Gradual Program Verification (with Implicit Dynamic Frames)

Johannes Bader, Karlsruhe Institute of Technology / Microsoft
Jonathan Aldrich, Carnegie Mellon University
Éric Tanter, University of Chile

```c
int getFour(int i)
    requires ?; // not sure what this should be yet
    ensures result = 4;
{
    i = i + 1;
    return i;
}
```
Motivation

• Program verification (against some specification)

• Two flavors: dynamic & static

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

• runtime checks
• testing techniques
• guarantee compliance at run time

```java
void withdrawCoins(int amount)
{
    assert this.balance >= amount;
    // business logic
    this.balance -= amount;
}
```
Dynamic Verification – Drawbacks

- runtime checks
- testing techniques
- guarantee compliance **at run time**

```java
void withdrawCoins(int amount) {
    assert this.balance >= amount;
    // business logic
    this.balance -= amount;
}
```
Static Verification

• declarative
• formal logic
• guarantee compliance in advance

```java
void withdrawCoins(int amount)
    requires this.balance >= amount;
{
    // business logic
    this.balance -= amount;
}
```
Static Verification – Drawbacks

- declarative
- formal logic
- guarantee compliance **in advance**

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

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

... acc.balance = 100;
acc.withdrawCoins(50); // statically checks OK!
acc.withdrawCoins(30); // oops, don’t know balance!

Can only remove false warnings by adding specifications

Specification becomes almost all-or-nothing; keep getting warnings until spec is highly complete. Want gradual return on investment—reasonable behavior at every level of specification! 
Solution: Combining Static + Dynamic

• Hybrid approach
  • Static checking, but failure is only a warning
  • Run-time assertions catch anything missed statically
• Benefits
  + Catch some errors early
  + Still catch remaining errors dynamically
  + Can eliminate run-time overhead if an assertion is statically discharged
• Drawbacks
  - Still false positive warnings / viral specification problem
  - Run-time checking may still impose too much overhead, and/or is an open problem
    (e.g. for implicit dynamic frames)
• Challenges / opportunities
  • Can we warn statically only if there is a definite error, and avoid viral specifications?
  • Can we reduce run-time overhead when we have partial information?
  • How to support dynamic checks for more powerful specification approaches (e.g. implicit dynamic frames)
Engineering Verification

• Ideal: an *engineering approach* to verification
  • Choose what to specify based on costs, benefits
  • May focus on critical components
    • Leave others unspecified
  • May focus on certain properties
    • Those most critical to users
    • Those easiest to verify
  • May add more specifications over time
    • Want incremental costs/rewards

• Viral nature of static checkers makes this difficult
  • Warnings when unspecified code calls specified code
  • May have to write many extra specifications to verify the ones you care about
Gradual Verification

A verification approach that supports \textit{gradually} adding specifications to a program

• Novel feature: support \textit{unknown and imprecise} specs

```java
void withdrawCoins(int amount)
    requires amount > 0 && this.balance >= amount;
    ensures this.balance = old(this.balance) – amount;
```

• Analogous to Gradual Typing [Siek & Taha, 2006]
Gradual Verification

A verification approach that supports \textit{gradually} adding specifications to a program

• Novel feature: support \textit{unknown and imprecise} specs

\begin{verbatim}
void withdrawCoins(int amount)
  requires this.balance >= amount;
  ensures  ? && this.balance < old(this.balance);
\end{verbatim}

• Warning if we statically detect an inconsistency
  • The spec above would be statically OK with a \textit{?} added to the precondition, or an assertion that amount > 0
  • But the given precondition alone can’t assure the part of the postcondition that we know
Gradual Verification

A verification approach that supports **gradually** adding specifications to a program

- Novel feature: support **unknown and imprecise** specs

```java
void withdrawCoins(int amount)
    requires ? && this.balance >= amount;
    ensures ? && this.balance < old(this.balance);
```

- Warning if we statically detect an inconsistency
- Warning if spec is violated at run time

```java
acc.balance = 100;
acc.withdrawCoins(50); // statically guaranteed safe
acc.withdrawCoins(30); // dynamic check OK
acc.withdrawCoins(30); // dynamic check: error!
```
Gradual Verification

A verification approach that supports **gradually** adding specifications to a program

- Novel feature: support **unknown and imprecise** specs
- Engineering properties
  - Same as dynamic verification when specs fully imprecise
  - Same as static verification when specs fully precise
    - Applies to any part of the program whose code and libraries used are specified precisely
- Smooth path from dynamic to static checking (non-viral)
  - Gradual Guarantee [Siek et al. 2015]: Given a verified program and correct input, no static or dynamic errors will be raised for the same program and input with a less-precise specification
True ≠ ?

