Gradual Program Verification
(with Implicit Dynamic Frames)

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

```
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

```
// 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**

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

Dynamic Verification – Drawbacks

- runtime checks — runtime overhead
- testing techniques — additional effort
- guarantee compliance **at run time** — late detection

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

Static Verification

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

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

Static Verification – Drawbacks

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

limited expressiveness and/or decidability

annotation overhead (viral)

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

Viral Specifications

```
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 **gradually** adding specifications to a program

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

```
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 **gradually** adding specifications to a program

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

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

- Warning if we statically detect an inconsistency
  - The spec above would be statically OK with a ? 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

```
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

```
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

```
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
```

True ≠ ?

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

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

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]

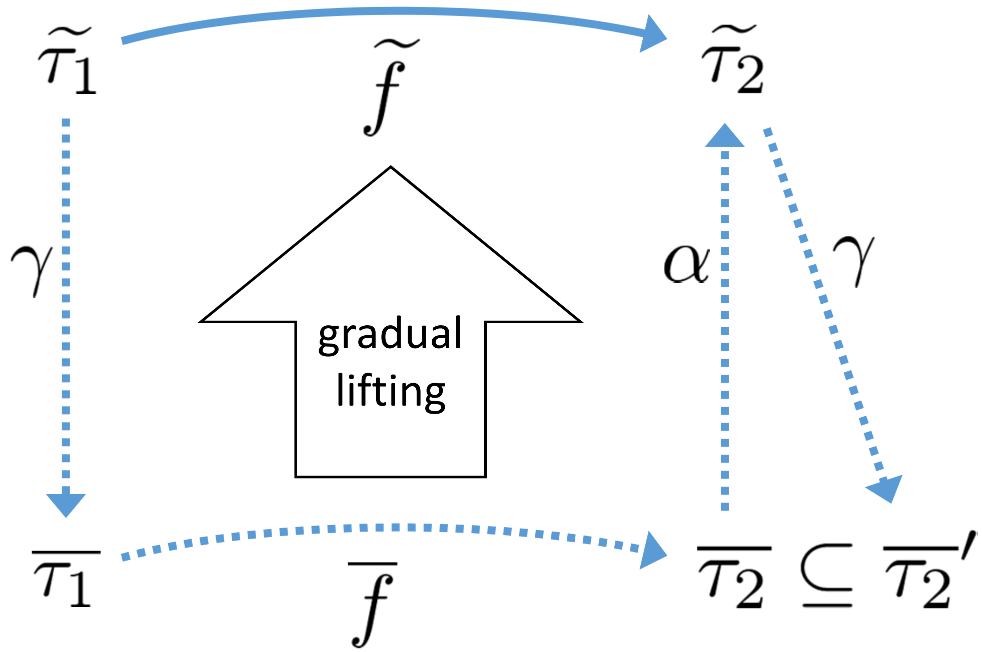
- Semantic foundation for Gradual Typing
  - Gradual types represent sets of possible static types
  - Use abstract interpretation to derive gradual type system from static type system

