Contents

1 Introduction **3**
- 1.1 Motivational Examples . . . 5
  - 1.1.1 Argument Validation . . . 5
  - 1.1.2 Limitations of Static Verification . . . 6

2 Background **9**
- 2.1 Gradual Typing . . . 10
- 2.2 Hoare Logic . . . 10
- 2.3 Implicit Dynamic Frames . . . 10

3 Gradualization of a Statically Verified Language **13**
- 3.1 A Generic Statically Verified Language (SVL) . . . 13
- 3.2 Gradual Formulas . . . 16
  - 3.2.1 Dedicated Wildcard Formula . . . 17
  - 3.2.2 Wildcard with Upper Bound . . . 17
  - 3.2.3 Precision . . . 18
  - 3.2.4 Gradual Statements . . . 18
  - 3.2.5 Gradual Program State . . . 19
- 3.3 Lifting Predicates and Functions . . . 20
  - 3.3.1 Gradual Guarantee of Verification . . . 20
  - 3.3.2 Lifting Predicates . . . 20
  - 3.3.3 Lifting Functions . . . 23
  - 3.3.4 Generalized Lifting . . . 25
- 3.4 Gradual Soundness vs Gradual Guarantee . . . 25
- 3.5 Abstracting Static Semantics . . . 26
  - 3.5.1 The Problem with Composite Predicate Lifting . . . 26
  - 3.5.2 The Deterministic Approach . . . 28
- 3.6 Abstracting Dynamic Semantics . . . 33
  - 3.6.1 Perfect Knowledge . . . 34
  - 3.6.2 Partial Knowledge . . . 34

1 Introduction

Program verification aims to check a compute 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	Dynamic verification
Approach	The program is not executed. Instead **formal methods** (like Hoare logic or separation logic) are used, trying to derive a proof for given assertions.	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	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 be unable to prove 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.	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, but 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 arose from drawbacks of static and dynamic type systems which are very similar to the drawbacks identified above. From a theoretical perspective, type systems are even 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 provides the background of our approach, 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 do just that in form of case study in chapter **??**, applying the approach to a statically verified language that uses implicit dynamic in order to enable race-free static reasoning about mutable state.

```
~\\ % more detail???


Most modern programming languages use static methods to some degree, ruling out at least
%% static typing
Static typing disciplines are among the most common representatives, guaranteeing type sa
Yet, the rigidity and limitations of static type systems resulted in the introduction of
Casts (e.g. as implemented in C\# or Java) overrule purely static reasoning, allowing the
At this location, a runtime check is introduced, resulting in a cast exception should the
Note that such deviations from a purely static type system (one where there is no need fo
It is still guaranteed that execution does not enter an inconsistent state by simply inte

Note that casts are necessary only because of a typical drawback of static systems, namel
More sophisticated type systems (e.g. the one in Haskell) might have been able to deduce

%% dynamic typing
%At the other end of the spectrum are dynamically typed languages.
%In scenarios where the limitations of a static type system would clutter up the source c

%% static verification
In contrast, general purpose static verification techniques are not common amongst popula
Note that such languages are usually driven by cost-benefit and usability considerations,

%% static verification
% example?

% research Eiffel!
% Design-by-Contract!!! Eiffel!
```

```
% D even has both

% this is more of a consequence of the "deep roots" of dynamic verification!!!
%But even preconditions at expression level are implemented as runtime checks, reflected
%Examples:
%\begin{description}
%    \item[Division by zero]~\\
%    Integer division performs a dynamic check...
%
%\end{description}

-
What is the thesis about?
Why is it relevant or important?
What are the issues or problems?
What is the proposed solution or approach?
What can one expect in the rest of the thesis?

"Static verification checks that properties are always true, but it can be difficult and
```

1.1 Motivational Examples

1.1.1 Argument Validation

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

```
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 to those cases (throwing an exception), we might still 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 verifies 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 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)
}
```

Note however that such assertions are dropped from regular builds, meaning that the method contract is no longer verified!

