical to SsCall.

Monotonicity

Reducing the precision of the precondition results in all permissions being passed to the topmost stack frame. Having more permissions than before cannot introduce runtime failures. Succeeding executions will thus be observationally identical after reducing precision.

$\widetilde{\text{SsCallFinish}}$

Introduction

For static postcondition, $\widetilde{\text{SsCallFinish}}$ is identical to SsCallFinish.

Monotonicity

The return value of $\widetilde{\text{SsCallFinish}}$ is independent of the postcondition thus monotonic w.r.t. to it. In terms of definedness, reducing the precision of the postcondition cannot result in premises that were satisfied before to be unsatisfied.

□

Proof of Lemma 4.15.
Goal:

$$\begin{aligned}&\forall \pi_1, \pi_2 \in \text{ProgramState}.\ \pi_1 \xrightarrow{s} \pi_2\\ \implies &\exists \Gamma \in \text{TypeEnv}, \phi_1, \phi_2 \in \text{Formula}.\ \Gamma \vdash \{\phi_1\}\ s\ \{\phi_2\} \land \pi_1 \vDash \phi_1\end{aligned}$$

for statements s mentioned in the lemma.

`skip`

$$\Gamma \vdash \{\phi_1\}\ s\ \{\phi_2\} \land \pi_1 \vDash \phi_1$$

trivially holds for $\phi_1 = \phi_2 = \texttt{true}$ and any Γ

$x.f$ `:=` y

Known: $\pi_1 \xrightarrow{x.f\ :=\ y} \pi_2$ for some $x, y \in \text{Var}$, $f \in \text{FieldName}$ and $\pi_1, \pi_2 \in \text{ProgramState}$.

We choose

$$\begin{aligned}\phi_1 &= \texttt{acc}(x.f)\\ \phi_2 &= \texttt{true} * \texttt{acc}(x.f) * (\texttt{x} \neq \texttt{null}) * (\texttt{x.f = y})\end{aligned}$$

Then $\Gamma \vdash \{\phi_1\}\ x.f\ :=\ y\ \{\phi_2\}$ can be deduced using HFieldAssign (for some Γ). Also $\pi_1 \vDash \phi_1$ holds by inversion of $\vec{\text{SsFieldAssign}}$.

x `:=` e

Known: $\pi_1 \xrightarrow{x \texttt{ := } e} \pi_2$ for some $x \in$ Var, $e \in$ Expr and $\pi_1, \pi_2 \in$ ProgramState.

We choose

$$\phi_1 = \texttt{acc(e)}$$
$$\phi_2 = \texttt{true * (}x \texttt{ = } e\texttt{)}$$

Then $\Gamma \vdash \{\phi_1\}\ x$ `:=` $e\ \{\phi_2\}$ can be deduced using HVarAssign (for some Γ). The remaining premises as well as $\pi_1 \vDash \phi_1$ hold by inversion of $\vec{\text{S}}$sVarAssign.

x `:= new` C

$\Gamma \vdash \{\texttt{true}\}\ x$ `:= new` $C\ \{\phi_2\}$ can be deduced using HAlloc (for some Γ and ϕ_2). Furthermore $\pi_1 \vDash$ `true` trivially holds.

`return` x

$\Gamma \vdash \{\texttt{true}\}$ `return` $x\ \{\phi_2\}$ can be deduced using HAlloc (for some Γ and ϕ_2). Furthermore $\pi_1 \vDash$ `true` trivially holds.

`assert` ϕ

$\Gamma \vdash \{\phi\}$ `assert` $\phi\ \{\phi\}$ can be deduced using HAlloc (for some Γ). Furthermore $\pi_1 \vDash \phi$ holds by inversion of $\vec{\text{S}}$sAssert.

`release` ϕ

$\Gamma \vdash \{\phi\}$ `release` $\phi\ \{\texttt{true}\}$ can be deduced using HRelease (for some Γ). Furthermore $\pi_1 \vDash \phi$ holds by inversion of $\vec{\text{S}}$sRelease.