Gradual System

$$\gamma(\tau) = \{ \tau \}$$

$$\gamma(?) = \textsc{Types}$$

Static System



How does this relate to Verification?

```
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

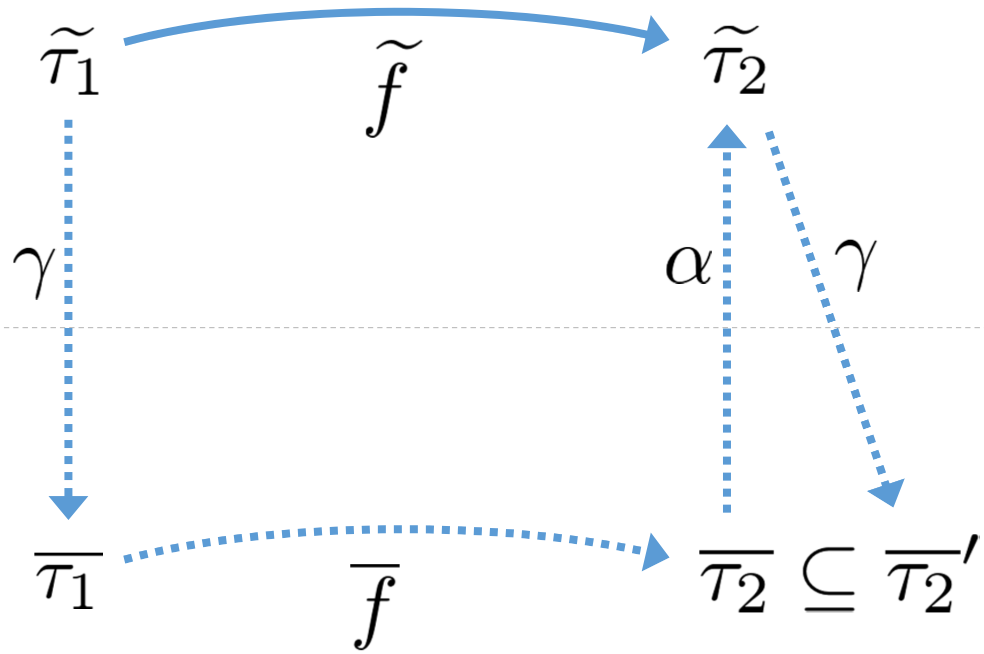
Abstracting Gradual Typing

Ronald Garcia, Alison M. Clark, and Éric Tanter

Gradual System

$\widetilde{\tau} \; ::= \; \tau \mid ?$

Static System



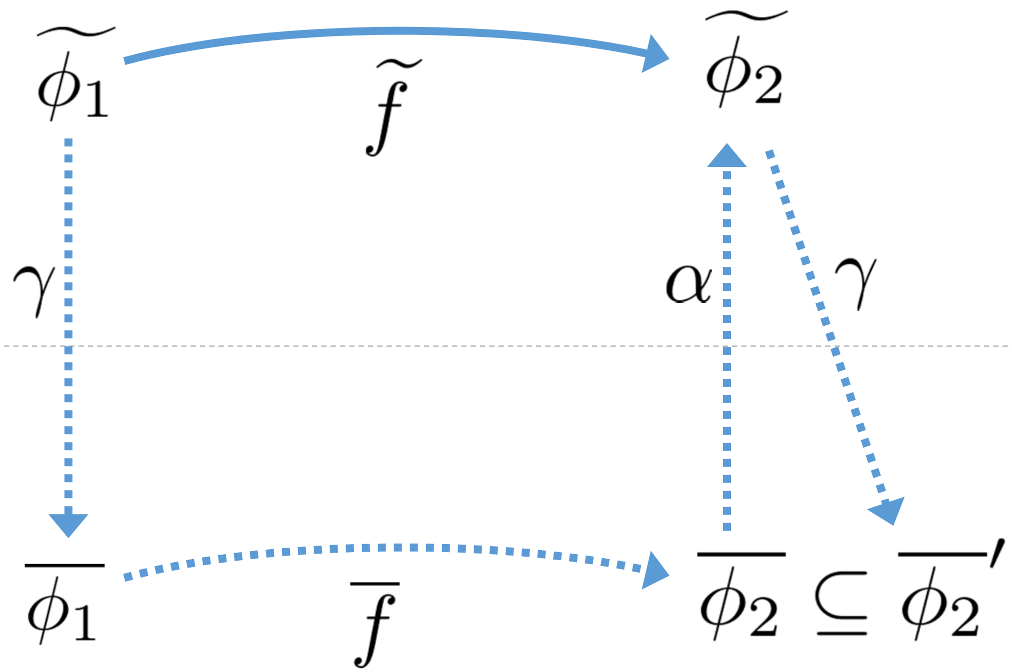
Abstracting Gradual ~~Typing~~ Verification

Ronald Garcia, Alison M. Clark, and Éric Tanter

Gradual System

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

Static System

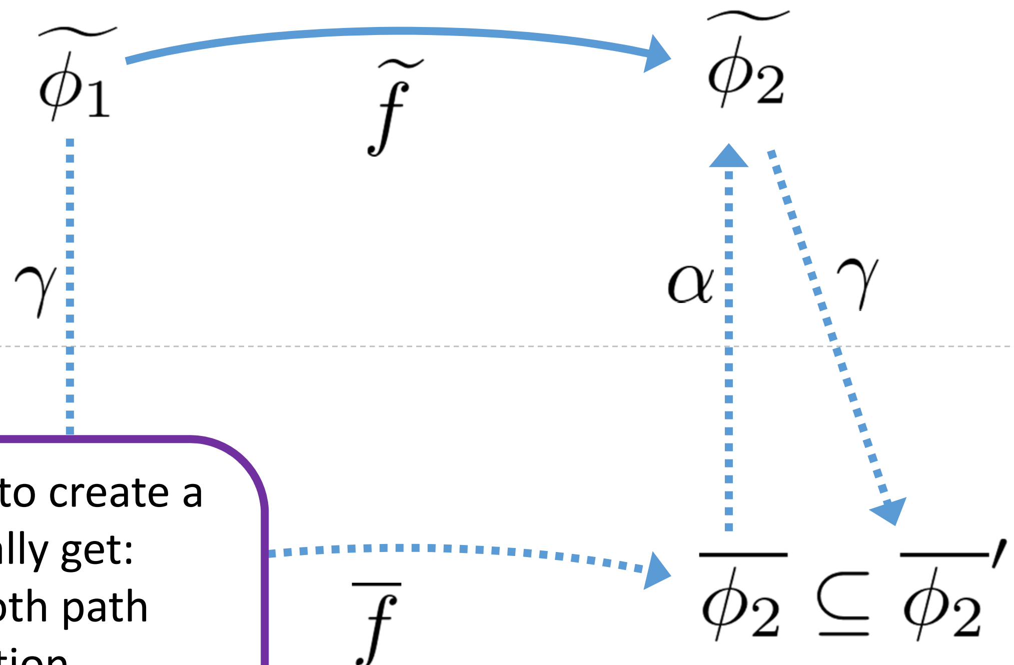


Abstracting Gradual ~~Typing~~ Verification

Ronald Garcia, Alison M. Clark, and Éric Tanter

Gradual System

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



Benefits: if we choose $\widetilde{f}$, α, and γ to create a sound abstraction, we automatically get:

- The gradual guarantee: a smooth path from dynamic to static verification
- A principled approach to optimizing run-time assertion checking

Gradualization – Overview

Syntax

$s \in \textsc{Stmt}$

$\phi \in \textsc{Formula}$

Program State

$\pi \in \textsc{ProgramState}$

Semantics

Static $\vdash \{\phi\}\ s\ \{\phi\}$

Dynamic $\pi \longrightarrow \pi$

Formula $\pi \vDash \phi$

Soundness

Gradualization

Syntax

$\widetilde{s} \in \widetilde{\textsc{Stmt}}$

$\widetilde{\phi} \in \widetilde{\textsc{Formula}}$

Program State

$\widetilde{\pi} \in \widetilde{\textsc{ProgramState}}$

Semantics

Static $\widetilde{\vdash}\ \{\widetilde{\phi}\}\ \widetilde{s}\ \{\widetilde{\phi}\}$

Dynamic $\widetilde{\pi} \ \widetilde{\longrightarrow}\ \widetilde{\pi}$

Formula $\widetilde{\pi}\ \widetilde{\vDash}\ \widetilde{\phi}$

Soundness

Gradualization – Starting Point

Syntax

$s \in \textsc{Stmt}$

$\phi \in \textsc{Formula}$

Program State

$\pi \in \textsc{ProgramState}$

Semantics

Static $\vdash \{\phi\}\ s\ \{\phi\}$

Dynamic $\pi \longrightarrow \pi$

Formula $\pi \vDash \phi$

Soundness

$$s \ ::= \ \texttt{skip} \mid x \ \texttt{:=}\ e \mid \texttt{assert}\ \phi \mid s_1\texttt{;}\ s_2$$

$$\phi \ ::= \ \texttt{true} \mid (e_1 \ \texttt{=}\ e_2) \mid \phi_1 \wedge \phi_2$$

$$= (\textsc{Var} \rightharpoonup \mathbb{N}_0) \times \textsc{Stmt}$$

