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

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

Reviewer: Prof. Dr.-Ing. Gregor Snelting, Karlsruhe Institute of Technology - Karlsruhe, Germany

Advisors: Assoc. Prof. Jonathan Aldrich, Carnegie Mellon University - Pittsburgh, USA
Assoc. Prof. Éric Tanter, University of Chile - Santiago, Chile

Duration: 2016-05-10 – 2016-09-28

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, 2016-09-??

. . . .

(**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.

Also I am very grateful to all my family and friends who encouraged and supported me during .

Contents

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 4, 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.

```java
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.

```java
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 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.

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

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. As we will demonstrate gradual verification on a Java-like language in chapter 4, 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 "ordinary" predicates like equality. 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 by ETH Zürich.

The static semantics of our example language in chapter 4 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 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. With the necessary tools for gradualization available, we apply them to the static semantics of SVL in section 3.4. 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

Operational 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.

Formula Semantics

Formulas are used to describe 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 $\phi_1 \underset{\phi}{\Longrightarrow} \phi_2$) 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*

$$\phi_1 \underset{\phi}{\Longrightarrow} \phi_2 \quad \wedge \quad \phi_2 \underset{\phi}{\Longrightarrow} \phi_1$$

Lemma 3.1.4 (Partial Order of Formulas).
The implication predicate is a partial order on $\textsc{Formula}$.

We assume that there is a largest element `true` $\in \textsc{Formula}$. 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 $\textsc{SatFormula}$, without explicitly stating that the presence of an unsatisfiable formula would result in failure.

Dynamic Semantics

We assume that there is a small-step semantics $\mathcal{S} \subseteq \textsc{ProgramState} \rightharpoonup \textsc{ProgramState}$ describing precisely how program state can be updated. Taking n steps at once can be abbreviated as $\mathcal{S}^n$, where undefinedness is propagated.

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

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

We further 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 $\mathsf{dom}(\mathcal{S}) \cap \textsc{ProgramStateFin} = \emptyset$, e.g. final states are stuck. Optionally, there may be a subset $\textsc{ProgramStateEx} \subseteq \textsc{ProgramStateFin}$ of states indicating exceptional termination of the program. To simplify reasoning about exceptional states, we assume

$$\forall \pi_X \in \textsc{ProgramStateEx}, \phi \in \textsc{Formula}.\ \pi_X \vDash \phi$$

and something with special statement set?

With this semantics we can formalize the notion of valid Hoare triples:

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

$$\vDash \{\phi_{pre}\}\ s\ \{\phi_{post}\} \quad \overset{\mathsf{def}}{\Longleftrightarrow} \quad \forall \langle \pi_{pre}, \pi_{post} \rangle \in \mathcal{S}^s.\ \pi_{pre} \vDash \phi_{pre} \implies \pi_{post} \vDash \phi_{post}$$

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

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

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:

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

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

Definition 3.1.5 (Weakest Static Precondition).
Let $\mathsf{wsp} : \textsc{Stmt} \to \mathcal{P}(\textsc{ProgramState})$ *be defined as*

$$\mathsf{wsp}(s) = \{\ \pi \in \textsc{ProgramState}_s \mid \exists \phi_1, \phi_2 \in \textsc{Formula}.\ \ \vdash \{\phi_1\}\ s\ \{\phi_2\}\ \wedge\ \pi \vDash \phi_1\ \}$$

Intuitively, the $\mathsf{wsp}(s)$ is a predicate on program states, indicating whether we could deduce anything about the state after executing s, using only our Hoare rules.

Example:

- Given that

$$\frac{}{\vdash \{\phi[e/x]\}\ x\ \texttt{:=}\ e\ \{\phi\}}\ \textsc{HoareAssign}$$

is the only Hoare rule for assignment, it follows that

$$\mathsf{wsp}(x\ \texttt{:=}\ e) = \textsc{ProgramState}$$

- Given that

$$\frac{\phi \underset{\phi}{\Longrightarrow} \phi_a}{\vdash \{\phi\}\ \texttt{assert}\ \phi_a\ \{\phi\}}\ \textsc{HoareStaticAssert}$$

is the only Hoare rule for assertions, it follows that

$$\mathsf{wsp}(\texttt{assert}\ \phi_a) = \{\ \pi \in \textsc{ProgramState} \mid \pi \vDash \phi_a\ \}$$

Soundness

We expect that given static semantics are sound w.r.t. given dynamic semantics.

$$\frac{\pi \in \mathsf{wsp}(s_1)}{\exists n \in \mathbb{N}_+,\ s_2 \in \textsc{Stmt}.\ \mathcal{S}^n(\pi) \in \textsc{ProgramState}_{s_2}}\ \textsc{Progress}$$

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

3.2 Gradual Formulas

We introduce the concepts of gradual verification by introducing a wildcard formula `?` into the formula syntax, resulting in a new set of gradual formulas G$\textsc{Formula}$. There are different ways to introduce the wildcard, we will describe two common options in the following sections.

Note that we want to strictly extend the existing formula syntax in order to maintain compatibility with the static system, i.e. $\textsc{Formula} \subset \textsc{GFormula}$ holds. This design goal ensures that any program considered syntactically valid by the static system will still be syntactically valid in the gradual system (motivated by gradual guarantee 2.1).

We decorate formulas $\widetilde{\phi} \in \textsc{GFormula}$ to distinguish them from formulas drawn from $\textsc{Formula}$. Using the concept of abstract interpretation, we want to reason about gradual formulas by mapping them back to a set of satisfiable static formulas (called "concretization") and then applying static reasoning to that set. Intuitively, a program state satisfies a gradual formula iff it satisfies (at least) on of the static formulas of the its concretization. (This intuition is formalized in section 3.3.2.)

Without knowing specifics of the syntax extension, we can already formalize this approach for static formulas:

Definition 3.2.1 (Concretization)**.**
Let $\gamma : \text{GF}\textsc{ormula} \to \mathcal{P}(\textsc{SatFormula})$ *be defined as follows:*

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

$$\gamma(\widetilde{\phi}) = \textit{to be defined when extending the syntax} \qquad \forall \phi \in \text{GF}\textsc{ormula}$$

There are two typical ways of extending the formula syntax.

3.2.1 Dedicated wildcard formula

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).

Since `?`is supposed to be a placeholder for an arbitrary formula, its concretization is defined as.

$$\gamma(\texttt{?}) = \textsc{SatFormula}$$

This approach is limited since programmers cannot express any additional static knowledge they might have. 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:

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

We pose $\texttt{?} \overset{\text{def}}{=} \texttt{true} \bigwedge \texttt{?}$.

We expect $\phi \bigwedge \texttt{?}$ to be a placeholder for a formula that also "contains" ϕ. There are two ways to express this containment, resulting in different concretizations.

