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

Formal verification using Hoare logic is a powerful tool to prove properties of imperative computer programs.
However, in practice programmers often face situations ... rigid... not flexible... - incomplete information about parts of the program - laziness, forced to annotate everything - unable to express due to limited syntax - unable to prove something facing undecidability
To counteract these limitations we introduce the notion of gradual formulas with an unknown part "?".
The main contribution of this work is presenting a gradual verification logic that covers the full range between completely unannotated programs and fully annotated programs. We prove the soundness of this logic and ... Siek et al. (2015).

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 any situation and every way, 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 throughout my life.

Contents

1 Introduction

Most modern programming languages use static analysis to some degree, ruling out certain types of runtime failure. Static analysis provides guarantees about the dynamic behavior of a program without actually running the program. Static typing disciplines are among the most common representatives of static analysis, guaranteeing type safety at compile time, obviating the need for dynamic checks.

Another powerful technique is static verification of programs against their specification, i.e. statically proving their "correctness". In practice this is achieved by checking that some annotated invariants or assertions (reflecting the specification) must always hold. Unfortunately, static verification has limitations and drawbacks:

- Syntax
- Decidability
- Difficult and Tedious to annotate programs
- ...

These limitations not only affect programmers trying to statically verify their program. Most general purpose programming languages (C/C++, C#, Java, ...), usually driven by cost-benefit and usability considerations, haven't adopted this level of static analysis in the first place.

The purpose of gradual verification is to weaken if not remove some of these limitations at the cost of turning some static checks into runtime checks, whenever inevitable. We will present a procedure of turning a static verification into a gradual one.

This idea is not new at all and actually common practice in type systems: In C# or Java, explicit type casts are assertions about the actual type of a value. This actual type (usually a subtype of the statically known type) could not be deduced by the static type system due to its limitations. Such an assertion/cast allows subsequent static reasoning about the value assuming its new type at the cost of an additional runtime check, ensuring the validity of the cast. Note that such deviations from a "purely" static type system (one where there is no need for runtime checks) do not affect type safety: It is still guaranteed that execution does not enter an invalid state (one where runtime types are incompatible with statically annotated types) by simply interrupting execution whenever a runtime type check fails. This is usually implemented by throwing an exception.

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 code, they allow expressing the same logic with less syntactic overhead, but at the cost of less static guarantees and early bug detection.

In terms of program verification, most general purpose languages are on the dynamic end of the spectrum. If they exist as designated syntax, assertions are usually implemented as runtime checks and often even dropped entirely for "release" builds (the Java compiler drops them by default). It is common practice to implement

A gradual type system is more flexible, as it provides the full continuum between static and dynamic typing, letting the programmer decide ... It can be seen as an extension "unknown" type

This work will also show that gradual verification ... other angle!

- 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 tedious to select a goal and to annotate programs for input to a static checker." (http://www.sciencedirect.com/

1.1 Motivation

1.1.1 As extension to unverified setting

Motivating example:

```
boolean hasLegalDriver(Car c)
{
    return c.driver.age >= 18;
}
```

Motivating example with argument validation:

```
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)
}
```

Motivating example with declarative approach (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 [12])

In the unlikely event that the verifier can prove the precondition at all call sites, our problem is solved. Otherwise, we have to enhance the call sites in order to convince the verifier. Choices:

- Add parameter validation, effectively duplicating the original runtime check across the program.
- Add further annotations, guiding the verifier towards a proof. This might not always work due to limitations of the verifier or decidability in general.

There are obvious limitations to this approach, static verification tends to be invasive. At least there is a performance benefit: Runtime checks (originally part of every call) are now only necessary in places where verification would not succeed otherwise.

Runtime Assertion Checking (e.g. run JML4c, TODO: http://www.cs.utep.edu/cheon/download/jml4

This approach basically converts the annotation back into a runtime check equivalent to our manual argument validation. It is therefore less invasive, not requiring further changes to the code, but also lacks the advantages of static verification.

