Karlsruhe Institute of Technology

Institute for Program Structures
and Data Organization (IPD)

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

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association **www.kit.edu**

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 3

2 Motivation 5

3 Background and Motivation 7

3.1 Categorization of existing programming languages 7

3.2 Abstract Gradual Typing 7

3.3 Implicit Dynamic Frames 7

3.3.1 Self-Framing 7

3.4 Hoare Logic 7

3.5 Motivation 7

4 Language 9

5 Abstracting Static Semantics 11

5.1 Syntax 11

5.2 Static Semantics 11

5.3 Dynamics Semantics 11

6 Abstracting Dynamic Semantics 13

7 Evaluation/Analysis 15

8 Conclusion 17

8.1 Limitations 17

8.2 Future Work 17

9 Appendix 19

10 UNSORTED 21

10.1 HoareMotivEx 21

10.2 MotivationExamples 21

10.2.1 As extension to unverified setting 21

10.2.2 As extension to fully verified setting 22

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, usually driven by cost-benefit and usability considerations, haven't adopted this level of static analysis in the first place. The purpose of this work is to weaken if not remove some of these limitations at the cost of turning some static checks into runtime checks, whenever inevitable.

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.

Gradual typing is the approach to provide the full continuum between static and dynamic typing, ...

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?

2 Motivation

more practical view? Intro? Background?

3 Background and Motivation

3.1 Categorization of existing programming languages

3.2 Abstract Gradual Typing

3.3 Implicit Dynamic Frames

3.3.1 Self-Framing

3.4 Hoare Logic

3.5 Motivation

or here?

4 Language

5 Abstracting Static Semantics

5.1 Syntax

5.2 Static Semantics

5.3 Dynamics Semantics

6 Abstracting Dynamic Semantics

7 Evaluation/Analysis

8 Conclusion

Recap, remind reader what big picture was. Briefly outline your thesis, motivation, problem, and proposed solution.

8.1 Limitations

8.2 Future Work

9 Appendix

10 UNSORTED

10.1 HoareMotivEx

Hoare Logic as formal setting

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

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

10.2 MotivationExamples

10.2.1 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/jm

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.

10.2.2 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
{
    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;
}
```

Bibliography

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