With support of 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)
}
```

There are two basic ways to turn this annotation into a guarantee:

Static Verification (e.g. ESC/Java, see [12])
Verification will only succeed if the precondition is provable at all call sites. This is achievable in two ways:

- Rigorously annotate the call sites, guiding the verifier towards a proof.
- Add parameter validation to the call sites, effectively duplicating the original runtime check across the program. Note that this approach combines static and dynamic validation in order to get a performance benefit (no more runtime checks required where precondition was provable) and circumvent rigorous annotation. The drawback is of course code duplication.

Dynamic Verification (e.g. run JML4c, see [19]
This approach basically converts the annotation back into a runtime check equivalent to our original argument validation.

Gradual verification would pursue the combined approach without (visible) code duplication: Static verification is used where possible, dynamic verification where needed. Note that for the programmer this means that adding the method contract comes with no further obligations.

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 reinterpreted arbitrarily, adding appropriate runtime checks to guarantee that this reinterpretation 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 realized this concept in his programming language Eiffel, providing compiler support for generating runtime checks required for dynamic verification (often called runtime verification). Combining design-by-contract with static verification techniques to was investigated by [6] as what they call "verified design-by-contract".

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

A more recent programming language that comes with integrated support for specification and both static and dynamic verification is Spec# [15]. Its compiler facilitates theorem provers for static verification and emits runtime checks for dynamic verification. It was developed further with current challenges of concurrent object-orientation in mind [3]. The concepts found their way to main stream programming in the form of "Code Contracts" [13], a tool-set deeply integrated with the .NET framework and thus available in a variety of programming languages.

The limitations of both static and dynamic verification lead to a recent trend of using both approaches at the same time. Static verification is meant as a best effort service and supplemented with dynamic verification to give the guarantee that static verification potentially failed to provide. Recent work focuses 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. [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 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 combines as smart as possible) but instead treated as complementary and tightly coupled.

Note that [1] mentions 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 and closing in on fully static verification. A similar metric implicitly arises from our notion of gradual verification: The amount of dynamic checks injected to ensure compliance with annotations is a direct indicator of locations where static verification 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 [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 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 [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). It furthermore states that well typed programs evaluating to a value will evaluate to the same value when removing type annotations (dynamic part).

With "Abstracting Gradual Typing" (AGT) [7] Garcia, Clark and Tanter propose a new formal foundation for gradual typing. Their approach draws on the principles from abstract interpretation, defining a gradual type system in terms of an existing static one. The resulting system satisfies the gradual guarantee by construction. Subsequent work by Garcia and Tanter demonstrates the flexibility of AGT by applying the concept to 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:

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

The method contract can directly 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 orientation) is not possible using traditional Hoare logic. The following Hoare triple should not be 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 post-condition invalid. As we will demonstrate gradual verification on a Java-like language in chapter **??**, 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. The following Hoare triple would thus be verifiable:

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

Note also the changed syntax explicitly tracking the values of certain heap locations.

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 for heap references. Implicit dynamic frames (IDF) [22] addresses this issue by decoupling the concept of access to a certain heap location from assertions about its value. It introduces an "accessibility predicate" `acc(`*`loc`*`)` that represents the permission to access *loc*. Parkinson and Summers [17] worked out the formal relationship between separation logic and IDF. Above example can be rewritten in terms of IDF:

$$\{\texttt{acc(p1.age)} * \texttt{acc(p2.age)} * \texttt{(p1.age = 19)} * \texttt{(p2.age = 19)}\}$$

$$\texttt{p1.age++}$$

$$\{\texttt{acc(p1.age)} * \texttt{acc(p2.age)} * \texttt{(p1.age = 20)} * \texttt{(p2.age = 19)}\}$$

The separating conjunction makes sure that the accessibility predicates mention disjoint memory locations, whereas it has no further meaning for non-access predicates like equality. As formulas can mention heap locations in arbitrary predicates, it has to be ensured that the same formula contains accessibility predicates to all heap locations mentioned. This property of formulas is called "self-framing". Above pre- and postconditions 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 **??** are 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 **??**.

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. 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.

Gradual soundness section???

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:

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

We assume that there is a sequence operator ; such that:

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

Program State

Dynamic semantics (see below) are formalized as discrete transitions between program states. Therefore a program state contains all information necessary to evaluate expressions and determine the next program state. We assume that ProgramState is the set of all possible program states in SVL.

