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

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

more practical view? Intro? Background?

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

[14] 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.

[8] JML => static verification

[4] JML => RAC

[13] Spec#

[3] Spec# extension (concurrent OO)

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

[11] Code Contracts! Combines runtime and static checking

[5] “verified design-by-contract”

2.2 Hoare Logic

...for static semantics

2.3 Related Work

2.3.1 Abstracting Gradual Typing

[16] Gradual Typing for Functional Languages

apply their gradual typing approach to other areas

[17] Refined criteria for gradual typing gradual guarantee

[6] 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.

[7] 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 [21] 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.

[20] 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

[18] IDF

[9] 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

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

[15] 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.

Self-Framing

3 Gradualization of a static... / 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.

We will now intrude a very simple Java-like language that uses Chalice/Eiffel/Spec# sub-syntax to express method contracts.

3.1 A Statically Verified Language

3.1.1 Syntax

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

Programs consist of classes and a main method, represented directly as the list of its instructions.

Further stuff

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

3.1.2 Static Semantics

3.1.3 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.

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

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

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

3.1.4 Dynamic Semantics

3.2 Deriving a Gradually Verified Language

3.2.1 Abstracting Static Semantics

3.2.2 Abstracting Dynamic Semantics

4 Implementation

...of 3.1

4.1 Prepare Hoare Rules

4.1.1 Deterministic Functions

4.2 Gradualize Hoare Rules

4.2.1 Gradualize Functions

4.3 Enhancing an Unverified Language

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 Limitations

6.2 Future Work

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 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 [10])

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.

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 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;
}
```

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] David Crocker. Safe object-oriented software: the verified design-by-contract paradigm. In *Practical Elements of Safety*, pages 19–41. Springer, 2004.

[6] Ronald Garcia, Alison M Clark, and Éric Tanter. Abstracting gradual typing. *ACM SIGPLAN Notices*, 51(1):429–442, 2016.

[7] Ronald Garcia and Eric Tanter. Deriving a simple gradual security language. *arXiv preprint arXiv:1511.01399*, 2015.

[8] 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.

[9] 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.

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

[11] Francesco Logozzo Manuel Fahndrich, Mike Barnett. Embedded contract languages. In *ACM SAC - OOPS*. Association for Computing Machinery, Inc., March 2010.

[12] Bertrand Meyer. *Design by contract*. Prentice Hall, 2002.

[13] 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.

[14] Greg Nelson. Extended static checking for java. In *International Conference on Mathematics of Program Construction*, pages 1–1. Springer, 2004.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

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