

# Gradual Verification and maybe something about 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

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Related Work . . . . .	5
2.2	Abstract Gradual Typing . . . . .	5
2.3	Implicit Dynamic Frames . . . . .	5
2.3.1	Self-Framing . . . . .	5
2.4	Hoare Logic . . . . .	5
<b>3</b>	<b>A Static Verification System</b>	<b>7</b>
3.1	Syntax . . . . .	7
3.2	Static Semantics . . . . .	7
3.3	Dynamics Semantics . . . . .	7
<b>4</b>	<b>A Gradual Verification System</b>	<b>9</b>
<b>5</b>	<b>Evaluation</b>	<b>11</b>
5.1	Limitations . . . . .	11
<b>6</b>	<b>Conclusion</b>	<b>13</b>
6.1	Future Work . . . . .	13
<b>7</b>	<b>Appendix</b>	<b>15</b>
<b>8</b>	<b>UNSORTED</b>	<b>17</b>
8.1	HoareMotivEx . . . . .	17
8.2	MotivationExamples . . . . .	17
8.2.1	As extension to unverified setting . . . . .	17
8.2.2	As extension to fully verified setting . . . . .	18



# **1 Introduction**

## **1.1 Motivation**





## **2 Background**

### **2.1 Related Work**

### **2.2 Abstract Gradual Typing**

### **2.3 Implicit Dynamic Frames**

#### **2.3.1 Self-Framing**

### **2.4 Hoare Logic**



## **3 A Static Verification System**

### **3.1 Syntax**

### **3.2 Static Semantics**

### **3.3 Dynamics Semantics**



## **4 A Gradual Verification System**



## **5 Evaluation**

### **5.1 Limitations**





## **6 Conclusion**

### **6.1 Future Work**



## 7 Appendix



# 8 UNSORTED

## 8.1 HoareMotivEx

Hoare Logic as formal setting

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

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

## 8.2 MotivationExamples

### 8.2.1 As extension to unverified setting

“Make dynamic setting more static”

Motivating example:

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

Motivating example with potential leak:

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

Motivating example with argument validation:

```

boolean hasLegalDriver(Car c)
{
    if (!(c != null))
        throw new IllegalArgumentException("expected c != null");
    if (!(c.driver != null))
        throw new IllegalArgumentException("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 (run ESC/Java [1])

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 (RAC, run JML4c, TODO: <http://www.cs.utep.edu/cheon/download/jml4c.jar>)

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.

### 8.2.2 As extension to fully verified setting

“Make static setting more dynamic”

```

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 is not satisfying, since (from a formal perspective) the problem is with the method's postcondition being too weak

```

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
{

```

## 8 UNSORTED

```
int result = collatzIterations(300, seed);
// we know: result  $\in$  { 1, 2, 4 }

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

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



# Bibliography

- [1] K Rustan M Leino, Greg Nelson, and James B Saxe. Esc/java user's manual. *ESC*, 2000:002, 2000.