How tests and proofs impede one another:

The need for always-on static and dynamic feedback

Michael Ernst
joint work with Michael Bayne
University of Washington

3rd Int'l Conference on Tests And Proofs
July 1, 2010

Tests and proofs

Tests and proofs are synergistic

[…, ISSTA 2006, SCP 2007, PLDI 2008, ECOOP 2008, ISSTA 2008, ASE 2009, …]

- Problem: attitudes and implementations put them at odds
- Goal: reconcile them

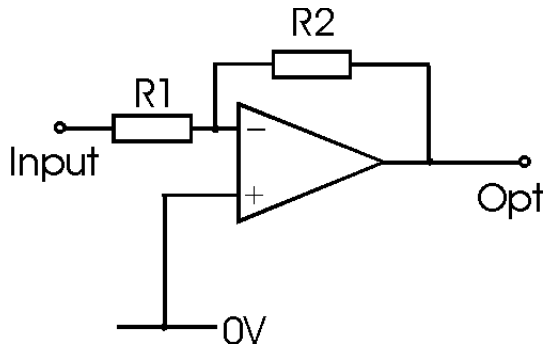
This talk: philosophy, lay of the land, problems, technical nugget, challenges

A theme: putting the developer in charge

Feedback is the essence of engineering

Software developers
get feedback from:

- dynamic analysis ("tests")
- static analysis ("proofs")
  - prime example: type systems



How do you choose a type system?

- Dynamically-typed scripting languages:
  - Faster and more flexible program development and modification
- Statically-typed programming languages:
  - More reliable and maintainable applications

You shouldn't have to choose!

Can we give programmers the benefits of both?

A type system checks some properties

Language designer chooses checks to address development needs

- Static checks
- Dynamic checks (early or late)

Later, tool developers add more checks

- Static: LCLint [Evans 94]
- Dynamic: Purify [Hastings 92]

Why don't tool developers remove checks?

Why we ♥ static typing (proofs)

- Documentation (machine-checked)
- Correctness/reliability
- Refactoring
- Performance (optimizations)
- Faster development? Yes [Gannon 77, Prechelt 98], no [Hannenberg 09, 10]

Why we ♥ dynamic typing (= Why we  static typing)

- Suited to rapidly-changing requirements
- More flexible code
  - Meta-programming: dynamic program behavior, add fields, change class
- No false positive warnings
  - Every static type system rejects some correct programs
- More interactive, more fun
- Ability to run tests at any time

```
Object now = new Date();
now.getMonth();   // illegal!
```

Other reasons we ♥ dynamic languages (besides the dynamic type system)

- Libraries (e.g., Rails)
- Conciseness (type inference, collection literals)
- Read-eval-print loop; no edit-compile-test cycle

Neither tests nor proofs dominates the other

- Tests:
  - Reveals emergent behavior
  - Quickly builds insight
  - Type system captures only a few properties
    - Not: user satisfaction, algorithmic properties
- Proofs:
  - Ensures consistency throughout the whole program
  - Guarantees absence of errors
  - Encourages good design
    - Caveat: some good designs are untypable

Two verification technologies that help programmers

Programmer should be able to choose the best one

Your programming language imposes a development model

Dynamically-typed languages favor testing

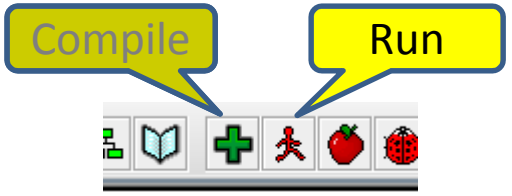
Dynamically-typed languages inhibit proofs