$$\langle [\mathrm{x} \mapsto 6, \mathrm{y} \mapsto 3], \texttt{x := y; assert (x = 3)} \rangle$$

Gradualization – Starting Point

Syntax

$s \in \textsc{Stmt}$

$\phi \in \textsc{Formula}$

Program State

$\pi \in \textsc{ProgramState}$

Semantics

Static $\vdash \{\phi\}\ s\ \{\phi\}$

Dynamic $\pi \longrightarrow \pi$

Formula $\pi \vDash \phi$

Soundness

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

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

$\vdots$

Gradualization – Starting Point

Syntax

$s \in \textsc{Stmt}$

$\phi \in \textsc{Formula}$

Program State

$\pi \in \textsc{ProgramState}$

Semantics

Static $\vdash \{\phi\}\ s\ \{\phi\}$

Dynamic $\pi \longrightarrow \pi$

Formula $\pi \vDash \phi$

Soundness

$$\langle [\mathrm{x} \mapsto 6, \mathrm{y} \mapsto 3], \texttt{x := y; assert (x = 3)} \rangle$$

$$\longrightarrow^*$$

$$\langle [\mathrm{x} \mapsto 3, \mathrm{y} \mapsto 3], \texttt{skip} \rangle$$

Gradualization – Starting Point

Syntax

$s \in \textsc{Stmt}$

$\phi \in \textsc{Formula}$

Program State

$\pi \in \textsc{ProgramState}$

Semantics

Static $\vdash \{\phi\}\ s\ \{\phi\}$

Dynamic $\pi \longrightarrow \pi$

Formula $\pi \vDash \phi$

Soundness

$$\langle [\mathtt{x} \mapsto 3], s \rangle \vDash (\mathtt{x = 3})$$

$$\langle [\mathtt{x} \mapsto 4, \mathtt{y} \mapsto 4], s \rangle \vDash (\mathtt{y = x})$$

Gradualization – Starting Point

Syntax

$s \in \textsc{Stmt}$

$\phi \in \textsc{Formula}$

Program State

$\pi \in \textsc{ProgramState}$

Semantics

Static $\vdash \{\phi\}\ s\ \{\phi\}$

Dynamic $\pi \longrightarrow \pi$

Formula $\pi \vDash \phi$

Soundness

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

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

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

$$\frac{\phi_{q1} \Rightarrow \phi_{q2} \qquad \vdash \{\phi_p\}\ s_1\ \{\phi_{q1}\} \qquad \vdash \{\phi_{q2}\}\ s_2\ \{\phi_r\}}{\vdash \{\phi_p\}\ s_1\texttt{;}\ s_2\ \{\phi_r\}}\ \textsc{HSeq}$$

Gradualization – Starting Point

Syntax

$s \in \textsc{Stmt}$

$\phi \in \textsc{Formula}$

Program State

$\pi \in \textsc{ProgramState}$

Semantics

Static $\vdash \{\phi\}\ s\ \{\phi\}$

Dynamic $\pi \longrightarrow \pi$

Formula $\pi \vDash \phi$

Soundness

Semantic validity of Hoare triples

$$\vDash \{\phi\}\ s\ \{\phi'\}$$