1.1.2 As extension to fully verified setting

```
int collatzIterations(int iter, int start)
    requires 0 < start;
    ensures  0 < result;
{
    // ...
}

int myRandom(int seed)
    requires 0 < seed   && seed   < 10000;
    ensures  0 < result && result < 4;     // not provable
{
    int result = collatzIterations(300, seed);
    // we know: result ∈ { 1, 2, 4 }

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

Non-solution:

```
int collatzIterations(int iter, int start)
    requires 0 < start;
    ensures  0 < result;
{
    // ...
}

int myRandom(int seed)
    requires 0 < seed   && seed   < 10000;
    ensures  0 < result && result < 4;
{
    int result = collatzIterations(300, seed);
    // we know: result ∈ { 1, 2, 4 }

    // "cast"
    if (!(result < 5))
        throw new IllegalStateException("expected␣result␣<␣5");

    // verifier now knows:  0 < result && result < 5

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

```
}
```

This solution is not satisfying, - much to write, have to think about what to write (requires you to kind of thing from verifiers perspective) - intuitively the problem is with the method's postcondition being too weak, i.e. we solved the problem at the wrong place!

```
int collatzIterations(int iter, int start)
    requires 0 < start;
    ensures  0 < result && ?;
{
    // ...
}

int myRandom(int seed)
    requires 0 < seed   && seed   < 10000;
    ensures  0 < result && result < 4;
{
    int result = collatzIterations(300, seed);
    // we know: result ∈ { 1, 2, 4 }

    // verifier allowed to
    //  assume 0 < result && result < 5
    //  from   0 < result && ?
    // (adding runtime check)

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

2 Background

2.1 Categorization of existing stuff

[1] GraVy: metric of progress of the verification process and allows the verification engineer to focus on the remaining statements. Gradual verification is not a new static verification technique. It is an extension that can be applied to any existing static verification techniques to provide additional information to the verification engineer. Thus, issues, such as handling of loops or aliasing are not addressed in this paper. These are problems related to sound verification, but gradual verification is about how to make the use of such verification more traceable and quantifiable

[16] ESC/Java Software development and maintenance are costly endeavors. The cost can be reduced if more software defects are detected earlier in the development cycle. This paper introduces the Extended Static Checker for Java (ESC/Java), an experimental compile-time program checker that finds common programming errors. The checker is powered by verification-condition generation and automatic theorem proving techniques. It provides programmers with a simple annotation language with which programmer design decisions can be expressed formally. ESC/Java examines the annotated software and warns of inconsistencies between the design decisions recorded in the annotations and the actual code, and also warns of potential runtime errors in the code. This paper gives an overview of the checker architecture and annotation language and describes our experience applying the checker to tens of thousands of lines of Java programs.

[10] JML => static verification

[4] JML => RAC

[15] Spec#

[3] Spec# extension (concurrent OO)

[14] Design-by-Contract then also: Eiffel by Bertrand Meyer

[13] Code Contracts! Combines runtime and static checking

[6] “verified design-by-contract”

[5] = static verification plus directed dynamic verification In this paper, we present a technique to complement partial verification results by automatic test case generation. In contrast to existing work, our technique supports the common case that the verification results are based on unsound assumptions. We annotate programs to reflect which executions have been verified, and under which assumptions. These annotations are then used to guide dynamic symbolic execution toward unverified program executions. Our main technical contribution is a code instrumentation that causes dynamic symbolic execution to abort tests that lead to verified executions, to prune parts of the search space, and to prioritize tests that cover more properties that are not fully verified.

2.2 Hoare Logic

...for static semantics

[9]

2.3 Related Work

2.3.1 Abstracting Gradual Typing

[18] Gradual Typing for Functional Languages
apply their gradual typing approach to other areas

[19] Refined criteria for gradual typing gradual guarantee: The gradual guarantee says that if a gradually typed program is well typed, then removing type annotations always produces a program that is still well typed. Further, if a gradually typed program evaluates to a value, then removing type annotations always produces a program that evaluates to an equivalent value.

[7] AGT In this paper, we propose a new formal foundation for gradual typing, drawing on principles from abstract interpretation to give gradual types a semantics in terms of preexisting static types. Abstracting Gradual Typing (AGT for short) yields a formal account of consistency—one of the cornerstones of the gradual typing approach—that subsumes existing notions of consistency, which were developed through intuition and ad hoc reasoning.