Statically-typed languages favor type-checking

Statically-typed languages inhibit testing

Problem: the programmer is not in charge

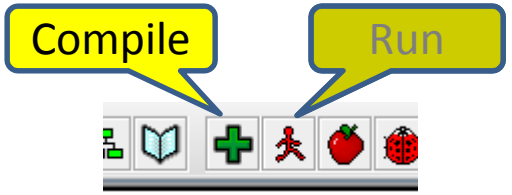
Dynamic languages inhibit proofs



- Good support for testing, at any moment
  - No fuss and bother of appeasing the type checker
- No possibility of static type checking
  - …and most other static analyses are also very hard
  - Programmers attempt to emulate a type system
    - Naming conventions, comments, extra assertions, tests
    - Inevitably, hard-to-debug type errors remain

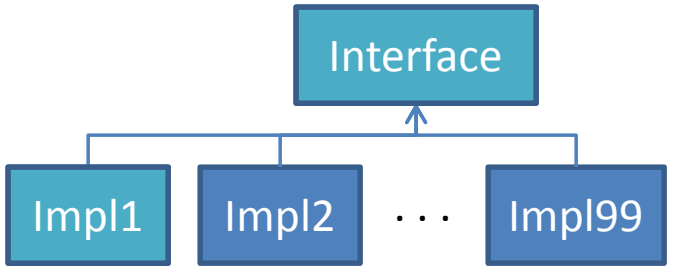
Example: a field crash after hours of execution

Static languages inhibit testing

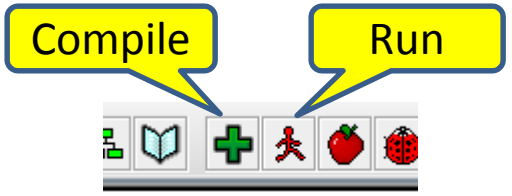


- Support *both* testing and type-checking
  - … in a specific order
- No tests are permitted until types are perfect
  - Delays learning from experimentation
  - Not all properties are captured by types
  - Assumption: what errors are important?

Example: change an interface & 1 implementation, then test



Putting the developer in charge



Each language-imposed approach leads to frustration and wasted effort

The developer should be able to get feedback from tests *or* proofs (or both!) at any time

- Knows the biggest risks or uncertainties
- Knows how to address them

Open research questions

1. When is static feedback (types, proofs) most useful?
2. When is dynamic feedback (testing) most useful?

- Not: "When does lack of *X* get most in the way?"

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

Enabling tests and proofs

- Workarounds: emulate dynamic typing
- Prototype in dynamic language, deliver in static
- Each program is half-static, half-dynamic
  - Add types to a dynamic language
  - Add `Dynamic` type to a static language
  - Result is unsafe and inflexible

- Full static and dynamic views of the program, at any moment during development

Outline

- Workarounds: emulate dynamic typing
- Prototype in dynamic language, deliver in static
- Each program is half-static and half-dynamic
  - Add types to a dynamic language
  - Add `Dynamic` type to a static language
- Full static and dynamic views of the program, at any moment during development
- Conclusion

Workarounds: emulate dynamic typing in a statically-typed language

1. Partial compilation or execution

   Don't compile code with type errors

   Comment out; modify build file; unexecuted casts

2. Explicit dynamic typing

   `Dynamic`, `Object`, `void*`

   No documentation benefits

   Prone to untypeable, confusing design

Problems with workarounds

- Reasoning burden
  - Identify boundary between checked & unchecked
- Transformation burden
  - Represent the boundary (coarsely) to the type system
  - Later, undo work (in an order dictated by the type system)
- Boundary changes with time

Workarounds indicate a need for better mechanisms!

