How analysis can hinder source code manipulaton

And what to do about it

Michael Ernst
University of Washington

Static analysis pros and cons

- Benefits of program analysis:
  - + identifies/prevents bugs
  - + enables transformation
  - + aids re-engineering & other development tasks
- Program analysis is mandatory for effective software development
- This reliance can have negative effects:
  - - delays testing
  - - discourages restructuring/refactoring

My background & biases

- My previous research thrusts:
  - slicing
  - dynamic analysis
  - testing
  - types
  - security
  - (not clones, much)
- Today’s talk: inspired by types and testing
  - Type-checking is the analysis
    - Many of the ideas generalize more broadly
  - Transformation is the manipulation
    - Type inference
    - General restructuring

Why is program transformation hard?

- Case study: convert a program to a stronger type system

Two conclusions of this talk

1. We need to strengthen type systems
2. We need to weaken type systems

Upgrading your type system

- For untyped programs:
  - Add types to Scheme, Python
- For typed programs:
  - Generics (parametric polymorphism): Java 5
  - Information flow (e.g., Jif)
  - Pluggable type qualifiers: nullness, immutability, …
- For stronger properties:
  - Extended Static Checking: array bounds
  - Discharge arbitrary assertions

Java Generics

- Convert Java 1.4 code to Java 5 code
  – Key benefit: type-safe containers
- Instantiation problem:

```
List myList = …;  ⇒  List<String> myList = …;
```

- Parameterization problem

```
class Map {
  Object get(Object key) { … }
}
```

⇒

```
class Map<K,V> {
  V get(K key) { … }
}
```

  – Adding parameters changes the type constraints
- Series of research papers
  – OOPSLA 2004, ECOOP 2005, ICSE 2007

Java standard libraries

- The JDK libraries are not generics-correct
  - The signatures use generic types
  - The implementations do not type-check
    - Retained from Java 1.4, usually without change
- Why?
  - Old design flaws were revealed
  - Danger of refactoring was too great
  - The code already worked
  - It wasn’t worth it

Immutability

- Goal: avoid unintended side effects
  - in the context of an imperative language
- Dozens of papers proposing type systems
- Little empirical evaluation
  - Javari: 160KLOC case studies (OOPSLA 2005)
  - IGJ: 106KLOC case studies (FSE 2007)

Obstacles to evaluation and use of a type system

- Building a type checker is hard
  - Solution: Checker Framework (ISSTA 2008)
- Building a type inference is hard
  - Example: Javarifier (ECOOP 2008)
  - Inference is much harder than checking
  - No general framework for implementations
- Programs resist transformation
  - Often type-incorrect
  - Difficult to make type-correct

Eliminating null pointer exceptions

- A common analysis
  - NPEs are pervasive and important
  - Problem is simple to express
  - We should be able to solve this problem
- Goal: prove all dereferences safe
  - Suppose: 95% success rate, program with 100,000 dereferences
  - 5000 dereferences to check manually!
  - Empirically, how many are false positives?

Analysis power vs. transparency

- A powerful analysis can prove many facts
  - Example: analysis that considers all possible execution flows in your program
  - Pointer analysis, type inference
- A transparent analysis has comprehensible behavior and results
  - Results depend on local information only
  - Small change to program $\Rightarrow$ small change in analysis results
- Making an analysis more transparent
  - Concrete error cases & counterexamples, …
  - User-supplied annotations
- Do programmers need more power or transparency?
  - We need research addressing this question

Feedback: the essence of engineering

- Feedback during the development process is essential for improving software quality.
- Two useful types of feedback:
  - static checking via type systems and code analysis
  - dynamic checking via tests and experimentation

Get the feedback you need

- Static analysis:
  - Is the design sound and consistent? Does the system fit together as a whole and implement the design?
  - Guarantees the absence of errors
    - Alternative: discover piecemeal during testing or in the field.
- Experimentation and testing:
  - Does this algorithm work? Do the pieces fit together to implement the desired functionality?
  - Many important properties are beyond any static analysis
    - User satisfaction, but algorithmic properties too
- Each is most useful in certain circumstances
  - Sound reasoning can trump quick experimentation
  - Quick experimentation can trump sound reasoning