[8] Abstracting Gradual Typing (AGT) is an approach to systematically deriving gradual counterparts to static type disciplines (Garcia et al. 2016). The approach consists of defining the semantics of gradual types by interpreting them as sets of static types, and then defining an optimal abstraction back to gradual types. These operations are used to lift the static discipline to the gradual setting. The runtime semantics of the gradual language then arises as reductions on gradual typing derivations. To demonstrate the flexibility of AGT, we gradualize a prototypical security-typed language with respect to only security labels rather than entire types, yielding a type system that ranges gradually from simply-typed to securely typed. We establish noninterference for our gradual language using Zdancewic's logical relation proof method. Whereas prior work presents gradual security cast languages, which require explicit security casts, this work yields the first gradual security source language, which requires no explicit casts.

prior to AGT [23] the language extends the notion of gradual typing to account for typestate: gradual typestate checking seamlessly combines static and dynamic checking by automatically inserting runtime checks into programs.

[2] develop a theory of gradual effect checking, which makes it possible to incrementally annotate and statically check effects, while still rejecting statically inconsistent programs. We extend the generic type-and-effect framework of Marino and Millstein with a notion of unknown effects, which turns out to be significantly more subtle than unknown types in traditional gradual typing. We appeal to abstract interpretation to develop and validate the concepts of gradual effect checking.

[22] Grad Effects in Scala, benchmarks on runtime impact!

2.3.2 Implicit Dynamic Frames

Race-free Assertion language! => static verification tool able to reason soundly about concurrent programs

[20] IDF

[11] Chalice, a verification methodology based on implicit dynamic frames

Chalice's verification methodology centers around permissions and permission transfer. In particular, a memory location may be accessed by a thread only if that thread has permission to do so. Proper use of permissions allows Chalice to deduce upper bounds on the set of locations modifiable by a method and guarantees the absence of data races for

concurrent programs. The lecture notes informally explain how Chalice works through various examples.

also: Viper (Verification Infrastructure for Permission-based Reasoning; is a suite of tools developed at ETH Zurich, providing an architecture on which new verification tools and prototypes can be developed simply and quickly.) has Chalice as front-end

[21] In this paper, we provide both an isorecursive and an equirecursive formal semantics for recursive definitions in the context of Chalice

[17] VERY IMPORTANT: chapter 2.2

Finally, we show that we can encode the separation logic fragment of our logic into the implicit dynamic frames fragment, preserving semantics. For the connectives typically supported by tools, this shows that separation logic can be faithfully encoded in a first-order automatic verification tool (Chalice).

Although IDF was partially inspired by separation logic, there are many differences between SL and IDF that make understanding their relationship difficult. SL does not allow expressions that refer to the heap, while IDF does. SL is defined on partial heaps, while IDF is defined using total heaps and permission masks. The semantics of IDF are only defined by its translation to first-order verification conditions, while SL has a direct Kripke semantics for its assertions.

3 Gradualization of a Statically Verified Language

As illustrated earlier gradual verification can be seen as an extension of both static and dynamic verification. Yet, our approach of "gradualization" formalizes the introduction of the dynamic aspect into a fully static system. Thus, this uses a statically verified language as starting point. Later we will show how a programming language without static verification can be approached.

3.1 A Generic Statically Verified Language (GSVL)

Requirements:

Syntax

We assume the existence of at least the following two syntactic categories:

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

We write $\textsc{Stmts}$ for the set of lists of instructions. ...and use $\bar{\cdot}$ to decorate variables representing lists of things.

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 current program state.

We assume that we are given a 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 Formula.

We assume that there is a largest element `true` $\in$ Formula. Note that the presence of an unsatisfiable formula (as invariant, pre-/postcondition, assertion, ...) in a sound verification system implies that the corresponding source code location is unreachable: Preservation guarantees that any reachable program state satisfies potentially annotated formulas, trivially implying that the formula is satisfiable.