Syntactic $\gamma_1(\phi \bigwedge \texttt{?}) = \{ \ \phi \wedge \phi' \mid \phi' \in \textsc{SatFormula} \ \}$

Semantic $\gamma_2(\phi \bigwedge \texttt{?}) = \{ \ \phi' \in \textsc{SatFormula} \mid \phi' \underset{\phi}{\Longrightarrow} \phi \ \}$

Lemma 3.2.2. $\forall \widetilde{\phi} \in \text{GF}\textsc{ormula}.\ \gamma_1(\widetilde{\phi}) \subseteq \gamma_2(\widetilde{\phi})$

Lemma 3.2.3. $\forall \widetilde{\phi} \in \text{GF}\textsc{ormula}.\ \gamma_1(\widetilde{\phi}) = \gamma_2(\widetilde{\phi})$ *modulo equivalence*

Note that $\gamma_1(\texttt{?}) = \gamma_2(\texttt{?}) = \textsc{SatFormula}$, meaning that this approach of extending the formula syntax is compatible with (but superior to) the approach introduced in the previous section.

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 still more precise than `?`. Using concretization, we can formalize this intuition.

Definition 3.2.4 (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 where formulas are even part of the syntax is the assertion statement `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 some "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

$$\begin{array}{l}\texttt{assert } \phi_{m_{pre}}\texttt{;} \\ \texttt{// body of } m \\ \texttt{assume } \phi_{m_{post}}\texttt{;}\end{array}$$

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 our desugared call statement is affected:

$$\begin{array}{l}\texttt{assert } \widetilde{\phi_{m_{pre}}}\texttt{;} \\ \texttt{// body of } m \\ \texttt{assume } \widetilde{\phi_{m_{post}}}\texttt{;}\end{array}$$

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

Definition 3.2.5 (Concretization of Gradual Statements)**.** *Let γ_s : GSTMT $\rightarrow \mathcal{P}$(STMT) be defined as*

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

Definition 3.2.6 (Precision of Gradual Statement)**.** *Let* $\sqsubseteq_s \subseteq \textsc{GStmt} \times \textsc{GStmt}$ *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 gradual static and dynamic semantics.

3.2.5 Gradual Program State

...continuation...

Therefore the introduction of gradual statements GStmt leads to a notion of gradual program states $\textsc{GProgramState} \supseteq \textsc{ProgramState}$. TODO: $\textsc{GProgramState}_{\widetilde{s}}$

Again, we give meaning to gradual program states using concretization.

Definition 3.2.7 (Concretization of Gradual Program States)**.** *Let* $\gamma_\pi : \textsc{GProgramState} \to \mathcal{P}(\textsc{ProgramState})$ *be defined as*

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

Definition 3.2.8 (Precision of Gradual Program States)**.** *Let* $\sqsubseteq_\pi \subseteq \textsc{GProgramState} \times \textsc{GProgramState}$ *be a predicate defined as*

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

Consequence:

$$\forall \widetilde{\pi_{\widetilde{s}}} \in \textsc{GProgramState}_{\widetilde{s}}, \pi \in \gamma_\pi(\widetilde{\pi_{\widetilde{s}}}).\ \exists s \in \gamma_s(\widetilde{s}).\ \pi \in \textsc{ProgramState}_s$$

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

$$\forall \phi \in \textsc{Formula}, \widetilde{\pi} \in \textsc{GProgramState}, \pi \in \gamma_\pi(\widetilde{\pi}).\ \ \widetilde{\pi} \vDash \phi \iff \pi \vDash \phi$$

3.3 Lifting Predicates and Functions

The Hoare logic of our language are defined in terms of predicates and functions that operate on formulas. Examples:

After introducing and giving meaning to gradual formulas, we will now describe how to redefine existing predicates and functions in order for them to deal with gradual formulas.

Definition 3.3.1 (Gradual Lifting)**.** *The process 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 *replaced by* GFormula*.*

Lifted predicates and functions are not allowed to deal with gradual formulas arbitrarily but must do so in a sound way. What soundness means is a direct consequence of the gradual guarantee (definition 3.3.2), i.e. an unsound predicate/function may cause the gradual verification system to break the gradual guarantee. This idea is formalized in the following sections.

How gradual parameters have to be handled is dictated by the gradual guarantee by Siek et al. [21]. , so we start with a formal definition of the gradual guarantee , reinterpreted from gradual type systems to gradual verification systems.

3.3.1 Gradual Guarantee of Verification

With the notion of precision, we can give a formal definition of the gradual guarantee (TODO: ref) that we are aiming to satisfy.

Definition 3.3.2 (Gradual Guarantee (for Gradual Verification Systems))**.**

PROBABLY UNNECESSARY:
Because of its generality, we will pursue the approach introduced in section 3.2.2 for the remainder of this chapter. As concretization we chose the semantic version, as it is more flexible than the syntactic one in practice. For reference, the full definitions:

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

Concretization:
$\gamma(\phi) = \{ \ \phi \ \} \qquad \forall \phi \in \textsc{SatFormula}$
$\gamma(\phi \wedge ?) = \{ \ \phi' \in \textsc{SatFormula} \mid \phi' \underset{\phi}{\Longrightarrow} \phi \ \}$
$\gamma(\widetilde{\phi}) = \emptyset \quad \textit{otherwise}$

3.3.2 Lifting Predicates

In this section, we assume that we are dealing with a binary predicate $P \subseteq \textsc{Formula} \times \textsc{Formula}$. The concepts are directly applicable to predicates with different arity or with additional non-formula parameters. The lifted version we are targeting has signature $\widetilde{P} \subseteq \text{G}\textsc{Formula} \times \text{G}\textsc{Formula}$. W.l.o.g. we further assume that P appears unnegated in the axiomatic semantics (otherwise we simply regard the negation of that predicate as P).

Rules emerging from the gradual guarantee:

Introduction

Having source code that is considered valid by the static verification system, the same source code must be considered valid by the gradual verification system. In other words, switching to the gradual system may never "break the code". This means that arguments satisfying P must satisfy $\widetilde{P}$:

$$\frac{P(\phi_1, \phi_2)}{\widetilde{P}(\phi_1, \phi_2)} \ \text{GP}\textsc{red}\text{I}\textsc{ntro}$$

Or equivalently, using set notation

$$P \subseteq \widetilde{P}$$

Monotonicity

A central point of a gradual verification system is enabling programmers to specify contracts with less precision. Source code that is rejected by the verifier might get accepted after reducing precision. If the opposite would happen, though, that would be highly counter-intuitive and ...??? workflow. To prevent such behavior, we expect satisfied predicates to still be satisfied after reducing the precision of arguments:

$$\frac{\widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2}) \qquad \widetilde{\phi_1} \sqsubseteq \widetilde{\phi'_1} \qquad \widetilde{\phi_2} \sqsubseteq \widetilde{\phi'_2}}{\widetilde{P}(\widetilde{\phi'_1}, \widetilde{\phi'_2})} \text{ GPredMon}$$

or equivalently, thinking of predicates as boolean functions

$$\widetilde{P} \text{ is monotonic w.r.t. } \sqsubseteq$$

or something with set terminology!???

$$\widetilde{P} \text{ is somewhat closed under weakening}$$

Definition 3.3.3 (Sound Predicate Lifting)**.** *A lifted predicate is* ***sound/valid*** *if it is closed under the above rules.*

Note that the rules for sound lifting only give a lower bound for the predicate. Thus $\widetilde{P} = \text{GFormula} \times \text{GFormula}$ is a sound predicate lifting of any binary predicate $P \subseteq \text{Formula} \times \text{Formula}$.

