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)

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

Duration: 2016-05-10 – 2016-??-??

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-??-??

. . . . . .

(**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 race-free reasoning.

Acknowledgments

I wish to thank my advisors Jonathan Aldrich and Éric 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.

TODO: Snelting, funding, this and that

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 given assertions.

Drawbacks

The syntax available for static verification is naturally limited by the underlying formal logic. Complex properties 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 annotate the API afterwards, lacking the source code the verifier is unable to prove those annotations. In case the annotation are wrong this results in inconsistent proves.

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 above drawbacks. The resulting system provides a continuum 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.

Our approach is based on recent formalizations regarding gradual typing, using the concept of abstract interpretation to define a gradual system in terms of a static one (this process is called "gradualization"). Gradual typing emerged from very similar drawbacks of static and dynamic type systems. 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 introducing 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 manual or template for designing gradual verification systems. We follow that manual in form of case study in chapter 4, applying the approach to a statically verified language that uses implicit dynamic in order to enable safe static reasoning about shared mutable state.

1.1. Motivational Examples

1.1.1. Argument Validation

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

```
// spec: no resource leaks
boolean hasLegalDriver(Car c)
{
    // business logic:
    resAllocate();
    boolean result = c.driver.age >= 18;
    resFree();
    return result;
}
```

A call to `hasLegalDriver` fails if `c` or `c.driver` evaluate to `null`. Note that, although the Java runtime has defined behavior in those cases (throwing an exception), we have created a resource leak. To prevent this from happening, arguments have to be validated before entering the business logic.

```
boolean hasLegalDriver(Car c)
{
    if (!(c != null))
        throw new IllegalArgException("expected c != null");
    if (!(c.driver != null))
        throw new IllegalArgException("expected c.driver != null");
```

```
    // business logic (requires 'c.driver.age' to evaluate)
}
```

Note that these runtime checks dynamically verify a method contract, having `c != null && c.driver != null` 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 which might not be acceptable in all scenarios.

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

```
boolean hasLegalDriver(Car c)
{
    assert c != null;
    assert c.driver != null;

    // business logic (requires 'c.driver.age' to evaluate)
}
```

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

```
//@ requires c != null && c.driver != null;
boolean hasLegalDriver(Car c)
{
    // business logic (requires 'c.driver.age' to evaluate)
}
```

Using JML4c (see [19]) or similar tools, this annotations is compiled back into equivalent runtime assertions. However, the declarative approach also enables the use of static verification tools like ESC/Java (see [12]) 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.

Gradual verification would pursue a combined approach without drawbacks: Static verification is used where possible, dynamic verification where needed. Where static verification proves a violation of the method contract, an error is found at compile time (which would have required tests with full code coverage to detect otherwise). Where static verification proves compliance with the method contract, no runtime checks are emitted in order to enforce the contract. Only call sites which are provable neither to violate nor to adhere with the contract will be enhanced with runtime assertions.

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 and Spec#). We assume that this language is statically verified, i.e. static verification is part of the compilation.

The example shows the limitations of static verification using the Collatz sequence as an algorithm too complex to describe 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 given number of times, starting at given value. We assume that the only provable contract is that positive start value results in 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 sequence after 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 result, but no specific range of values. Again, we can resort to dynamic methods to aid verification:

```
...
{
    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 verifier. Furthermore, the solution is in an unintuitive location: The problem is not caused by `myRandom`, yet it is solved there. The actual problem is that the postcondition of `collatzIterations` is too weak, causing the verifier to fail deducing our

knowledge.

Gradual verification allows enhancing the postcondition with "unknown" knowledge that can be interpreted by the verifier as an arbitrary plausible static formula. At the same time appropriate runtime checks have to be injected to guarantee that this treatment was 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 <= start && result <= 4
    //  from   1 <= start && ?
    // (adding runtime check)

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

Note the `?` in the postcondition of `collatzIterations`.

2. Background

Design-by-Contract, a term coined by Bertrand Meyer [14], 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, providing compiler support for generating runtime checks required for dynamic verification (also called runtime verification). Combining design-by-contract with static verification techniques lead to the concept of "verified design-by-contract" [6].

Similar developments took place regarding Java and JML specifications. Static verification using theorem provers was investigated by Jacobs and Poll [10] and is implemented as part of ESC/Java [16]. Turning JML specifications into runtime assertion checks (RAC) to drive dynamic verification was described by Cheon and Leavens [4] and lead up to the development of JML4c [19].

A more recent programming language with built-in support to express specification, coming with both static and dynamic verification tools is Spec# [15]. 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 [3]. The concepts found their way to main stream programming in the form of "Code Contracts" [13], 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 led to a recent trend of using both approaches at the same time (as observable in above programming languages). Due to its rigidity, static verification is treated more as a best effort service, meant to detect big inconsistencies or contract violations (the more coverage, the better). 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 [5] 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 smart 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 amount 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 typing for functional programming languages was formalized by Siek and Taha [20]. They describe a λ-calculus with optional type annotations, which is sound w.r.t. simply-typed λ-calculus for fully annotated terms. Static and dynamic type type checking is seamlessly combined by automatically inserting runtime checks (casts) where necessary.

This work has later been extended in a variety of ways. Wolff et al. introduced "gradual typestate" [25], circumventing the rigidity of static typestate checking. Schwerter, Garcia and Tanter developed a theory of gradual effect systems [2], 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 [24].

Siek et al. recently formalized refined criteria for gradual typing, called "gradual guarantee" [21]. The gradual guarantee states that well typed programs will stay well typed when removing type annotations (static part of the guarantee). It furthermore states that well typed programs evaluating to a value will evaluate to the same value when removing type annotations (dynamic part of the guarantee).

With "Abstracting Gradual Typing" (AGT) [7] 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 a to security-typed language, yielding a gradual security language [8], which in contrast to prior work does not require explicit security casts.

2.2. Hoare Logic

We use Hoare logic [9] as the formal logic used for static verification. We assume that source code annotations can be translated into Hoare logic.

Example 2.1 (Hoare Logic for Contract Verification)**.**
Consider a programming language with 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 can 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 would 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 [18] 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.

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

The use of 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.

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) [22] 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 [17] worked out the formal relationship between separation logic and IDF.

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, it has to be ensured 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-framing whereas the sub-formula `(p1.age = 20)` would not be. It is essential that access cannot be duplicated and thus also not be shared between threads, allowing race-free reasoning about concurrent programs.

Implicit dynamic frames was implemented as part of the Chalice verifier [11]. 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.

The static semantics of our example language in chapter 4 is based on the Hoare logic for Chalice given by Summers and Drossopoulou [23].

3. Gradualization of a Statically Verified Language

As illustrated in section 1.1 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 a gradually verified language "**GVL**" starting with a generic statically verified language "**SVL**". An informal description of how to tackle the opposite direction can be found in section 5.1.

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 3.3 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 3.4 and point out the associated challenges.

With the necessary tools for gradualization available, we apply them to the static semantics of **SVL** in section 3.5. Finally, we develop gradual dynamic semantics in section 3.6 and show how gradual soundness is achievable.

3.1. A Generic Statically Verified Language SVL

While aiming to give a general procedure for deriving 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 statically verified programming languages satisfy the following assumptions and thus qualify as starting point for our procedure.

Assumptions about **SVL**:

Syntax

We assume the existence of the following two syntactic categories:

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

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)**.**

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

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...

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 can be updated. 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 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.

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

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

This notation is useful as it abstracts from the implementation details of the small-step semantics. We want to be able to reason about small-step derivations that correspond to executing certain statements without worrying about the exact number of steps taken.

Note that the judgment $\pi_1 \longrightarrow^* \pi_2$ is also independent from the number of steps taken, yet it does not encode what is supposed to happen during those steps:

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

or

$$\pi_2 = \pi_1$$

would be a valid instantiations.

We assume that there is a designated non-empty set $\textsc{ProgramStateFin} \subseteq \textsc{ProgramState}$ of states indicating regular termination of the program. W.l.o.g. we assume $\forall \pi \in \textsc{ProgramStateFin}.\ \pi \not\rightarrow$, i.e. final states are stuck.

Formula Semantics

While types restrict which values 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 in the program state at the call site.

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*

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

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 $\textsc{Formula}$*.*

We assume that there is a largest element `true` $\in \textsc{Formula}$ that describes every program state. Note that the presence of an unsatisfiable formula (as invariant, pre-/postcondition, assertion, ...) in a sound verification system implies that the corresponding source code location is unreachable: Preservation guarantees that any

reachable program state satisfies potentially annotated formulas, trivially ensuring that the formula is satisfiable.

This property is true regardless of whether **SVL** forbids usage of unsatisfiable formulas entirely or whether it only fails when trying to use the corresponding code (which would involve proving that a satisfiable formula implies an unsatisfiable one). Therefore we will often restrict our reasoning on the satisfiable formulas SatFormula, without explicitly stating that the presence of an unsatisfiable formula would result in failure.

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

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

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

- It is non-empty. This ensures that every program state is describable by at least one formula. This is true as we demanded a formula `true` that describes every program state.
- If $\pi \vDash \phi_a$ and $\pi \vDash \phi_b$, then there exists $\phi \in \textsc{Formula}$ with $\pi \vDash \phi \wedge \phi \Rightarrow \phi_a \wedge \phi \Rightarrow \phi_b$. 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.
- If $\pi \vDash \phi_a$ and $\phi_a \Rightarrow \phi_b$ then $\pi \vDash \phi_b$. This is true by definition of $\cdot \Rightarrow \cdot$.

Static Semantics

We assume that there is a Hoare logic (HL)

$$\vdash \{\cdot\}\ \cdot\ \{\cdot\} \quad \subseteq \quad \textsc{SatFormula} \times \textsc{Stmt} \times \textsc{SatFormula}$$

describing which programs (together with pre- and postconditions about the program state) are accepted. While the Hoare logic might be defined for arbitrary formulas in practice, we only ever reason about it in presence of satisfiable formulas, hence the "restricted domain"???.

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.

We assume that

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

is derivable from given Hoare rules.

We further assume that this predicate is monotonic in the precondition w.r.t. implication:

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

Intuitively, this means that more knowledge about the initial program state can not 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.

$$\frac{???}{???}\ \textsc{Progress}$$

$$\frac{\vdash \{\phi_1\}\ s\ \{\phi_2\}}{\vDash \{\phi_1\}\ s\ \{\phi_2\}}\ \textsc{Preservation}$$

3.2. Gradual Formulas

The fundamental concept of gradual verification is the introduction of a wildcard formula `?` into the formula syntax. The first difference of **GVL** in comparison to **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 presence of the wildcard, should it be plausible that there exists a static formula that would make a static verifier succeed. Example:

```
int increment(int i)
    requires ?;
    ensures  result == 3;
{
    return i + 1;
}
```

The gradual verifier is expected to successfully verify this method contract since there exists a static formula (`i == 2`), 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 work:

```
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 concept 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 intuition is formalized in section 3.3.2.)

Definition 3.10 (Concretization)**.**
Let $\gamma : \widetilde{\text{Formula}} \to \mathcal{P}(\text{Formula})$ *be defined as*

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

Static formula 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. Dedicated Wildcard Formula

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 most gradually typed languages are realized (e.g. `dynamic`-type in C# 4.0 and upward).

The approach is limited since programmers cannot express any additional static knowledge they might have in the presence of `?`. For example, a programmer might resort to using the wildcard lacking some knowledge about variable `x` (or being unable to express it), whereas he could give a static formula for `y`, say `(y = 3)`. Yet, there is no way to express this information as soon as the wildcard is used.

3.2.2. Wildcard with Upper Bound

To allow combining wildcards with static knowledge, we might view ?merely as an unknown conjunctive term within a formula:

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

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

We expect $\phi \ \wedge \ \texttt{?}$ to be a placeholder for a formula that implies ϕ.

Definition 3.11 (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 the previous one.

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 3.12 (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. Gradual Statements

Formulas play a role for some statements, extending their syntax may thus also affect the syntax of statements.

A common example are assertion statements `assert` ϕ. Having a gradual formula syntax available does not necessary mean that all statements have to adopt it. In case of the assertion statement there might be little benefit in allowing gradual formulas.