$$\overset{\text{def}}{\iff}$$

$$\forall \pi, \pi'.\ \pi \xrightarrow{s} \pi' \land \pi \vDash \phi \implies \pi' \vDash \phi'$$

$$\frac{\vdash \{\phi\}\ s\ \{\phi'\}}{\vDash \{\phi\}\ s\ \{\phi'\}}\ \textsc{Soundness}$$

Gradualization – Overview

Syntax

$s \in \textsc{Stmt}$

$\phi \in \textsc{Formula}$

Program State

$\pi \in \textsc{ProgramState}$

Semantics

Static $\vdash \{\phi\}\ s\ \{\phi\}$

Dynamic $\pi \longrightarrow \pi$

Formula $\pi \vDash \phi$

Soundness

Gradualization

Syntax

$\widetilde{s} \in \widetilde{\textsc{Stmt}}$

$\widetilde{\phi} \in \widetilde{\textsc{Formula}}$

Program State

$\widetilde{\pi} \in \widetilde{\textsc{ProgramState}}$

Semantics

Static $\widetilde{\vdash} \{\widetilde{\phi}\}\ \widetilde{s}\ \{\widetilde{\phi}\}$

Dynamic $\widetilde{\pi} \ \widetilde{\longrightarrow}\ \widetilde{\pi}$

Formula $\widetilde{\pi}\ \widetilde{\vDash}\ \widetilde{\phi}$

Soundness

Gradualization – Approach

Syntax

$s \in \textsc{Stmt}$

$\phi \in \textsc{Formula}$

Program State

$\pi \in \textsc{ProgramState}$

syntax extension →

Syntax

$\widetilde{s} \in \widetilde{\textsc{Stmt}}$

$\widetilde{\phi} \in \widetilde{\textsc{Formula}}$

Program State

$\widetilde{\pi} \in \widetilde{\textsc{ProgramState}}$

Design Principles

$\textsc{Formula} \subset \widetilde{\textsc{Formula}}$

$? \in \widetilde{\textsc{Formula}}$

$? \notin \textsc{Formula}$

Concrete Design

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

$$\gamma(\phi) = \{ \phi \}$$

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

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

$$\text{where } \textsc{SatFormula} \stackrel{\text{def}}{=} \{ \phi \mid \exists \pi.\ \pi \models \phi \}$$

$$? \neq \texttt{true} \wedge ?$$

Gradualization – Approach

Syntax

$s \in \textsc{Stmt}$

$\phi \in \textsc{Formula}$

Program State

$\pi \in \textsc{ProgramState}$

syntax extension →

Syntax

$\widetilde{s} \in \widetilde{\textsc{Stmt}}$

$\widetilde{\phi} \in \widetilde{\textsc{Formula}}$

Program State

$\widetilde{\pi} \in \widetilde{\textsc{ProgramState}}$

Design Principles

$\textsc{Formula} \subset \widetilde{\textsc{Formula}}$

Concrete Design

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

$$\gamma(\phi) = \{ \phi \}$$