• Prior verifiers are not “gradual”
  • No support for imprecise/unknown specifications
• Treating missing specs as “true” is insufficient

```java
class Account {
    void withdrawCoins(int amount)
    {
        requires this.balance >= amount;
        ensures true;
        ...
    }
}

Account a = new Account(100);
a.withdrawCoins(40);
a.withdrawCoins(30); // error: only know “true” here
```

Johannes Bader
True ≠ ?

• Prior verifiers are not “gradual”
  • No support for imprecise/unknown specifications

• Treating missing specs as “true” is insufficient

```java
class Account {
    void withdrawCoins(int amount)
        requires this.balance >= amount;
        ensures ?;
    ...
}
```

```java
Account a = new Account(100)
a.withdrawCoins(40);
a.withdrawCoins(30);
// OK: ? consistent with precondition
```
Gradual Verification Roadmap

• Motivation and Intuition
  • Engineering: need good support for partial specs
  • Key new idea: a (partly) unknown spec: “?”

• Overview: Abstracting Gradual Verification

• A static verification system

• Deriving a gradual verification system

• Demonstration!

• Extension to Implicit Dynamic Frames
Gradual Verification Roadmap

• Motivation and Intuition
  • Engineering: need good support for partial specs
  • Key new idea: a (partly) unknown spec: “?”

• Overview: Abstracting Gradual Verification

• A static verification system
• Deriving a gradual verification system
• Demonstration!
• Extension to Implicit Dynamic Frames
Inspiration: Gradual Typing [Siek & Taha, 2006]

- Allows programmers to selectively omit types
  - Mixing dynamically-typed code (e.g. as in Python) with statically-typed code
  - Missing types denoted with a “?” or “dynamic” keyword
  - Can have “partly dynamic” types like “? -> int”
Abstracting Gradual Typing [Garcia et al., 2016]

- Semantic foundation for Gradual Typing
  - Gradual types represent sets of possible static types
  - Use abstract interpretation to derive gradual type system from static type system
How does this relate to Verification?

```java
int getFour(int i)
    requires ?; // not sure what this should be yet
    ensures result = 4;
{
    i = i + 1;
    return i;
}
```

**Types** restrict which **values** are valid for a certain variable

**Formulas** restrict which **program states** are valid at a certain point during execution
Abstracting Gradual Typing

Ronald Garcia, Alison M. Clark, and Éric Tanter

Gradual System

\[ \tilde{\tau} ::= \tau | ? \]

Static System

\[ \tau_1 \xrightarrow{f} \tau_2 \subseteq \tau_2' \]
Abstracting Gradual Typing Verification

Ronald Garcia, Alison M. Clark, and Éric Tanter

Gradual System

\[ \tilde{\phi} ::= \phi \mid ? \]

Static System

\[ \overline{\phi_1} \xrightarrow{f} \overline{\phi_2} \subseteq \overline{\phi_2}' \]
Abstracting Gradual Typing Verification

Ronald Garcia, Alison M. Clark, and Éric Tanter

Gradual System

\[ \tilde{\phi} ::= \phi \mid \gamma \]

Benefits: if we choose \( \tilde{f}, \alpha, \) and \( \gamma \) to create a sound abstraction, we automatically get:

- The gradual guarantee: a smooth path from dynamic to static verification
- A principled approach to optimizing runtime assertion checking
Gradualization – Overview

**Syntax**

\[ s \in \text{Stmt} \]
\[ \phi \in \text{Formula} \]

**Program State**

\[ \pi \in \text{ProgramState} \]

**Semantics**

Static \[ \vdash \{ \phi \} \ s \ \{ \phi \} \]
Dynamic \[ \pi \rightarrow \pi \]
Formula \[ \pi \models \phi \]

**Soundness**

---

**Syntax**

\[ \tilde{s} \in \tilde{\text{Stmt}} \]
\[ \tilde{\phi} \in \tilde{\text{Formula}} \]