Definition 3.3.4 (Optimal Predicate Lifting)**.** *A sound lifted predicate is* ***optimal*** *if it is the smallest set closed under the above rules.*

This definition coincides with the definition of consistent predicate lifting in AGT:

Lemma 3.3.5 (Optimal Predicate Lifting (Direct Definition))**.** *Let* $\widetilde{P} \subseteq \text{GFormula} \times \text{GFormula}$ *be defined as*

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

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

Consistent lifting of common predicates:

Lemma 3.3.6 (Consistent Lifting of Evaluation)**.**
Let $\cdot \mathrel{\widetilde{\models}} \cdot \subseteq \text{ProgramState} \times \text{GFormula}$ *be defined as*

$$\pi \mathrel{\widetilde{\models}} \widetilde{\phi} \overset{\mathsf{def}}{\iff} \pi \models \mathsf{static}(()\phi)$$

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

We define $\text{SatGFormula} = \{\ \widetilde{\phi} \in \text{GFormula} \mid \exists \pi.\ \pi \mathrel{\widetilde{\models}} \widetilde{\phi}\ \}$ as the set of satisfiable gradual formulas.

Lemma 3.3.7 (Restricted Domain of Concretization)**.**
$\gamma|_{\text{SatGFormula}}$ *never returns the empty set.*

Lemma 3.3.8 (Consistent Lifting of Implication)**.**
Let $\cdot \underset{\phi}{\widetilde{\Longrightarrow}} \cdot \subseteq \text{GFormula} \times \text{GFormula}$ *be defined as*

$$\widetilde{\phi_1} \underset{\phi}{\widetilde{\Longrightarrow}} \widetilde{\phi_2} \overset{\mathsf{def}}{\iff} \exists \phi_1 \in \gamma(\widetilde{\phi_1}), \phi_2 \in \gamma(\widetilde{\phi_2}).\ \phi_1 \underset{\phi}{\Longrightarrow} \phi_2$$

Then $\cdot \underset{\phi}{\widetilde{\Longrightarrow}} \cdot$ *is a consistent lifting of* $\cdot \underset{\phi}{\Longrightarrow} \cdot$ *.*

3.3.3 Lifting Functions

Static verification rules may contain functions manipulating formulas. We can also derive rules for lifting such functions from the gradual guarantee. In this section, we assume that we are dealing with a function $f : \textsc{Formula} \to \textsc{Formula}$. Again, the concepts are directly applicable to functions with higher arity.

Restrictions imposed by the gradual guarantee:

Introduction

We ensure that our verification system is "immune" to reduction of precision. Thus, when passing a static formula ϕ to $\widetilde{f}$, the result must be the same or less precise than $f(\phi)$.

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

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. $\sqsubseteq$ (in every argument).

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

Definition 3.3.9 (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 $\widetilde{f} : \text{G}\textsc{Formula} \to \text{G}\textsc{Formula}$ constantly returning ? is a sound lifting of any function $f : \textsc{Formula} \to \textsc{Formula}$.

Definition 3.3.10 (Optimal Function Lifting)**.** *A sound lifted function is **optimal** if its return values are as precise as possible.*

This definition coincides with the definition of consistent function lifting in AGT:

Lemma 3.3.11 (Optimal Function Lifting (Direct Definition))**.**
Let $\alpha : \mathcal{P}(\text{Sat}\text{F}\textsc{ormula}) \rightharpoonup \text{G}\textsc{Formula}$ be a partial function such that $\langle \gamma, \alpha \rangle$ is a $\{f\}$-partial Galois connection.
Let $\widetilde{f} : \text{G}\textsc{Formula} \to \text{G}\textsc{Formula}$ be defined as

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

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

Examples

$$\alpha(\overline{\phi}) = \min_{\sqsubseteq} \{\ \widetilde{\phi} \mid \overline{\phi} \subseteq \gamma(\widetilde{\phi})\ \}$$

The logical and operator $\cdot \bigwedge \cdot$ of our formula syntax can be viewed as a binary function on formulas.

$$\widetilde{f}(\widetilde{\phi_1}, \widetilde{\phi_2}) = \alpha(\{\ \phi_1 \bigwedge \phi_2 \mid \phi_1 \in \gamma(\widetilde{\phi_1}) \wedge \phi_2 \in \gamma(\widetilde{\phi_2})\ \})$$

PARTIAL:

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

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

3.3.4 Generalized Lifting

The previous sections describe how lifting is performed in order to deal with $\text{GF}\textsc{ormula}$ 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 Abstracting Static Semantics

With the rules for lifting set up we can apply them to the static verification predicate: Lifting

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

w.r.t. all parameters yields

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

Optimality discussion:

```
{i = 10000}
n = collatzIterations(300, i);
{1 <= n * n <= 4}
{n = 4}
staticAssert (n = 4);
{n = 4}
```

...not verifiable with optimal lifting!

3.4.1 The Problem with a Predicate Lifting

As seen in section 3.5, the lifted Hoare predicate in general requires an additional assertion to guarantee preservation. Yet, there is a more fundamental design issue connected to the gradual lifting approach which we will illustrate in this section.

...rule-wise lifting yields overall lifting... neat.

Problem: non-deterministic! Compiler has to find “good” intermediate formulas

too weak could always choose ?

too strong could choose stuff that is not guaranteed by runtime... (so: inject runtime assertions? yes: could be wrong! no: could enter method violating precondition)

3.4.2 The Deterministic Approach

The approach we propose is based on the idea to treat the Hoare predicate as a (multivalued) function, mapping preconditions to the set of possible/verifiable postconditions. We can obtain a lifted version of this hypothetical construct and demand certain properties similar to the ones defined in section **??**:

Definition 3.4.1 (Deterministic Lifting)**.** *Given a binary predicate $P \subseteq \textsc{Formula} \times \textsc{Formula}$ we call a partial function $\vec{P} : \textsc{Formula} \rightharpoonup \textsc{Formula}$* ***deterministic lifting*** *of P if the following conditions are met:*

Introduction

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

Preservation

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

$$\Longrightarrow$$

$$\forall \phi_1 \in \gamma(\widetilde{\phi_1}), \phi_2 \in \text{F}\textsc{ormula}.\ P(\phi_1, \phi_2) \implies \exists \phi \in \gamma(\widetilde{\phi_2}).\ P(\phi_1, \phi) \ \wedge\ \phi \underset{\phi}{\Longrightarrow} \phi_2$$

Monotonicity

Note: Identical to monotonicity condition of lifted partial functions.

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

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

Obtaining a Sound Gradual Lifting

Lemma 3.4.2 (Deterministic Gradual Lifting)**.**
Let $\vec{P}$ be a deterministic lifting of P. Then

$$\widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2}) \quad \overset{\mathsf{def}}{\Longleftrightarrow} \quad \exists \widetilde{\phi_2'}.\ \vec{P}(\widetilde{\phi_1}) = \widetilde{\phi_2'} \wedge \widetilde{\phi_2'} \underset{\phi}{\widetilde{\Longrightarrow}} \widetilde{\phi_2}$$

is a sound gradual lifting of P.