□

Proof of Lemma 4.16.
Semi-optimality of the gradual small-step semantics is trivially the case due to being identical with the small-step semantics of $\mathbf{SVL_{IDF}}$.

Semi-optimality of the deterministic gradual Hoare logic: Goal:

$$\forall \Gamma \in \text{TypeEnv}, \phi_1 \in \text{Formula}, \widetilde{\phi_2} \in \widetilde{\text{F}}\text{ormula}.\ \Gamma \vec{\vdash} \{\phi_1\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}$$
$$\implies \exists s \in \gamma(\widetilde{s}), \phi_2 \in \text{Formula}.\ \Gamma \vdash \{\phi_1\}\ s\ \{\phi_2\}$$

for statements s mentioned in the lemma.

`skip`

Known: $\Gamma \vec{\vdash} \{\phi_1\}$ `skip` $\{\widetilde{\phi_2}\}$ for some $\phi_1 \in$ ProgramState, $\widetilde{\phi_2} \in \widetilde{\text{P}}$rogramState.

Then $\Gamma \vdash \{\phi_1\}$ `skip` $\{\phi_1\}$ holds (HSkip).

$x\texttt{.}f$ `:=` y

Known: $\Gamma \vec{\vdash} \{\phi_1\}\ x\texttt{.}f$ `:=` $y\ \{\widetilde{\phi_2}\}$ for some $x, y \in$ Var, $f \in$ FieldName and $\phi_1 \in$

ProgramState, $\widetilde{\phi_2} \in \widetilde{\text{ProgramState}}$. Applying inversion (rule $\vec{\text{H}}$FieldAssign) we can deduce:

$$\phi_1 \div \texttt{acc(x.f)} = \widetilde{\phi'}$$

for some $\widetilde{\phi'} \in \widetilde{\text{Formula}}$. Expanding the definition of $\cdot \div \cdot$

$$\min_{\Rightarrow}\{\ \phi_2 \in \text{SfrmFormula} \mid \phi_1 \Rightarrow \texttt{acc(x.f) * } \phi_2\ \} = \widetilde{\phi'}$$

It follows that $\phi_1 \Rightarrow \texttt{acc(x.f)}$.

Then $\Gamma \vdash \{\phi_1\}\ x\texttt{.}f\ \texttt{:=}\ y\ \{\texttt{true * acc(}x\texttt{.}f\texttt{) * (x} \neq \texttt{null) * (x.f = y)}\}$ holds (HFieldAssign).

x `:=` e

Known: $\Gamma \vec{\vdash} \{\phi_1\}\ x\ \texttt{:=}\ e\ \{\widetilde{\phi_2}\}$ for some $x \in$ Var, $e \in$ Expr and $\phi_1 \in$ ProgramState, $\widetilde{\phi_2} \in \widetilde{\text{ProgramState}}$. Applying inversion (rule $\vec{\text{H}}$VarAssign) we can deduce:

$$\phi_1 \Rightarrow \mathsf{acc}(e)$$
$$x \notin \mathsf{FV}(e)$$

Then $\Gamma \vdash \{\phi_1\}\ x\ \texttt{:=}\ e\ \{\texttt{true * (}x\ \texttt{=}\ e\texttt{)}\}$ holds (HVarAssign).

x `:= new` C

Known: $\Gamma \vec{\vdash} \{\phi_1\}\ x\ \texttt{:= new}\ C\ \{\widetilde{\phi_2}\}$ for some $x \in$ Var, $C \in$ ClassName and $\phi_1 \in$ ProgramState, $\widetilde{\phi_2} \in \widetilde{\text{ProgramState}}$.

Then $\Gamma \vdash \{\phi_1\}\ x\ \texttt{:= new}\ C\ \{\texttt{true * (}x\ \neq\ \texttt{null) * ...}\}$ holds (HAlloc).