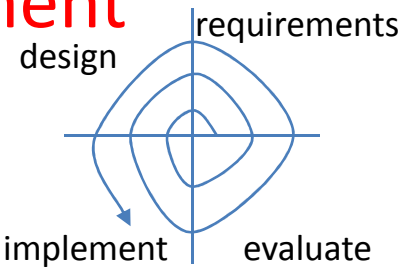
Outline

- Workarounds: emulate dynamic typing
- Prototype in dynamic language, deliver in static
- Each program is half-static and half-dynamic
  - Add types to a dynamic language
  - Add `Dynamic` type to a static language
- Full static and dynamic views of the program, at any moment during development
- Conclusion

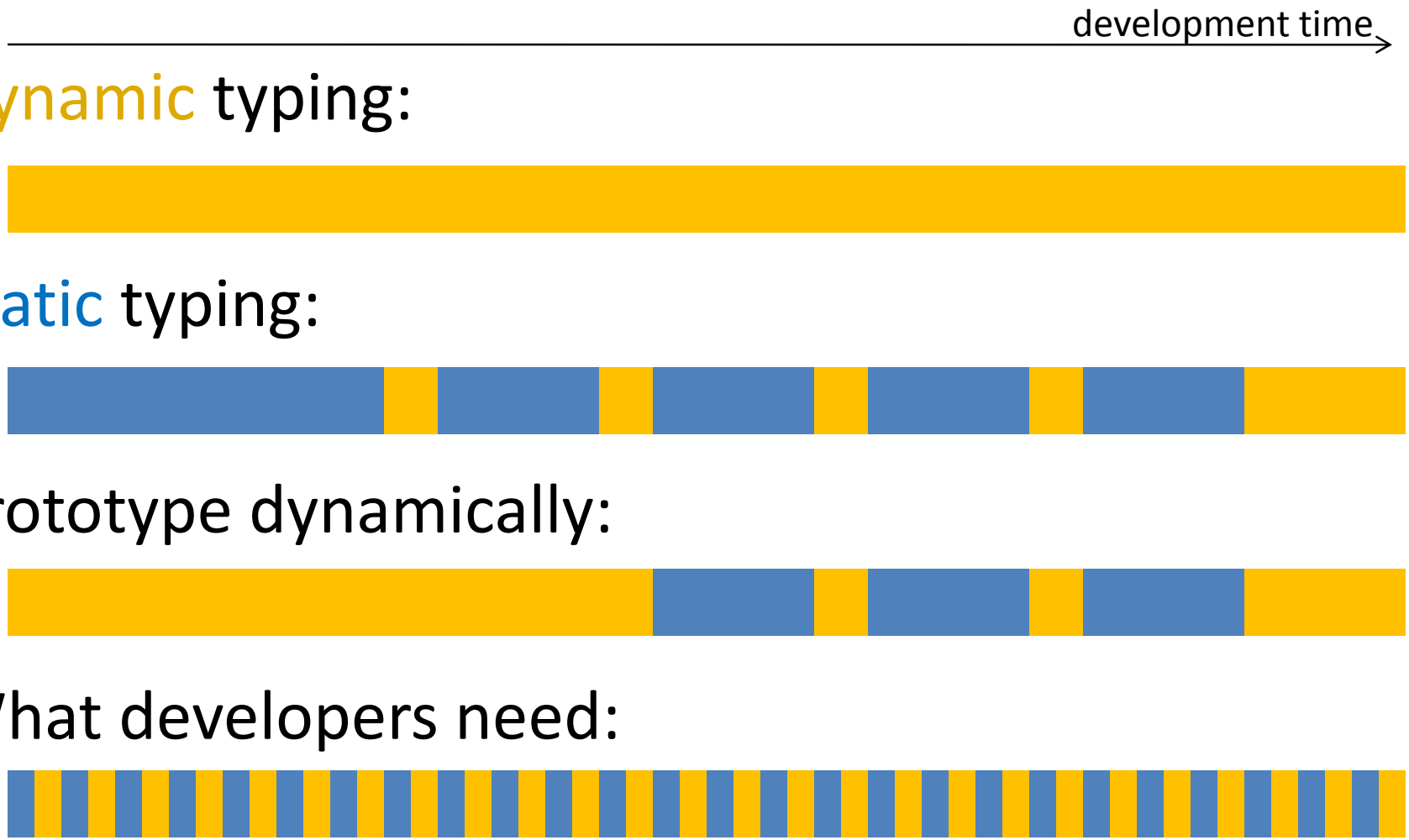
Dynamic prototype ⇒ static product

1. Start coding in a dynamic language
2. Throw away the prototype
3. Rewrite in a static language
4. Ship the statically-typed version