Determinism

A verifier dealing with deterministic liftings has no more obligation of finding good intermediate formulas.

Preservation

A (gradual) postcondition returned by the lifted function is guaranteed to reflect the execution state after executing the statements in question (given that the precondition was met). Almost. Combines all the knowledge of static rules.

Composability

Lemma 3.4.3 (Composability of Deterministic Lifting)**.**
Let $\vec{P_1}, \vec{P_2}$ be deterministic liftings of predicates P_1, P_2. Then

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

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

3.5 Gradual Soundness vs Gradual Guarantee

Valid Hoare triples for gradual system

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

$$\widetilde{\models} \{\widetilde{\phi_{pre}}\}\ \widetilde{s}\ \{\widetilde{\phi_{post}}\} \quad \overset{\mathsf{def}}{\iff} \quad \forall \langle \widetilde{\pi_{pre}}, \widetilde{\pi_{post}} \rangle \in \widetilde{\mathcal{S}}^{\widetilde{s}}.\ \widetilde{\pi_{pre}}\ \widetilde{\models}\ \widetilde{\phi_{pre}} \implies \widetilde{\pi_{post}}\ \widetilde{\models}\ \widetilde{\phi_{post}}$$

(Note: NOT A sound gradual lifting! Would accept $\widetilde{\models}$ {?} `x := 3` {`(y = 4)` ∧ ?})

Soundness of gradual system:

$$\frac{\widetilde{\pi} \in \widetilde{\mathsf{wsp}}(\widetilde{s_1})}{\exists n \in \mathbb{N}_+,\ \widetilde{s_2} \in \textsc{GStmt}.\ \widetilde{\mathcal{S}}^n(\widetilde{\pi}) \in \textsc{ProgramState}_{\widetilde{s_2}}}\ \textsc{GProgress}$$

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

Gradual guarantee: Let $\widetilde{\vdash} \{\cdot\} \cdot \{\cdot\}$ be gradual lifting of $\vdash \{\cdot\} \cdot \{\cdot\}$. Then:

$$\begin{aligned} & \vdash \{\texttt{(x = 2)}\}\ \texttt{y := 3}\ \{\texttt{(x = 2)} \land \texttt{(y = 3)}\} \\ \xRightarrow{Introduction}\quad & \widetilde{\vdash} \{\texttt{(x = 2)}\}\ \texttt{y := 3}\ \{\texttt{(x = 2)} \land \texttt{(y = 3)}\} \\ \xRightarrow{Monotonicity}\quad & \widetilde{\vdash} \{?\}\ \texttt{y := 3}\ \{\texttt{(x = 2)} \land \texttt{(y = 3)}\} \end{aligned}$$

Preservation is obviously not satisfied!

Reiteration:

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

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

TODO: more bla, like "there is fundamentally no way around this - the programmer *can* specify postconditions that..."

3.6 Abstracting Dynamic Semantics

Let $\widetilde{\mathcal{S}}$ be gradual lifting of $\mathcal{S}$.

Progress: Note that premise is tautology. So we artificially make conclusion true by demanding that lifting is total. This always works since the lifting can be defined arbitrarily wherever the original function is undefined.

Preservation: Conclusion is already a tautology. This is not really satisfying: An arbitrary verification predicate would satisfy this kind of preservation. Also, this is no guarantee for all the formulas describing intermediate program states. A stronger notion of preservation gives this guarantee:

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

Making this guarantee work is trickier and there are different trade-offs available. Without further assumptions, $\overrightarrow{\vdash} \{\cdot\} \cdot \{\cdot\}$ is not a subset of $\widetilde{\models} \{\cdot\} \cdot \{\cdot\}$.

Running example:

$$\vec{\vdash} \{?\}\ \texttt{assert (x = 3)}\ \{\texttt{(x = 3)} \wedge ?\}$$

holds but not

$$\widetilde{\models} \{?\}\ \texttt{assert (x = 3)}\ \{\texttt{(x = 3)} \wedge ?\}$$

So far, our definition of $\widetilde{\mathcal{S}}$ as a total lifting of $\mathcal{S}$ may be to weak, breaking the subset relationship:

$\mathcal{S}$ too weak It is possible that the dynamic semantics of SVL defines

$$\mathcal{S}^{\texttt{assert (x = 3)}}(\pi_{\texttt{(x = 4)}}) = \pi'_{\texttt{(x = 4)}}$$

This is not unreasonable, since this function is guaranteed to be only called with "valid" program states in the static system! An additional runtime check would be overhead.

$\widetilde{\mathcal{S}}$ too weak If $\mathcal{S}^{\texttt{assert (x = 3)}}(\pi_{\texttt{(x = 4)}})$ is undefined due to runtime checks. Yet, the lifting is supposed to be total, so passing along the program state unchecked is again a valid realization:

$$\widetilde{\mathcal{S}}^{\texttt{assert (x = 3)}}(\pi_{\texttt{(x = 4)}}) = \pi'_{\texttt{(x = 4)}}$$

Mapping to an exception would have been better in this case.

Note that both problems are unrelated to optimality of the lifting.

3.6.1 Perfect Knowledge

Choose $\widetilde{\mathcal{S}} : \text{G}\textsc{ProgramState} \rightarrow \text{G}\textsc{ProgramState}$ as lifted version of $\mathcal{S} : \textsc{ProgramState} \rightharpoonup \textsc{ProgramState}$ with $\widetilde{\mathcal{S}}(\widetilde{\pi}) = \pi_{EX}$ if stuck for all concretizations.

$$\mathsf{wsp} : \text{G}\textsc{Stmt} \rightarrow \textsc{ProgramState}$$
$$\mathsf{wsp}(\widetilde{s}) \overset{\mathsf{def}}{=} \bigcup_{s \in \gamma_s(\widetilde{s})} \mathsf{wsp}(s)$$

$$\forall \widetilde{s} \in \text{G}\textsc{Stmt}.$$
$$\widetilde{\pi_{\widetilde{s}}} \in \text{G}\textsc{ProgramState}_{\widetilde{s}}.\ \mathsf{wsp}(\widetilde{s}) \cap \gamma_\pi(\widetilde{\pi_{\widetilde{s}}}) = \emptyset \implies \widetilde{\mathcal{S}}^{\widetilde{s}}(\widetilde{\pi_{\widetilde{s}}}) = \pi_{EX}$$

MINUS: - need above knowledge... - not always desirable

```
{i = 10000}
n = collatzIterations(300, i);
{1 <= n * n <= 4}
{n = 4}
staticAssert (n = 4);
{n = 4}
```

would throw exception!?