A more complex example affected by gradualization of formulas is a call statement m`();` in presence of method contracts. Although not directly visible, this statement's semantics

(static and dynamic) is affected by the contract of m, consisting of pre- and postcondition. One can think of m as a reference to some method definition including method contract. Note that in practice such method definitions usually reside in a "program context" that is then passed to static and dynamic semantics. 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 φ_{m_pre};
// body of m
assume φ_{m_post};
```

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

```
assert ~φ_{m_pre}~;
// body of m
assume ~φ_{m_post}~;
```

In general, 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 3.13 (Concretization of Gradual Statements)**.** *Let* $\gamma_s : \widetilde{\text{STMT}} \to \mathcal{P}(\text{STMT})$ *be defined as*

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

Definition 3.14 (Precision of Gradual Statement)**.** *Let* $\sqsubseteq_s \subseteq \widetilde{\text{STMT}} \times \widetilde{\text{STMT}}$ *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**.

3.2.5. Gradual Program State

Recall that program state has a notion of remaining work, see section 3.1 for examples. As the set of possible statements has been augmented from STMT to $\widetilde{\text{STMT}}$, the notion of remaining work 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:

$$\text{ProgramState} \;=\; (\text{Var} \rightharpoonup \mathbb{Z}) \;\times\; \text{Stmt}$$

is extended to

$$\widetilde{\text{ProgramState}} \;=\; (\text{Var} \rightharpoonup \mathbb{Z}) \;\times\; \widetilde{\text{Stmt}}$$

We give meaning to gradual program states using concretization.

Definition 3.15 (Concretization of Gradual Program States)**.** *Let* $\gamma_\pi : \widetilde{\text{ProgramState}} \to \mathcal{P}(\text{ProgramState})$ *be defined as*

$\gamma_\pi(\widetilde{\pi}) = \{\, \pi \in \text{ProgramState} \mid \pi$ *is* $\widetilde{\pi}$ *with all continuations replaced by a concretization* $\}$

Definition 3.16 (Precision of Gradual Program States)**.** *Let* $\sqsubseteq_\pi \subseteq \widetilde{\text{ProgramState}} \times \widetilde{\text{ProgramState}}$ *be a predicate defined as*

$$\widetilde{\pi_a} \sqsubseteq_\pi \widetilde{\pi_b} \quad\iff\quad \gamma_\pi(\widetilde{\pi_a}) \subseteq \gamma_\pi(\widetilde{\pi_b})$$

We demand that formula semantics are not affected by this extension, which is trivially the case if evaluation does not depend on the continuation in the first place:

$$\forall \phi \in \text{Formula}, \widetilde{\pi} \in \widetilde{\text{ProgramState}}, \pi \in \gamma_\pi(\widetilde{\pi}).\;\; \widetilde{\pi} \vDash \phi \iff \pi \vDash \phi$$

3.3. Lifting Predicates and Functions

The Hoare logic of **SVL** is a ternary predicate $\vdash \{\cdot\}\ \cdot\ \{\cdot\} \subseteq \text{Formula} \times \text{Stmt} \times \text{Formula}$. Since **GVL** contains gradual formulas and gradual statements, the gradualized Hoare logic is expected to have signature $\widetilde{\vdash} \{\cdot\}\ \cdot\ \{\cdot\} \subseteq \widetilde{\text{Formula}} \times \widetilde{\text{Stmt}} \times \widetilde{\text{Formula}}$. Similarly, the gradualized small-step semantics is expected to have signature $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot : \widetilde{\text{ProgramState}} \rightharpoonup \widetilde{\text{ProgramState}}$ instead of $\cdot \longrightarrow \cdot : \text{ProgramState} \rightharpoonup \text{ProgramState}$. Usually semantics are 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 "lifting", which formalizes this adaptation of predicates and functions.

Definition 3.17 (Gradual Lifting)**.** *The procedure of extending an existing predicate/-function in order to deal with gradual formulas. The resulting predicate/function has the same signature as the original one, with occurrences of* Formula, Stmt *and* ProgramState *replaced by* $\widetilde{\text{Formula}}$, $\widetilde{\text{Stmt}}$, $\widetilde{\text{ProgramState}}$.

Our rules for 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, whereas they are directly applicable to other gradualized sets.

3.3.1. 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. [21].

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

Definition 3.18 (Gradual Guarantee (Static Semantics))**.** *Correct programs remain correct when reducing precision of any formula.*

Definition 3.19 (Gradual Guarantee (Dynamic Semantics))**.** *Correct programs with certain observational behavior (termination, values of variables, output, etc.) will have the same observational behavior after reducing precision of any formula.*

3.3.2. 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:

$$\frac{P(\phi_1, \phi_2)}{\widetilde{P}(\phi_1, \phi_2)}\ \textsc{GPredIntro}$$

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

$$\frac{\widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2}) \qquad \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_1'} \qquad \widetilde{\phi_2} \sqsubseteq \widetilde{\phi_2'}}{\widetilde{P}(\widetilde{\phi_1'}, \widetilde{\phi_2'})}\ \textsc{GPredMon}$$

Definition 3.20 (Soundness of Predicate Lifting)**.** *A lifted predicate is* ***sound/valid*** *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 3.21 (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 [7].

Lemma 3.22 (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 back to plausible static formulas. We noticed early that this definition is too strong for gradual verification rules in general, due to the complexity of verification rules compared to typing rules. In chapter 3.5 we will see examples of gradual predicates which are not optimal and would thus not fit into AGT's model of consistent lifting.

Realizing that consistent lifting is actually not necessary for ending up with a sound gradual verification system, we took a different 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 soundness. The optional notion of optimality bridges the gap between both approaches.

3.3.2.1. Examples

Lemma 3.23 (Optimal Lifting of Implication)**.**
Let $\widetilde{\textsc{Formula}}$ *be extended using the "dedicated wildcard" approach (see section 3.2.1). Let* $\cdot \mathrel{\widetilde{\Rightarrow}} \cdot \subseteq \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}}$ *be defined inductively as*

$$\frac{\phi_1 \Rightarrow \phi_2}{\phi_1 \mathrel{\widetilde{\Rightarrow}} \phi_2}\ \textsc{GImplStatic}$$

$$\frac{\phi \in \textsc{SatFormula}}{\texttt{?} \mathrel{\widetilde{\Rightarrow}} \phi}\ \textsc{GImplGrad1}$$

$$\frac{}{\widetilde{\phi} \mathrel{\widetilde{\Rightarrow}} \texttt{?}}\ \textsc{GImplGrad2}$$

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

Proof. Goal:

$$\widetilde{\phi_1} \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi_2} \iff \exists \phi_1 \in \gamma(\widetilde{\phi_1}), \phi_2 \in \gamma(\widetilde{\phi_2}).\ \phi_1 \Rightarrow \phi_2$$

$\Longrightarrow$: Case GImplStatic:

$$\begin{aligned} & \phi_1 \mathrel{\widetilde{\Rightarrow}} \phi_2 \\ \Longrightarrow\ & \phi_1 \Rightarrow \phi_2 \\ \Longrightarrow\ & (\exists \phi_1' \in \gamma(\phi_1), \phi_2' \in \gamma(\phi_2).\ \phi_1' \Rightarrow \phi_2') \end{aligned}$$

Case GImplGrad1:

$$\begin{aligned} & \texttt{?} \mathrel{\widetilde{\Rightarrow}} \phi \\ \Longrightarrow\ & \phi \in \textsc{SatFormula} \\ \Longrightarrow\ & (\exists \phi_1 \in \textsc{SatFormula}.\ \phi_1 \Rightarrow \phi) \\ \Longrightarrow\ & (\exists \phi_1 \in \gamma(\texttt{?}), \phi_2 \in \gamma(\phi).\ \phi_1 \Rightarrow \phi_2) \end{aligned}$$

Case GImplGrad2:

$$\begin{aligned} & \gamma(\widetilde{\phi}) \neq \emptyset\ \wedge\ \texttt{true} \in \gamma(\texttt{?}) \\ \Longrightarrow\ & (\exists \phi_1 \in \gamma(\widetilde{\phi}).\ \phi_1 \Rightarrow \texttt{true})\ \wedge\ \texttt{true} \in \gamma(\texttt{?}) \\ \Longrightarrow\ & (\exists \phi_1 \in \gamma(\widetilde{\phi}), \phi_2 \in \gamma(\texttt{?}).\ \phi_1 \Rightarrow \phi_2) \end{aligned}$$

$\Longleftarrow$:

$$\phi_1 \in \gamma(\widetilde{\phi_1}) \wedge \phi_2 \in \gamma(\widetilde{\phi_2}) \wedge \phi_1 \Rightarrow \phi_2$$

Case $\widetilde{\phi_2} = \texttt{?}$: Apply GImplGrad2. Case $\widetilde{\phi_2} = \phi_2 \wedge \widetilde{\phi_1} = \texttt{?}$:

$$\begin{aligned} & \phi_1 \in \gamma(\texttt{?}) \wedge \phi_1 \Rightarrow \phi_2 \\ \Longrightarrow\ & \phi_1 \in \textsc{SatFormula} \wedge \phi_1 \Rightarrow \phi_2 \\ \Longrightarrow\ & \phi_2 \in \textsc{SatFormula} \end{aligned}$$

Apply GImplGrad1. Case $\widetilde{\phi_2} = \phi_2 \wedge \widetilde{\phi_1} = \phi_1$: Apply GImplStatic. □

Lemma 3.24 (Optimal Lifting of Evaluation)**.**
Let $\widetilde{\textsc{Formula}}$ *be extended using the "dedicated wildcard" approach (see section 3.2.1). Let* $\cdot \mathrel{\widetilde{\vDash}} \cdot \subseteq \textsc{ProgramState} \times \widetilde{\textsc{Formula}}$ *be defined inductively as*

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

$$\frac{}{\pi \mathrel{\widetilde{\models}} \texttt{?}}\ \textsc{GEvalGrad}$$

Then $\cdot \mathrel{\widetilde{\models}} \cdot$ *is a consistent lifting of* $\cdot \models \cdot$.

Proof. Goal:

$$\pi \mathrel{\widetilde{\models}} \widetilde{\phi} \iff \exists \phi \in \gamma(\widetilde{\phi}).\ \pi \models \phi$$

$\Longrightarrow$: Case GEvalStatic:

$$\begin{aligned} & \pi \mathrel{\widetilde{\models}} \phi \\ \Longrightarrow\ & \pi \models \phi \\ \Longrightarrow\ & (\exists \phi' \in \gamma(\phi).\ \pi \models \phi') \end{aligned}$$

Case GEvalGrad:

$$\begin{aligned} & \pi \models \texttt{true} \ \wedge\ \texttt{true} \in \gamma(\texttt{?}) \\ \Longrightarrow\ & (\exists \phi \in \gamma(\texttt{?}).\ \pi \models \phi) \end{aligned}$$

$\Longleftarrow$:

$$\phi \in \gamma(\widetilde{\phi}) \wedge \pi \models \phi$$

Case $\widetilde{\phi} = \texttt{?}$: Apply GEvalGrad. Case $\widetilde{\phi} = \phi$: Apply GEvalStatic. □

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 3.2.5). We define denotational semantics of gradual formulas analogous to the non-gradual variant (see definition 3.4):

Definition 3.25 (Denotational Formula Semantics $[\![\cdot]\!]$ of Gradual Formulas)**.**
Let $[\![\cdot]\!] : \widetilde{\text{Formula}} \to \mathcal{P}^{\text{ProgramState}}$ *be defined as*

$$[\![\widetilde{\phi}]\!] \stackrel{\text{def}}{=} \{\ \pi \in \text{ProgramState} \mid \pi \mathrel{\widetilde{\models}} \widetilde{\phi}\ \}$$

Lemma 3.26 (Sound Lifting of Composite Predicate)**.**
Let $P, Q \subseteq \text{Formula} \times \text{Formula}$ *be arbitrary binary predicates. Let* $(P \circ Q) \subseteq \text{Formula} \times \text{Formula}$ *be defined as*

$$(P \circ Q)(\phi_1, \phi_3) \iff \exists \phi_2 \in \text{Formula}.\ P(\phi_1, \phi_2) \wedge Q(\phi_2, \phi_3)$$

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

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

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* $\widetilde{(P \circ Q)}$.

Proof. Introduction

$$\begin{array}{ll} & (P \circ Q)(\phi_1, \phi_3) \\ \overset{Definition}{\Longrightarrow} & (\exists \phi_2 \in \textsc{Formula}.\ P(\phi_1, \phi_2) \wedge Q(\phi_2, \phi_3)) \\ \overset{Introduction}{\Longrightarrow} & (\exists \phi_2 \in \textsc{Formula}.\ \widetilde{P}(\phi_1, \phi_2) \wedge \widetilde{Q}(\phi_2, \phi_3)) \\ \overset{Definition}{\Longrightarrow} & (\widetilde{P} \circ \widetilde{Q})(\phi_1, \phi_3) \\ \overset{Definition}{\Longrightarrow} & \widetilde{(P \circ Q)}(\phi_1, \phi_3) \end{array}$$

Monotonicity

$$\begin{array}{ll} & \widetilde{(P \circ Q)}(\widetilde{\phi_1}, \widetilde{\phi_3}) \ \wedge\ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_1'} \ \wedge\ \widetilde{\phi_3} \sqsubseteq \widetilde{\phi_3'} \\ \overset{Definition}{\Longrightarrow} & (\widetilde{P} \circ \widetilde{Q})(\widetilde{\phi_1}, \widetilde{\phi_3}) \ \wedge\ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_1'} \ \wedge\ \widetilde{\phi_3} \sqsubseteq \widetilde{\phi_3'} \\ \overset{Definition}{\Longrightarrow} & (\exists \widetilde{\phi_2} \in \widetilde{\textsc{Formula}}.\ \widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2}) \wedge \widetilde{Q}(\widetilde{\phi_2}, \widetilde{\phi_3})) \ \wedge\ \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_1'} \ \wedge\ \widetilde{\phi_3} \sqsubseteq \widetilde{\phi_3'} \\ \overset{Monotonicity}{\Longrightarrow} & (\exists \widetilde{\phi_2} \in \widetilde{\textsc{Formula}}.\ \widetilde{P}(\widetilde{\phi_1'}, \widetilde{\phi_2}) \wedge \widetilde{Q}(\widetilde{\phi_2}, \widetilde{\phi_3'})) \\ \overset{Definition}{\Longrightarrow} & (\widetilde{P} \circ \widetilde{Q})(\widetilde{\phi_1'}, \widetilde{\phi_3'}) \\ \overset{Definition}{\Longrightarrow} & \widetilde{(P \circ Q)}(\widetilde{\phi_1'}, \widetilde{\phi_3'}) \end{array}$$