The great fallacy of hybrid static/dynamic typing:
A transition from testing to proving exists
Programmers need early static checking
Programmers need late dynamic development
(Consider spiral [Boehm 86],
agile [Beck 01] development)



Switching between dynamic and static feedback



The need for early static feedback

- Documentation
  - Provisional type
  - Types are more inviting than comments
  - Type-checker shows where `Dynamic` is needed
- Correctness
  - Aids in debugging
  - Some aspects of the code can be finalized early
- Improves program design
  - Keeps type-checking in the developer's mind
  - Encourages statically-typeable design
  - Indicates divergence quickly, when changes are easier

The need for early dynamism

- Prototyping
  - Structure changes quickly
  - Code is written quickly and thrown away
    - Wasteful to create separate implementations and interfaces
  - Code fits “in the developer’s head”
  - Code becomes obsolete, inconsistent
- API sketching
  - Iterate on interface, implementation, and client
  - Developer can focus on vertical slices of functionality

The need for late dynamism in software evolution

- Programmer wants to make and test a change
  - Doesn't want to delay the tests
- Representation or interface changes
  - Change `String` to a data structure
- Library replacement
  - Test parts of the client
- Exploratory changes
  - Restructuring may require many type changes and workarounds before determining viability
  - Likely to be wasted work

Problems with prototyping in a dynamic language

- No benefit of early static checking
- No benefit of late dynamic development

- Design may not be implementable in the static language
  - Due to features of the dynamic language
- Forced to throw away the prototype
  - The freedom to throw it away is good
  - Being forced to throw it away is bad
  - The developer should decide what to discard
- Prototyping approach is inefficient
- We need seamless integration in a single language

Outline

- Workarounds: emulate dynamic typing
- Prototype in dynamic language, deliver in static
- Each program is half-static and half-dynamic
  - Add types to a dynamic language
  - Add `Dynamic` type to a static language
- Full static and dynamic views of the program, at any moment during development
- Conclusion