Proof:

$\widetilde{s} \in \text{GS}\textsc{tmt}$

$\widetilde{\phi_1}, \widetilde{\phi_2} \in \text{GF}\textsc{ormula}$

$\widetilde{\pi_1}, \widetilde{\pi_2} \in \text{GP}\textsc{rogram}\text{S}\textsc{tate}$

1 = Premise $\vec{\vdash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}$

2 = HoareIntrosA $\widetilde{\mathcal{S}}^{\widetilde{s}}(\widetilde{\pi_1}, \widetilde{\pi_2})$

3 = HoareIntrosB $\widetilde{\pi_1} \mathrel{\widetilde{\vDash}} \widetilde{\phi_1}$

4 = Case $\exists \pi_s \in \gamma(\widetilde{\pi_1}).\ \pi_s \in \mathsf{wsp}(s)$

5 = 4 + wsp def $\exists \phi_1', \phi' \in \text{F}\textsc{ormula}.\ \pi_s \vDash \phi_1' \ \wedge \ \vdash \{\phi_1'\}\ s\ \{\phi'\}$

6 = 4 + 5 + rule42 $\exists \phi_1 \in \gamma(\widetilde{\phi_1}).\ \phi_1 \underset{\phi}{\Longrightarrow} \phi_1' \wedge \pi_s \vDash \phi_1$

7 = 5 + 6 + mono $\exists \phi \in \text{F}\textsc{ormula}.\ \ \vdash \{\phi_1\}\ s\ \{\phi\}$

8 = 7 + intro $\exists \widetilde{\phi} \in \text{GF}\textsc{ormula}.\ \ \vec{\vdash} \{\phi_1\}\ s\ \{\widetilde{\phi}\}$

9 = 1 + 6 + 8 + mono_det_hoare $\widetilde{\phi} \sqsubseteq \widetilde{\phi_2}$

10 = 8 + pres $\exists \phi_2 \in \gamma(\widetilde{\phi}).\ \ \vdash \{\phi_1\}\ s\ \{\phi_2\}$

11 = 6 + 10 + snd $\mathcal{S}^s(\pi_s) \vDash \phi_2$

12 = 11 + intro $\widetilde{\mathcal{S}}^s(\pi_s) \vDash \phi_2$

13 = 3 + 12 + mono $\widetilde{\mathcal{S}}^{\widetilde{s}}(\pi_{\widetilde{s}}) \vDash \phi_2$

14 = 13 + intro $\widetilde{\mathcal{S}}^{\widetilde{s}}(\widetilde{\pi_{\widetilde{s}}}) \mathrel{\widetilde{\vDash}} \phi_2$

15 = 10 + 14 + mono $\widetilde{\mathcal{S}}^{\widetilde{s}}(\widetilde{\pi_{\widetilde{s}}}) \mathrel{\widetilde{\vDash}} \widetilde{\phi}$

16 = 9 + 15 + mono $\widetilde{\mathcal{S}}^{\widetilde{s}}(\widetilde{\pi_{\widetilde{s}}}) \mathrel{\widetilde{\vDash}} \widetilde{\phi_2}$

$\widetilde{s} \in \text{GS}\textsc{tmt}$

$\widetilde{\phi_1}, \widetilde{\phi_2} \in \text{GF}\textsc{ormula}$

$\widetilde{\pi_{\widetilde{s}}} \in \text{GP}\textsc{rogram}\text{S}\textsc{tate}_{\widetilde{s}}$

1 = PremiseA $\vec{\vdash} \{\widetilde{\phi_1}\}\ \widetilde{s}\ \{\widetilde{\phi_2}\}$

2 = PremiseB $\widetilde{\pi_{\widetilde{s}}} \mathrel{\widetilde{\vDash}} \widetilde{\phi_1}$

3 = Case $\neg\exists \pi_s \in \gamma(\widetilde{\pi_{\widetilde{s}}}).\ \pi_s \in \mathsf{wsp}(s)$

4 = 3 + completeness $\forall \pi_s \in \gamma(\widetilde{\pi_{\widetilde{s}}}).\ \mathcal{S}^s(\pi_s)\ \textit{stuck}$

5 = 4 + def $\widetilde{\mathcal{S}}^{\widetilde{s}}(\widetilde{\pi_{\widetilde{s}}}) = \pi_{EX}$

6 = 5 + precision $\widetilde{\mathcal{S}}^{\widetilde{s}}(\widetilde{\pi_{\widetilde{s}}}) \mathrel{\widetilde{\vDash}} \widetilde{\phi_2}$

3.6.2 Partial Knowledge

wsp not always known, think of sequence operator. Turns out we don't need it for sequence operator. Assume approach of previous section, but not for sequences. Preservation still holds:

$$\dfrac{\dfrac{\dfrac{\vec{\vdash}\ \{\widetilde{\phi_1}\}\ \tilde{s}_1\,;\ \ \tilde{s}_2\ \{\widetilde{\phi_3}\}}{\vec{\vdash}\ \{\widetilde{\phi_1}\}\ \tilde{s}_1\ \{\widetilde{\phi_2}\} \qquad \vec{\vdash}\ \{\widetilde{\phi_2}\}\ \tilde{s}_2\ \{\widetilde{\phi_3}\}}\ \textsc{inversion}}{\widetilde{\vDash}\ \{\widetilde{\phi_1}\}\ \tilde{s}_1\ \{\widetilde{\phi_2}\} \qquad \widetilde{\vDash}\ \{\widetilde{\phi_2}\}\ \tilde{s}_2\ \{\widetilde{\phi_3}\}}\ \text{GS}\textsc{oundness}}{\widetilde{\vDash}\ \{\widetilde{\phi_1}\}\ \tilde{s}_1\,;\ \ \tilde{s}_2\ \{\widetilde{\phi_3}\}}\ \textsc{seq}$$

4 Case Study: Implicit Dynamic Frames

4.1 Language

We now introduce a simplified Java-like statically verified language SVLidf that uses Chalice/Eiffel/Spec# sub-syntax to express method contracts.

4.1.1 Syntax

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

Figure 4.1. SVL: Syntax

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

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

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

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

Figure 4.2. SVL: Dynamic Footprint

4.1.2 Program State

The program state of SVL is defined as $\textsc{ProgramState} = \textsc{Heap} \times \textsc{Stack}$ with

$$\begin{aligned}
H &\in \textsc{Heap} && = \textsc{Loc} \rightharpoonup (\textsc{ClassName} \times (\textsc{FieldName} \rightharpoonup \textsc{Val})) \\
\rho &\in \textsc{VarEnv} && = \textsc{Var} \rightharpoonup \textsc{Val} \\
\Gamma &\in \textsc{TypeEnv} && = \textsc{Var} \rightharpoonup \textsc{Type} \\
A_s &\in \textsc{StaticFootprint} && = \mathcal{P}^{\textsc{Expr} \times \textsc{FieldName}} \\
A_d &\in \textsc{DynamicFootprint} && = \mathcal{P}^{\textsc{Loc} \times \textsc{FieldName}} \\
E &\in \textsc{StackEntry} && = \textsc{VarEnv} \times \textsc{DynamicFootprint} \times \textsc{Stmt} \\
S &\in \textsc{Stack} && ::= E \cdot S \mid \mathsf{nil}
\end{aligned}$$

REQUIRED?