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

Static Semantics

We assume that there is a Hoare logic (HL)

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

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.

Notion of Well-Formedness

Dynamic Semantics

We assume that there is a dynamic semantics (e.g. small-step) describing precisely whether and how program state is updated. This is required mainly for reasoning about soundness of the static semantics.

Soundness

We expect that given static semantics are sound w.r.t. given dynamic semantics. This means that programs satisfying ...

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 GFormula. 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. Formula $\subset$ 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 2.3.1).

We decorate formulas $\widetilde{\phi} \in$ GFormula to distinguish them from formulas drawn from 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 3.3.1.)

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

Definition 3.2.1 (Concretization)**.**
Let γ : GFormula $\to \mathcal{P}$(SatFormula) *be defined as follows:*

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

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

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.

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

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:

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

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

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

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

Semantic $\gamma_2(\phi \wedge \texttt{?}) = \{\ \phi' \in \textsc{SatFormula} \mid \phi' \implies \phi\ \}$

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

Lemma 3.2.3. $\forall \widetilde{\phi} \in \textsc{GFormula}.\ \gamma_1(\widetilde{\phi}) = \gamma_2(\widetilde{\phi})$ *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`, $x = 3 \wedge \texttt{?}$, `?`) gives rise to a notion of "precision". Intuitively, `x = 3` is more precise than $x = 3 \wedge \texttt{?}$ 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.

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

Definition 3.2.5 (Gradual Guarantee (for Gradual Verification Systems))**.**

3.2.4 Conclusion

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

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

3.3 Lifting Predicates and Functions

The axiomatic semantics of our language are (PROBABLY!?) 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 creating a "gradual version"/"lifted version" of a predicate/function. The resulting predicate/function has the same signature as the original one, with occurrences of* Formula *replaced by* GFormula*.*

The more important question is of course how to define such lifted versions in a consistent way. What consistency means is a direct consequence of the gradual guarantee (definition 3.2.5), i.e. an inconsistently lifted predicate/function may cause the gradual verification system to break the gradual guarantee. The specifics are described in the following sections.

3.3.1 Lifting Predicates

In this section, we assume that we are dealing with a binary predicate $P \subseteq \text{Formula} \times \text{Formula}$. The concepts are directly applicable to predicates with different arity or with additional non-formula parameters. The lifted version we are targeting has signature $\widetilde{P} \subseteq \text{GFormula} \times \text{GFormula}$. 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 $\widetilde{P}$:

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

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 3.3.2 (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 3.3.3 (Optimal Predicate Lifting)**.** *A sound lifted predicate is* ***consistent/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 3.3.4 (Consistent Predicate Lifting (Direct Definition))**.** *Let* $\widetilde{P} \subseteq \text{GFormula} \times \text{GFormula}$ *be defined as*

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

Then $\widetilde{P}$ *is the only consistent lifting of* P*.*

Consistent lifting of common predicates:

Lemma 3.3.5 (Consistent Lifting of Evaluation)**.**
Let $\cdot \mathrel{\widetilde{\models}} \cdot \subseteq \text{ProgramState} \times \text{GFormula}$ *be defined as*

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

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

Lemma 3.3.7 (Consistent Lifting of Implication)**.**
Let $\cdot \mathrel{\widetilde{\underset{\phi}{\Longrightarrow}}} \cdot \subseteq \text{GFormula} \times \text{GFormula}$ *be defined as*

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

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

3.3.2 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).

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