$$\gamma(? * \phi) = \{ \phi' \in \textsc{SatFormula} \mid \phi' \Rightarrow \phi \} \quad \text{if } \phi \in \textsc{SatFormula}$$

$$\gamma(? * \phi) \quad \text{undefined otherwise}$$

Sidebar: Why Must ? Be Satisfiable?

$$\gamma(\phi) = \{ \ \phi \ \}$$

$$\gamma(? * \phi) = \{ \ \phi' \in \textsc{SatFormula} \mid \phi' \Rightarrow \phi \ \} \quad \text{if } \phi \in \textsc{SatFormula}$$

$$\gamma(? * \phi) \quad \text{undefined otherwise}$$

- Should "? ∧ (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		Syntax
$s \in \textsc{Stmt}$	syntax extension →	$\widetilde{s} \in \widetilde{\textsc{Stmt}}$
$\phi \in \textsc{Formula}$	syntax extension →	$\widetilde{\phi} \in \widetilde{\textsc{Formula}}$
Program State		**Program State**
$\pi \in \textsc{ProgramState}$		$\widetilde{\pi} \in \widetilde{\textsc{ProgramState}}$

Design Principles

$\textsc{Stmt} \subseteq \widetilde{\textsc{Stmt}}$

Concrete Design

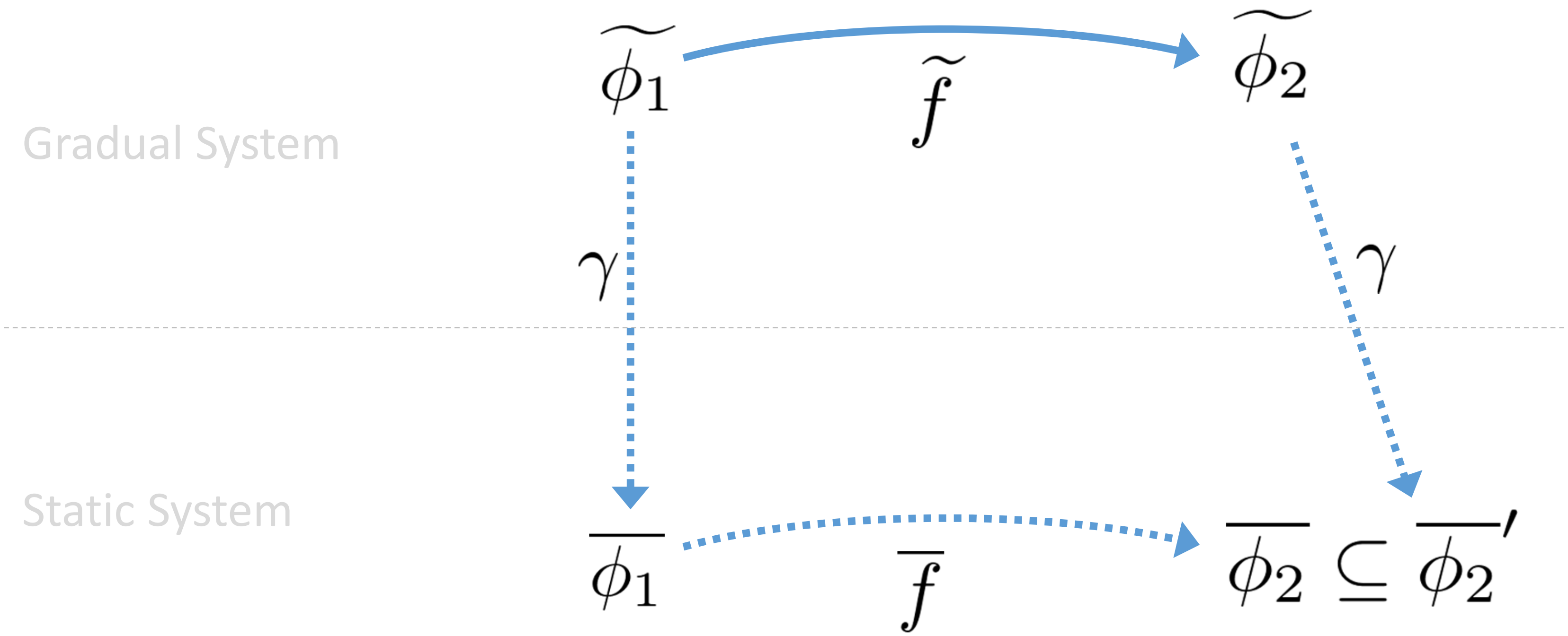
$$\widetilde{s} \quad ::= \quad x \; \texttt{:=} \; e \;\mid\; \texttt{assert} \; \widetilde{\phi} \;\mid\; \widetilde{s_1} \texttt{;} \; \widetilde{s_2}$$

$$\gamma : \widetilde{\textsc{Stmt}} \rightarrow \mathcal{P}^{\textsc{Stmt}}$$

$$\gamma(\texttt{assert} \; \widetilde{\phi}) = \{ \; \texttt{assert} \; \phi \mid \phi \in \gamma(\widetilde{\phi}) \; \}$$

...

Gradual Lifting



Gradualization – Approach

Syntax

$s \in \textsc{Stmt}$ —syntax extension→ $\widetilde{s} \in \widetilde{\textsc{Stmt}}$

$\phi \in \textsc{Formula}$ —syntax extension→ $\widetilde{\phi} \in \widetilde{\textsc{Formula}}$

Program State

$\pi \in \textsc{ProgramState}$ —extension→ $\widetilde{\pi} \in \widetilde{\textsc{ProgramState}}$

Design Principles

$$\textsc{ProgramState} \subseteq \widetilde{\textsc{ProgramState}}$$

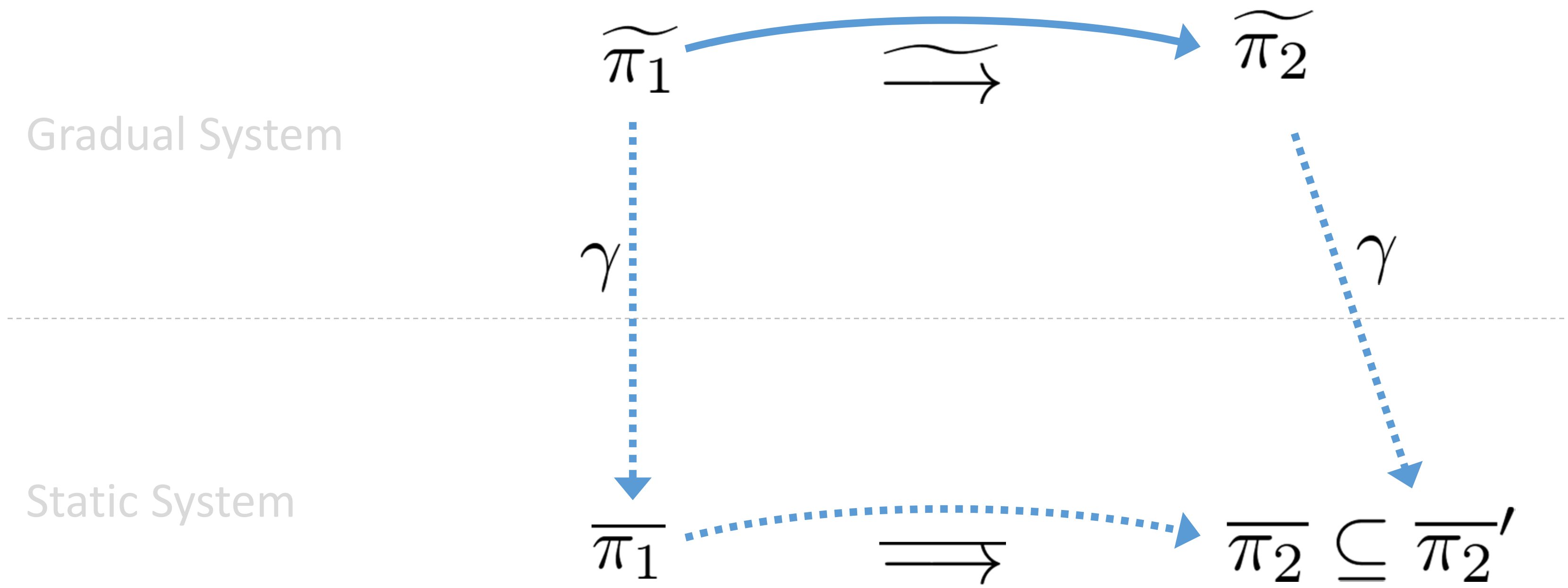
Concrete Design

$$\textsc{ProgramState} = (\textsc{Var} \rightharpoonup \mathbb{N}_0) \times \textsc{Stmt}$$

$$\widetilde{\textsc{ProgramState}} = (\textsc{Var} \rightharpoonup \mathbb{N}_0) \times \widetilde{\textsc{Stmt}}$$

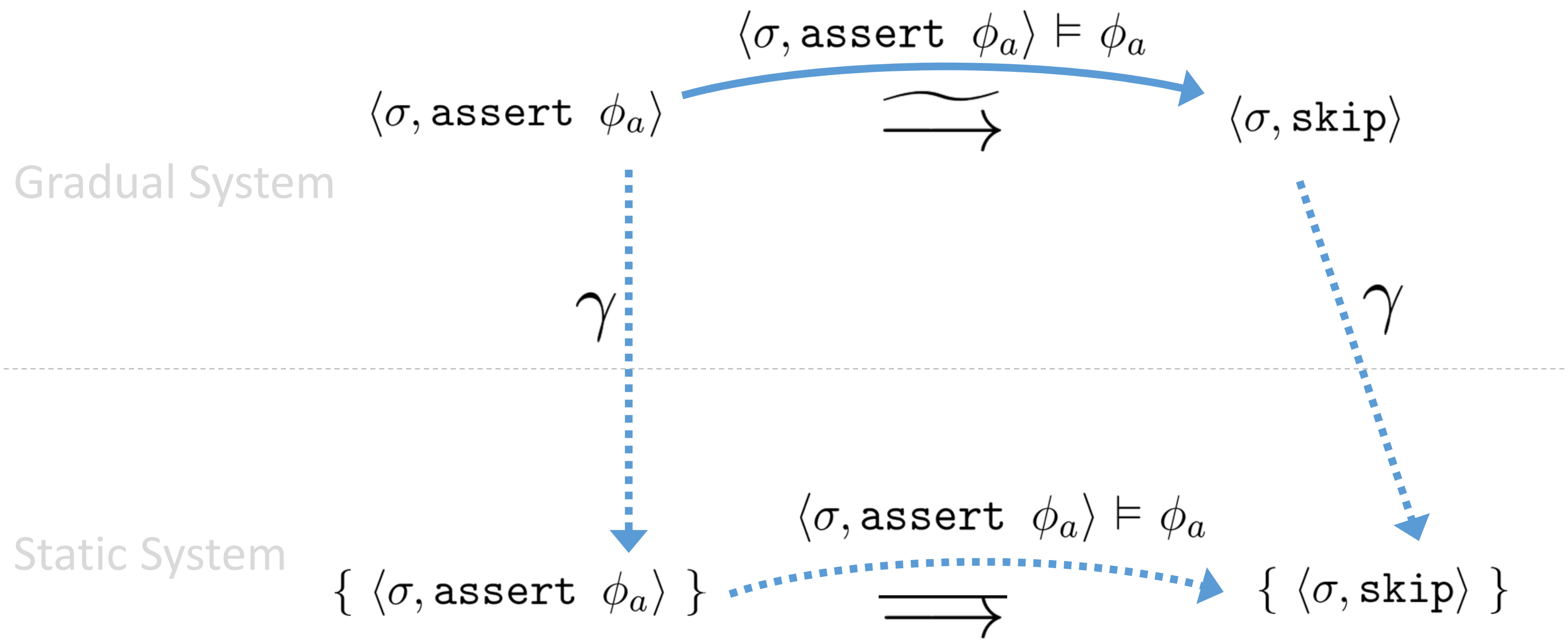
$$\gamma(\langle \sigma, \widetilde{s} \rangle) = \{\sigma\} \times \gamma(\widetilde{s})$$

Gradual Lifting



Gradual Lifting

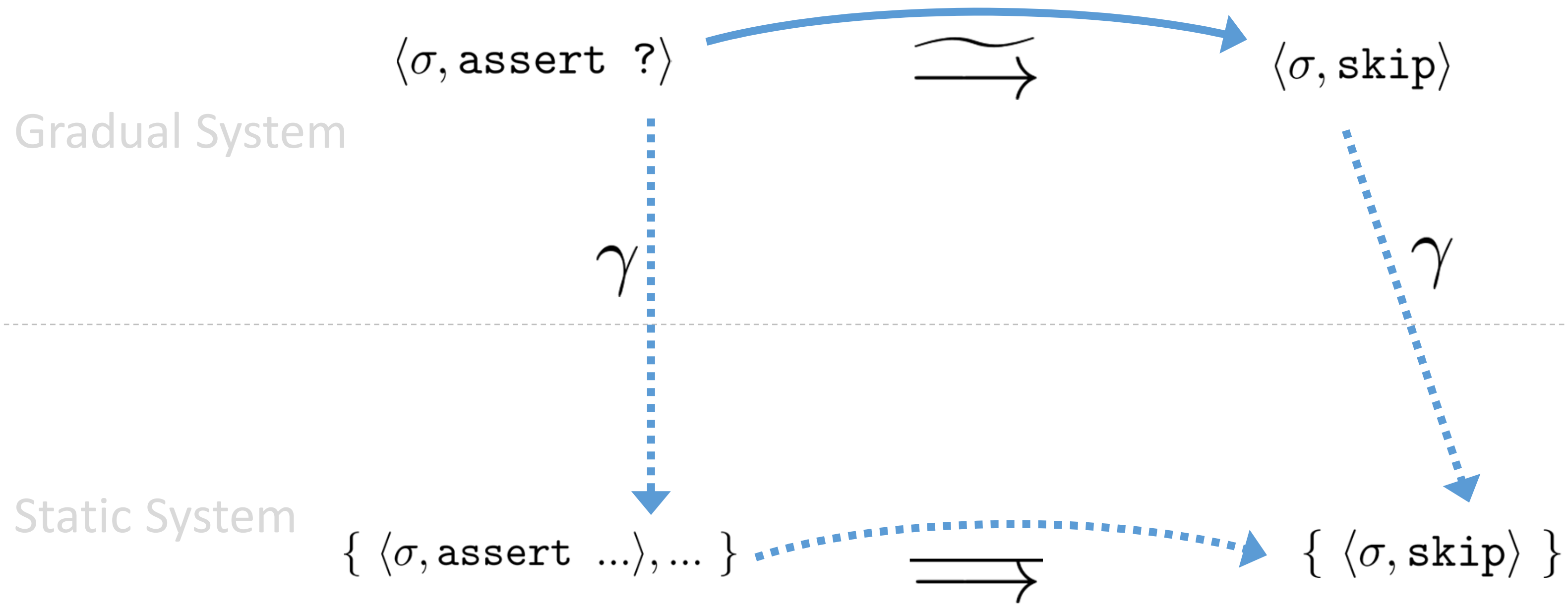