Definition 4.1.1 (Topmost Stack Entry). *Let* $\mathsf{topmost} : \textsc{Stack} \rightharpoonup \textsc{StackEntry}$ *be defined as*

$$\begin{aligned}
&\mathsf{topmost}(E \cdot S) = E \\
&\mathsf{topmost}(\mathsf{nil}) \quad \textit{undefined}
\end{aligned}$$

Program states with scheduled statement s are defined as

$$\textsc{ProgramState}_s \overset{\mathsf{def}}{=} \textsc{Heap} \times \{\ (\rho, A_d, s) \cdot S \mid \rho \in \textsc{VarEnv},\ A_d \in \textsc{DynamicFootprint},\ S \in \textsc{Stack}\ \}$$

4.1.3 Formula Semantics

Framing

SVL uses the concepts of implicit dynamic frames to ensure that a statement can only access memory locations (more specifically: fields) which it is guaranteed to have exclusive access to. This is achieved by explicitly tracking access tokens `acc(<expression>.<field>)` as part of formulas throughout the entire program during verification.

The Hoare rules of SVL also make sure that access is never duplicated within or across stack frames, effectively ruling out concurrent access to any field during runtime.

Implicit dynamic frames also allows static reasoning about the values of fields during verification, i.e. as part of verification formulas. In order to guarantee that such formulas

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

$$\frac{}{H, \rho \vdash x \Downarrow \rho(x)}\ \textsc{EEVar}$$

$$\frac{}{H, \rho \vdash v \Downarrow v}\ \textsc{EEValue}$$

$$\frac{H, \rho \vdash e \Downarrow o}{H, \rho \vdash e.f \Downarrow H(o)(f)}\ \textsc{EEAcc}$$

Figure 4.3. SVL: Evaluating Expressions

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

$$\frac{}{H, \rho, A \vDash \texttt{true}}\ \textsc{EATrue}$$

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

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

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

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

Figure 4.4. SVL: Evaluating Expressions

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

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

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

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

Figure 4.5. SVL: Framing Expressions

always reflect the program state (preservation), formulas mentioning a certain field must also contain the access token to that very field:

Definition 4.1.2 (Self-Framing)**.** *A formula is **self-framing/self-framed** if it contains access to all fields it mentions.*

We omit the emptyset...

Definition 4.1.3 (Formula Self-Framedness)**.** *A formula ϕ is **self-framed** iff*

$$\vdash_{sfrm} \phi$$

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

As illustrated in example??? self-framed formulas are required for race-free verification.

SVL will thus only consider method contracts using self-framed and satisfiable formulas well-formed (see section 4.1.5).

4.1.4 Static Semantics

The static semantics of SVL consist of typing rules and a Hoare calculus 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 in the following typing rules.

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

$$\frac{}{A \vdash_{\texttt{sfrm}} \texttt{true}}\ \textsc{WFTrue}$$

$$\frac{A \vdash_{\texttt{frm}} e_1 \qquad A \vdash_{\texttt{frm}} e_2}{A \vdash_{\texttt{sfrm}} \texttt{(}e_1 \texttt{ = } e_2\texttt{)}}\ \textsc{WFEqual}$$

$$\frac{A \vdash_{\texttt{frm}} e_1 \qquad A \vdash_{\texttt{frm}} e_2}{A \vdash_{\texttt{sfrm}} \texttt{(}e_1 \texttt{ ≠ } e_2\texttt{)}}\ \textsc{WFNEqual}$$

$$\frac{A \vdash_{\texttt{frm}} e}{A \vdash_{\texttt{sfrm}} \texttt{acc(}e\texttt{.}f\texttt{)}}\ \textsc{WFAcc}$$

Figure 4.6. SVL: Framing Formulas

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

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

Figure 4.7. SVL: Static Footprint

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

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \textsc{STValNum}$$

$$\frac{}{\Gamma \vdash \texttt{null} : C}\ \textsc{STValNull}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \textsc{STVar}$$

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

Figure 4.8. SVL: Static Typing of Expressions

Typing

Verification

Let $\mathsf{wsp} : \textsc{Stmt} \to \mathcal{P}(\textsc{ProgramState})$ be defined as

$$\mathsf{wsp}(s) \ = \{\ \pi \in \textsc{ProgramState}_s \mid \exists \phi_1, \phi_2 \in \textsc{Formula},\ \Gamma \in \textsc{TypeEnv}.\ \Gamma \vdash \{\phi_1\}\ s\ \{\phi_2\} \quad \wedge \quad \pi \vDash \phi_1\ \}$$

$$\mathsf{wsp}(s) \ = \begin{cases} \textsc{ProgramState}_s & \text{if } s = x \texttt{ := new } C \\ \{\ \pi \in \textsc{ProgramState}_s \mid \pi \vDash \texttt{acc}(x.f)\ \} & \text{if } s = x.f \texttt{ := } y \\ \{\ \pi \in \textsc{ProgramState}_s \mid \pi \vDash \mathsf{acc(e)}\ \} & \text{if } s = x \texttt{ := } e \\ \textsc{ProgramState}_s & \text{if } s = \texttt{return } x \\ \{\ \pi \in \textsc{ProgramState}_s \mid \pi \vDash (y \ne \texttt{null}) * \mathsf{mpre}_p(m)\ \} & \text{if } s = x \texttt{ := } y.m(z) \\ \{\ \pi \in \textsc{ProgramState}_s \mid \pi \vDash \phi\ \} & \text{if } s = \texttt{assert } \phi \\ \{\ \pi \in \textsc{ProgramState}_s \mid \pi \vDash \phi\ \} & \text{if } s = \texttt{release } \phi \end{cases}$$

4.1.5 Well-Formedness

With static semantics in place, we can define what makes programs well-formed. Well-formedness is required to ... The following predicates

A program is well-formed if both its classes and main method are. For the main method to be well-formed, it must satisfy our Hoare predicate, given no assumptions.

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

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

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

4.1.6 Dynamic Semantics

4.1.7 Soundness

4.2 Gradualization

We will now follow along the procedure introduced in chapter 3 to design a gradually verified language "GVL " based on SVL.

The path we take:

Syntax:

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

Concretization:

$\gamma(\phi) = \{\ \phi\ \} \qquad \forall \phi \in \textsc{SfrmFormula}$

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

$\gamma(\widetilde{\phi}) = \emptyset \quad \textit{otherwise}$

4.2.1 Extension: Statements

In GVL we want the programmer to specify gradual method contracts. Therefore we extend their syntax as follows.

$$\widetilde{contract} \in \textsc{GContract} \qquad ::= \texttt{requires}\ \widetilde{\phi}\texttt{;}\ \texttt{ensures}\ \widetilde{\phi}\texttt{;}$$

This extension is propagated to method declarations (now accepting gradual contracts but not changing otherwise), yielding GMethod. Carrying on with the same logic, we get an extended set of class definitions GClass and finally an extended set of programs GProgram. Again, note that the only syntactical difference is the acceptance of gradual formulas in method contracts.

We see no motive to extend the syntax of statements themselves and define GStmt = Stmt. As postulated in section 3.2.4, the call statement hides away gradualized syntax by referencing a method with gradual contract. This becomes obvious when looking at its static or dynamic semantics (see HCall and ESCall???/ESCallFinish) where the method name is effectively dereferenced.