Examples:

Primitive language with integer variables

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

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 $\mathcal{S} : \textsc{ProgramState} \rightharpoonup \textsc{ProgramState}$, describing precisely how program state can be updated. Taking n steps at once can be abbreviated as $\mathcal{S}^n$ (undefinedness is propagated).

$\mathcal{S}^s \subseteq \textsc{ProgramState}_s \times \textsc{ProgramState}$

$\mathcal{S}^s(\pi_s, \pi) \quad \overset{\mathsf{def}}{\iff} \quad \exists n \in \mathbb{N}_+.\ \mathcal{S}^n(\pi_s) = \pi \;\wedge\; \pi$ is the first state after s is fully consumed

We further assume that there is a designated non-empty set $\textsc{ProgramStateFin} \subseteq \textsc{ProgramState}$ of states indicating regular or exceptional termination of the program. W.l.o.g. we assume $\mathsf{dom}(\mathcal{S}) \cap \textsc{ProgramStateFin} = \emptyset$, i.e. final states are stuck. We say a statement "throws an exception" if its small-step semantics transitions into an exceptional state.

Formula Semantics

Formulas are used for annotations like method contracts or invariants. Formally they constrain program states. 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.

We can derive a notion of satisfiability, implication and equivalence from this evaluation predicate.

Definition 3.1.1 (Formula Satisfiability)**.**
*A formula ϕ is **satisfiable** iff*

$$\exists \pi \in \textsc{ProgramState}.\ \pi \vDash \phi$$

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

Definition 3.1.2 (Formula Implication)**.**
*A formula ϕ_1 **implies** formula ϕ_2 (written $\phi_1 \Rightarrow \phi_2$) iff*

$$\forall \pi \in \textsc{ProgramState}.\ \pi \vDash \phi_1 \implies \pi \vDash \phi_2$$

Definition 3.1.3 (Formula Equivalence)**.**
*Two formulas ϕ_1 and ϕ_2 are **equivalent** (written $\phi_1 \equiv \phi_2$) iff*

$$\phi_1 \Rightarrow \phi_2 \quad \wedge \quad \phi_2 \Rightarrow \phi_1$$

Lemma 3.1.4 (Partial Order of Formulas)**.**
The implication predicate is a partial order on Formula.

We assume that there is a largest element `true` $\in$ Formula. 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:

$$\vDash \{\cdot\} \cdot \{\cdot\} \quad \subseteq \quad \textsc{Formula} \times \textsc{Stmt} \times \textsc{Formula}$$
$$\vDash \{\phi_{pre}\}\ s\ \{\phi_{post}\} \quad \overset{\mathsf{def}}{\iff} \quad \forall \langle \pi_{pre}, \pi_{post} \rangle \in \mathcal{S}^s.\ \pi_{pre} \vDash \phi_{pre} \implies \pi_{post} \vDash \phi_{post}$$

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

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

is derivable from given Hoare rules.

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

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

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 GFormula $\supset$ Formula with `?` $\in$ GFormula but `?` $\notin$ 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 $\widetilde{\phi} \in \text{GFormula}$ 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 3.2.1 (Concretization)**.**
Let $\gamma : \text{GFormula} \to \mathcal{P}(\text{SatFormula})$ *be defined as*

$$\gamma(\phi) = \begin{cases} \{\ \phi\ \} & \phi \in \text{SatFormula} \\ \emptyset & \textit{otherwise} \end{cases}$$
$$\gamma(\texttt{?}) = \text{SatFormula}$$

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 GFormula. In fact, this is only one possible way to realize GFormula. In the following two sections we will further analyze both this and a more powerful alternative.

We illustrate two typical ways of extending the formula syntax.