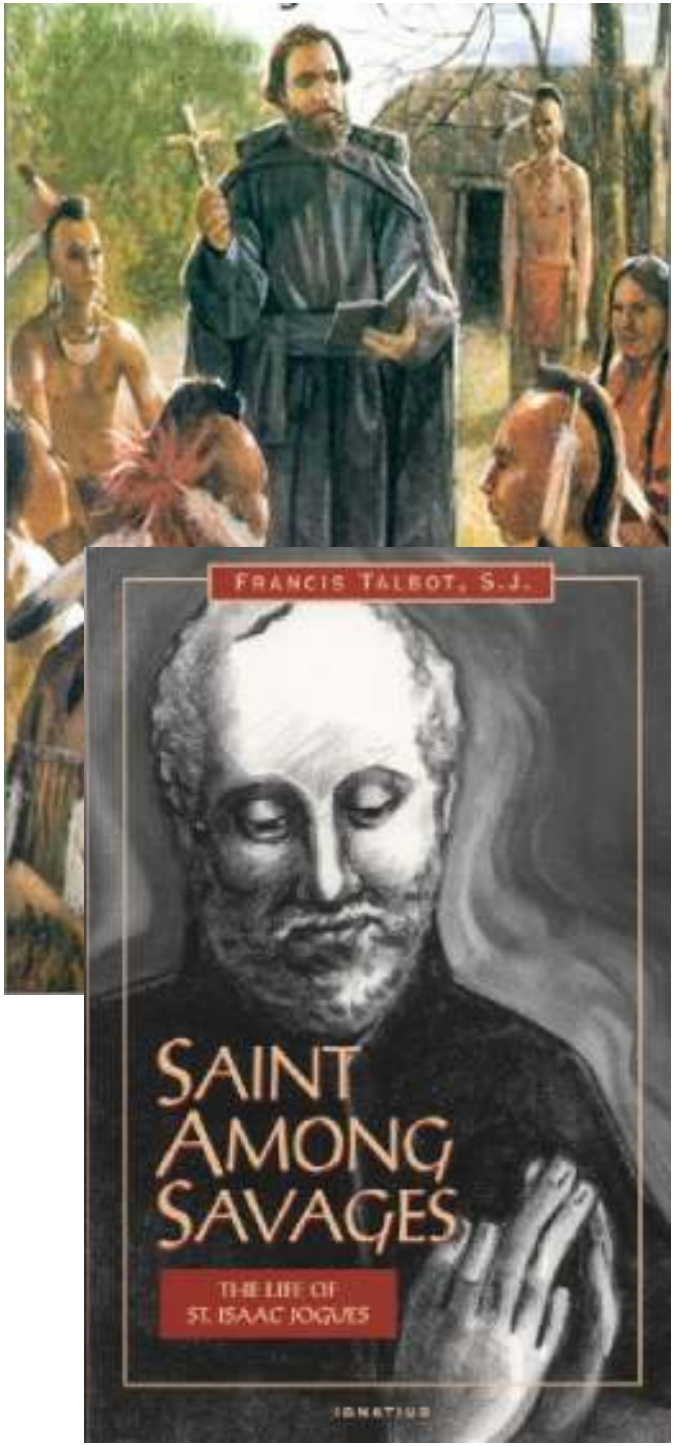
Add types to a dynamic language

Popular among academics

Less popular among practitioners

Realities:

- Dynamism is a key selling point, not incidental
- Types cannot handle all language features
  - Programming paradigms do not translate
- Poor cost/benefit



Upgrading your type system

- For untyped programs:
  - Add types to Scheme [Cartwright 91] , Erlang [Nyström 03], Python [Ancona 2007], Java [Lagorio 07], Ruby [Furr 09], PHP [Camphuijsen 09], …
- For typed programs:
  - Immutability: C/C++ **`const`**
  - Generics (parametric polymorphism): Java 5, C# 2
  - Haskell type classes
  - Information flow: Jif [Myers 97]
  - Pluggable types [Bracha 04, ISSTA 08]: nullness, immutability, …
- For even stronger properties:
  - Extended Static Checking: array bounds [Leino 98, Flanagan 02]
  - Discharge arbitrary assertions: theorem proving [Ou 04, Flanagan 06]

Java Generics

Convert Java 1.4 code to Java 5 code

Key benefit: type-safe containers

[OOPSLA 04, ECOOP 05, ICSE 07]

- 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

Java standard libraries

The JDK libraries are not generics-correct

- Most 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 05]
- IGJ: 106KLOC case studies [FSE 07]

Building a type checker is hard

- Solution: Checker Framework [ISSTA 08]

What bugs can you find & prevent?

	The annotation you write:
• Null dereferences	`@NonNull`
• Mutation and side-effects	`@Immutable`
• Concurrency: locking	`@GuardedBy`
• Security: encryption, tainting	`@Encrypted` `@Untainted`
• Aliasing	`@Linear`
• Equality tests	`@Interned`
• Strings: localization, regular expression syntax	`@Localized` `@Regex`
• Typestate (e.g., open/closed files)	`@State`
• You can write your own checker!	