4.2.2 Extension: Program State

GProgramState = ProgramState

4.3 Gradualize Hoare Rules

4.4 Gradual Dyn. Semantics

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

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

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

$$\frac{\phi \underset{\phi}{\Longrightarrow} \texttt{acc(}x\texttt{.}f\texttt{)} * \phi' \qquad \vdash_{\texttt{sfrm}} \phi' \qquad \Gamma \vdash x : C \qquad \Gamma \vdash y : T \qquad \vdash C.f : T}{\Gamma \vdash \{\phi\}\ x\texttt{.}f\ \ \texttt{:=}\ y\ \{\phi' * \texttt{acc(}x\texttt{.}f\texttt{)} * \texttt{(}x\ \neq\ \texttt{null)} * \texttt{(}x\texttt{.}f\ \ \texttt{=}\ \ y\texttt{)}\}}\ \textsc{HFieldAssign}$$

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

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

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

$$\frac{\phi \underset{\phi}{\Longrightarrow} \phi'}{\Gamma \vdash \{\phi\}\ \texttt{assert}\ \ \phi'\ \{\phi\}}\ \textsc{HAssert}$$

$$\frac{\phi \underset{\phi}{\Longrightarrow} \phi_r * \phi' \qquad \vdash_{\texttt{sfrm}} \phi'}{\Gamma \vdash \{\phi\}\ \texttt{release}\ \ \phi_r\ \{\phi'\}}\ \textsc{HRelease}$$

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

$$\frac{\begin{array}{c}\vdash_{\texttt{sfrm}} \phi \qquad \phi_f \underset{\phi}{\Longrightarrow} \phi_r * \phi' \\ \phi' \underset{\phi}{\Longrightarrow} \phi \qquad FV(\phi') = FV(\phi) \qquad \neg\mathsf{writesTo}(FV(\phi), s) \qquad \Gamma \vdash \{\phi_r\}\ s\ \{\phi'_r\}\end{array}}{\Gamma \vdash \{\phi_f\}\ \texttt{hold}\ \ \phi\ \ \texttt{\{}\ \ s\ \ \texttt{\}}\ \{\phi'_r * \phi'\}}\ \textsc{HHold}$$

$$\boxed{(H, S) \rightarrow (H, S)}$$

$$\frac{}{(H, (\rho, A, \texttt{skip}s) \cdot S) \rightarrow (H, (\rho, A, s) \cdot S)} \ \textsc{ESSkip}$$

$$\frac{H, \rho \vdash x \Downarrow o \qquad H, \rho \vdash y \Downarrow v_y \qquad (o, f) \in A \qquad H' = H[o \mapsto [f \mapsto v_y]]}{(H, (\rho, A, x\,\texttt{.}\,f \ \ \texttt{:=} \ \ ys) \cdot S) \rightarrow (H', (\rho, A, s) \cdot S)} \ \textsc{ESFieldAssign}$$

$$\frac{H, \rho \vdash e \Downarrow v \qquad \rho' = \rho[x \mapsto v]}{(H, (\rho, A, x \ \ \texttt{:=} \ \ es) \cdot S) \rightarrow (H, (\rho', A, s) \cdot S)} \ \textsc{ESVarAssign}$$

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

$$\frac{H, \rho \vdash x \Downarrow v_x \qquad \rho' = \rho[\texttt{result} \mapsto v_x]}{(H, (\rho, A, \texttt{return} \ \ xs) \cdot S) \rightarrow (H, (\rho', A, s) \cdot S)} \ \textsc{ESReturn}$$

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

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

$$\frac{H, \rho, A \vDash \phi}{(H, (\rho, A, \texttt{assert} \ \ \phi s) \cdot S) \rightarrow (H, (\rho, A, s) \cdot S)} \ \textsc{ESAssert}$$

$$\frac{H, \rho, A \vDash \phi \qquad A' = A \backslash \lfloor \phi \rfloor_{H, \rho}}{(H, (\rho, A, \texttt{release} \ \ \phi s) \cdot S) \rightarrow (H, (\rho, A', s) \cdot S)} \ \textsc{ESRelease}$$

$$\frac{\rho' = \rho[x \mapsto \mathsf{defaultValue}(T)]}{(H, (\rho, A, T \ \ xs) \cdot S) \rightarrow (H, (\rho', A, s) \cdot S)} \ \textsc{ESDeclare}$$

$$\frac{H, \rho, A \vDash \phi \qquad A' = \lfloor \phi \rfloor_{H, \rho}}{(H, (\rho, A, \texttt{hold} \ \ \phi \ \ \texttt{\{} \ \ \overline{s'} \ \ \texttt{\}}s) \cdot S) \rightarrow (H, (\rho, A \backslash A', \overline{s'}) \cdot (\rho, A', \texttt{hold} \ \ \phi \ \ \texttt{\{} \ \ \overline{s'} \ \ \texttt{\}}s) \cdot S)} \ \textsc{ESHold}$$

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

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

$$\frac{\widetilde{\phi} \div \texttt{acc(}x\texttt{.}f\texttt{)} = \widetilde{\phi'} \qquad \Gamma \vdash x : C \qquad \Gamma \vdash y : T \qquad \vdash C.f : T}{\Gamma \overset{\rightarrow}{\vdash} \{\widetilde{\phi}\}\ x\texttt{.}f\ \ \texttt{:=}\ y\ \{\widetilde{\phi'} \mathbin{\widetilde{*}} \texttt{acc(}x\texttt{.}f\texttt{)} \mathbin{\widetilde{*}} \texttt{(}x\ \neq\ \texttt{null)} \mathbin{\widetilde{*}} \texttt{(}x\texttt{.}f\ \texttt{=}\ y\texttt{)}\}}\ \textsc{GHFieldAssign}$$

$$\frac{\widetilde{\phi} \underset{\phi}{\widetilde{\Longrightarrow}} \mathsf{acc}(e) \qquad \widetilde{\phi} \div x = \widetilde{\phi'} \qquad x \notin FV(e) \qquad \Gamma \vdash x : T \qquad \Gamma \vdash e : T}{\Gamma \overset{\rightarrow}{\vdash} \{\widetilde{\phi}\}\ x\ \ \texttt{:=}\ e\ \{\widetilde{\phi'} \mathbin{\widetilde{*}} \texttt{(}x\ \texttt{=}\ e\texttt{)}\}}\ \textsc{GHVarAssign}$$

$$\frac{\widetilde{\phi} \div \texttt{result} = \widetilde{\phi'} \qquad \Gamma \vdash x : T \qquad \Gamma \vdash \texttt{result} : T}{\Gamma \overset{\rightarrow}{\vdash} \{\widetilde{\phi}\}\ \texttt{return}\ \ x\ \{\widetilde{\phi'} \mathbin{\widetilde{*}} \texttt{(result = }x\texttt{)}\}}\ \textsc{GHReturn}$$