□

Lemma 3.27 (Sound Lifting of Conjunctive Predicate)**.**
Let $P, Q \subseteq$ Formula be arbitrary binary predicates. Let $(P \wedge Q) \subseteq$ Formula be defined as

$$(P \wedge Q)(\phi) \quad \overset{\mathsf{def}}{\Longleftrightarrow} \quad P(\phi) \wedge Q(\phi)$$

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

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

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

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

Proof. 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

$$\begin{array}{ll} & (\widetilde{P} \wedge \widetilde{Q})(\widetilde{\phi}) \ \wedge \ \widetilde{\phi} \sqsubseteq \widetilde{\phi'} \\ \overset{Definition}{\Longrightarrow} & \widetilde{P}(\widetilde{\phi}) \ \wedge \ \widetilde{Q}(\widetilde{\phi}) \ \wedge \ \widetilde{\phi} \sqsubseteq \widetilde{\phi'} \\ \overset{Monotonicity}{\Longrightarrow} & \widetilde{P}(\widetilde{\phi'}) \ \wedge \ \widetilde{Q}(\widetilde{\phi'}) \\ \overset{Definition}{\Longrightarrow} & (\widetilde{P} \wedge \widetilde{Q})(\widetilde{\phi'}) \\ \overset{Definition}{\Longrightarrow} & (\widetilde{P \wedge Q})(\widetilde{\phi'}) \end{array}$$

□

Lemma 3.28 (Optimal Lifting of Disjunctive Predicate)**.**
Let $P, Q \subseteq$ Formula be arbitrary binary predicates. Let $(P \vee Q) \subseteq$ Formula be defined as

$$(P \vee Q)(\phi) \quad \overset{\mathsf{def}}{\Longleftrightarrow} \quad P(\phi) \vee Q(\phi)$$

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

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

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 $\widetilde{Q}$ does imply optimality of $(\widetilde{P \vee Q})$.

Proof.

$$\begin{array}{ll} & (\widetilde{P \vee Q})(\widetilde{\phi}) \\ \overset{Definition}{\Longleftrightarrow} & (\widetilde{P} \vee \widetilde{Q})(\widetilde{\phi}) \\ \overset{Definition}{\Longleftrightarrow} & \widetilde{P}(\widetilde{\phi}) \vee \widetilde{Q}(\widetilde{\phi}) \\ \overset{AGTDef.}{\Longleftrightarrow} & (\exists \phi \in \gamma(\widetilde{\phi}).\ P(\phi)) \vee (\exists \phi \in \gamma(\widetilde{\phi}).\ Q(\phi)) \\ \Longleftrightarrow & (\exists \phi \in \gamma(\widetilde{\phi}).\ P(\phi) \ \vee \ Q(\phi)) \\ \overset{Definition}{\Longleftrightarrow} & (\exists \phi \in \gamma(\widetilde{\phi}).\ (P \vee Q)(\phi)) \end{array}$$

□

3.3.3. Lifting Functions

In this section, we assume that we are dealing with a total function f : Formula $\to$ Formula. The concepts are directly applicable to functions with higher arity.

Introduction

By making sure to comply with the gradual guarantee, we made design a gradual verification system immune to reduction of precision at any stage. 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 3.29 (Sound Function Lifting)**.** *A lifted function is **sound/valid** 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 3.30 (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, definition of optimal function lifting coincides with the definition of "consistent function lifting" given by AGT.

Lemma 3.31 (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.
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.

Proof. Helping

$$\alpha(\overline{f}(\gamma(\phi))) = f(\phi)$$

Proof $\alpha(\overline{f}(\gamma(\phi)))$ defined, since $\{\overline{f}\}$-partial Galois connection, i.e.

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

Then Rule 1

$$\{\ f(\phi)\ \} \subseteq \gamma(\widetilde{\phi})$$

Rule 2

$$\begin{aligned} & \{\ f(\phi)\ \} \subseteq \gamma(f(\phi)) \\ \Longrightarrow\ & \widetilde{\phi} \sqsubseteq f(\phi) \end{aligned}$$

Combining:

$$\begin{aligned} & \{\ f(\phi)\ \} \subseteq \gamma(\widetilde{\phi}) \subseteq \gamma(f(\phi)) \\ \Longrightarrow\ & \gamma(\widetilde{\phi}) = \{\ f(\phi)\ \} \\ \Longrightarrow\ & \widetilde{\phi} = f(\phi) \end{aligned}$$

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

$$\begin{aligned} & \overline{f}(\gamma(\widetilde{\phi})) \subseteq \gamma(\widetilde{f}'(\widetilde{\phi})) \\ \Longrightarrow\ & \alpha(\overline{f}(\gamma(\widetilde{\phi}))) \sqsubseteq \widetilde{f}'(\widetilde{\phi}) \\ \Longrightarrow\ & \widetilde{f}(\widetilde{\phi}) \sqsubseteq \widetilde{f}'(\widetilde{\phi}) \end{aligned}$$

Contradiction. □

3.3.3.1. Examples

Lemma 3.32 (Optimal Lifting of And)**.**
Let $\widetilde{\text{Formula}}$ *be extended using the "wildcard with upper bound" approach (see section 3.2.2). We assume that* $\vee$ *is part of the formula syntax such that* $[\![\phi_1 \vee \phi_2]\!] = [\![\phi_1]\!] \cup [\![\phi_2]\!]$*. phiOr can be viewed as a binary function on formulas. Let* $\widetilde{\vee} : \widetilde{\text{Formula}} \times \widetilde{\text{Formula}} \to \widetilde{\text{Formula}}$ *be defined as*

$$\begin{aligned}
\phi_1 \mathrel{\widetilde{\vee}} \phi_2 &\stackrel{\mathsf{def}}{=} \phi_1 \vee \phi_2 \\
\phi_1 \mathrel{\widetilde{\vee}} (\phi_2 \wedge ?) &\stackrel{\mathsf{def}}{=} \\
(\phi_1 \wedge ?) \mathrel{\widetilde{\vee}} \phi_2 &\stackrel{\mathsf{def}}{=} \\
(\phi_1 \wedge ?) \mathrel{\widetilde{\vee}} (\phi_2 \wedge ?) &\stackrel{\mathsf{def}}{=} (\phi_1 \vee \phi_2) \wedge ?
\end{aligned}$$

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

Proof. Soundness Introduction

$$\begin{aligned}
&\phi_1 \wedge \phi_2 \\
=&\phi_1 \mathrel{\widetilde{\vee}} \phi_2
\end{aligned}$$

Monotonicity Large case analysis. Omitted.

Optimality?

□

Lemma 3.33 (Sound Lifting of Composed Function)**.**
Let $g, f : \text{Formula} \to \text{Formula}$ *be arbitrary functions.*

Let $\widetilde{(g \circ f)} : \widetilde{\text{Formula}} \to \widetilde{\text{Formula}}$ *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* $\widetilde{(g \circ f)}$*.*

Proof. Introduction

$$
\begin{aligned}
& g(f(\phi)) \\
\overset{Introduction\ \widetilde{g}}{\sqsubseteq}\ & \widetilde{g}(f(\phi)) \\
\overset{\substack{Introduction\ \widetilde{f} \\ \& \\ Monotonicity\ \widetilde{g}}}{\sqsubseteq}\ & \widetilde{g}(\widetilde{f}(\phi)) \\
=\ & \widetilde{g}(\widetilde{f}(\phi)) \\
=\ & (\widetilde{g} \circ \widetilde{f})(\phi) \\
=\ & \widetilde{(g \circ f)}(\phi)
\end{aligned}
$$

Monotonicity

$$
\begin{aligned}
& \widetilde{\phi_1} \sqsubseteq \widetilde{\phi_2} \\
\overset{Monotonicity\ \widetilde{f}}{\Longrightarrow}\ & \widetilde{f}(\widetilde{\phi_1}) \sqsubseteq \widetilde{f}(\widetilde{\phi_2}) \\
\overset{Monotonicity\ \widetilde{g}}{\Longrightarrow}\ & \widetilde{g}(\widetilde{f}(\widetilde{\phi_1})) \sqsubseteq \widetilde{g}(\widetilde{f}(\widetilde{\phi_2})) \\
\overset{Definition}{\Longrightarrow}\ & \widetilde{(g \circ f)}(\widetilde{\phi_1}) \sqsubseteq \widetilde{(g \circ f)}(\widetilde{\phi_2})
\end{aligned}
$$

□

3.3.3.2. 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 and a predicate indicating the domain of the partial function.

Definition 3.34 (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

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

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 usual.

3.3.4. Generalized Lifting

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

Example: The signature of Hoare rules contain $\textsc{Stmt}$ and can therefore be lifted w.r.t. this parameter using the definitions in section 3.2.4.

3.4. 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 $\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\} \subseteq \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Stmt}} \times \widetilde{\textsc{Formula}}$ (see section 3.5). Furthermore, function lifting is applied to the small-step semantics of **SVL**, resulting in gradual small-step semantics $\cdot \widetilde{\longrightarrow} \cdot : \widetilde{\textsc{ProgramState}} \to \widetilde{\textsc{ProgramState}}$ (see section 3.6). 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. We define gradual soundness of **GVL** as follows:

$$\frac{???}{???}\ \textsc{GProgress}$$

$$\frac{\widetilde{\vdash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}}{\widetilde{\vDash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}}\ \textsc{GPreservation}$$

Valid Hoare triples for the gradual system are defined as

$$\widetilde{\vDash} \{\cdot\} \cdot \{\cdot\} \quad \subseteq \quad \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Stmt}} \times \widetilde{\textsc{Formula}}$$

$$\widetilde{\vDash} \{\widetilde{\phi_{pre}}\}\ \widetilde{s}\ \{\widetilde{\phi_{post}}\} \overset{\mathsf{def}}{\iff} \forall \widetilde{\pi_{pre}}, \widetilde{\pi_{post}} \in \widetilde{\textsc{ProgramState}}.\ \widetilde{\pi_{pre}} \overset{\widetilde{s}}{\widetilde{\longrightarrow}} \widetilde{\pi_{post}} \implies (\widetilde{\pi_{pre}} \mathrel{\widetilde{\vDash}} \widetilde{\phi_{pre}} \implies \widetilde{\pi_{post}} \mathrel{\widetilde{\vDash}} \widetilde{\phi}$$

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 $\{?\}$ `x := 3` $\{$`(y = 4)`$\}$ valid ()due to the existence of a valid instantiation, e.g. $\{$`(y = 4)`$\}$ `x := 3` $\{$`(y = 4)`$\}$). However, this triple is clearly not valid as the postcondition is not guaranteed for all executions satisfying the precondition (?is always satisfied). Recall that sound lifting was introduced in order to comply with the expectations a programmer would have when using a gradual verification system. The validity predicate $\widetilde{\models}\ \{\cdot\}\cdot\{\cdot\}$ plays a higher conceptual role (correctness proofs), is invisible to the programmer and therefore not affected by any user experience expectations.

Unfortunately, the different concepts collide in the gradual preservation condition. On the one hand gradual Hoare logic must comply with the gradual guarantee and thus verify $\widetilde{\vdash}\ \{?\}$ `x := 3` $\{$`(y = 4)`$\}$. On the other hand $\widetilde{\models}\ \{?\}$ `x := 3` $\{$`(y = 4)`$\}$ does not hold since the Hoare triple is invalid. Gradual preservation 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 reiterate our notion of preservation. We introduce an assertion statement `assert` ϕ (if not already available) that throws an exception should the condition not hold. Preservation of **SVL** can then be rewritten as

$$\frac{\vdash \{\phi_1\}\ s\ \{\phi_2\}}{\models \{\phi_1\}\ s\texttt{; assert}\ \phi_2\ \{\phi_2\}}\ \textsc{Preservation'}$$

Intuitively, this definition states that an assertion is inserted during compilation whenever Hoare logic is used to derive the premise. Note that Preservation' is derivable from the original definition of Preservation in section 3.1. Updating gradual preservation accordingly:

$$\frac{\widetilde{\vdash}\ \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}}{\widetilde{\models}\ \{\widetilde{\phi_1}\}\ \widetilde{s}\texttt{; assert}\ \widetilde{\phi_2}\ \{\widetilde{\phi_2}\}}\ \textsc{GPreservation'}$$

Note that there is room for an optimized implementation of the injected assertions. Example: Deducing $\widetilde{\vdash}\ \{?\}$ `y := 2` $\{$`(x = 3)` $\wedge$ `(y = 2)`$\}$ would lead to the injection of `assert (x = 3)` $\wedge$ `(y = 2)`. However, it is known that `(y = 2)` holds after the assignment, making it legal to instead only assert `(x = 3)`.

3.5. Abstracting Static Semantics

Lifting

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

w.r.t. all parameters yields

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

3.5.1. The Problem with Composite Predicate Lifting

As seen in section 3.4, the lifted Hoare predicate in general requires an additional runtime assertion to guarantee preservation. 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 3.3.2.1). The rules allow us to soundly lift each individual inductive rule in order to end up with a sound lifting of the overall predicate predicate.

Example Hoare logic:

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

Rule-wise lifting (assuming that `?`is introduced as single dedicated formula):

$$\frac{\begin{array}{c}\widetilde{\phi_{q1}} \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi_{q2}} \\ \widetilde{\vdash} \{\widetilde{\phi_p}\}\ s_1\ \{\widetilde{\phi_{q1}}\} \qquad \widetilde{\vdash} \{\widetilde{\phi_{q2}}\}\ s_2\ \{\widetilde{\phi_r}\}\end{array}}{\widetilde{\vdash} \{\widetilde{\phi_p}\}\ s_1\texttt{;}\ s_2\ \{\widetilde{\phi_r}\}}\ \textsc{GHSeq} \qquad \frac{}{\widetilde{\vdash} \{\phi[e/x]\}\ x\ \texttt{:=}\ e\ \{\phi\}}\ \textsc{GHAssign1}$$

$$\frac{}{\widetilde{\vdash} \{\texttt{?}\}\ x\ \texttt{:=}\ e\ \{\widetilde{\phi}\}}\ \textsc{GHAssign2} \qquad \frac{}{\widetilde{\vdash} \{\widetilde{\phi}\}\ x\ \texttt{:=}\ e\ \{\texttt{?}\}}\ \textsc{GHAssign3}$$

Note the usage of composite predicate lifting for GHSeq. The overall Hoare predicate $\vdash \{\cdot\} \cdot \{\cdot\}$ can be thought of as a disjunction of all inductive rules. It follows that $\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\}$, being the disjunction of above lifted rules is also a sound lifting.

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 it can deduce that

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

has to hold for some $\widetilde{\phi_{q1}}, \widetilde{\phi_{q2}} \in \widetilde{\text{Formula}}$. 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 aims to use static formulas as early as possible. The implication trivially holds.

Too strict: $\widetilde{\phi_{q1}} = \texttt{(x = 3)} \wedge \texttt{(y = 2)}$, $\widetilde{\phi_{q2}} = \texttt{(y = 2)}$
The instantiation is stricter than necessary – but nevertheless valid according to the rules. The implication holds (the knowledge about `x` is dropped), and $\widetilde{\vdash} \{\texttt{?}\}\ \texttt{y := 2}\ \{\texttt{(x = 3)} \wedge \texttt{(y =}$

holds since $\vdash \{\texttt{(x = 3) } \wedge \texttt{ (2 = 2)}\}\ \texttt{y := 2}\ \{\texttt{(x = 3) } \wedge \texttt{ (y = 2)}\}$ does. This instantiation illustrates the requirement of runtime checks to ensure preservation as described in section 3.4. The judgment $\widetilde{\vdash} \{\texttt{?}\}\ \texttt{y := 2}\ \{\texttt{(x = 3) } \wedge \texttt{ (y = 2)}\}$ must lead to the injection of an assertion of `assert (x = 3) ∧ (y = 2)` right after the assignment. Unfortunately, this assertion alters runtime behavior: Code that would have never thrown an exception when evaluated with the runtime of **SVL** might now throw an exception. This is a violation of the dynamic part of the gradual guarantee. Note that it is actually the runtime semantics breaking the guarantee, yet it is the verifiers "bad decision" that leads up to it.

Consequently, further rules are necessary to prevent the verifier from making decisions that are, as in this case, not general.

Too weak: $\widetilde{\phi_{q1}} = \texttt{?},\ \widetilde{\phi_{q2}} = \texttt{?}$
Using the wildcard is a valid option which also circumvents the trouble illustrated above. 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. In general, choosing `?`as intermediate gradual formulas allows verifying arbitrary inconsistent judgments (a manifestation of the lack of optimality of $\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\}$). For example,

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

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

Consequently, the verifier should somehow try to be "as static as possible" in order to detect inconsistencies at compile time.

Note that above observations do not apply to a static verifier: Its only goal is to find a proof for given Hoare triple. There is no wildcard, meaning that inconsistencies are guaranteed to be detected statically. There is also no notion of choosing an intermediate formula that is too strict since preservation does not rely on runtime checks which might fail if the formula is not general enough.

On the other side, decisions of the gradual verifier have the power to change the runtime behavior or let obvious inconsistencies go unnoticed. We propose a different approach, formalizing above intuition about being neither too weak nor too strong as part of a deterministic gradual Hoare logic which also obviates the need of injecting runtime assertions.

3.5.2. 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) not too strict (making assumptions that are neither general nor required by the following judgment and thus alter runtime behavior). 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 define a deterministic gradual

Hoare logic $\vec{\vdash} : \widetilde{\text{Formula}} \times \widetilde{\text{Stmt}} \rightharpoonup \widetilde{\text{Formula}}$ which has very desirable properties and fits well into our existing model, as described later. We will use our 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 3.3.3.2).

Given a binary predicate $P \subseteq \text{Formula} \times \text{Formula}$ we call a partial function $\vec{P} : \text{Formula} \rightharpoonup \text{Formula}$ **deterministic lifting** of P if the following 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 light gray using function notation for $\vec{\vdash}$.

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.

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

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

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

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

Then valid return values for the deterministic lifting are

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

Not valid are weaker static values like

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

or stronger values like

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

Monotonicity

Identical to monotonicity condition of lifted partial functions (see section 3.3.3.2).

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

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

Soundness and optimality are defined as usual (see sections 3.3.2 or 3.3.3).

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

(a) Obtaining a Sound Gradual Lifting

Lemma 3.36 (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.

Proof.
Introduction

$$\begin{aligned}
P(\phi_1,\phi_2) &\overset{Introduction}{\Longrightarrow} \exists\widetilde{\phi_2}.\ \widetilde{P}(\phi_1)=\widetilde{\phi_2}\\
&\overset{Strength}{\Longrightarrow} \exists\widetilde{\phi_2}.\ \widetilde{P}(\phi_1)=\widetilde{\phi_2}\ \wedge\ \exists\phi\in\gamma(\widetilde{\phi_2}).\ P(\phi_1,\phi)\wedge\phi\Rightarrow\phi_2\\
&\Longrightarrow \exists\widetilde{\phi_2}.\ \widetilde{P}(\phi_1)=\widetilde{\phi_2}\ \wedge\ \exists\phi\in\gamma(\widetilde{\phi_2}).\ \phi\Rightarrow\phi_2\\
&\Longrightarrow \exists\widetilde{\phi_2}.\ \widetilde{P}(\phi_1)=\widetilde{\phi_2}\ \wedge\ \widetilde{\phi_2}\mathrel{\widetilde{\Rightarrow}}\phi_2\\
&\overset{Def}{\Longrightarrow} \widetilde{P}(\phi_1,\phi_2)
\end{aligned}$$

Monotonicity

$$\begin{aligned}
\widetilde{P}(\widetilde{\phi_1},\widetilde{\phi_2})\ \wedge\ \widetilde{\phi_1}\sqsubseteq\widetilde{\phi_1'}\ \wedge\ \widetilde{\phi_2}\sqsubseteq\widetilde{\phi_2'} &\overset{Def}{\Longrightarrow} (\exists\widetilde{\phi}.\ \vec{P}(\widetilde{\phi_1})=\widetilde{\phi}\wedge\widetilde{\phi}\mathrel{\widetilde{\Rightarrow}}\widetilde{\phi_2})\ \wedge\ \widetilde{\phi_1}\sqsubseteq\widetilde{\phi_1'}\ \wedge\ \widetilde{\phi_2}\sqsubseteq\widetilde{\phi_2'}\\
&\overset{Monotonicity}{\Longrightarrow} (\exists\widetilde{\phi},\widetilde{\phi'}.\ \vec{P}(\widetilde{\phi_1'})=\widetilde{\phi'}\wedge\widetilde{\phi}\sqsubseteq\widetilde{\phi'}\wedge\widetilde{\phi}\mathrel{\widetilde{\Rightarrow}}\widetilde{\phi_2})\ \wedge\ \widetilde{\phi_2}\sqsubseteq\widetilde{\phi_2'}\\
&\Longrightarrow (\exists\widetilde{\phi'}.\ \vec{P}(\widetilde{\phi_1'})=\widetilde{\phi'}\wedge\widetilde{\phi'}\mathrel{\widetilde{\Rightarrow}}\widetilde{\phi_2})\ \wedge\ \widetilde{\phi_2}\sqsubseteq\widetilde{\phi_2'}\\
&\Longrightarrow (\exists\widetilde{\phi'}.\ \vec{P}(\widetilde{\phi_1'})=\widetilde{\phi'}\wedge\widetilde{\phi'}\mathrel{\widetilde{\Rightarrow}}\widetilde{\phi_2'})\\
&\overset{Def}{\Longrightarrow} \widetilde{P}(\phi_1',\phi_2')
\end{aligned}$$

□

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

$$\vdash \{\texttt{(y = 4)}\}\ \texttt{x := 3}\ \{\texttt{(y = 4) ∧ (x = 3)}\}$$

and

$$\vdash \{\texttt{(y = 5)}\}\ \texttt{x := 3}\ \{\texttt{(y = 5) ∧ (x = 3)}\}$$

using Hoare logic.

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

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

The unknown formula as postcondition is necessary, as anything else (e.g. `(x = 3)`) would break the strength condition (take the above static judgments, both of their postconditions cannot be implied by any static formula). Note that the "wildcard with upper bound" approach (see 3.2.2) would allow specifying `(x = 3) ∧ ?` as postcondition.

Since `?` $\widetilde{\Rightarrow}$ `(x = 1)` holds, we can deduce

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

An optimal gradual lifting would not be able to deduce this judgment, as it requires the existing of some $\phi \in \gamma(?) = \textsc{SatFormula}$ 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 with `(x = 3)` $\wedge$ `?` as a postcondition, one cannot deduce above above gradual judgment (since `(x = 3)` $\wedge$ `?` $\widetilde{\Rightarrow}$ `(x = 1)`) does not hold. This example motivates the use of more powerful gradual syntax extensions, as apparently they result in more optimal gradual liftings.

(b) Determinism of Verifier

As the name suggests, deterministic liftings leave no room for choice. For the gradual verifier this means that there is no need to infer intermediate formulas, averting the risk of choosing the wrong formulas (as illustrated in section 3.5.1).

(c) Free Transitivity

Furthermore, applying Hoare rules transitively induces no more additional runtime cost: As described in 3.4, every judgment of the form $\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\}$ must be accompanied with the injection of a runtime check in order to guarantee preservation. Note that applying gradual Hoare logic to a sequence of statements (using GHSec) requires such judgment for every single statement, resulting in assertions between every pair of statements.

Using a deterministic lifting transitively induces no such runtime cost since the no judgment of the form $\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\}$ is made. Only after applying lemma 3.36 in the very end of verification, an assertion will be injected to ensure preservation for the overall judgment.

(d) Preservation

Deterministic liftings are designed in a way that enables defining a stronger notion of preservation that does not rely on runtime assertions.

$$\frac{\vec{\widetilde{\vdash}}\ \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}}{\widetilde{\vDash}\ \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}}\ \textsc{GDPreservation}$$

Note however, that this rule is not automatically satisfied, this also depends on the gradual dynamic semantics of **GVL**.

Example 3.38 (GDPreservation Counterexample)**.** Assume that **SVL** contains an assertion statement with corresponding Hoare rule.

$$\frac{\phi \Rightarrow \phi_a}{\vdash \{\phi\}\ \texttt{assert}\ \ \phi_a\ \{\phi\}}\ \textsc{HAssert}$$

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.

A valid deterministic lifting of HAssert is able to verify

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

However, if the gradual dynamic semantics of **GVL** still implement assertions as a no-operation, then $\widetilde{\models} \{?\}\ \texttt{assert}\ \phi_a\ \{\phi_a \wedge ?\}$ does not hold. On the other hand, adding a runtime check that throws an exception on failure would restore preservation.

3.5.2.1. Examples

Lemma 3.39 (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

$$\vec{P_3} \overset{\text{def}}{=} \vec{P_2} \circ \vec{P_1}$$

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

Proof. Introduction

$$\begin{array}{lcl}
P_3(\phi_1, \phi_3) & \Longrightarrow & \exists \phi_2.\ P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi_3) \\