Pluggable type-checking

Untyped legacy language: Java
Typed language: Java with type qualifiers

Checker Framework eases type system construction

- Previous frameworks were unsatisfactory
- Academics built dozens of type-checkers
- Teaching: usable in first class in the major
- Industry: in daily use
- Influenced the design of Java 7 (type annotations)

Cost/benefit: A mixed success

+ Final design is clearer, better, documented, checked
+ You can ignore the type-checking when you want to
– It takes a lot of time
  - Annotation burden: minor
  - Understanding & improving design: major
– Adds no new features
– You already fixed the worst bugs
– May introduce new errors

It's hard to add stronger types

- The design is inherently not type-safe
  - Uses dynamic typing, dynamic checks, heterogeneity
  - Requires many loopholes (casts, suppressed warnings)
- This instantiation of the design is not type-safe
  - Programmer had no pressure to make the program type-safe
  - The type-safe version is better, but you learn that in retrospect
  - Another implementation would have been easy at the start
  - Design decisions proliferate and ossify
- Design is too hard to understand
- There are too many errors to correct or suppress
  - Every change carries a risk; cost may outweigh benefits
- Problem: type-checking is too late (not: too much of it)
  - Late changes are costly

Write and check types from the beginning

- Adding types after the fact is doomed
  - Whole-program type inference probably is, too
- New code is a good testing ground for new type systems
  - But, most development effort is maintenance
- We can't ignore legacy code
  - Evaluation
  - Practicality and impact

Another stymied project: Eliminating null pointer exceptions

- Goal: prove all dereferences safe in an existing program
  - NPEs are pervasive and important
  - Problem is simple to express

[Barnett 04, Spoto 08]

- Suppose: 99% success rate, 200 KLOC program
  - 1000 dereferences to check manually!
  - Program changes will be necessary

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

To make an analysis more transparent:

- Concrete error cases & counterexamples, …
- User-supplied annotations & restructuring

Open question: Do programmers need more power or transparency? When?

Outline

- Workarounds: emulate dynamic typing
- Prototype in dynamic language, deliver in static
- Each program is half-static and half-dynamic
  - Add types to a dynamic language
  - Add `Dynamic` type to a static language
- Full static and dynamic views of the program, at any moment during development
- Conclusion

Incremental/gradual/hybrid typing

Statically-typed portion is safe

Run-time type errors are the fault of the dynamic code or the boundary

Programmer must:

- Decide the boundary
- Indicate it with type annotations

Research challenge: behavior at the boundary

- Correctness
- Blame control
- Efficiency

Correctness

- Retain all assertions that may fail [Ou 04, Flanagan 06]
- Objects may need to carry types [Siek 07, Herman 07]
- Contracts at the boundary [Findler 02, Gray 05]

Blame assignment

- A run-time error might arise in typed code
- Find the root cause in untyped code
- Usually located at the boundary
- Stop the program as soon as detected

[Findler 01, Tobin-Hochstadt 06,08, Furr 09, Wadler 09]

Efficiency

- Statically discharge assertions
  - Omit run-time checks
  - Remove wrappers
  - Smaller representations
- Permit tail recursion
- Extension to higher-order functions
- Omit blame assignment [Bloom 09]

[Herman 09, Siek 09,10]

Disadvantages

- Mostly starts from a dynamic mindset
  - But, programmers are more likely to accept
- Little encouragement for good design
  - Types not required, type checker errors assumed
- Few run-time guarantees
- Programmer must manage boundary
- Evaluation needed

Outline

- Workarounds: emulate dynamic typing
- Prototype in dynamic language, deliver in static
- Each program is half-static and half-dynamic
  - Add types to a dynamic language
  - Add `Dynamic` type to a static language
- Full static and dynamic views of the program, at any moment during development
- Conclusion

Dynamic interpretation of static code

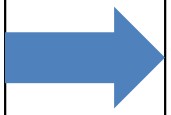
Write in a statically-typed language

The developer may always execute the code

To execute, ignore the types (mostly)

Convert every type to **`Dynamic`**

```
class MyClass {
 List<String> names;
 int indexOf(String name) {
  …
 }
}
```



```
class MyClass {
 Object names;
 Object indexOf(Object name) {
  …
 }
}
```

Type-removing transformation

- Primitive operations (`+`, `>`, `[]`, `if`) dynamically check their argument types
- Method invocations and field accesses are performed reflectively
  - Run-time system re-implements dynamic dispatch, etc.