$$\frac{\begin{array}{c} \widetilde{\phi} \div x \div \lfloor\widetilde{\phi_p}\rfloor_{\Gamma,y,z'} = \widetilde{\phi'} \\ \Gamma \vdash y : C \qquad \mathsf{method}_p(C,m) = T_r\ \ m\texttt{(}T_p\ \ z\texttt{)}\ \texttt{requires}\ \phi_{pre}\texttt{;}\ \texttt{ensures}\ \phi_{post}\texttt{;}\ \texttt{\{ \_ \}} \\ \Gamma \vdash x : T_r \qquad \Gamma \vdash z' : T_p \qquad \widetilde{\phi} \underset{\phi}{\widetilde{\Longrightarrow}} \texttt{(}y\ \neq\ \texttt{null)} \mathbin{\widetilde{*}} \widetilde{\phi_p} \\ x \neq y \wedge x \neq z' \qquad \widetilde{\phi_p} = \widetilde{\phi_{pre}}[y, z'/\texttt{this}, z] \qquad \widetilde{\phi_q} = \widetilde{\phi_{post}}[y, z', x/\texttt{this}, z, \texttt{result}] \end{array}}{\Gamma \overset{\rightarrow}{\vdash} \{\widetilde{\phi}\}\ x\ \ \texttt{:=}\ y\texttt{.}m\texttt{(}z'\texttt{)}\ \{\widetilde{\phi'} \mathbin{\widetilde{*}} \widetilde{\phi_q}\}}\ \textsc{GHCall}$$

$$\frac{\widetilde{\phi'} \vdash \widetilde{\phi} \underset{\phi}{\widetilde{\Longrightarrow}} \phi_a}{\Gamma \overset{\rightarrow}{\vdash} \{\widetilde{\phi}\}\ \texttt{assert}\ \ \phi'\ \{\widetilde{\phi'}\}}\ \textsc{GHAssert}$$

$$\frac{\widetilde{\phi'} \vdash \widetilde{\phi} \underset{\phi}{\widetilde{\Longrightarrow}} \phi_r \qquad \widetilde{\phi'} \div \lfloor\phi_r\rfloor = \widetilde{\phi''}}{\Gamma \overset{\rightarrow}{\vdash} \{\widetilde{\phi}\}\ \texttt{release}\ \ \phi_r\ \{\widetilde{\phi'}\}}\ \textsc{GHRelease}$$

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

$$\frac{\begin{array}{c} \vdash_{\texttt{sfrm}} \phi \qquad \widetilde{\phi'_f} \vdash \widetilde{\phi_f} \underset{\phi}{\widetilde{\Longrightarrow}} \phi \qquad \widetilde{\phi'_f} \div \lfloor\phi\rfloor = \widetilde{\phi_r} \\ \widetilde{\phi'_f} \div \lfloor\widetilde{\phi_r}\rfloor \div FV(\widetilde{\phi'_f}) \backslash FV(\widetilde{\phi}) = \widetilde{\phi'} \qquad \neg\mathsf{writesTo}(FV(\phi), s) \qquad \Gamma \vdash \{\widetilde{\phi_r}\}\ s\ \{\widetilde{\phi'_r}\} \end{array}}{\Gamma \overset{\rightarrow}{\vdash} \{\widetilde{\phi_f}\}\ \texttt{hold}\ \ \phi\ \texttt{\{}\ \ s\ \ \texttt{\}}\ \{\widetilde{\phi'_r} \mathbin{\widetilde{*}} \widetilde{\phi'}\}}\ \textsc{GHHold}$$

$$\frac{\Gamma \overset{\rightarrow}{\vdash} \{\widetilde{\phi_p}\}\ s_1\ \{\widetilde{\phi_q}\} \qquad \Gamma \overset{\rightarrow}{\vdash} \{\widetilde{\phi_q}\}\ s_2\ \{\widetilde{\phi_r}\}}{\Gamma \overset{\rightarrow}{\vdash} \{\widetilde{\phi_p}\}\ s_1\texttt{;}\ \ s_2\ \{\widetilde{\phi_r}\}}\ \textsc{GHSeq}$$

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 an Unverified Language

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

no shared access...

6.3 Future Work

$$\mathsf{wlp}(\text{“}\texttt{x := a.f}\text{”}, \texttt{acc(b.f)}) = \begin{cases} \texttt{acc(b.f)} * \texttt{acc(a.f)} \\ \texttt{acc(b.f)} * \texttt{(a = b)} \end{cases}$$

7 Appendix

8 UNSORTED

8.1 HoareMotivEx

Hoare Logic as formal setting

```
class Point
{
    int manhattenDistance(Point p)
        requires \phi_{pre};
        ensures  \phi_{post};
    {
        s1;
        s2;
        .
        .
        .
    }
}
```

$$\texttt{this} : \texttt{Point}, \texttt{p} : \texttt{Point}, \texttt{result} : \texttt{int} \vdash \{\phi_{pre}\}\ s1; s2; ...\ \{\phi_{post}\}$$

8.2 NPC formula

Checking a formula at runtime, i.e. performing a runtime assertion check, is the integral part of dynamic verification and thus plays a role in gradual verification. Formally, a runtime assertion check corresponds to evaluating a closed formula since the environment provides an instantiation of the formula's free variables. It is reasonable to demand that this check can be performed in a time polynomial, if not linear to the formula's length (the specifics are up to the language designer, of course).

Such a requirement effectively restricts the formula syntax. For example, a syntax containing universal quantification generally violates above runtime limitations: A formula $\forall x_1 \in M, x_2 \in M, ..., x_n \in M.P(x_1, x_2, ..., x_n)$ would require $|M|^n$ steps to evaluate. As a result, the execution time is already exponential if M is finite – and unbounded otherwise.

Putting quantification (and therefore the introduction of new variables) aside, there are little restrictions to formula syntax, essentially allowing any predicates or operations that can be evaluated in linear/polynomial time. This includes equality/inequality relations, arithmetic and even own predicates that might be recursive to some extent.

Nevertheless such "easily" evaluable formulas are also subject to higher order reasoning in the static verification rules, including checks like satisfiability of or implication between formulas. Those judgments basically introduce quantification of the free variables, whereas evaluation works on a concrete instantiation. This makes static verification NP-hard in general:

NPC One can easily encode SAT instances as formulas, either directly (if the syntax covers boolean variables, conjunction and disjunction) or using arithmetic (if the syntax covers addition and a comparison relation like "greater-than"). Note that although evaluating such formulas is trivial, checking for satisfiability is NP-complete.

Undecidability ...Paeno-arithmetic

We chose the formula syntax of ... specifically to ensure that even static semantics are decidable in polynomial time. This allowed applying the procedures of AGT directly, as they are based on a decidable type system, i.e. decidable .

8.2.1 Impact of NP-hard Verification Predicates

Let's assume that our rules for static verification indeed contain an NP-hard predicate P. (NOTE: need positive occurrence for following reasoning!) The immediate consequence is that any working verifier would have to realize a conservative approximation of the actual predicate.

Under-approximation: for static guarantees to hold, verifier must under-approximate P... blabla

Over-approximation: for (det.) gradual lifting to be ?sound?, it must over-approximate P... blabla

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.