$$\frac{\langle \sigma, \texttt{assert}\ \ \phi_a \rangle \models \phi_a}{\langle \sigma, \texttt{assert}\ \ \phi_a \rangle \ \widetilde{\longrightarrow}\ \langle \sigma, \texttt{skip} \rangle}\ \widetilde{\text{SsAssert1}} \qquad \frac{}{\langle \sigma, \texttt{assert}\ \ ? \rangle \ \widetilde{\longrightarrow}\ \langle \sigma, \texttt{skip} \rangle}\ \widetilde{\text{SsAssert2}}$$



$$\frac{\langle \sigma, \texttt{assert}\ \ \phi_a \rangle \models \phi_a}{\langle \sigma, \texttt{assert}\ \ \phi_a \rangle \longrightarrow \langle \sigma, \texttt{skip} \rangle}\ \text{SsAssert}$$

Gradual Lifting

$$\frac{\langle \sigma, \texttt{assert}\ \ \phi_a \rangle \vDash \phi_a}{\langle \sigma, \texttt{assert}\ \ \phi_a \rangle \ \widetilde{\longrightarrow}\ \langle \sigma, \texttt{skip} \rangle}\ \widetilde{\text{SsAssert1}} \qquad \frac{}{\langle \sigma, \texttt{assert}\ \ ? \rangle \ \widetilde{\longrightarrow}\ \langle \sigma, \texttt{skip} \rangle}\ \widetilde{\text{SsAssert2}}$$



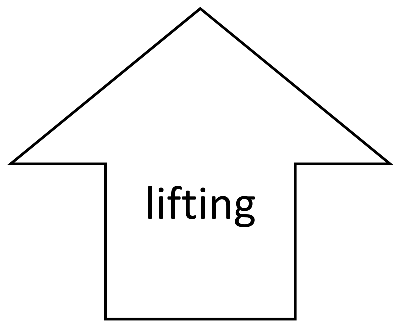
$$\frac{\langle \sigma, \texttt{assert}\ \ \phi_a \rangle \vDash \phi_a}{\langle \sigma, \texttt{assert}\ \ \phi_a \rangle \longrightarrow \langle \sigma, \texttt{skip} \rangle}\ \text{SsAssert}$$

Gradual Verification - Approach

Syntax		**Syntax**
$s \in \textsc{Stmt}$	syntax extension →	$\widetilde{s} \in \widetilde{\textsc{Stmt}}$
$\phi \in \textsc{Formula}$	syntax extension →	$\widetilde{\phi} \in \widetilde{\textsc{Formula}}$
Program State		**Program State**
$\pi \in \textsc{ProgramState}$	extension →	$\widetilde{\pi} \in \widetilde{\textsc{ProgramState}}$
Semantics		**Semantics**
Static $\vdash \{\phi\}\ s\ \{\phi\}$	predicate lifting →	Static $\widetilde{\vdash} \{\widetilde{\phi}\}\ \widetilde{s}\ \{\widetilde{\phi}\}$
Dynamic $\pi \longrightarrow \pi$	function lifting →	Dynamic $\widetilde{\pi} \ \widetilde{\longrightarrow}\ \widetilde{\pi}$
Formula $\pi \vDash \phi$	predicate lifting →	Formula $\widetilde{\pi} \ \widetilde{\vDash}\ \widetilde{\phi}$
Soundness		**Soundness**

Predicate Lifting in a Nutshell

$$\widetilde{P} \subseteq \widetilde{\text{Formula}} \times \widetilde{\text{Stmt}} \times \widetilde{\text{Formula}}$$



$$P \subseteq \text{Formula} \times \text{Stmt} \times \text{Formula}$$

Predicate Lifting in a Nutshell

all gradually lifted predicates satisfy