**Program State**

\[ \tilde{\pi} \in \tilde{\text{ProgramState}} \]

**Semantics**

Static \[ \vdash \{ \tilde{\phi} \} \ \tilde{s} \ \{ \tilde{\phi} \} \]
Dynamic \[ \tilde{\pi} \rightarrow \tilde{\pi} \]
Formula \[ \tilde{\pi} \models \tilde{\phi} \]

**Soundness**
Gradualization – Starting Point

Syntax
\[ s \in \text{Stmt} \]
\[ \phi \in \text{Formula} \]

Program State
\[ \pi \in \text{ProgramState} \]

Semantics
Static \[ \vdash \{ \phi \} s \{ \phi \} \]
Dynamic \[ \pi \rightarrow \pi \]
Formula \[ \pi \vDash \phi \]

Soundness

\[ s ::= \text{skip} \mid x := e \mid \text{assert } \phi \mid s_1; s_2 \]
\[ \phi ::= \text{true} \mid (e_1 = e_2) \mid \phi_1 \land \phi_2 \]

\[ = (\text{VAR} \rightarrow \mathbb{N}_0) \times \text{Stmt} \]
\[ \langle [x \mapsto 6, y \mapsto 3], x := y; \text{assert } (x = 3) \rangle \]
Gradualization – Starting Point

Syntax
\[ s \in \text{Stmt} \]
\[ \phi \in \text{Formula} \]

Program State
\[ \pi \in \text{ProgramState} \]

Semantics

- Static \[ \vdash \{ \phi \} \ s \ \{ \phi \} \]
- Dynamic \[ \pi \longrightarrow \pi \]
- Formula \[ \pi \models \phi \]

Soundness

\[ \vdash \{ \phi \} \text{skip} \{ \phi \} \]

\[ \vdash \{ \phi[e/x] \} \ x := e \ \{ \phi \} \]

- 
- 
-
Gradualization – Starting Point

Syntax
\( s \in \text{Stmt} \)
\( \phi \in \text{Formula} \)

Program State
\( \pi \in \text{ProgramState} \)

Semantics
Static \( \vdash \{ \phi \} s \{ \phi \} \)
Dynamic \( \pi \longrightarrow \pi \)
Formula \( \pi \models \phi \)

Soundness

\[ \langle [x \mapsto 6, y \mapsto 3], x := y; \text{assert } (x = 3) \rangle \]
\[ \longrightarrow^* \]
\[ \langle [x \mapsto 3, y \mapsto 3], \text{skip} \rangle \]
Gradualization – Starting Point

Syntax
\[ s \in \text{Stmt} \]
\[ \phi \in \text{Formula} \]

Program State
\[ \pi \in \text{ProgramState} \]

Semantics
Static \[ \vdash \{ \phi \} \ s \ \{ \phi \} \]
Dynamic \[ \pi \rightarrow \pi \]
Formula \[ \pi \vdash \phi \]

Soundness

\[ \langle [x \mapsto 3], s \rangle \vdash (x = 3) \]
\[ \langle [x \mapsto 4, y \mapsto 4], s \rangle \vdash (y = x) \]
Gradualization – Starting Point

Syntax
\[ s \in \text{Stmt} \]
\[ \phi \in \text{Formula} \]

Program State
\[ \pi \in \text{ProgramState} \]

Semantics

Static \[ \vdash \{ \phi \} \ s \ \{ \phi \} \]
Dynamic \[ \pi \rightarrow \pi \]

Formula \[ \pi \models \phi \]

Soundness

\[
\frac{\vdash \{ \phi \} \ \text{skip} \ \{ \phi \}}{\text{HSkip}}
\]

\[
\frac{\vdash \{ \phi[e/x] \} \ x := e \ \{ \phi \}}{\text{HAssign}}
\]

\[
\frac{\phi \Rightarrow \phi_a}{\vdash \{ \phi \} \ \text{assert} \ \phi_a \ \{ \phi \}} \quad \text{HAssert}
\]

\[
\frac{\phi_{q1} \Rightarrow \phi_{q2}}{\vdash \{ \phi_p \} \ s_1 \ \{ \phi_{q1} \} \ \vdash \{ \phi_{q2} \} \ s_2 \ \{ \phi_r \}} \quad \text{HSeq}
\]