`return` x

Known: $\Gamma \vec{\vdash} \{\phi_1\}\ \texttt{return}\ x\ \{\widetilde{\phi_2}\}$ for some $x \in$ Var and $\phi_1 \in$ ProgramState, $\widetilde{\phi_2} \in \widetilde{\text{ProgramState}}$.

Then $\Gamma \vdash \{\phi_1\}\ \texttt{return}\ x\ \{\texttt{true * (result = }x\texttt{)}\}$ holds (HReturn).

`assert` ϕ

Known: $\Gamma \vec{\vdash} \{\phi_1\}\ \texttt{assert}\ \phi\ \{\widetilde{\phi_2}\}$ for some $\phi \in$ Formula and $\phi_1 \in$ ProgramState, $\widetilde{\phi_2} \in \widetilde{\text{ProgramState}}$. Applying inversion (rule $\vec{\text{H}}$Assert) we can deduce:

$$\widetilde{\phi'} \vdash \phi_1 \mathrel{\widetilde{\Rightarrow}} \phi$$

for some $\widetilde{\phi'} \in \widetilde{\text{Formula}}$. By definition of $\cdot \vdash \cdot \mathrel{\widetilde{\Rightarrow}} \cdot$ it follows that $\phi_1 \Rightarrow \phi$.

Then $\Gamma \vdash \{\phi_1\}\ \texttt{assert}\ \phi\ \{\phi_1\}$ holds (HAssert).

`release` ϕ

Known: $\Gamma \vec{\vdash} \{\phi_1\}\ \texttt{release}\ \phi\ \{\widetilde{\phi_2}\}$ for some $\phi \in$ Formula and $\phi_1 \in$ ProgramState, $\widetilde{\phi_2} \in \widetilde{\text{ProgramState}}$. Applying inversion (rule $\vec{\text{H}}$Release) we can deduce:

$$\widetilde{\phi'} \vdash \phi_1 \mathrel{\widetilde{\Rightarrow}} \phi$$

for some $\widetilde{\phi'} \in \widetilde{\text{Formula}}$. By definition of $\cdot \vdash \cdot \mathrel{\widetilde{\Rightarrow}} \cdot$ it follows that $\phi_1 \Rightarrow \phi$.

Then $\Gamma \vdash \{\phi_1\}\ \texttt{release}\ \phi\ \{\texttt{true}\}$ holds (HRelease).

□

Proof of Lemma 4.17.
Assumption:

$$\Gamma \vec{\vdash} \{\widetilde{\phi}\}\ T\ \ x;\ \ s\ \{\widetilde{\phi'}\} \tag{A.35}$$

Using inversion of rule $\vec{\text{H}}$DECLARE we know that

$$x \notin \mathsf{dom}(\Gamma) \tag{A.36}$$
$$x \notin \mathsf{FV}(\widetilde{\phi}) \tag{A.37}$$
$$\Gamma, x : T \vec{\vdash} \{(x\ =\ \mathsf{defaultValue}(T))\ \widetilde{*}\ \widetilde{\phi}\}\ s\ \{\widetilde{\phi'}\} \tag{A.38}$$

Using the induction hypothesis we may apply $\vec{\text{S}}$OUNDNESS to s, deducing

$$\widetilde{\vDash} \{(x\ =\ \mathsf{defaultValue}(T))\ \widetilde{*}\ \widetilde{\phi}\}\ s\ \{\widetilde{\phi'}\} \tag{A.39}$$

Furthermore we can show that

$$\widetilde{\vDash} \{\widetilde{\phi}\}\ T\ \ x\ \{(x\ =\ \mathsf{defaultValue}(T))\ \widetilde{*}\ \widetilde{\phi}\} \tag{A.40}$$

holds (using A.37, find proof below). Using lemma 3.39 we can combine A.39 and A.40 to derive the goal:

$$\widetilde{\vDash} \{\widetilde{\phi}\}\ T\ \ x;\ \ s\ \{\widetilde{\phi'}\} \tag{A.41}$$