& \overset{\textit{Introduction}}{\Longrightarrow} & \exists \phi_2.\ P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi_3) \wedge (\exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2}) \\
& \overset{\textit{Strength}}{\Longrightarrow} & \exists \phi_2.\ P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi_3) \wedge (\exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2} \wedge (\exists \phi_2' \in \gamma(\widetilde{\phi_2}).\ \phi_2' \Rightarrow \phi_2)) \\
& \overset{\textit{Mono } P_2}{\Longrightarrow} & \exists \phi_2.\ P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi_3) \wedge (\exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2} \wedge (\exists \phi_2' \in \gamma(\widetilde{\phi_2}), \phi_3'.\ \phi_2' \Rightarrow \phi_2 \wedge \\
& \Longrightarrow & \exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2} \wedge (\exists \phi_2' \in \gamma(\widetilde{\phi_2}), \phi_3'.\ P_2(\phi_2', \phi_3')) \\
& \overset{\textit{Introduction}}{\Longrightarrow} & \exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2} \wedge (\exists \phi_2' \in \gamma(\widetilde{\phi_2}), \widetilde{\phi_3'}.\ \vec{P_2}(\phi_2') = \widetilde{\phi_3'}) \\
& \overset{\textit{Monotonicity}}{\Longrightarrow} & \exists \widetilde{\phi_2}.\ \vec{P_1}(\phi_1) = \widetilde{\phi_2} \wedge (\exists \widetilde{\phi_3}.\ \vec{P_2}(\widetilde{\phi_2}) = \widetilde{\phi_3}) \\
& \Longrightarrow & \exists \widetilde{\phi_3}.\ \vec{P_2}(\vec{P_1}(\phi_1)) = \widetilde{\phi_3} \\
& \Longrightarrow & \exists \widetilde{\phi_3}.\ \vec{P_3}(\phi_1) = \widetilde{\phi_3}
\end{array}$$

Strength

$$
\begin{array}{ll}
\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} & \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}) \wedge P_1(\phi_1, \phi') \wedge P_2(\phi', \phi) \\
\overset{Strength}{\Longrightarrow} & \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}) \wedge P_1(\phi_1, \phi') \wedge P_2(\phi', \phi) \wedge (\exists \phi_2 \in \gamma(\widetilde{\phi_2}).\ P_1 \\
\Longrightarrow & \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}) \wedge P_1(\phi_1, \phi_2) \wedge P_2(\phi', \phi) \wedge \phi_2 = \\
\Longrightarrow & \exists \widetilde{\phi_2}, \phi', \phi_2 \in \gamma(\widetilde{\phi_2}).\ \vec{P}_2(\widetilde{\phi_2}) = \widetilde{\phi_3} \wedge P_1(\phi_1, \phi_2) \wedge P_2(\phi', \phi) \wedge \phi_2 \Rightarrow \phi' \\
\overset{Mono\ P_2}{\Longrightarrow} & \exists \widetilde{\phi_2}, \phi_2 \in \gamma(\widetilde{\phi_2}), \phi''.\ \vec{P}_2(\widetilde{\phi_2}) = \widetilde{\phi_3} \wedge P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi'') \wedge \phi'' \Rightarrow \phi \\
\overset{Strength}{\Longrightarrow} & \exists \widetilde{\phi_2}, \phi_2 \in \gamma(\widetilde{\phi_2}), \phi''.\ \vec{P}_2(\widetilde{\phi_2}) = \widetilde{\phi_3} \wedge P_1(\phi_1, \phi_2) \wedge P_2(\phi_2, \phi'') \wedge \phi'' \Rightarrow \phi \wedge (\exists \phi_3 \in \gamma(\widetilde{\phi_3}).\ P_2(\phi_2 \\
\Longrightarrow & \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} & \exists \phi_3 \in \gamma(\widetilde{\phi_3}).\ P_3(\phi_1, \phi_3) \wedge \phi_3 \Rightarrow \phi
\end{array}
$$

Monotonicity holds by transitivity □

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 3.5.1, where we demonstrated the problems that ultimately lead to deterministic liftings.

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

Lemma 3.40 (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* $\cdot \Rightarrow \cdot \subseteq \textsc{Formula} \times \textsc{Formula}$.

Proof. Soundness Introduction 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. □

Using lemma ??? and lemma ???, the deterministic lifting of HSeq reduces to

$$\frac{\vec{\vdash} \{\widetilde{\phi_p}\}\ \widetilde{s_1}\ \{\widetilde{\phi_q}\} \qquad \vec{\vdash} \{\widetilde{\phi_q}\}\ \widetilde{s_2}\ \{\widetilde{\phi_r}\}}{\vec{\vdash} \{\widetilde{\phi_p}\}\ \widetilde{s_1}\texttt{;}\ \widetilde{s_2}\ \{\widetilde{\phi_r}\}}\ \text{DGHSeq}$$

Note that we used inductive notation to define a function – there are no free variables.

For the assignment rule we can derive

$$\frac{x \notin \mathsf{FV}(\phi) \qquad x \notin \mathsf{FV}(e)}{\vec{\vdash} \{\phi\}\ x\ \texttt{:=}\ e\ \{\phi \wedge (x\ \texttt{=}\ e)\}}\ \text{DGHAssign1}$$

$$\frac{\text{DGHAssign1 } \textit{does not apply}}{\vec{\vdash} \{\widetilde{\phi}\}\ x\ \texttt{:=}\ e\ \{\texttt{?}\}}\ \text{DGHAssign2}$$

as a sound deterministic lifting.

3.5.2.2. 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 3.5.2.1.

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

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

As established in section 3.4, runtime assertions have to be injected whenever such a judgment is made, in order to guarantee preservation. Resulting program, as executed by small-step semantics:

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

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

Note that the gradual verifier has not yet verified above judgments. It does so by applying the deterministic lifting (as determined in section 3.5.2.1) and then using lemma 3.36 to deduce the gradual lifting.

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

$$\dfrac{\dfrac{\dfrac{\texttt{i} \notin \mathsf{FV}(\texttt{true})\quad \texttt{i} \notin \mathsf{FV}(4)}{\mathcal{D}_1}\ \textsc{DGHAssign1} \qquad \dfrac{\texttt{result} \notin \mathsf{FV}(\texttt{true} \wedge \texttt{(i = 4)})\quad \texttt{result} \notin \mathsf{FV}(\texttt{i})}{\mathcal{D}_2}\ \textsc{DGHAssign1}}{\mathcal{D}}\ \textsc{DGHSeq} \qquad \mathcal{I}}{\widetilde{\vdash} \{\texttt{true}\}\ \texttt{i := 4; result := i}\ \{\texttt{(result = 4)}\}}\ \textsc{Lemma 3.36}$$

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

$$\dfrac{\dfrac{\dfrac{}{\mathcal{D}_1}\ \textsc{DGHAssign2} \qquad \dfrac{}{\mathcal{D}_2}\ \textsc{DGHAssign2}}{\mathcal{D}}\ \textsc{DGHSeq} \qquad \mathcal{I}}{\widetilde{\vdash} \{\texttt{?}\}\ \texttt{i := i + 1; result := i}\ \{\texttt{(result = 4)}\}}\ \textsc{Lemma 3.36}$$

where

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

Comparing `getFourA` and `getFourB` one can observe a difference regarding the injected runtime assertions. On the one hand, the check in `getFourA` is unnecessary since it will always succeed. On the other hand, the check in `getFourB` seems necessary in order for the postcondition to be satisfied (for all executions that reach it). This difference is also indicated in the deduction of both contracts. Verifying `getFourA`, the deterministic lifting determined `true ∧ (i = 4) ∧ (result = i)` as postcondition, before being weakened to `(result = 4)`. However, verifying `getFourB`, the deterministic lifting ends up with `?` as knowledge, from which `(result = 4)` can only be implied by instantiating `?` accordingly.

The intuition behind the negligible assertion can be formalized: In case GDPreservation holds, we are guaranteed that the postcondition returned by the deterministic lifting actually holds for every execution. In other words, it is static knowledge that can be used to reason about the outcome of runtime assertions at verification time. In case of function `getFourA`, the compiler would be able to treat `true ∧ (i = 4) ∧ (result = i)` as static knowledge that is usable to formally guarantee the success of `assert (result = 4)`.

3.6. Abstracting Dynamic Semantics

Let $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot : \widetilde{\text{ProgramState}} \rightharpoonup \widetilde{\text{ProgramState}}$ be a sound gradual lifting of $\cdot \longrightarrow \cdot : \text{ProgramState} \rightharpoonup \text{ProgramState}$.

$$\frac{\widetilde{\vdash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}}{\widetilde{\vDash} \{\widetilde{\phi_1}\}\ \widetilde{s}\texttt{; assert }\widetilde{\phi_2}\ \{\widetilde{\phi_2}\}}\ \text{GPreservation'}$$

The conclusion is a tautology.

In case the deterministic lifting approach is used to derive a gradual Hoare logic it is desirable to satisfy the stronger notion of preservation introduced in section 3.5.2:

$$\frac{\vec{\vdash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}}{\widetilde{\vDash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}}\ \text{GDPreservation}$$

This stronger notion allows considering return values of the deterministic lifting as static knowledge. As illustrated in section 3.5.2.2, such static knowledge can be used for optimizations.

To summarize, GPreservation' is sufficient for soundness of the gradual system whereas GDPreservation is valuable for optimizations. In the remainder of this section we will thus determine sufficient criteria for GDPreservation.

3.6.1. Perfect Knowledge

When first introducing GDPreservation in section 3.5.2, we also gave an example (3.38) of a language violating GDPreservation. 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 3.38, incompleteness of the Hoare logic may not necessarily indicate weakness of the Hoare logic, but may also be due to 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 3.41 (Completeness)**.**
A semantics is ***complete*** *if every execution of a statement s can be supported with a matching Hoare logic derivation:*

$$\forall s \in \textsc{Stmt},\, \pi_1, \pi_2 \in \textsc{ProgramState}.\ \pi_1 \xrightarrow{s} \pi_2 \implies \exists \phi_1, \phi_2 \in \textsc{Formula}.\ \vdash \{\phi_1\}\ s\ \{\phi_2\} \land \pi_1 \vDash \phi_1$$

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 3.42 (Completion of Semantics)**.** *Deriving sound, complete semantics by restricting the domain of the small-step semantics or extending the Hoare logic of* ***SVL*** *does not have observable effects.*

Proof. **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. □

Based on a complete semantics of **SVL**, the following requirements for **GVL** are sufficient to satisfy GDPreservation.

Definition 3.43 (Perfect Gradual Hoare Logic)**.** *We call a deterministically lifted Hoare logic $\vec{\vdash}\ \{\cdot\}\ \cdot\ \{\cdot\}$ perfect iff*

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

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

deterministic liftings are perfect. Perfection can be viewed as a very weak optimality measure.

Definition 3.44 (Perfect Gradual Small-Step Semantics)**.** *We call a total, lifted small-step semantics* $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot$ *perfect iff*

$$\forall \widetilde{\pi_1}, \widetilde{\pi_2} \in \widetilde{\text{ProgramState}},\ \widetilde{s} \in \widetilde{\text{Stmt}}.\ \widetilde{\pi_1} \overset{\widetilde{s}}{\mathrel{\widetilde{\longrightarrow}}} \widetilde{\pi_2} \implies \exists \pi_1 \in \gamma(\widetilde{\pi_1}),\ s \in \gamma(\widetilde{s}), \pi_2 \in \text{Formula}.\ \pi_1 \xrightarrow{s} \pi_2$$

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). Optimal deterministic liftings are perfect. Perfection can be viewed as a very weak optimality measure.

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

Theorem 3.45. *If the semantics of* ***SVL*** *are complete and the semantics of* ***GVL*** *are perfect, then* GDPreservation *is satisfied.*

Proof. To prove GDPreservation we may assume:

$$\vec{\vdash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\} \tag{3.1}$$

$$\widetilde{\pi_1} \overset{\widetilde{s}}{\mathrel{\widetilde{\longrightarrow}}} \widetilde{\pi_2} \tag{3.2}$$

$$\widetilde{\pi_1} \mathrel{\widetilde{\vDash}} \widetilde{\phi_1} \tag{3.3}$$

Goal:

$$\widetilde{\pi_2} \mathrel{\widetilde{\vDash}} \widetilde{\phi_2} \tag{3.4}$$

Case $\widetilde{\phi_1} = \phi_1$ for some $\phi_1 \in$ Formula: We can simplify 3.3 as

$$\widetilde{\pi_1} \vDash \phi_1 \tag{3.5}$$

From perfection of 3.1 (see definition 3.44) we can derive

$$\vdash \{\phi_1\}\ s\ \{\phi\} \qquad \text{for some } s \in \gamma(\widetilde{s}),\ \phi_2 \in \text{Formula} \tag{3.6}$$

Applying the strength rule for deterministic liftings we can derive from 3.6 and 3.1 that

$$\vdash \{\phi_1\}\ s\ \{\phi_2\}\ \wedge\ \phi_2 \Rightarrow \phi \qquad \text{for some } \phi_2 \in \gamma(\widetilde{\phi_2}) \tag{3.7}$$

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 3.2 we can conclude that executing s must terminate (otherwise, $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot$, being the gradual lifting, could not terminate for $\widetilde{s} \sqsupseteq s$).