Johannes Bader
Gradual Verification
Gradualization – Starting Point

Syntax
\[ s \in \text{Stmt} \]
\[ \phi \in \text{Formula} \]

Program State
\[ \pi \in \text{ProgramState} \]

Semantics
Static \[ \vdash \{ \phi \} s \{ \phi' \} \]
Dynamic \[ \pi \rightarrow \pi \]
Formula \[ \pi \models \phi \]

Semantic validity of Hoare triples
\[ \vdash \{ \phi \} s \{ \phi' \} \]
\[ \iff \]
\[ \forall \pi, \pi'. \pi \xrightarrow{s} \pi' \land \pi \models \phi \implies \pi' \models \phi' \]

Soundness
\[ \vdash \{ \phi \} s \{ \phi' \} \]
\[ \vdash \{ \phi \} s \{ \phi' \} \text{ Soundness} \]
Gradualization – Overview

Syntax
\[ s \in \text{Stmt} \]
\[ \phi \in \text{Formula} \]

Program State
\[ \pi \in \text{ProgramState} \]

Semantics
Static \[ \vdash \{ \phi \} s \{ \phi \} \]
Dynamic \[ \pi \rightarrow \pi \]
Formula \[ \pi \models \phi \]

Soundness

Syntax
\[ \tilde{s} \in \tilde{\text{Stmt}} \]
\[ \tilde{\phi} \in \tilde{\text{Formula}} \]

Program State
\[ \tilde{\pi} \in \tilde{\text{ProgramState}} \]

Semantics
Static \[ \tilde{\vdash} \{ \tilde{\phi} \} \tilde{s} \{ \tilde{\phi} \} \]
Dynamic \[ \tilde{\pi} \tilde{\rightarrow} \tilde{\pi} \]
Formula \[ \tilde{\pi} \tilde{\models} \tilde{\phi} \]

Soundness
Gradualization – Approach

Syntax

\[ s \in \text{Stmt} \]
\[ \phi \in \text{Formula} \]

Program State

\[ \pi \in \text{ProgramState} \]

Syntax

\[ \tilde{s} \in \tilde{\text{Stmt}} \]
\[ \tilde{\phi} \in \tilde{\text{Formula}} \]

Program State

\[ \tilde{\pi} \in \tilde{\text{ProgramState}} \]

Design Principles

\[ \text{Formula} \subseteq \tilde{\text{Formula}} \]
\[ ? \in \tilde{\text{Formula}} \]
\[ ? \notin \text{Formula} \]

Concrete Design

\[ \tilde{\phi} ::= \phi \mid ? \]
\[ \gamma(\phi) = \{ \phi \} \]
\[ \gamma(?) = \begin{cases} \phi & \text{if } \phi \land ? \\ \text{def} \text{true} & \text{else} \end{cases} \]

where

\[ ? \{ \phi \mid \text{true} \land \nexists ? \phi \} \]
Gradualization – Approach

### Syntax
- $s \in \text{Stmt}$
- $\phi \in \text{Formula}$

### Program State
- $\pi \in \text{ProgramState}$

### Syntax Extension
- $\tilde{s} \in \tilde{\text{Stmt}}$
- $\tilde{\phi} \in \tilde{\text{Formula}}$

### Program State Extension
- $\tilde{\pi} \in \tilde{\text{ProgramState}}$

### Design Principles
- $\text{Formula} \subseteq \tilde{\text{Formula}}$
- $\gamma(\phi) = \{ \phi \}$
- $\gamma(\tilde{\phi}) = \{ \phi' \in \text{SatFormula} \mid \phi' \Rightarrow \phi \}$ if $\phi \in \text{SatFormula}$
- $\gamma(\tilde{\phi})$ undefined otherwise

### Concrete Design
- $\tilde{\phi} ::= \phi \mid \tilde{	ext{phi}} \cdot \phi$
Sidebar: Why Must ? Be Satisfiable?

\[ \gamma(\phi) = \{ \phi \} \]
\[ \gamma(? \ast \phi) = \{ \phi' \in \text{SatFormula} \mid \phi' \Rightarrow \phi \} \text{ if } \phi \in \text{SatFormula} \]
\[ \gamma(? \ast \phi) \text{ undefined otherwise} \]

• Should “? \wedge (x = 3)” imply “x = 2”?
  • Intuitively, no
  • But if we choose ? to be 0=1, the implication would (vacuously) hold
  • (x = 2) would be similarly problematic
  • Thus the completed formula must be satisfiable
Gradualization – Approach