$$\frac{\phi_1 \in \gamma(\widetilde{\phi_1}) \qquad \phi_2 \in \gamma(\widetilde{\phi_2}) \qquad P(\phi_1, \phi_2)}{\widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2})}$$

$$\cdot \models \cdot \subseteq \textsc{ProgramState} \times \textsc{Formula}$$

$$\cdot \mathrel{\widetilde{\models}} \cdot \subseteq \widetilde{\textsc{P}}\textsc{rogramState} \times \widetilde{\textsc{F}}\textsc{ormula}$$

$$\langle [x \mapsto 3], s \rangle \models \texttt{(x = 3)}$$

$$\langle [x \mapsto 3], s \rangle \mathrel{\widetilde{\models}} \texttt{(x = 3)}$$

$$\langle [x \mapsto 3], s \rangle \mathrel{\widetilde{\models}} \texttt{?}$$

Predicate Lifting in a Nutshell

all gradually lifted predicates satisfy

$$\frac{\phi_1 \in \gamma(\widetilde{\phi_1}) \qquad \phi_2 \in \gamma(\widetilde{\phi_2}) \qquad \phi_3 \in \gamma(\widetilde{\phi_3}) \qquad P(\phi_1, \phi_2, \phi_3)}{\widetilde{P}(\widetilde{\phi_1}, \widetilde{\phi_2}, \widetilde{\phi_3})}$$

$$\frac{\phi \Rightarrow \phi_a}{\vdash \{\phi\}\ \texttt{assert}\ \phi_a\ \{\phi\}}\ \text{HASSERT} \qquad P(\phi_1, \phi_a, \phi_2) = (\phi_1 = \phi_2) \wedge (\phi_1 \Rightarrow \phi_a)$$

$$\vdash \{(\texttt{x = 3})\ \wedge\ (\texttt{y = 4})\}\ \texttt{assert (x = 3)}\ \{(\texttt{x = 3})\ \wedge\ (\texttt{y = 4})\}$$

$$\widetilde{\vdash} \{(\texttt{x = 3})\ \wedge\ (\texttt{y = 4})\}\ \texttt{assert (x = 3)}\ \{(\texttt{x = 3})\ \wedge\ (\texttt{y = 4})\}$$

$$\widetilde{\vdash} \{?\}\ \texttt{assert (x = 3)}\ \{(\texttt{x = 3})\ \wedge\ (\texttt{y = 4})\}$$

$$\widetilde{\vdash} \{(\texttt{x = 3})\ \wedge\ (\texttt{y = 4})\}\ \texttt{assert ?}\ \{(\texttt{x = 3})\ \wedge\ (\texttt{y = 4})\}$$

$$\widetilde{\vdash} \{(\texttt{x = 3})\ \wedge\ (\texttt{y = 4})\}\ \texttt{assert (x = 3)}\ \{?\}$$

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 $? * (x = 4) \mathrel{\widetilde{\Rightarrow}} ? * (y = 3)$

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 ϕ_B and we know ϕ_A. Assume ϕ_B is in conjuctive normal form. Example:
    - $\phi_A = (x > 5)$
    - $\phi_B = (y > x \wedge y > 4)$
  - We remove any conjunct of ϕ_B that is implied by ϕ_A and the remaining conjuncts of ϕ_B.
- Example: residual is (y > x)
- Best case: static verification (ϕ_A implies ϕ_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.age = 19) ∧ (p2.age = 19)}
p1.age++
{(p1.age = 20) ∧ (p2.age = 19)}
```