$$\pi_1 \xrightarrow{s} \pi_2 \qquad \text{for some } \pi_1 \in \gamma(\widetilde{\pi_1}),\ \pi_2 \in \text{ProgramState} \tag{3.8}$$

From soundness (preservation) of the static Hoare logic we can deduce

$$\pi_2 \vDash \phi_2 \tag{3.9}$$

Applying the introduction rule of gradually lifted functions to 3.8 we can derive

$$\pi_1 \overset{s}{\widetilde{\longrightarrow}} \widetilde{\pi} \ \wedge \ \pi_2 \sqsubseteq \widetilde{\pi} \qquad \text{for some } \widetilde{\pi} \in \widetilde{\text{ProgramState}} \tag{3.10}$$

It follows that

$$\widetilde{\pi} \vDash \phi_2 \tag{3.11}$$

Applying the monotonicity rule of gradually lifted functions to 3.10 and 3.2 we can derive that

$$\widetilde{\pi} \sqsubseteq \widetilde{\pi_2} \tag{3.12}$$

and thus

$$\widetilde{\pi_2} \vDash \phi_2 \tag{3.13}$$

We can generalize this using $\phi_2 \in \gamma(\widetilde{\phi_2})$ (see 3.7)

$$\widetilde{\pi_2} \mathrel{\widetilde{\vDash}} \widetilde{\phi_2} \tag{3.14}$$

Case $\widetilde{\phi_1}$ partially unknown, i.e. $\gamma(\widetilde{\phi_1})$ closed over implication: From perfection of 3.2 (see definition 3.44) we can derive

$$\pi_1 \xrightarrow{s} \pi_2 \qquad \text{for some } \pi_1 \in \gamma(\widetilde{\pi_1}),\ s \in \gamma(\widetilde{s}),\ \pi_2 \in \text{ProgramState} \tag{3.15}$$

Recall that formula evaluation is immune to concretization (REF), so from 3.3 follows

$$\pi_1 \mathrel{\widetilde{\vDash}} \widetilde{\phi_1} \qquad \text{or equivalently} \qquad \pi_1 \vDash \phi_{1a} \qquad \text{for some } \phi_{1a} \in \gamma(\widetilde{\phi_1}) \tag{3.16}$$

Using the monotonicity of $\cdot \overset{\widetilde{}}{\longrightarrow} \cdot$ we can deduce from 3.2 and 3.15 that

$$\pi_2 \in \gamma(\widetilde{\pi_2}) \tag{3.17}$$

From completeness of the static system (see definition 3.41) it follows from 3.15 that

$$\vdash \{\phi_{1b}\}\ s\ \{\phi\} \ \wedge \ \pi_1 \vDash \phi_{1b} \qquad \text{for some } \phi_{1b}, \phi \in \text{Formula} \tag{3.18}$$

Due to $[\![\pi_1]\!]$ being a filter we can derive from $\pi_1 \vDash \phi_{1a}$ (3.16) and $\pi_1 \vDash \phi_{1b}$ (3.18) that

$$\pi_1 \vDash \phi_1 \ \wedge \ \phi_1 \Rightarrow \phi_{1a} \ \wedge \ \phi_1 \Rightarrow \phi_{1b} \qquad \text{for some } \phi_1 \in \text{Formula} \tag{3.19}$$

From monotonicity of $\vdash \{\cdot\} \cdot \{\cdot\}$ in its first argument (

$$\vdash \{\phi_1\}\ s\ \{\phi'\} \ \wedge \ \phi' \Rightarrow \phi \qquad \text{for some } \phi_2 \in \text{Formula} \tag{3.20}$$

Applying the introduction rule for deterministic liftings we can deduce

$$\vec{\vdash} \{\phi_1\}\ s\ \{\widetilde{\phi}\} \qquad \text{for some } \widetilde{\phi} \in \widetilde{\text{Formula}} \tag{3.21}$$

Applying the strength rule for deterministic liftings we can derive from 3.20 and 3.21 that

$$\vdash \{\phi_1\}\ s\ \{\phi_2\}\ \wedge\ \phi_2 \Rightarrow \phi' \qquad \text{for some } \phi_2 \in \gamma(\widetilde{\phi}) \tag{3.22}$$

From soundness of the static system we can deduce (using 3.21, 3.15, 3.19) that

$$\pi_2 \vDash \phi_2 \tag{3.23}$$

and therefore (using 3.17)

$$\widetilde{\pi_2} \mathrel{\widetilde{\vDash}} \phi_2 \tag{3.24}$$

and therefore (using 3.22)

$$\widetilde{\pi_2} \mathrel{\widetilde{\vDash}} \widetilde{\phi} \tag{3.25}$$

Now, using $\phi_1 \in \gamma(\widetilde{\phi_1})$ (implied from the fact that $\phi_{1a} \in \gamma(\widetilde{\phi_1})$, $\phi_1 \Rightarrow \phi_{1a}$) we can apply monotonicity of deterministic liftings to 3.21, we can derive

$$\widetilde{\phi} \sqsubseteq \widetilde{\phi_2} \tag{3.26}$$

and therefore

$$\widetilde{\pi_2} \mathrel{\widetilde{\vDash}} \widetilde{\phi_2} \tag{3.27}$$

□

3.6.2. Partial Knowledge

Completeness as defined before tightly couples Hoare logic and small-step semantics. In practice this can be impossible to implement due to decidability, i.e. not all valid small-step derivations can be modeled using Hoare logic. As a result, completeness is impossible to achieve. Similarly, perfection of the gradual semantics might be hard to ensure. Perfection 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 completion and perfection may be sufficient to prove GDPreservation.

The sequence operator ; is key to defining composite statements in most programming languages. Fortunately, GDPreservation can be proved for sequences inductively.

Lemma 3.46 (GDPreservation for Sequences)**.** *If* GDPreservation *holds for statements s_1 and s_2 then it holds for $s_1; s_2$*

Proof.

$$\dfrac{\dfrac{\dfrac{\vec{\vdash}\ \{\widetilde{\phi_1}\}\ \widetilde{s_1};\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}{\vec{\vdash}\ \{\widetilde{\phi_1}\}\ \widetilde{s_1}\ \{\widetilde{\phi_2}\} \qquad \vec{\vdash}\ \{\widetilde{\phi_2}\}\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}\ \text{INVERSION}}{\widetilde{\vDash}\ \{\widetilde{\phi_1}\}\ \widetilde{s_1}\ \{\widetilde{\phi_2}\} \qquad \widetilde{\vDash}\ \{\widetilde{\phi_2}\}\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}\ \text{GDPreservation}}{\widetilde{\vDash}\ \{\widetilde{\phi_1}\}\ \widetilde{s_1};\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}\ \text{SEQ}$$

□

As we will show in the case study (section 4.5) this inductive approach can be applied to other composite statements like method calls. As a result, it is sufficient to prove GDPreservation for "primitive" statements, e.g. by using the approach introduced in the previous section: Note that the definitions of completeness and perfection 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 GDPreservation will apply only to this set of statements.

Again, we want to point out that we only give examples of sufficient criteria to prove GDPreservation. It is possible that the approaches do not work for a certain programming language, or even that it is entirely impossible to satisfy GFPreservation. However, recall that GDPreservation is not necessary for **GVL** to be sound.

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 3.5.2 we will obtain gradual Hoare logic of $\mathbf{GVL_{IDF}}$ in section 4.3. In section 4.4 we derive gradual small-step semantics in a way that makes the gradual verification system sound and also complies with the stronger notion of preservation postulated in section 3.4.

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/Eiffel/Spec# sub-syntax to express method contracts.

4.1.1. Syntax

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

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.

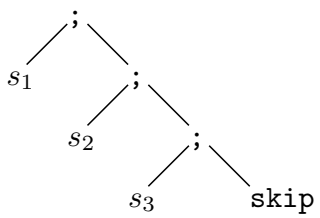
```
...
{
    s1;
    s2;
    s3;
}
```

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

We pose $\texttt{false} \stackrel{\text{def}}{=} \texttt{(null ≠ null)}$.

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

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 fol-

lowing functions:

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

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

Let $\mathsf{FV} : (\textsc{Expr} \cup \textsc{Formula}) \to \mathcal{P}^{\textsc{Var}}$ be defined as

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

Default Value of Type

$\mathbf{SVL_{IDF}}$ assigns default values to declared variables.

Let $\mathsf{defaultValue} : \textsc{Type} \to \textsc{Val}$ be defined 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 expressions in a uniform fashion, we define a pseudo accessibility-predicate which is also defined for expressions that do not mention fields.

Let $\mathsf{acc} : \textsc{Expr} \to \textsc{Formula}$ be defined 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 rare circumstances, 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.

Let $\mathsf{mod} : \textsc{Stmt} \to \mathcal{P}^{\textsc{Var}}$ be defined 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 variable storage is abstracted in $\mathbf{SVL_{IDF}}$.

In figure 4.2 we define a ternary evaluation function

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

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

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

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

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

access to a certain field. As illustrated in section 2.3 sound reasoning is only possible if a formula contains access `acc` for all the fields it mentions. This property is called self-framing and will be formalized in this section.

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 $\lfloor \cdot \rfloor$: Formula $\to$ StaticFootprint, a function for obtaining static footprints where

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

Static footprints are required for static reasoning, mainly for determining whether a formula is self-framing. Later, they will become 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 $\lfloor \cdot \rfloor_{\cdot,\cdot}$: Heap $\times$ VarEnv $\times$ Formula $\rightharpoonup$ StaticFootprint, a function for obtaining dynamic footprints where

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

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 ir 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 for the dynamic footprint.

Example 4.3 (Aliasing Access)**.**
Let H and ρ be defined such that $a, b, c \in$ Expr all evaluate to the same memory

location $o \in$ Loc. Then:

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

Note that this scenario 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.

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

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

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

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

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

What about undefinedness of acc case? Guess: propagates to undefinedness of small-step rule => covered by soundness

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 in figure 4.5.

At last, we can define a predicate $\vdash_{\texttt{sfrm}} \subseteq$ StaticFootprint $\times$ 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, however only to the right.

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

$$\frac{}{A \vdash_{\texttt{frm}} x}\ \textsc{WFVar}$$

$$\frac{}{A \vdash_{\texttt{frm}} v}\ \textsc{WFValue}$$

$$\frac{(e, f) \in A \qquad A \vdash_{\texttt{frm}} e}{A \vdash_{\texttt{frm}} e\,.\,f}\ \textsc{WFField}$$

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

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-Framing Formula)**.** *A formula ϕ is **self-framing** iff*

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

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

For the remainder of this work we will only encounter $\vdash_{\texttt{sfrm}}$ in connection with self-framing and thus omit $\emptyset$ from the judgment.

As we introduced SatFormula back in section 3.1, we argued that a sound verification system should not be able to derive unsatisfiable formulas, as they correspond to unreachable code. Similarly, we argue that a sound verification system based on IDF should not be able to derive formulas that are not self-framing. Inconsistent information could be derived from such formulas, rendering the verification system unsound. For $\mathbf{SVL_{IDF}}$ will thus only consider method contracts with formulas drawn from SfrmFormula. We will actually enforce this condition as part of well-formedness conditions in section 4.1.8.

4.1.4. Program State

The set of program states is defined as ProgramState = Heap × Stack where

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

A program state of $\mathbf{SVL_{IDF}}$ consists of a single heap and a stack. Each stack frame has an environment VarEnv for local variables, tracks a set of accessible fields DynamicFootprint and stores the remaining work Stmt. Stack frames will be introduced by method calls, but also by special scopes as introduced 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,

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

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

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

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

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

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