**Syntax**
- $s \in \text{Stmt}$
- $\phi \in \text{Formula}$

**Program State**
- $\pi \in \text{ProgramState}$

**Syntax**
- $\tilde{s} \in \tilde{\text{Stmt}}$
- $\tilde{\phi} \in \tilde{\text{Formula}}$

**Program State**
- $\tilde{\pi} \in \tilde{\text{ProgramState}}$

---

**Design Principles**
- $\text{Stmt} \subseteq \tilde{\text{Stmt}}$

**Concrete Design**

\[
\tilde{s} ::= x := e \mid \text{assert } \tilde{\phi} \mid \tilde{s}_1; \tilde{s}_2
\]

\[
\gamma : \tilde{\text{Stmt}} \rightarrow \mathcal{P}^{\text{Stmt}}
\]

\[
\gamma(\text{assert } \tilde{\phi}) = \{ \text{assert } \phi \mid \phi \in \gamma(\tilde{\phi}) \}
\]

...
Gradual Lifting

Gradual System

Static System

\[ \phi_1 \xrightarrow{f} \phi_2 \]

\[ \phi_1 \xleftarrow{f} \phi_2 \subseteq \phi_2' \]
Gradualization – Approach

**Syntax**
- $s \in \text{Stmt}$
- $\phi \in \text{Formula}$

**Program State**
- $\pi \in \text{ProgramState}$

---

**Syntax**
- $\tilde{s} \in \tilde{\text{Stmt}}$
- $\tilde{\phi} \in \tilde{\text{Formula}}$

**Program State**
- $\tilde{\pi} \in \tilde{\text{ProgramState}}$

---

**Design Principles**
- $\text{ProgramState} \subseteq \tilde{\text{ProgramState}}$

**Concrete Design**
- $\text{ProgramState} = (\text{VAR} \to \mathbb{N}_0) \times \text{Stmt}$
- $\tilde{\text{ProgramState}} = (\text{VAR} \to \mathbb{N}_0) \times \tilde{\text{Stmt}}$
- $\gamma(\langle \sigma, \tilde{s} \rangle) = \{\sigma\} \times \gamma(\tilde{s})$

Johannes Bader  
Gradual Verification  
38
Gradual Lifting

Gradual System

Static System
Gradual Lifting

\[ \frac{\langle \sigma, \text{assert } \phi_a \rangle \models \phi_a}{\langle \sigma, \text{assert } \phi_a \rangle \leadsto \langle \sigma, \text{skip} \rangle} \quad \text{SsAssert1} \quad \frac{\langle \sigma, \text{assert } ? \rangle \leadsto \langle \sigma, \text{skip} \rangle}{\text{SsAssert2}} \]

**Gradual System**

\[ \langle \sigma, \text{assert } \phi_a \rangle \leadsto \langle \sigma, \text{skip} \rangle \]

**Static System**

\[ \{ \langle \sigma, \text{assert } \phi_a \rangle \} \quad \frac{\langle \sigma, \text{assert } \phi_a \rangle \models \phi_a}{\langle \sigma, \text{skip} \rangle} \]

Johannes Bader

Gradual Verification
Gradual Lifting

\[ \langle \sigma, \text{assert } \phi_a \rangle \models \phi_a \quad \text{SS_ASSERT1} \]

\[ \langle \sigma, \text{assert } \phi_a \rangle \xrightarrow{\sim} \langle \sigma, \text{skip} \rangle \quad \tilde{\text{SS_ASSERT1}} \]

\[ \langle \sigma, \text{assert } ? \rangle \xrightarrow{\sim} \langle \sigma, \text{skip} \rangle \quad \tilde{\text{SS_ASSERT2}} \]

Gradual System

\[ \langle \sigma, \text{assert } ? \rangle \xrightarrow{\sim} \langle \sigma, \text{skip} \rangle \]

\[ \gamma \]

Static System

\{ \langle \sigma, \text{assert } \ldots \rangle, \ldots \} \xrightarrow{\sim} \{ \langle \sigma, \text{skip} \rangle \}

\[ \langle \sigma, \text{assert } \phi_a \rangle \models \phi_a \quad \text{SS_ASSERT} \]

\[ \langle \sigma, \text{assert } \phi_a \rangle \rightarrow \langle \sigma, \text{skip} \rangle \quad \text{SS_ASSERT} \]
Gradual Verification - Approach