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:

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

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 3.2.2 (Concretization)**.**
Let $\gamma : \text{GFormula} \to \mathcal{P}(\text{SatFormula})$ *be defined as*

$$\gamma(\phi) = \begin{cases} \{\ \phi\ \} & \phi \in \text{SatFormula} \\ \emptyset & \textit{otherwise} \end{cases}$$
$$\gamma(\phi\ \wedge\ \texttt{?}) = \{\ \phi' \in \text{SatFormula} \mid \phi' \Rightarrow \phi\ \}$$

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

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

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

Definition 3.2.4 (Concretization of Gradual Statements)**.** *Let* $\gamma_s : \text{GSTMT} \to \mathcal{P}(\text{STMT})$ *be defined as*

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

Definition 3.2.5 (Precision of Gradual Statement)**.** *Let* $\sqsubseteq_s \subseteq \text{GSTMT} \times \text{GSTMT}$ *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 GSTMT, 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 $\text{GPROGRAMSTATE} \supseteq \text{PROGRAMSTATE}$ of gradual program states. Example:

$$\text{PROGRAMSTATE} \ = \ (\text{VAR} \rightharpoonup \mathbb{Z}) \ \times \ \text{STMT}$$

is extended to

$$\text{GPROGRAMSTATE} \ = \ (\text{VAR} \rightharpoonup \mathbb{Z}) \ \times \ \text{GSTMT}$$

We give meaning to gradual program states using concretization.

Definition 3.2.6 (Concretization of Gradual Program States)**.** *Let* $\gamma_\pi : \text{GPROGRAMSTATE} \to \mathcal{P}(\text{PROGRAMSTATE})$ *be defined as*

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

Definition 3.2.7 (Precision of Gradual Program States)**.** *Let* $\sqsubseteq_\pi \subseteq \text{GPROGRAMSTATE} \times \text{GPROGRAMSTATE}$ *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:

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

3.3 Lifting Predicates and Functions

The Hoare logic of SVL is a ternary predicate $\vdash \{\cdot\} \cdot \{\cdot\} \subseteq \textsc{Formula} \times \textsc{Stmt} \times \textsc{Formula}$. Since GVL contains gradual formulas and gradual statements, the gradualized Hoare logic is expected to have signature $\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\} \subseteq \textsc{GFormula} \times \textsc{GStmt} \times \textsc{GFormula}$. Similarly, the gradualized small-step semantics is expected to have signature $\widetilde{\mathcal{S}} : \textsc{GProgramState} \rightharpoonup \textsc{GProgramState}$ instead of $\mathcal{S} : \textsc{ProgramState} \rightharpoonup \textsc{ProgramState}$. 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 3.3.1 (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* GFormula, GStmt, GProgramState.

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

Definition 3.3.3 (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 $\widetilde{P} \subseteq \textsc{GFormula} \times \textsc{GFormula}$. 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 3.3.4 (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:

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

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 3.3.5 (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 3.3.6 (Equivalence with Consistent Predicate Lifting (AGT))**.** *Let* $\widetilde{P} \subseteq \textsc{GFormula} \times \textsc{GFormula}$ *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

For the following examples we assume that gradual formulas were defined as in section 3.2.1:

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

$$\gamma(\phi) = \begin{cases} \{\ \phi\ \} & \phi \in \textsc{SatFormula} \\ \emptyset & \textit{otherwise} \end{cases}$$
$$\gamma(?) = \textsc{SatFormula}$$

Lemma 3.3.7 (Optimal Lifting of Implication)**.**
Let $\cdot \mathrel{\widetilde{\Rightarrow}} \cdot \subseteq \textsc{GFormula} \times \textsc{GFormula}$ *be defined inductively as*

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

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

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

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

Lemma 3.3.8 (Optimal Lifting of Evaluation)**.**
Let $\cdot \mathrel{\widetilde{\vDash}} \cdot \subseteq \textsc{ProgramState} \times \textsc{GFormula}$ *be defined inductively as*

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

$$\frac{}{\pi \mathrel{\widetilde{\vDash}} ?}\ \textsc{GEvalGrad}$$

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

Note that the definition of lifted evaluation was lifted only w.r.t. the second parameter. While theoretically possible, 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).