Definition 3.3.8 (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 $\widetilde{f} : \textsc{GFormula} \to \textsc{GFormula}$ constantly returning `?` is a sound lifting of any function $f : \textsc{Formula} \to \textsc{Formula}$.

Definition 3.3.9 (Optimal Function Lifting)**.** *A sound lifted function is **consistent/optimal** if its return values are as precise as possible.*

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

Lemma 3.3.10 (Consistent Function Lifting (Direct Definition))**.**
Let $\alpha : \mathcal{P}(\textsc{SatFormula}) \rightharpoonup \textsc{GFormula}$ be a partial function such that $\langle \gamma, \alpha \rangle$ is a $\{f\}$-partial Galois connection.
Let $\widetilde{f} : \textsc{GFormula} \to \textsc{GFormula}$ be defined as

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

Then $\widetilde{f}$ is the only consistent 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:

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

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

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. both pre- and postcondition results in a predicate

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

The remainder of this chapter will show how to tackle lifting the verification predicate in practice. For this detailed description, we assume that ...

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

$$\frac{\vdash \{\phi_p\}\ \overline{s_1}\ \{\phi_{q1}\} \qquad \phi_{q1} \implies \phi_{q2} \qquad \vdash \{\phi_{q2}\}\ \overline{s_2}\ \{\phi_r\}}{\Gamma \vdash \{\phi_p\}\ \overline{s_1};\ \overline{s_2}\ \{\phi_r\}}\ \textsc{VerSeqFlexible}$$

$$\begin{aligned}
\mathcal{H}_s &: \textsc{SfrmFormula} \rightharpoonup \textsc{SfrmFormula} \\
\mathcal{H}_s &= \mathcal{M}_s \circ \mathcal{I}_s \circ \mathcal{F}_s \\
\mathcal{H}_s(\phi^\circ) &= \mathcal{M}_s(\phi^\circ) \quad \text{if } \mathcal{P}(s) \wedge \phi^\circ \implies \mathcal{J}(s) \\
\mathcal{F}_s &: \textsc{SfrmFormula} \rightharpoonup \textsc{SfrmFormula} \\
\mathcal{F}_s(\phi^\circ) &= \phi^\circ \quad \text{if } \mathcal{P}(s) \\
\mathcal{I}_s &: \textsc{SfrmFormula} \rightharpoonup \textsc{SfrmFormula} \\
\mathcal{I}_s(\phi^\circ) &= \phi^\circ \quad \text{if } \phi^\circ \implies \mathcal{J}(s) \\
\mathcal{M}_s &: \textsc{SfrmFormula} \to \textsc{SfrmFormula}
\end{aligned}$$

3.4.1 The Problem with a Rulewise??? Lifting

Recall what soundness means for a static verification system: Assume that $\vdash \{\phi_{pre}\}\ \overline{s}\ \{\phi_{post}\}$ holds in the static verification system. Given a program state that satisfies ϕ_{pre}, soundness guarantees us both that execution won't get blocked when executing $\overline{s}$ (progress) and that the program state satisfies ϕ_{post} afterwards (preservation).

It turns out that additional measures are necessary to ensure for both progress and preservation. We will first focus on preservation, assuming that statements do not cause the runtime to be in a blocked state.

Consider a sound static verification system that allows verifying:

$$\frac{\ldots}{\vdash \{\texttt{(x = 2)}\}\ \texttt{y := 3;}\ \{\texttt{(x = 2)} \wedge \texttt{(y = 3)}\}}\ \textsc{VerAssign}$$

With the rules of lifting we can deduce

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

Preservation is apparently not ensured in a runtime that lets the assignment succeed: While the precondition does not put any restriction on the value of `x`, the postcondition claims that `x` does have a certain value. From a logical perspective the runtime can fix this problem in two ways:

"Fix" the program state
: The only way to actually make the postcondition true would involve altering the program state to satisfy the formula. This is not an option as it actively interferes with the program state. Verification conditions should not bend reality in order to hold, they have a rather passive role of checking reality.

Termination
: Preservation is only expected to hold in case execution actually reaches the postcondition. The runtime could artificially terminate in case the postcondition is not satisfied. This approach is unsatisfying as it trivially achieves soundness, effectively making the notion worthless: Any system can be made sound by just terminating exceptionally whenever progress or preservation are about to be violated.

Note that the intermediate formula `(x = 2)` does not show up in the final judgment, i.e. it was up to the verifier to "come up" with a formula that makes the overall proof work – a valid alternative would have been `(x ≠ 0)`. Still, preservation also applies to "hidden" judgments (after all we could look at them isolated, so soundness must apply), so we are guaranteed that the program state satisfies `(x = 2)` (and also `(x ≠ 0)`) after the first statement is executed.

The same reasoning does not apply to the lifted verification predicate:

Yet, soundness guarantees that

Apparently the precondition does not restrict the program state before executing the assertion. Let's assume that we

In the static system, we are guaranteed that the runtime satisfies a postcondition, given that the precondition should execution There is something fundamentally wrong with this