**Syntax**
- $s \in \text{Stmt}$
- $\phi \in \text{Formula}$

**Program State**
- $\pi \in \text{ProgramState}$

**Semantics**
- Static $\vdash \{\phi\} s \{\phi\}$
- Dynamic $\pi \rightarrow \pi$
- Formula $\pi \models \phi$

**Soundness**

- Syntax extension
- Predicate lifting
- Function lifting
- Predicate lifting

**Syntax**
- $\tilde{s} \in \tilde{\text{Stmt}}$
- $\tilde{\phi} \in \tilde{\text{Formula}}$

**Program State**
- $\tilde{\pi} \in \tilde{\text{ProgramState}}$

**Semantics**
- Static $\tilde{\vdash} \{\tilde{\phi}\} \tilde{s} \{\tilde{\phi}\}$
- Dynamic $\tilde{\pi} \tilde{\rightarrow} \tilde{\pi}$
- Formula $\tilde{\pi} \tilde{\models} \tilde{\phi}$

**Soundness**
Predicate Lifting in a Nutshell

\[ \tilde{P} \subseteq \tilde{\text{Formula}} \times \tilde{\text{Stmt}} \times \tilde{\text{Formula}} \]

\[ P \subseteq \text{Formula} \times \text{Stmt} \times \text{Formula} \]
Predicate Lifting in a Nutshell

all gradually lifted predicates satisfy

\[
\begin{align*}
\phi_1 &\in \gamma(\bar{\phi}_1) & \phi_2 &\in \gamma(\bar{\phi}_2) & P(\phi_1, \phi_2) \\
\bar{P}(\bar{\phi}_1, \bar{\phi}_2)
\end{align*}
\]

\[
\cdot \models \cdot \subseteq \text{ProgramState} \times \text{Formula}
\]

\[
\cdot \bar{\models} \cdot \subseteq \bar{\text{ProgramState}} \times \bar{\text{Formula}}
\]

\[
\llbracket x \mapsto 3, s \rrbracket \models (x = 3)
\]

\[
\llbracket x \mapsto 3, s \rrbracket \bar{\models} (x = 3)
\]

\[
\llbracket x \mapsto 3, s \rrbracket \bar{\models} ?
\]
Predicate Lifting in a Nutshell

all gradually lifted predicates satisfy

\[
\phi_1 \in \gamma(\bar{\phi}_1) \quad \phi_2 \in \gamma(\bar{\phi}_2) \quad \phi_3 \in \gamma(\bar{\phi}_3) \quad P(\phi_1, \phi_2, \phi_3)
\]

\[
\overline{P(\bar{\phi}_1, \bar{\phi}_2, \bar{\phi}_3)}
\]

\[
\frac{\phi \Rightarrow \phi_a}{\vdash \{\phi\} \text{ assert } \phi_a \{\phi\}}
\]

\[
\text{HASSERT} \quad P(\phi_1, \phi_a, \phi_2) = (\phi_1 = \phi_2) \land (\phi_1 \Rightarrow \phi_a)
\]

\[
\vdash \{(x = 3) \land (y = 4)\} \text{ assert } (x = 3)\{(x = 3) \land (y = 4)\}
\]

\[
\frac{}{\vdash \{(x = 3) \land (y = 4)\} \text{ assert } (x = 3)\{(x = 3) \land (y = 4)\}}
\]

\[
\frac{}{\vdash \{?\} \text{ assert } (x = 3)\{(x = 3) \land (y = 4)\}}
\]

\[
\frac{}{\vdash \{(x = 3) \land (y = 4)\} \text{ assert } ?\{(x = 3) \land (y = 4)\}}
\]

\[
\frac{}{\vdash \{(x = 3) \land (y = 4)\} \text{ assert } (x = 3)\{?\}}
\]
Lifting Dynamic Semantics

• We borrow the idea of evidence from AGT
  • Intuitively, a witness for why a judgment holds, e.g.
    • The contents of variables witnesses a well-formed configuration
    • A pair of representative concrete formulas witnesses that one gradual formula can imply another