Not valid if p1 = p2!

Traditional Hoare Logic solution

```
{(p1.age = 19) ∧ (p2.age = 19) ∧ p1 ≠ p2}
p1.age++
{(p1.age = 20) ∧ (p2.age = 19) ∧ p1 ≠ p2}
```

Issue: scalability. What if we have 4 pointers?

```
{... ∧ p1 ≠ p2 ∧ p1 ≠ p3 ∧ p1 ≠ p4 ∧ p2 ≠ p3 ∧ p2 ≠ p4 ∧ p3 ≠ p4}
```

Alias information scales quadratically (n * n-1) with the number of pointer variables!

Implicit Dynamic Frames [Smans et al. 2009]

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

`p1.age++`

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

Not valid if p1 = p2!

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

`p1.age++`

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

OK! p1 and p2 may not overlap

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

$$\frac{\{\,P\,\}\ S\ \{\,Q\,\} \qquad R \text{ is self-framed}}{\{\,P * R\,\}\ S\ \{\,Q * R\,\}}$$

- Example application

```
{ acc(p1.age) * p1.age = 19 * acc(p2.age) * p2.age = 19 }
p1.age++
{ /* what goes here? */ }
```

The Frame Rule

$$\frac{\{\ P * R\ \}\ S\ \{\ Q * R\ \} \qquad \text{R is self-framed}}{\{\ P * R\ \}\ S\ \{\ Q * R\ \}}$$

- Example application

note: R is self-framed!

```
{ acc(p1.age) * p1.age = 19 * acc(p2.age) * p2.age = 19 }
p1.age++
{ /* what goes here? */ }
```

The Frame Rule

$$\frac{\{ P * R \}\ S\ \{ Q * R \} \qquad R \text{ is self-framed}}{\{ P * R \}\ S\ \{ Q * R \}}$$

- Example application

```
{ acc(p1.age) * p1.age = 19 * [ R ] }
p1.age++
{ acc(p1.age) * p1.age = 20 * [ R ] }
```

Apply the normal assignment rule

The Frame Rule

$$\frac{\{ P * R \} \; S \; \{ Q * R \} \qquad R \text{ is self-framed}}{\{ P * R \} \; S \; \{ Q * R \}}$$

- Example application

```
{ acc(p1.age) * p1.age = 19 * acc(p2.age) * p2.age = 19 }
p1.age++
{ acc(p1.age) * p1.age = 20 * acc(p2.age) * p2.age = 19 }
```

Frame back on the rest of the formula

The Frame Rule

$$\frac{\{\, P * R \,\}\; S \;\{\, Q * R \,\} \qquad R \text{ is self-framed}}{\{\, P * R \,\}\; S \;\{\, Q * R \,\}}$$

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

- Anti-example

```
{ acc(p1.age) * p1.age = 19 * p2.age = 19 }
p1.age++
{ /* what goes here? */ }
```

The Frame Rule

$$\frac{\{\,P * R\,\}\,S\,\{\,Q * R\,\} \qquad R \text{ is self-framed}}{\{\,P * R\,\}\,S\,\{\,Q * R\,\}}$$

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

- Anti-example

```
{ acc(p1.age) * p1.age = 19 * p2.age = 19 }
p1.age++
{ acc(p1.age) * p1.age = 20 }
```

The best we can do is drop the unframed information from the formula

Gradual Implicit Dynamic Frames

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

`p1.age++`

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

Not valid if p1 = p2!

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

`p1.age++`

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

OK! p1 and p2 may not overlap

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

`p1.age++`

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

OK statically; requires run-time check
Useful if you don't want to specify whether p1 and p2 alias:
? could be "acc(p1.age) && p1 = p2"

Consequences of Implicit Dynamic Frames

- Gradual types can help with self-framing
  - We can ignore frames just by writing “? $\wedge$ *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:
  - ϕ_A = (? $\wedge$ x.f = 2)
  - ϕ_B = (acc(x.f) $\wedge$ x.f =2 $\wedge$ 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