```
{ ? }
x := 2;
{ ? } // too weak, not optimal
assert (x = 3);

{ ? }
assert (x != 0);
{ (x = 42) } // too strong, "somewhat unsound!!!" (no guarantee of holding at runtime, as

// BUT supported by instantiation!!!

{ (x = 42) }
assert (x != 0);
{ (x = 42) }
```

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

Note: Similar to introduction condition of lifted partial functions, but using a predicate instead of function.

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

Monotonicity

Note: Identical to monotonicity condition of lifted partial functions.

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

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

Obtaining Sound Gradual Lifting

Lemma 3.4.2 (Obtaining Sound Gradual Lifting)**.**

$$\widetilde{\vdash} \{\widetilde{\phi_1}\}\ \overline{s}\ \{\widetilde{\phi_2}\} \quad \overset{\mathit{def}}{\iff} \quad \exists \widetilde{\phi_2'}.\ \ \vec{\vdash} \{\widetilde{\phi_1}\}\ \overline{s}\ \{\widetilde{\phi_2'}\} \wedge \widetilde{\phi_2'} \underset{\phi}{\widetilde{\Longrightarrow}} \widetilde{\phi_2}$$

is a sound gradual lifting.

Determinism

The verifier has no more degrees of freedom as it is now dealing with a function instead of a predicate.

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).

Composability

Composing two ...

3.5 Abstracting Dynamic Semantics

Gradualization of dynamic semantics is driven by our goal to produce a sound system. Gradualizing static semantics potentially introduced spots where progress is no longer guaranteed.

Emit “runtimeAssert” prechecks.

4 Implementation / Case Study / ...

4.1 Language

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

4.1.1 Syntax

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

Figure 4.1. SVL: Syntax

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

$$
\begin{array}{ll}
H & \in \ (o \rightharpoonup (C, \overline{(f \rightharpoonup v)})) \\
\rho & \in \ (x \rightharpoonup v) \\
\Gamma & \in \ (x \rightharpoonup T) \\
A_s & ::= \overline{(e, f)} \\
A_d & ::= \overline{(o, f)} \\
S & ::= (\rho, A_d, \overline{s}) \cdot S \mid nil
\end{array}
$$

4.1.2 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.

Typing

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

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

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

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

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

Figure 4.2. SVL: Static Typing of Expressions

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

Definition 4.1.1 (Self-Framing)**.** *A formula is* ***self-framing/self-framed*** *if it contains access to all fields it mentions.*

We omit the emptyset... bla self-framing

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

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

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

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

Figure 4.3. SVL: Framing Expressions

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

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

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

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

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

Figure 4.4. SVL: Framing Formulas

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

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

Figure 4.5. SVL: Static Footprint

Verification

4.1.3 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.

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

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

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

4.1.4 Dynamic Semantics

4.1.5 Soundness

4.2 Gradual Formulas

The path we take:

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

4.2.1 Gradualize Functions

4.3 Gradualize Hoare Rules

4.4 Gradual Dyn. Semantics

4.5 Enhancing an Unverified Language

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

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

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

$$\frac{\begin{array}{c}\phi \implies \phi' \\ \phi \implies [\![e]\!] \qquad \vdash_{\texttt{sfrm}} \phi' \qquad x \notin FV(\phi') \qquad x \notin FV(e) \qquad \Gamma \vdash x : T \qquad \Gamma \vdash e : T\end{array}}{\Gamma \vdash \{\phi\}\ x\ \texttt{:=}\ e\texttt{;}\ \{\phi' \texttt{ * (}x \texttt{ = } e\texttt{)}\}}\ \textsc{HVarAssign}$$

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

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

$$\frac{\phi \implies \phi'}{\Gamma \vdash \{\phi\}\ \texttt{assert}\ \phi'\texttt{;}\ \{\phi\}}\ \textsc{HAssert}$$

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

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

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

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

Figure 4.6. SVL: Hoare Calculus

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

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

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

Figure 4.7. SVL: Dynamic Footprint

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

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

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

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

Figure 4.8. 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 4.9. SVL: Evaluating Expressions

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

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

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

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

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

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

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

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

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

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

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

$$\frac{}{(H, (\rho', A', \emptyset) \cdot (\rho, A, \texttt{hold}\ \phi\ \texttt{\{}\ \overline{s'}\ \texttt{\};}\,\overline{s}) \cdot S) \to (H, (\rho', A \cup A', \overline{s}) \cdot S)}\ \textsc{ESHoldFinish}$$

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).

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

[19] 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.

[20] 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.

[21] 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.

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

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