Proof for A.40: Assumptions:

$$\pi_1 \overset{T\ x}{\rightsquigarrow} \pi_2 \tag{A.42}$$
$$\pi_1 \widetilde{\vDash} \widetilde{\phi} \tag{A.43}$$

for some $\pi_1, \pi_2 \in \widetilde{\text{P}}$ROGRAMSTATE. Goal:

$$\pi_2 \widetilde{\vDash} (x\ =\ \mathsf{defaultValue}(T))\ \widetilde{*}\ \widetilde{\phi} \tag{A.44}$$

Applying rule inversion (SSDECLARE) to A.42 we can see that π_2 is like π_1, but with x set to $\mathsf{defaultValue}(T)$ (and the assertion consumed). It follows that

$$\pi_2 \widetilde{\vDash} (x\ =\ \mathsf{defaultValue}(T)) \tag{A.45}$$

Since we know that $\widetilde{\phi}$ does not contain x (see A.37), we can deduce that

$$\pi_2 \widetilde{\vDash} \widetilde{\phi} \tag{A.46}$$

The goal follows (the separating conjunction acts like classical conjunction since $(x\ =\ \mathsf{defaultValue}(T))$ does not mention the heap). □

Proof of Lemma 4.18.
Assumption:

$$\Gamma \vec{\vdash} \{\widetilde{\phi}\}\ x\ \texttt{:=}\ y.m(z')\ \{\widetilde{\phi'} \mathbin{\widetilde{*}} \widetilde{\phi_q}\} \tag{A.47}$$

Using inversion of rule $\vec{\text{H}}$CALL we know that

$$\widetilde{\phi} \div x \div \widetilde{\phi_p} = \widetilde{\phi'} \tag{A.48}$$

$$\mathsf{method}_p(C, m) = T_r\ \ m(T_p\ \ z)\ \texttt{requires}\ \widetilde{\phi_{pre}}\texttt{;}\ \texttt{ensures}\ \widetilde{\phi_{post}}\texttt{;}\ \texttt{\{}\ s\ \texttt{\}} \tag{A.49}$$

$$\widetilde{\phi} \mathrel{\widetilde{\Rightarrow}} (y \neq \texttt{null}) \mathbin{\widetilde{*}} \widetilde{\phi_p} \tag{A.50}$$

$$x \neq y \wedge x \neq z' \tag{A.51}$$

$$\widetilde{\phi_p} = \widetilde{\phi_{pre}}[y, z'/\texttt{this}, z] \tag{A.52}$$

$$\widetilde{\phi_q} = \widetilde{\phi_{post}}[y, z', x/\texttt{this}, z, \texttt{result}] \tag{A.53}$$

From gradual well-formedness (section 4.6) it follows that

$$z : T_p, \texttt{this} : C, \texttt{result} : T_r \mathrel{\widetilde{\vdash}} \{\widetilde{\phi_{pre}}\}\ s\ \{\widetilde{\phi_{post}}\} \tag{A.54}$$

$$\mathsf{FV}(\widetilde{\phi_{pre}}) \subseteq \{z, \texttt{this}\} \tag{A.55}$$

$$\mathsf{FV}(\widetilde{\phi_{post}}) \subseteq \{z, \texttt{this}, \texttt{result}\} \tag{A.56}$$

$$\widetilde{\vdash}_{\texttt{sfrm}}\ \widetilde{\phi_{pre}} \tag{A.57}$$

$$\widetilde{\vdash}_{\texttt{sfrm}}\ \widetilde{\phi_{post}} \tag{A.58}$$

$$z, \texttt{this} \notin \mathsf{mod}(s) \tag{A.59}$$

As shown below, it follows that

$$\widetilde{\vDash} \{\widetilde{\phi}\}\ x\ \texttt{:=}\ y.m(z')\ \{\widetilde{\phi'} \mathbin{\widetilde{*}} \widetilde{\phi_q}\} \tag{A.60}$$