Lemma 3.3.9 (Sound Lifting of Composite Predicate)**.**
Let $P, Q \subseteq \textsc{Formula} \times \textsc{Formula}$ *be arbitrary binary predicates. Let* $(P \circ Q) \subseteq \textsc{Formula} \times \textsc{Formula}$ *be defined as*

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

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

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

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

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

Lemma 3.3.10 (Sound Lifting of Conjunctive Predicate)**.**
Let $P, Q \subseteq$ Formula be arbitrary binary predicates. Let $(P \wedge Q) \subseteq$ Formula be defined as

$$(P \wedge Q)(\phi) \iff P(\phi) \wedge Q(\phi)$$

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

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

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 $(\widetilde{P \wedge Q})$.

Lemma 3.3.11 (Optimal Lifting of Disjunctive Predicate)**.**
Let $P, Q \subseteq$ Formula be arbitrary binary predicates. Let $(P \vee Q) \subseteq$ Formula be defined as

$$(P \vee Q)(\phi) \iff P(\phi) \vee Q(\phi)$$

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

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

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 $(\widetilde{P \vee Q})$.

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.

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

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. $\sqsubseteq$.

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

Definition 3.3.12 (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 $\widetilde{f}$: GFormula $\to$ GFormula constantly returning `?` is a sound lifting of any function f : Formula $\to$ Formula. This observation motivates an additional notion of optimality.

Definition 3.3.13 (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 3.3.14 (Equivalence with Consistent Function Lifting (AGT))**.**
Let $\alpha : \mathcal{P}(\text{SatFormula}) \rightharpoonup \text{GFormula}$ *be a partial function such that* $\langle \gamma, \alpha \rangle$ *is a* $\{f\}$*-partial Galois connection.*
Let $\widetilde{f} : \text{GFormula} \to \text{GFormula}$ *be defined as*

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

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

3.3.3.1 Examples

Assuming that the formula syntax of SVL contains a logic and operator $\wedge$, we can view it as a binary function on formulas.

Lemma 3.3.15 (Optimal Lifting of And)**.**
Let $\widetilde{\wedge} : \text{GFormula} \times \text{GFormula} \to \text{GFormula}$ *be defined as*

$$\begin{aligned}
\phi_1 \mathbin{\widetilde{\wedge}} \phi_2 &\stackrel{\text{def}}{=} \phi_1 \wedge \phi_2 \\
\phi \mathbin{\widetilde{\wedge}} ? &\stackrel{\text{def}}{=} \phi \wedge ? \\
? \mathbin{\widetilde{\wedge}} \phi &\stackrel{\text{def}}{=} \phi \wedge ? \\
? \mathbin{\widetilde{\wedge}} ? &\stackrel{\text{def}}{=} ?
\end{aligned}$$

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

Lemma 3.3.16 (Sound Lifting of Composed Function)**.**
Let $g, f : \text{Formula} \to \text{Formula}$ *be arbitrary functions.*
Let $\widetilde{(g \circ f)} : \text{GFormula} \to \text{GFormula}$ *be defined as*

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

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

3.3.3.2 Lifting Partial Functions

Semantics can be defined in terms of partial functions or even be a partial function as is the case for the small-step semantics of SVL. We derive rules for lifting partial functions using the following decomposition:

Lemma 3.3.17 (Partial Function Decomposition)**.** *Let* $f : \text{Formula} \rightharpoonup \text{Formula}$ *be a partial function. Then there exists a total function* $f' : \text{Formula} \to \text{Formula}$ *and a predicate* $F \subseteq \text{Formula}$ *such that*

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

Composing $\widetilde{f}$ from the gradual liftings of f's decomposition gives rise to the following rules for lifting partial functions.