Want evidence for $? * (x = 4) \implies ? * (y = 3)$

Example evidence: $\varepsilon_1 = \langle (x = 4) * (y = 3), (y = 3) \rangle$

Want most general evidence – a valid piece of evidence that generalizes all others (e.g. pre- and post-states are implied by those of other valid evidence). The evidence above is the most general evidence for the example implication.
Lifting Dynamic Semantics

• We borrow the idea of evidence from AGT
  • Intuitively, a witness for why a judgment holds, e.g.
    • The contents of variables witnesses a well-formed configuration
    • A pair of representative concrete formulas witnesses that one gradual formula can imply another

• When program executes, we combine evidence
  • E.g. combine the evidence for the current program configuration with the evidence for the next statement, to yield the next program configuration
    • Or an error if the next program configuration is not well-formed – could happen if gradual spec was too approximate
  • Conveniently, combining evidence is equivalent to checking assertions in program text!
Optimization: Checking Residuals

• If we know some information statically, we may not need to verify all of an assertion

• We compute the *residual* of a run-time check
  • Assume we are checking $\phi_B$ and we know $\phi_A$. Assume $\phi_B$ is in conjunctive normal form. Example:
    • $\phi_A = (x > 5)$
    • $\phi_B = (y > x \land y > 4)$
    • We remove any conjunct of $\phi_B$ that is implied by $\phi_A$ and the remaining conjuncts of $\phi_B$.

• Example: residual is $(y > x)$

• Best case: static verification ($\phi_A$ implies $\phi_B$)
  • All run-time checking is removed!
Some Theorems
(stated formally in our draft paper, but have not laid the groundwork here)

- **Soundness**: standard progress and preservation
  - Note: run-time errors may occur due to assertion failures
- **Static gradual guarantee**: if a program checks statically, it will still do so if the precision of its specifications is reduced
- **Dynamic gradual guarantee**: if a program executes without error, it will still do so if the precision of its specifications is reduced
- We get the last two “for free” based on the properties of abstract interpretation
Demonstration

http://olydis.github.io/GradVer/impl/HTML5wp/
The Challenge of Aliasing

\{(p1.\text{age} = 19) \land (p2.\text{age} = 19)\}

\text{p1.\text{age}}++

\{(p1.\text{age} = 20) \land (p2.\text{age} = 19)\}

Traditional Hoare Logic solution

\{(p1.\text{age} = 19) \land (p2.\text{age} = 19) \land p1 \neq p2\}

\text{p1.\text{age}}++

\{(p1.\text{age} = 20) \land (p2.\text{age} = 19) \land p1 \neq p2\}

Issue: scalability. What if we have 4 pointers?

\{\ldots \land p1 \neq p2 \land p1 \neq p3 \land p1 \neq p4 \land p2 \neq p3 \land p2 \neq p4 \land p3 \neq p4\}

Alias information scales quadratically \((n \times n-1)\) with the number of pointer variables!
Implicit Dynamic Frames [Smans et al. 2009]

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

\{acc(p1.age) \ast acc(p2.age) \ast (p1.age = 19) \ast (p2.age = 19)\}
\(p1\).age++
\{acc(p1.age) \ast acc(p2.age) \ast (p1.age = 20) \ast (p2.age = 19)\}

Implicit Dynamic Frames rules:
• acc(p1.age) denotes permission to access p1.age
• if p1.age is used in a formula, acc(p1.age) must appear earlier (‘self-framing’)
• acc(x.f) may only appear once for each object/field combination
The Frame Rule

\[
\{ P \} S \{ Q \} \quad \text{R is self-framed} \\
\{ P \ast R \} S \{ Q \ast R \}
\]

• Example application

\{
\text{acc(p1.age)} \ast p1.age = 19 \ast \text{acc(p2.age)} \ast p2.age = 19
\}

p1.age++

\{
/* what goes here? */
\}
The Frame Rule

\[ \{ P \land R \} S \{ Q \land R \} \quad \text{R is self-framed} \]
\[ \{ P \land R \} S \{ Q \land R \} \]

- Example application
  \{ acc(p1.age) \land p1.age = 19 \land \text{acc(p2.age) \land p2.age = 19} \}
  p1.age++
  \{ /* what goes here? */ \}
The Frame Rule

\[
\{ P \ast R \} S \{ Q \ast R \} \quad \text{R is self-framed}
\]

\[
\{ P \ast R \} S \{ Q \ast R \}
\]