Proof for A.60: Assumptions:

$$\pi_1 \overset{x\ \texttt{:=}\ y.m(z')}{\widetilde{\longrightarrow}} \pi_2 \tag{A.61}$$

$$\pi_1 \widetilde{\vDash} \widetilde{\phi} \tag{A.62}$$

for some $\pi_1, \pi_2 \in \widetilde{\text{P}}$ROGRAMSTATE. Goal:

$$\pi_2 \widetilde{\vDash} \widetilde{\phi'} \mathbin{\widetilde{*}} \widetilde{\phi_q} \tag{A.63}$$

Applying rule inversion to A.61 we derive that there exists $\pi_{pre}, \pi_{post} \in \widetilde{\text{P}}$ROGRAMSTATE such that

$$\pi_1 \mathrel{\widetilde{\longrightarrow}} \pi_{pre} \tag{A.64}$$

$$\pi_{pre} \overset{s}{\widetilde{\longrightarrow}} \pi_{post} \tag{A.65}$$

$$\pi_{post} \mathrel{\widetilde{\longrightarrow}} \pi_2 \tag{A.66}$$

where A.64 is determined by $\widetilde{\text{S}}$SCALL and A.66 by $\widetilde{\text{S}}$SCALLFINISH. Applying rule inversion to A.66 it follows that

$$\pi_{post} \widetilde{\vDash} \widetilde{\phi_{post}} \tag{A.67}$$

Combining A.53 and A.56 we can show that $\widetilde{\phi_q}$ may only contain y, z', x. $\widetilde{\text{SsCallFinish}}$ assigns `result` to x, which reflects the substitution in A.53. Because of A.59 $\widetilde{\phi_q}$ does not contain false information about y and z' (as described in section 4.1.8). It follows that

$$\pi_2 \mathrel{\widetilde{\models}} \widetilde{\phi_q} \tag{A.68}$$

Since the variables and fields mentioned in $\widetilde{\phi'}$ are guaranteed to be disjoint from those in $\widetilde{\phi_p}$ and due to A.48, those memory locations cannot be changed by the call. Furthermore x is not in $\widetilde{\phi'}$ such that there cannot be a contradiction to the knowledge in $\widetilde{\phi_q}$ (which may contain information of x, provided by the call). It follows that

$$\pi_2 \mathrel{\widetilde{\models}} \widetilde{\phi'} \tag{A.69}$$

The goal follows from A.68 and A.69 using the fact that the formulas contain disjoint memory locations. □

Proof of Lemma 4.19. Analogous to previous proofs. □

Proof of Lemma 4.20. Analogous to lemma 3.63. □

Proof of Theorem 4.21.
Structural induction over statement s. Most cases are handled by theorem 3.62 and lemmas 4.15 and 4.16. The remaining cases are covered by lemmas 4.17, 4.18, 4.19 and 4.20. □

Proof of Lemma 5.1.
We prove that

$$\begin{aligned} &\forall \phi_1, \phi_{2b} \in \text{Formula}, s \in \text{Stmt}.\ \vdash \{\phi_1\}\ s\ \{\phi_{2b}\} \\ &\qquad \Longrightarrow\ (\exists \phi_{2a} \in \text{Formula}.\ \vec{\vdash}\ \{\phi_1\}\ s\ \{\phi_{2a}\} \wedge \phi_{2a} \Rightarrow \phi_{2b}) \end{aligned} \tag{A.70}$$

As a consequence, all reasoning that allows the static system to be runtime check free must also apply to the gradual system (if all annotations are static).

Proof for A.70:

$$\vdash \{\phi_1\}\ s\ \{\phi_{2b}\}$$

for some $\phi_1, \phi_{2b} \in \text{Formula}$, $s \in \text{Stmt}$. Using the introduction rule of deterministic liftings it follows that

$$\vec{\vdash}\ \{\phi_1\}\ s\ \{\widetilde{\phi_2}\}$$

for some $\widetilde{\phi_2} \in \widetilde{\text{Formula}}$. According to assumption 5.1, $\widetilde{\phi_2} = \phi_{2a}$ for some $\phi_{2a} \in \text{Formula}$. Using the strength rule of deterministic liftings it follows immediately that $\phi_{2a} \Rightarrow \phi_{2b}$.