Introduction

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

Monotonicity

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

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 GFormula instead of 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 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 $\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\} \subseteq \textsc{GFormula} \times \textsc{GStmt} \times \textsc{GFormula}$ (see section 3.5). Furthermore, function lifting is applied to the small-step semantics of SVL, resulting in gradual small-step semantics $\widetilde{\mathcal{S}} : \textsc{GProgramState} \to \textsc{GProgramState}$ (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

$$\widetilde{\vDash} \{\cdot\} \cdot \{\cdot\} \quad \subseteq \quad \textsc{GFormula} \times \textsc{GStmt} \times \textsc{GFormula}$$
$$\widetilde{\vDash} \{\widetilde{\phi_{pre}}\}\ \widetilde{s}\ \{\widetilde{\phi_{post}}\} \overset{\mathsf{def}}{\iff} \forall \langle \widetilde{\pi_{pre}}, \widetilde{\pi_{post}} \rangle \in \widetilde{\mathcal{S}^s}.\ \widetilde{\pi_{pre}} \widetilde{\vDash} \widetilde{\phi_{pre}} \implies \widetilde{\pi_{post}} \widetilde{\vDash} \widetilde{\phi_{post}}$$

Note that $\widetilde{\vDash} \{\cdot\} \cdot \{\cdot\}$ is not a sound gradual lifting of $\vDash \{\cdot\} \cdot \{\cdot\}$. 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) ∧ (y = 2)`$\}$ would lead to the injection of `assert (x = 3) ∧ (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

$$\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\} \quad \subseteq \quad \textsc{GFormula} \times \textsc{GStmt} \times \textsc{GFormula}$$

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:

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

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

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

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

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

$$\begin{aligned}&\widetilde{\phi_{q1}} \mathrel{\widetilde{\Rrightarrow}} \widetilde{\phi_{q2}}\\ &\widetilde{\vdash} \{\texttt{?}\}\ \texttt{y := 2}\ \{\widetilde{\phi_{q1}}\}\\ &\widetilde{\vdash} \{\widetilde{\phi_{q2}}\}\ \texttt{x := 3}\ \{\texttt{(x = 3) } \wedge \texttt{ (y = 2)}\}\end{aligned}$$

has to hold for some $\widetilde{\phi_{q1}}, \widetilde{\phi_{q2}} \in$ GFormula. 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 $\vec{\vdash} \{\cdot\} \cdot \{\cdot\} : \text{GFormula} \times \text{GStmt} \rightharpoonup \text{GFormula}$ which has very desirable properties and fits well into our existing model, as described later.

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 same rules can easily be adapted to different parameter types and higher arity.

Introduction

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

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

Strength

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

$$\forall \widetilde{\phi_1}, \widetilde{\phi_2} \in \text{GFormula}.\ \vec{P}(\widetilde{\phi_1}) = \widetilde{\phi_2}$$

$$\Longrightarrow$$

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

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: 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).

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

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

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

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

is a sound gradual lifting of P.

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.

(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 3.5.1 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.

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

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

Example: 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 $\widetilde{\vDash} \{?\}\ \texttt{assert}\ \phi_a\ \{\phi_a \wedge ?\}$ 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 3.5.2 (Composability of Deterministic Lifting)**.**
Let $\vec{P_1}, \vec{P_2}$ be sound deterministic liftings of predicates P_1, P_2. Then

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

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

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:

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

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

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

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

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

For the assignment rule we can derive

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

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

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

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

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 3.5.1 to deduce the gradual lifting.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{i} \notin \mathsf{FV}(\texttt{true})}{\texttt{i} \notin \mathsf{FV}(4)}}{\mathcal{D}_1}\ \textsc{DGHAssign1} \qquad \dfrac{\dfrac{\texttt{result} \notin \mathsf{FV}(\texttt{true} \wedge (\texttt{i = 4}))}{\texttt{result} \notin \mathsf{FV}(\texttt{i})}}{\mathcal{D}_2}\ \textsc{DGHAssign1}}{\mathcal{D}}\ \textsc{DGHSeq} \qquad \mathcal{I}}{\widetilde{\vdash} \{\texttt{true}\}\ \texttt{i := 4; result := i}\ \{(\texttt{result = 4})\}}\ \textsc{Lemma 3.5.1}$$

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

$$\dfrac{\dfrac{\dfrac{}{\mathcal{D}_1}\ \text{DGHAssign2} \qquad \dfrac{}{\mathcal{D}_2}\ \text{DGHAssign2}}{\mathcal{D}}\ \text{DGHSeq} \qquad \mathcal{I}}{\widetilde{\vdash}\ \{?\}\ \texttt{i := i + 1; result := i}\ \{\texttt{(result = 4)}\}}\ \text{Lemma 3.5.1}$$

where

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

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` $\wedge$ `(i = 4)` $\wedge$ `(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` $\wedge$ `(i = 4)` $\wedge$ `(result = i)` as static knowledge that is usable to formally guarantee the success of `assert (result = 4)`.