- Example application

\[
\{ \text{acc(p1.age)} \ast \ p1.\text{age} = 19 \ast \} R \]
\]

\[
\text{p1.\text{age}}++
\]

\[
\{ \text{acc(p1.age)} \ast \ p1.\text{age} = 20 \ast \} R \]
\]

Apply the normal assignment rule
The Frame Rule

\[
\{ P \land R \} S \{ Q \land R \} \quad \text{R is self-framed} \\
\{ P \land R \} S \{ Q \land R \}
\]

• Example application

\[
\{ \text{acc(p1.age) \land p1.age = 19} \} \quad \text{acc(p2.age) \land p2.age = 19} \\
p1.\text{age}++ \\
\{ \text{acc(p1.age) \land p1.age = 20} \} \quad \text{acc(p2.age) \land p2.age = 19} 
\]

Frame back on the rest of the formula
The Frame Rule

\[
\{ P \land R \} \triangleright \{ Q \land R \} \quad \text{R is self-framed}
\]

\[
\{ P \land R \} \triangleright \{ Q \land R \}
\]

- Anti-example

\[
\{ \text{acc}(p1.\text{age}) \land p1.\text{age} = 19 \land p2.\text{age} = 19 \} \\
p1.\text{age}++ \\
\{ /* what goes here? */ \}
\]

R is not self-framed. Cannot apply the frame rule!
The Frame Rule

\[ \{ P \land R \} S \{ Q \land R \} \quad R \text{ is self-framed} \]

\[ \{ P \land R \} S \{ Q \land R \} \]

- Anti-example

\{ acc(p1.age) \land p1.age = 19 \land p2.age = 19 \}

p1.age++

\{ acc(p1.age) \land p1.age = 20 \}

R is not self-framed. Cannot apply the frame rule!

The best we can do is drop the unframed information from the formula.
Gradual Implicit Dynamic Frames

\{(p1\text{.}age = 19) \land (p2\text{.}age = 19)\}\n
\text{p1\text{.}age++}

\{(p1\text{.}age = 20) \land (p2\text{.}age = 19)\}\n
\{\text{acc}(p1\text{.}age) \land \text{acc}(p2\text{.}age) \land (p1\text{.}age = 19) \land (p2\text{.}age = 19)\}\n
\text{p1\text{.}age++}

\{\text{acc}(p1\text{.}age) \land \text{acc}(p2\text{.}age) \land (p1\text{.}age = 20) \land (p2\text{.}age = 19)\}\n
\{? \land (p1\text{.}age = 19) \land (p2\text{.}age = 19)\}\n
\text{p1\text{.}age++}

\{? \land (p1\text{.}age = 20) \land (p2\text{.}age = 19)\}\n
\text{Not valid if } p1 = p2!\n
\text{OK! } p1 \text{ and } p2 \text{ may not overlap}\n
\text{OK statically; requires run-time check}

\text{Useful if you don’t want to specify whether } p1 \text{ and } p2 \text{ alias:}

\text{? could be “}\text{acc}(p1\text{.}age) \land \land p1 = p2”\text{”}
Consequences of Implicit Dynamic Frames

• Gradual types can help with self-framing
  • We can ignore frames just by writing “? ∧ P” where P does not include acc(...)
    • Any invalid assumptions due to framing will be caught at run time
    • We can always add framing later

• Evidence: must track ownership of heap in the runtime
  • Allows for testing acc(x.f) in assertions
  • Of course, in statically verified code we can optimize this away!

• Residual testing gets more interesting. Example:
  • \( \phi_A = (? ∧ x.f = 2) \)
  • \( \phi_B = (acc(x.f) ∧ x.f = 2 ∧ y = 5) \)
  • Residual is \( y = 5 \)
    • Don’t need to check acc(x.f) because ? must include acc(x.f) for the \( x.f = 2 \) statement to be well-formed
Demonstration: Implicit Dynamic Frames
Gradual Verification

• Engineering approach to verification
  • Choose what properties & components to specify

• Support for unknown formulas?
  • Model partly specified properties, components
  • Semantically: replace with anything that leaves the formula satisfiable

• Gradual Verification
  • Derived as an abstraction of static verification
  • Gradual guarantee: making formulas less precise will not cause compile-time or run-time failures

• Future work
  • Efficient implementation
  • Richer verification system