□

Proof of Lemma A.2.
It is essential that FORMULA cannot express knowledge like

$$\texttt{x} \in \{1, 2, 3\} \tag{A.71}$$

since there is neither a logical disjunction (that would allow writing `(x = 1) ∨ (x = 2) ∨ (x = 3)`) nor an inequality operator `<=` (that would allow writing `(1 <= x) * (x <= 3)`) nor any other sufficiently expressive syntax.

Hence, this knowledge can only be approximated, e.g. as

$$\texttt{(x} \neq \texttt{0)} \land \texttt{(x} \neq \texttt{4)} \land \texttt{(x} \neq \texttt{5)}$$

Now, let $\overline{\phi}$ be the set of all approximation for $A.71$, i.e.

$$\overline{\phi} = \{\ \phi \in \text{FORMULA} \mid [\![\texttt{(x = 1)}]\!] \cup [\![\texttt{(x = 2)}]\!] \cup [\![\texttt{(x = 3)}]\!] \subseteq [\![\phi]\!]\ \}$$

Then there exists no least (most precise) element $\widetilde{\phi}$ that conservatively over-approximates $\overline{\phi}$. As a consequence $\alpha(\overline{\phi})$ cannot be defined.

Proof by contradiction:
Assume there exists a least element $\widetilde{\phi}$ that conservatively approximates $\overline{\phi}$, i.e. $\overline{\phi} \subseteq \gamma(\widetilde{\phi})$. Then $\widetilde{\phi}$ cannot contain an equality including `x`. As a result, $\widetilde{\phi}$ can only make statements about `x` using inequalities. Since $\widetilde{\phi}$ is finite, there must be an integer that has not yet been excluded from the set of possible values for `x`. Therefore, adding a corresponding inequality to $\widetilde{\phi}$ yields a gradual formula that is more precise than $\widetilde{\phi}$ and still conservatively approximates $\overline{\phi}$.
This contradicts the assumption that $\widetilde{\phi}$ is the least element with that property. □

Proofs of Lemma A.3, A.4, A.5, A.6, A.2. Omitted (proofs in [2]). □

Proof of Theorem A.10.
Omitted.

Intuition behind the normal form:
Recall that gradual formulas `?` $*$ ϕ_1 and `?` $*$ ϕ_2 are considered equal iff $\gamma(\texttt{?} * \phi_1) = \gamma(\texttt{?} * \phi_1)$. The normal form makes use of the fact that concretizations of partially unknown formulas contain only self-framed formulas (see section 4.2).

Lemma A.11 (Mentioning a Field Implies Access)**.**
For any formula ϕ mentioning field $x\,.\,f$:

$$\forall \phi' \in \gamma(\texttt{?} * \phi), \phi' \Rightarrow \texttt{acc}(x\,.\,f)$$

In other words: Merely mentioning a field will make sure that the concretization contains appropriate framing. We say access to a field is "restored" by the concretization. This is a helpful observation for justifying *removal* of accessibility predicates from the static part.

Example A.12 (Removal of Accessibility Predicate)**.**
Using lemma A.11 one can derive

$$\gamma(\texttt{? * acc(x.f) * (x.f = 3)}) = \gamma(\texttt{? * (x.f = 3)})$$

The accessibility predicate can be dropped since it is restored by concretization due to `(x.f = 3)` mentioning field `x.f`.

Note, however, that simply dropping access from the static part may not result in an equivalent gradual formula (like in example A.12) for two reasons:

1. No more mentions

Dropping $\texttt{acc}(x.f)$ might result in $x.f$ not being mentioned in the formula anymore, so there would be no more reason for the concretization to "restore" $\texttt{acc}(x.f)$ in all concretizations.