3.6 Abstracting Dynamic Semantics

Let $\widetilde{\mathcal{S}}$: GProgramState $\rightharpoonup$ GProgramState be a sound gradual lifting of $\mathcal{S}$: ProgramState $\rightharpoonup$ ProgramState.

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

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:

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

This stronger notion is 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.

When first introducing GDPreservation in section 3.5.2, we also gave an example illustrating that whether it holds or not can depend on the small-step semantics of GVL.

3.6.1 Perfect Knowledge

Let $\mathcal{S}' : \textsc{ProgramState} \rightharpoonup \textsc{ProgramState}$ be defined such that

$$\mathcal{S}'^{s}(\pi_1, \pi_2) \overset{\text{def}}{\iff} \mathcal{S}^{s}(\pi_1, \pi_2) \land \exists \phi_1, \phi_2 \in \textsc{Formula}.\ \vdash \{\phi_1\}\ s\ \{\phi_2\} \land \pi_1 \vDash \phi_1 \land \pi_2 \vDash \phi_2$$

Intuitively, $\mathcal{S}'$ is the same as $\mathcal{S}$, but is artificially made exceptional whenever there exists no Hoare rule that would be able to deduce static information about the execution.

Lemma 3.6.1 (Restricted Domain of Small-Step Semantics)**.** *Replacing the small-step semantics $\mathcal{S}$ of SVL with $\mathcal{S}'$ as defined above would have no observable effects.*

Let $\widetilde{\mathcal{S}} : \textsc{GProgramState} \to \textsc{GProgramState}$ be a total gradual lifting of $\mathcal{S}'$ that is optimal in its exceptional behavior:

$$\forall \widetilde{\pi}.\ (\forall \pi \in \gamma(\widetilde{\pi}).\ \mathcal{S}'(\pi) \in \textsc{ProgramStateEx}) \implies \widetilde{\mathcal{S}}(\widetilde{\pi}) \in \textsc{ProgramStateEx}$$

Theorem 3.6.2. *If the gradual small-step semantics $\widetilde{\mathcal{S}}$ of GVL are defined as above, then GVL satisfies* GDPreservation*.*

3.6.2 Partial Knowledge

$\mathcal{S}'$ as defined previously requires a priori knowledge about whether Hoare logic can derive a proof for arbitrarily complex statements. In SVL the sequence operator is key to defining more complex statements.

Assume $\mathcal{S}'$ is artificially made exceptional as described in the previous section, but not for sequences. Example: If $\mathcal{S}^{s_1,s_2}(\pi_1, \pi_2)$ holds, then $\mathcal{S}'^{s_1,s_2}(\pi_1, \pi_2)$ may hold even if there exists no proof.

GDPreservation still holds:

$$\dfrac{\dfrac{\dfrac{\vec{\vdash}\ \{\widetilde{\phi_1}\}\ \widetilde{s_1};\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}{\vec{\vdash}\ \{\widetilde{\phi_1}\}\ \widetilde{s_1}\ \{\widetilde{\phi_2}\} \qquad \vec{\vdash}\ \{\widetilde{\phi_2}\}\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}\ \textsc{inversion}}{\widetilde{\vDash}\ \{\widetilde{\phi_1}\}\ \widetilde{s_1}\ \{\widetilde{\phi_2}\} \qquad \widetilde{\vDash}\ \{\widetilde{\phi_2}\}\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}\ \textsc{GSoundness}}{\widetilde{\vDash}\ \{\widetilde{\phi_1}\}\ \widetilde{s_1};\ \widetilde{s_2}\ \{\widetilde{\phi_3}\}}\ \textsc{seq}$$

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. *ACM SIGPLAN Notices*, 51(1):429–442, 2016.

[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.