but not the continuation statement). To drastically simplify notation, we will therefore define $\vDash \subseteq \textsc{ProgramState} \times \textsc{Formula}$ (as postulated in section 3.1) indirectly:

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

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 ($\mathbf{SVL_{IDF}}$: Derivable Rules)**.**

Decomposition

$$\frac{}{\phi_1 * \phi_2 \Rightarrow \phi_1}\ \textsc{FrmDecomp1}$$

$$\frac{}{\phi_1 * \phi_2 \Rightarrow \phi_2}\ \textsc{FrmDecomp2}$$

Transitivity

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

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

Aliasing Prevention

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

Conjunction

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

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 proves 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 $\phi \stackrel{\text{def}}{=} \texttt{acc(a.f)} \wedge \texttt{acc(b.f)}$. According to above equation, ϕ is satisfied by an environment π, if it contains access to `a.f` and `b.f`. Since we want to define $\cdot \wedge \cdot$ in terms of existing syntax, it should be possible to define ϕ using that syntax.

Attempt: We can prove that

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

and

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

are formulas that would imply ϕ and that there exist no larger formulas (w.r.t. $\Rightarrow$) with that property.

Unfortunately, neither ϕ_1 nor ϕ_2 are implied by ϕ, which means that $\phi \notin \textsc{Formula}$.

Above examples proves by shows that $\cdot \wedge \cdot$ cannot be defined using existing syntax of $\mathbf{SVL_{IDF}}$. 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)` but never be unspecific about it, as would be necessary in the example above.

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. All the rules are implicitly parameterized over some program $p \in$ 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 $\mathbf{SVL_{IDF}}$.

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) obviates 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.*

Proof. 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.

$\widetilde{\text{SsSkip}}$ H, ρ, A, s, S are directly drawn from the input.

$\widetilde{\text{SsFieldAssign}}$ 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.

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

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

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

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

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

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

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

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

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

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

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

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

The same approach can be used for all remaining rules. □

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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.

Example 4.9 (Leaking Postcondition)**.**

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

While the method passes static verification, it could lead to unsound proofs. 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.

For similar reasons, we prevent writes to the method's parameter. Changes to the parameter could legally be stated in the postcondition which is forwarded to the call site. Information about the parameter is then reflected back on the variable passed for the

method call. However, since variables are passed by value, this information would be false.

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).*

Proof. □

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 "gradual formulas with upper bound" as introduced in section 3.2.2:

Syntax

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

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

We call formulas containing ?"partly unknown". Consequence: A gradual formulas is either static or partly-unknown.

Concretization

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

Note that we do not require the static part of $\texttt{?} * \phi$ 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.color = b.color)`, 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:

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

We want to only allow gradual formulas in method contracts, resulting in the following gradual syntax: Note how the small change propagates throughout other constructs, all

$$
\begin{aligned}
\widetilde{program} \in \widetilde{\text{PROGRAM}} &::= \overline{\widetilde{cls}}\ \ s \\
\widetilde{cls} \in \widetilde{\text{CLASS}} &::= \texttt{class}\ C\ \texttt{\{}\ \overline{field}\ \ \overline{\widetilde{method}}\ \texttt{\}} \\
\widetilde{method} \in \widetilde{\text{METHOD}} &::= T\ \ m\texttt{(}T\ \ x\texttt{)}\ \ \widetilde{contract}\ \texttt{\{}\ s\ \texttt{\}} \\
\widetilde{contract} \in \widetilde{\text{CONTRACT}} &::= \texttt{requires}\ \widetilde{\phi}\texttt{;}\ \texttt{ensures}\ \widetilde{\phi}\texttt{;} \\
\widetilde{\phi} \in \widetilde{\text{FORMULA}} &::= \phi \mid \texttt{?} * \phi
\end{aligned}
$$

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

the way up to gradual programs. The change has no direct impact on statement syntax, allowing us to define $\widetilde{\text{STMT}} \stackrel{\text{def}}{=} \text{STMT}$ and therefore $\widetilde{\text{PROGRAMSTATE}} \stackrel{\text{def}}{=} \text{PROGRAMSTATE}$. Note that we also have not decorated s in figure 4.12 although it is formally drawn from $\widetilde{\text{STMT}}$. There is no point in decorating statements or program state in of $\mathbf{GVL_{IDF}}$. However, note that static and dynamic semantics of the call statement $x\ \texttt{:=}\ y\texttt{.}m\texttt{(}z\texttt{)}$ will still have to be lifted as m now references a gradual method contract (see section 3.2.4 for detailed discussion).

4.2.1. Access-Free Gradual Normal Form

In section 4.1.5.1 we showed that regular non-separating conjunction $\cdot \wedge \cdot$ is not expressible using existing syntax of $\mathbf{SVL_{IDF}}$. Fortunately, the same is not true for $\mathbf{GVL_{IDF}}$:

$$\cdot \wedge \cdot : \widetilde{\text{FORMULA}} \times \widetilde{\text{FORMULA}} \to \widetilde{\text{FORMULA}}$$

is definable (and computable), as we will show in this section. Key to the definition is the existence of a normal form for partly unknown formulas $\texttt{?} * \phi$, that is free of accessibility-predicates. Having a non-separating conjunction at hand will prove useful for defining some gradual liftings.

Theorem 4.12 (Gradual Normal Form)**.**
There exists a computable function $|\cdot| : \text{FORMULA} \to \text{FORMULA}$*, such that*

(a) $||\texttt{?} * \phi|| \stackrel{\text{def}}{=} \texttt{?} * |\phi|$ *is equivalent to* $\texttt{?} * \phi$ *(for all* $\phi \in \text{FORMULA}$*)*

(b) $\phi \implies |\phi|$

(c) $|\phi|$ *contains no accessibility-predicates*

(d) $\texttt{?} * \phi_1 \sqsubseteq \texttt{?} * \phi_2 \iff |\phi_1| \implies |\phi_2|$

Theorem 4.13 (Non-Separating Conjunction)**.**
Let $\cdot \wedge \cdot : \widetilde{\text{FORMULA}} \times \widetilde{\text{FORMULA}} \to \widetilde{\text{FORMULA}}$ *be defined as*

$$\widetilde{\phi_1} \wedge \widetilde{\phi_2} \stackrel{\text{def}}{=} \texttt{?} * |\mathsf{static}(\widetilde{\phi_1})| * |\mathsf{static}(\widetilde{\phi_2})|$$

Then $[\![\widetilde{\phi_1} \wedge \widetilde{\phi_2}]\!] = [\![\widetilde{\phi_1}]\!] \cap [\![\widetilde{\phi_2}]\!]$

Proof.

$$
\begin{aligned}
[\![\widetilde{\phi_1} \wedge \widetilde{\phi_2}]\!] &= [\![? * |\mathsf{static}(\widetilde{\phi_1})| * |\mathsf{static}(\widetilde{\phi_2})|]\!] \\
&= \{\ \pi \in \textsc{ProgramState} \mid \pi \mathrel{\widetilde{\vDash}} ? * |\mathsf{static}(\widetilde{\phi_1})| * |\mathsf{static}(\widetilde{\phi_2})|\ \} \\
&= \{\ \pi \in \textsc{ProgramState} \mid \exists \phi \in \gamma(? * |\mathsf{static}(\widetilde{\phi_1})| * |\mathsf{static}(\widetilde{\phi_2})|).\ \pi \vDash \phi\ \} \\
&= \{\ \pi \in \textsc{ProgramState} \mid \exists \phi \in \textsc{SfrmFormula}.\ \phi \Rightarrow |\mathsf{static}(\widetilde{\phi_1})| * |\mathsf{static}(\widetilde{\phi_2})| \ \wedge\ \pi \vDash \phi\ \} \\
&= \{\ \pi \in \textsc{ProgramState} \mid \exists \phi \in \textsc{SfrmFormula}.\ \phi \Rightarrow |\mathsf{static}(\widetilde{\phi_1})| \ \wedge\ \phi \Rightarrow |\mathsf{static}(\widetilde{\phi_2})| \ \wedge\ \pi \vDash \phi \\
&= \{\ \pi \in \textsc{ProgramState} \mid \exists \phi \in \textsc{SfrmFormula}.\ \phi \in \gamma(? * |\mathsf{static}(\widetilde{\phi_1})|) \ \wedge\ \phi \in \gamma(? * |\mathsf{static}(\widetilde{\phi_2})| \\
&= \{\ \pi \in \textsc{ProgramState} \mid \exists \phi \in \gamma(? * |\mathsf{static}(\widetilde{\phi_1})|) \cap \gamma(? * |\mathsf{static}(\widetilde{\phi_2})|).\ \pi \vDash \phi\ \} \\
&= \{\ \pi \in \textsc{ProgramState} \mid (\exists \phi \in \gamma(? * |\mathsf{static}(\widetilde{\phi_1})|).\ \pi \vDash \phi) \ \wedge\ (\exists \phi \in \gamma(? * |\mathsf{static}(\widetilde{\phi_2})|).\ \pi \vDash \phi)\ \} \\
&= \{\ \pi \in \textsc{ProgramState} \mid \exists \phi \in \gamma(? * |\mathsf{static}(\widetilde{\phi_1})|).\ \pi \vDash \phi\ \} \cap \{\ \pi \in \textsc{ProgramState} \mid \exists \phi \in \gamma(? \\
&= [\![? * |\mathsf{static}(\widetilde{\phi_1})|]\!] \cap [\![? * |\mathsf{static}(\widetilde{\phi_2})|]\!]
\end{aligned}
$$

□

The normal form is not only useful for the defining a non-separating conjunction but also provides efficient implementations for a lot of concepts. For example, gradual formula precision $\sqsubseteq$ is originally defined as comparing (possibly infinite) concretizations for inclusion. However, with above theorem the problem is reducible to checking an implication.

4.3. Gradual Static Semantics

Figure 4.13 shows a deterministic lifting of the Hoare logic (figure 4.9). It uses a number of helper functions that we define in section 4.3.1.

Lemma 4.14 (**$\mathbf{GVL_{IDF}}$**: 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 3.5.2) of the Hoare logic of* ***SVL_{IDF}*** *defined in section 4.1.6.*

Proof. Well-definedness: The rules are syntax directed and only can deduce at most one result per input.

Deterministic lifting: rule-wise HSkip: See lemma 3.40.

HAlloc, HFieldAssign, HReturn, HAssert: Composition of lifted components as defined in 4.3.1. □

A sound predicate lifting $\cdot \mathrel{\widetilde{\vDash}} \{\cdot\} \cdot \{\cdot\}$ can be derived using lemma 3.36.

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

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

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

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

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

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

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

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

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

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

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

Figure 4.13.. GVL: Gradual Hoare Logic

4.3.1. Lifted Components

In section 3.5.2.1 we established how inductively defined predicates can be lifted rule-wise and how composite rules can be lifted by lifting them piecewise and composing the resulting functions (lemma 3.39). We will thus identify the components the Hoare rules

are made of (e.g. implications and self-framing checks) and give sound deterministic liftings for them.

Implication of fixed static formula: $P_{\phi_t}(\phi_1, \phi_2) \overset{\mathsf{def}}{=} \phi_1 \Rightarrow \phi_t \wedge \phi_1 = \phi_2$

This predicate is not only a component of some rules but also the overall shape of HAssert.

Lemma 4.15.
Let $\vec{P}_{\phi_t} : \widetilde{\text{Formula}} \rightharpoonup \widetilde{\text{Formula}}$ *be defined as*

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

Then $\vec{P}_{\phi_t}$ *is an optimal deterministic lifting of* P_{ϕ_t}.

We define a more intuitive notation that corresponds closely with the original static predicate:

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

Formula extraction: $P_{\phi_t}(\phi_1, \phi_2) \overset{\mathsf{def}}{=} \phi_1 \Rightarrow \phi_t * \phi_2 \ \wedge \ \vdash_{\mathtt{sfrm}} \phi_2$

This predicate is a component of HFieldAssign, HCall, HHold and also is the overall shape of HRelease.

Lemma 4.16.
Let $\vec{P}_{\phi_t} : \widetilde{\text{Formula}} \rightharpoonup \widetilde{\text{Formula}}$ *be defined as*

$$\begin{aligned}\vec{P}_{\phi_t}(\phi_1) &= \max_{\Rightarrow}\{\ \phi_2 \in \text{SfrmFormula} \mid \phi_1 \Rightarrow \phi_t * \phi_2\ \} \\ \vec{P}_{\phi_t}(\mathsf{?} * \phi_1) &= \mathsf{?} * \vec{P}_{\phi_t}(\phi_1)\end{aligned}$$

Then $\vec{P}_{\phi_t}$ *is well-defined and an optimal deterministic lifting of* P_{ϕ_t}.

We define a more intuitive notation that reflects the dual nature with $*$:

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

We extend the domain of $\div$ to deal with gradual parameters:

$$\widetilde{\phi} \div (\mathsf{?} * \phi_t) \overset{\mathsf{def}}{=} \widetilde{\phi} \div \mathsf{static}(\widetilde{\phi})$$

Formally, this extension soundly lifts the function w.r.t. its second parameter.

Variable extraction: $P_x(\phi_1, \phi_2) \overset{\mathsf{def}}{=} \phi_1 \Rightarrow \phi_2 \ \wedge \ x \notin \mathsf{FV}(\phi_2) \ \wedge \ \vdash_{\mathtt{sfrm}} \phi_2$

This predicate is a component of HAlloc, HVarAssign, HReturn, HCall and in a sense HHold (the equality between the free variable sets can be reformulated as repeated application of P_x).

Lemma 4.17.
Let $\vec{P}_x : \widetilde{\text{Formula}} \rightharpoonup \widetilde{\text{Formula}}$ *be defined as*

$$\begin{aligned}\vec{P}_x(\phi_1) &= \max_{\Rightarrow}\{\ \phi_2 \in \text{SfrmFormula} \mid \phi_1 \Rightarrow \phi_2 \wedge x \notin \mathsf{FV}(\phi_2)\ \} \\ \vec{P}_x(\mathsf{?} * \phi_1) &= \mathsf{?} * \vec{P}_x(\phi_1)\end{aligned}$$

Then $\vec{P}_x$ *is well-defined and an optimal deterministic lifting of* P_x.

We extend the domain of the extraction function $\div$ to deal with variables:

$$\widetilde{\phi} \div x \stackrel{\mathsf{def}}{=} \vec{P}_x(\widetilde{\phi})$$

We further extend the function to deal with sets of variables element-wise

$$\begin{aligned} \widetilde{\phi} \div \emptyset &\stackrel{\mathsf{def}}{=} \widetilde{\phi} \\ \widetilde{\phi} \div (\{x\} \cup X) &\stackrel{\mathsf{def}}{=} (\widetilde{\phi} \div x) \div X \end{aligned}$$

4.4. 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. 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 $\widetilde{\text{ProgramStateFin}}$ with a designated exceptional state π_{EX}, representing dynamic verification failure. If no other derivation exists, we map to this state.

Lemma 4.18 ($\cdot \mathrel{\widetilde{\longrightarrow}} \cdot$ Well-Defined)**.** *The small-step semantics of* $\boldsymbol{GVL_{IDF}}$ *is well-defined.*

Proof. For $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot$ 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 ???). 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 $\widetilde{\text{SsCall}}$ 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 $\widetilde{\text{SsCallFinish}}$ is deterministic: H, ρ, x, A, A', s, S are forwarded from the input, v_r is uniquely determined by the premises. □

Lemma 4.19 ($\mathbf{GVL_{IDF}}$: Sound Lifting of Small-Step-Semantics)**.** *The lifting we propose is sound as defined in section 3.3.3.2.*

Proof. 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, which are not handled by those rules.

$\widetilde{\text{SsCall}}$ 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.

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

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

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

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

$\widetilde{\textsc{SsCallFinish}}$ Introduction For static postcondition, $\widetilde{\textsc{SsCallFinish}}$ is identical to SsCallFinish.

Monotonicity The return value of $\widetilde{\textsc{SsCallFinish}}$ 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. □

4.5. Gradual Soundness

Proposed gradual Hoare logic is sound w.r.t. the small-step semantics.

However, $\mathbf{GVL_{IDF}}$ also complies with the stronger notion of soundness GDPreservation. We will show how the small-step semantics $\cdot \longrightarrow \cdot$ and $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot$ realize the approach introduced in section 3.6.2.

Specifically, we show that for some statements, $\cdot \longrightarrow \cdot$ successfully executes the statement precisely iff Hoare logic is able to deduce information about the execution. For all composite statements (sequence, calls and hold), we prove by induction that GDPreservation holds for them as well.

5. Evaluation/Analysis

> E: with gradual typestates the same problem happened: as soon as the potential for unknown annotations was accepted, there was a "baseline cost" just to maintain the necessary infrastructure. With simple gradual types, it's almost nothing. With gradual effects, we've shown that it can boil down to very little (a thread-local variable with little overhead, see OOPSLA'15).

5.1. 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. precondition, postcondition, or even both) corresponds to an annotation of `?`. A dynamically verified language (say, Java) can be imitated using this process. Given such a language, the steps are thus as follows:

First one has to 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. One can then develop a sound Hoare logic that achieves that goal, including syntax extensions to the language that allow leveraging that Hoare logic. Gradualization can be applied to that system, introducing `?`as a "default formula", which allows making all newly introduced syntax extensions optional again. The syntax of the resulting gradually verified language should thus be a superset of the dynamically verified one. Contracts that were before explicitly realized as runtime checks can now be removed and encoded using the extended syntax.

6. Conclusion

Recap, remind reader what big picture was. Briefly outline your thesis, motivation, problem, and proposed solution.

6.1. Conceptual Nugget: Comparison/Implication to AGT!

6.2. Limitations and Future Work

After realizing that optimality of gradual liftings (or "consistent lifting", as called in AGT [7]) is likely impossible to achieve 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 3.37). We have also shown that a more expressive gradual syntax can reduce that problem, yet we have not formalized that relationship or determined what is necessary to preserve optimality.

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 holds:
>
> $$\{\texttt{(x = 10000)}\}\ s\ \{\texttt{(x = 4)}\}$$
>
> 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 (note that $\texttt{1 <= x} \land \texttt{x <= 4} \notin \gamma(\texttt{(x = 4)} \land \texttt{?})$).

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

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

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

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

is deducible.

We conjecture that a notion of consistency that relies on valid Hoare triples instead of deducible Hoare triples 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 semantics (from which validity of Hoare triples arises) instead of static semantics (Hoare logic).

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 (see [23] gives a 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.

A. Appendix

A.1. Proofs

Recall that gradual formulas $\texttt{?} * \phi_1$ and $\texttt{?} * \phi_2$ are considered equal iff $\gamma(\texttt{?} * \phi_1) = \gamma(\texttt{?} * \phi_1)$. The normal form makes use of the fact that concretizations contain only self-framed formulas.

Lemma: For any formula ϕ mentioning $x.f$:

$$\forall \hat{\phi} \in \gamma(\texttt{?} * \phi), \hat{\phi} \implies \texttt{acc}(x.f)$$

In other words: Merely mentioning a field will make sure that the concretization contains appropriate framing. This is a good starting point to justify removal of access-terms from the static part. Note, however, that just dropping all access from the static part may not result in an equivalent gradual formula for two reasons:

1. Mentioning
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 access to be restored by concretization.

2. Aliasing
In general there are different ways in which access to multiple fields can be restored (this is were linear logic plays in). Example: Dropping all access from `acc(a.f) * acc(b.f) * (a.f = 3) * (b.f = x)` results in `(a.f = 3) * (b.f = x)` which might be re-framed as `acc(a.f) * (a = b) * (a.f = 3) * (a.f = x)`. In other words, the possibility of aliasing may result in a variety of re-framed formulas that are not equivalent with the original one.

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 simply spelling out certain implications of the access-terms:

$\texttt{acc}(x.f) \implies (x.f = x.f)$
Access to a field implicitly guarantees that it actually evaluates to some (arbitrary) value. Note that $(x.f = x.f)$ is not a logical tautology (i.e. it is not

implies by `true`), since it indeed makes sure that $x.f$ evaluates, whereas `true` does not. The bottom line is that $x.f$ is being mentioned even after dropping `acc(`$x.f$`)`, therefore solving the first problem.

`acc(`$x.f$`) * acc(`$y.f$`)` $\implies$ `(`$x \neq y$`)`
: Having access to the same field of different expressions actively prevents those expressions to ever alias. Spelling out this restriction by adding the corresponding inequality also prevents re-framing in a way that relies on aliasing. The bottom line is that any valid re-framing must restore two distinct access-terms, therefore solving the second problem.

We enhance the non-linear part of our formula by spelling out above implications in every possible way, i.e. accounting for all (pairs of) access-terms. It is worth noting that this enhancement preserves equality of the formula as only terms are added that were implied by the original formula, anyway.

2. Delinearization
: All access-terms are dropped.

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] 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.

[3] 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.

[4] Yoonsik Cheon and Gary T Leavens. A runtime assertion checker for the java modeling language (jml). 2002.

[5] 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.

[6] David Crocker. Safe object-oriented software: the verified design-by-contract paradigm. In *Practical Elements of Safety*, pages 19–41. Springer, 2004.

[7] 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.

[8] Ronald Garcia and Eric Tanter. Deriving a simple gradual security language. *arXiv preprint arXiv:1511.01399*, 2015.

[9] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[10] 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.

[11] 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.

[12] K Rustan M Leino, Greg Nelson, and James B Saxe. Esc/java user's manual. *ESC*, 2000:002, 2000.

[13] Francesco Logozzo Manuel Fahndrich, Mike Barnett. Embedded contract languages. In *ACM SAC - OOPS*. Association for Computing Machinery, Inc., March 2010.

[14] Bertrand Meyer. *Design by contract*. Prentice Hall, 2002.

[15] 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.

[16] Greg Nelson. Extended static checking for java. In *International Conference on Mathematics of Program Construction*, pages 1–1. Springer, 2004.

[17] 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.

[18] 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.

[19] 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.

[20] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[21] 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.

[22] 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.

[23] 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.

[24] Matías Toro and Eric Tanter. Customizable gradual polymorphic effects for scala. In *ACM SIGPLAN Notices*, volume 50, pages 935–953. ACM, 2015.

[25] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming*, pages 459–483. Springer, 2011.