Example A.13.

$$\gamma(\texttt{? * acc(x.f) * (p = 3)}) \neq \gamma(\texttt{? * (p = 3)})$$

2. Aliasing

In general there are different ways in which access to multiple (syntactically different) fields can be restored.

Example A.14 (Ambiguous Framing)**.**
Dropping all accessibility predicates from

$$\texttt{acc(a.f) * acc(b.f) * (a.f = 3) * (b.f = x)}$$

results in

$$\texttt{(a.f = 3) * (b.f = x)}$$

However, $\gamma(\texttt{? * (a.f = 3) * (b.f = x)})$ contains

$$\texttt{acc(a.f) * (a = b) * (a.f = 3) * (a.f = x)}$$

whereas $\gamma(\texttt{? * acc(a.f) * acc(b.f) * (a.f = 3) * (b.f = x)})$ does not, i.e.

$$\gamma(\texttt{? * acc(a.f) * acc(b.f) * (a.f = 3) * (b.f = x)}) \neq \gamma(\texttt{? * (a.f = 3) * (b.f = x)})$$

Fortunately, we can prevent both problems from occurring by carefully preparing the static part before dropping all access, resulting in the following two-step approach:

1. Enhancement

Enrich the static part to counteract above problems, i.e. to enforce that access is restored exactly the right way. This is achieved by appending certain implications of the access-terms as conjunctive terms:

$$\texttt{acc}(x.f) \implies (x.f = x.f)$$

Access to a field implicitly guarantees that it evaluates to some value (see formula semantics in figure 4.7). The formula $(x.f = x.f)$ has the same effect, i.e. it can be appended as a conjunctive term without altering the formula semantics, but making sure that information is not lost when dropping the accessibility predicates.

Example A.15 (Example A.13 revised)**.**

$$\begin{aligned} & \gamma(? * \texttt{acc(x.f)} * \texttt{(p = 3)}) \\ = \; & \gamma(? * \texttt{acc(x.f)} * \texttt{(p = 3)} * \texttt{(x.f = x.f)}) \\ = \; & \gamma(? * \texttt{(p = 3)} * \texttt{(x.f = x.f)}) \end{aligned}$$

$\texttt{acc}(x.f) * \texttt{acc}(y.f) \implies (x \neq y)$

Having access to the same field of different expressions actively prevents those expressions to alias due to the formula semantics. A corresponding inequality has the same effect, i.e. it can be appended as a conjunctive term without altering the formula semantics, but making sure that information is not lost when dropping the accessibility predicates.

Example A.16 (Example A.14 revised)**.**

$$\begin{aligned} & \gamma(? * \texttt{acc(a.f)} * \texttt{acc(b.f)} * \texttt{(a.f = 3)} * \texttt{(b.f = x)}) \\ = \; & \gamma(? * \texttt{acc(a.f)} * \texttt{acc(b.f)} * \texttt{(a.f = 3)} * \texttt{(b.f = x)} * \texttt{(a} \neq \texttt{b)}) \\ = \; & \gamma(? * \texttt{(a.f = 3)} * \texttt{(b.f = x)} * \texttt{(a} \neq \texttt{b)}) \end{aligned}$$

As illustrated above, the normal form $|\cdot|$ will append corresponding conjunctive terms in every possible way, i.e. accounting for all (pairs of) access-terms. It is worth noting that this process preserves equality of the formula as only terms are added that were implied by the original formula, anyway.

2. Delinearization

All accessibility predicates are dropped. Due to above measures, they are restored by concretization correctly.

□

Proof of Lemma A.11.
Note that a formula mentioning the field $x.f$ cannot be implied by a formula that does not mention that field. This is a direct consequence of the formula semantics (see figure 4.7).

E.g. `true` $\Rightarrow$ `(x.f = x.f)` is not true, since `(x.f = x.f)` actually ensures that `x.f` evaluates (to any value), whereas `true` does not (there, `x` may not be defined or field `f` may not exist on the value `x` evaluates to).