Problem: programmers cannot choose between these approaches

- A programmer should always be able to
  - execute the software to run tests
  - statically verify (types or other properties)
- Current languages and environments impose their own model of feedback
  - Depends largely on the typing discipline

Dynamically-typed languages

- Good support for testing
- At any moment, run tests
  - No need to bother with irritating type declarations
- No support for type-checking
  - And most other static analyses are also very hard

- Errors persist, and are even discovered in the field
- Simple errors turn into debugging marathons
  - Also true for statically-typed languages without support for immutability or other propertes
- Programmers attempts to emulate a type system
  - Naming conventions, comments, assertions, unit tests
  - Less effective and more work than type-checking
  - Despite its roots, aggressive testing is still desirable

Statically-typed languages

- Supports both testing and type-checking — in a specific order:

1. First, type-check: all types must be perfect
2. Then, experiment with the system

- You cannot run the system if type errors exist
- This delays the learning that comes from experimentation.
- Type system fails to capture many important properties of software
- A quick experiment might have indicated a problem and saved time overall
  - In other cases, type-checking saves time overall
- This approach stems from:
  - assumptions about what errors are important
  - desire to simplify compilers & other tools

Who is in charge?

- Both language-imposed approaches lead to frustration and wasted effort
- The programmer should be able to do either type of checking at any time
  - knows the biggest risks or uncertainties at the moment
  - knows how to address them
- … or even do both at the same time

Two research questions

- When is static feedback (types) most useful?
- When is dynamic feedback (testing) most useful?
- Not: "When does lack of *X* get most in the way?"

- What language model or toolset gives the benefits of both approaches?

Giving the benefits of both approaches

- Add a static type system to a dynamically-typed language
- Relax a static type system

Add types to a dynamically-typed language

- Popular approach among academics
  - Bring religion to the heathens
- Not yet successful
  - Java generics can be viewed as an exception
  - Maybe pluggable types will be an exception

Difficulties with re-engineering a program to add stronger types

- The design may not expressible in a statically type-safe way
  - May take advantage of the benefits of dynamic typing
  - Requires many loopholes (casts, suppressed warnings), limiting its value
- The high-level design may be instantiable type-safely, but this instantiation is not
  - Programmer had no pressure to make the program type-safe
  - Programmer chose an implementation strategy that does not preserve type safety
  - Would have been easy to chose the other implementation approach from the start
  - The type-safe version is better, but you learn that in retrospect
- There may be too many errors to correct
  - Cost of change may outweigh benefits
  - Especially if the program works now
  - Every change carries a risk
- Design may be too hard to understand
  - Better analysis tools are required

A few reasons for type errors

- Heterogeneity
- Application invariants
- Simple bugs

- Especially when the property is checkable at run time

The transformation is possible

- But if you want a typed program, it’s best to aim at that from the beginning

Relax a statically-typed language

- This approach has not been attempted before (to my knowledge)
- Start with a typed program
- Temporarily, at defined moments in the development lifecycle, ignore the types
  - Treat the program as if it had been written in a dynamically typed language
- Ignore during compilation and execution.
  - Exception: resolving overloading/dispatch
  - Exception: log any errors that would have been thrown
- “Weekend release” for type systems

Not just dynamic typing or optional typing

- Different philosophy from dynamically typed languages
- Dynamically typed execution *temporarily*, during testing or development
  - Programmer will re-introduce types after the dynamic experiment
  - Programmer views loopholes in the static type system as rarities, not commonplaces

Implementation approach

- Mask but log all errors
- If the execution fails, show the log to find the first evidence of the failure

- Prototype shows promise of this approach
- User testing is necessary

Changing type systems

- Strengthen them to enforce more guarantees
- Weaken them to enable better experimentation
- Enables better transformations as well

- Instances of:
  - combining static and dynamic analysis
  - inverting common approaches
- Can be extended to other analyses and transformations