Due to self-framing, all formulas in $\gamma(? * \phi)$ must therefore contain `acc(x.f)`. □

Bibliography

[1] Stephan Arlt, Cindy Rubio-González, Philipp Rümmer, Martin Schäf, and Natarajan Shankar. The gradual verifier. In *NASA Formal Methods Symposium*, pages 313–327. Springer, 2014.

[2] Johannes Bader. GradVer, GitHub repository. `https://github.com/olydis/GradVer`, 2016.

[3] Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *ACM SIGPLAN Notices*, volume 49, pages 283–295. ACM, 2014.

[4] Frank Piessens Wolfram Schulte Bart Jacobs, Jan Smans. A statically verifiable programming model for concurrent object-oriented programs. In *ICFEM*, volume 4260, pages 420–439. Springer, January 2006.

[5] Yannis Bres, Bernard Paul Serpette, and Manuel Serrano. Compiling scheme programs to .net common intermediate language. *.NET Technologies 2004*, page 25, 2004.

[6] Yoonsik Cheon and Gary T Leavens. A runtime assertion checker for the java modeling language (jml). 2002.

[7] M. Christakis, P. Müller, and V. Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In L. K. Dillon, W. Visser, and L. Williams, editors, *International Conference on Software Engineering (ICSE)*, pages 144–155. ACM, 2016.

[8] David Crocker. Safe object-oriented software: the verified design-by-contract paradigm. In *Practical Elements of Safety*, pages 19–41. Springer, 2004.

[9] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. ACM.

[10] Ronald Garcia and Eric Tanter. Deriving a simple gradual security language. *arXiv preprint arXiv:1511.01399*, 2015.

[11] Kathryn E Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. *ACM SIGPLAN Notices*, 40(10):231–245, 2005.

[12] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[13] Bart Jacobs and Erik Poll. A logic for the java modeling language jml. In *International Conference on Fundamental Approaches to Software Engineering*, pages 284–299. Springer, 2001.

[14] Nico Lehmann and Éric Tanter. Gradual refinement types. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, to appear*, POPL '17, 2017.

[15] 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.

[16] K Rustan M Leino, Greg Nelson, and James B Saxe. Esc/java user's manual. *ESC*, 2000:002, 2000.

[17] Francesco Logozzo Manuel Fahndrich, Mike Barnett. Embedded contract languages. In *ACM SAC - OOPS*. Association for Computing Machinery, Inc., March 2010.

[18] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. Citeseer, 2004.

[19] Bertrand Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[20] Bertrand Meyer. *Design by contract*. Prentice Hall, 2002.

[21] Wolfram Schulte Mike Barnett, Rustan Leino. The spec# programming system: An overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362, pages 49–69. Springer, January 2005.

[22] Antoine Miné. *Weakly relational numerical abstract domains*. PhD thesis, Ecole Polytechnique X, 2004.

[23] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[24] Greg Nelson. Extended static checking for java. In *International Conference on Mathematics of Program Construction*, pages 1–1. Springer, 2004.

[25] Matthew J Parkinson and Alexander J Summers. The relationship between separation logic and implicit dynamic frames. In *European Symposium on Programming*, pages 439–458. Springer, 2011.

[26] John C Reynolds. Separation logic: A logic for shared mutable data structures. In

Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on, pages 55–74. IEEE, 2002.

[27] Amritam Sarcar and Yoonsik Cheon. A new eclipse-based jml compiler built using ast merging. In *Software Engineering (WCSE), 2010 Second World Congress on*, volume 2, pages 287–292. IEEE, 2010.

[28] Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007.

[29] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[30] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[31] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming*, pages 148–172. Springer, 2009.

[32] 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.

[33] Matías Toro and Eric Tanter. Customizable gradual polymorphic effects for scala. In *ACM SIGPLAN Notices*, volume 50, pages 935–953. ACM, 2015.

[34] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming*, pages 459–483. Springer, 2011.