- Compilation always succeeds
  - Code must be syntactically correct
- Code can be run

Why wasn't this done before?

- Rigid attitudes about the "best" feedback
- Divide between static and dynamic researchers
- Aping of developer workarounds
- Choices made for the convenience of tools
- Difficult to get right

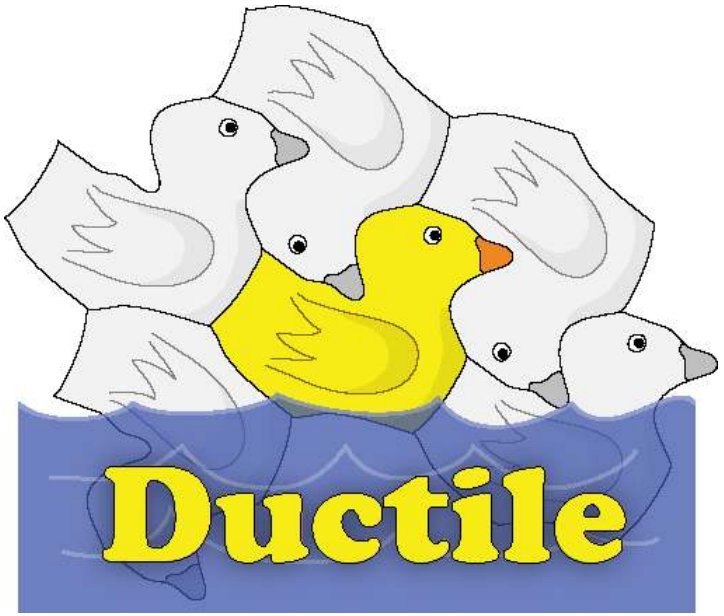
Question: What other problems have this feel?

Challenges to dynamic interpretation

1. Preserve semantics for type-correct programs
2. Useful semantics for type-incorrect programs

- Exploration of these challenges: Ductile
  - DuctileJ is a dialect of Java

http://code.google.com/p/ductilej/



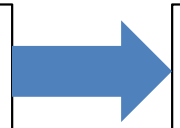
Preserve semantics of well-typed programs

1. Static types affect semantics (e.g., overloading)
2. Reflective calls yield different exceptions
3. Interoperation with un-transformed code
4. Meta-programming model limitations

Method overloading

Transformed declarations have same signature

```
void foo(int x) { … }
void foo(Date x) { … }
```



```
void foo(Object x) { … }
void foo(Object x) { … }
```

Overload resolution depends on static types

- Do not implement multi-method dispatch!

Solution:

- Dummy type-carrying arguments to disambiguate
- Resolution at run time if necessary

Exceptions

Reflective calls have different checked exceptions

- Compiler error
- Different run-time behavior

Solution:

- Wrap exceptions
- Catch, unwrap, and re-throw with correct type

Interfacing with non-transformed code

- Tool must operate on source code
  - Because the code doesn't compile!
- Bytecode transformation is possible
  - Libraries are usually general enough already
- Solution: untransformed code is treated like a primitive operation
  - Signatures inherited from libraries remain un-transformed – e.g., **`hashCode()`**

Reflection

Reflection results reflect transformed program

Solution: Un-transform signatures in results

Cannot reflectively call:

- `super` constructor
- `super` method call
- Chained constructor call
- Anonymous inner class constructor

Solution: Fight magic with more magic

Useful semantics for ill-typed programs

Give a semantics to ill-typed programs

- Formalization is a research challenge

Best-effort interpretation of the program

Accommodations for ill-typed programs

Each of these accommodations could be disabled:

- Assignment: permitted, regardless of declared and actual types
- Missing fields: add new field
- Method invocation
  - Search for closest matching signature in run-time type ("duck typing")
  - If none, generalize or refine type

Leave these on even in code that type-checks

Example code paradigms:

- Interface declarations: no **`implements`** is needed
- Type sketching: make up a name, or use **`var`**

Debugging and blame assignment

At each assignment and pseudo-assignment:

- Check against static type and record the result

If the program fails:

- Show relevant type failures (true positives)

If the program succeeds:

- User can choose to ignore or examine the log

Blame assignment as late as possible

- Contrasts with other work: as early as possible

Feedback vs. action

A user has a choice to interact with, or to ignore:

- tests
- version control conflicts
- performance tuning
- lint
- theorem-proving

Why doesn't the type-checker provide this choice?

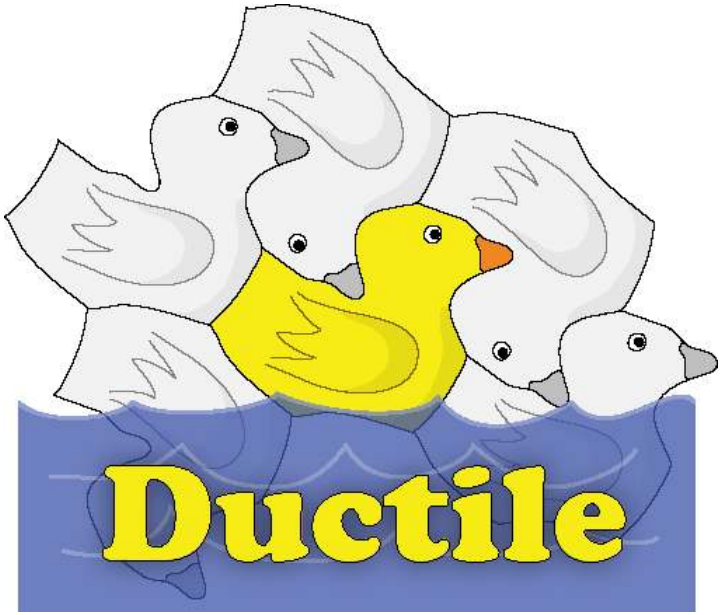
- Tool builder convenience and doctrine

Should separate when feedback is discovered and acted upon

Weekend release for type systems

- Goal is a statically-typed program
  - Occasionally, the type-checker gets in the way
- Disable type-checker temporarily
  - Continue thinking about types!
- Write a slice of a correctly-typed program
  - Missing branches, exception handling, etc.
- If you need **`Dynamic`**, use it
  - Or design a better type system!
- Efficiency is not a major concern
  - Human attention is the scarce resource

Prototype implementation



DuctileJ: a dialect of Java

Publicly available at
http://code.google.com/p/ductilej/

To use: add **`detyper.jar`** to your classpath

Try it and give us feedback

Assessment: Preserving semantics

Program	sLOC	Tests
Google Collections	51,000	44,760
HSQLDB	76,000	3,783
JODA Time	79,000	3,688

Assessment: Usefulness for prototyping

Prototyping an address book manager

Delayed specification of:

- Interfaces
  - After implementation complete, defined the interface
- Checked exceptions
  - No dummy **`try`** or **`throws`** constructs
- Access control
  - Didn't make members **`public`** unless necessary

Partial implementation of:

- Interfaces
  - Object that implemented only **`add`** acted as a **`List`**
  - **`Iterable`**
- Exception handling
  - Missing **`catch`** clauses

Outline

- Workarounds: emulate dynamic typing
- Prototype in dynamic language, deliver in static
- Each program is half-static and half-dynamic
  - Add types to a dynamic language
  - Add `Dynamic` type to a static language
- Full static and dynamic views of the program, at any moment during development
- Conclusion

Static ♥ dynamic

Synergy and duality between tests and proofs [PASTE 04]

- Type-checking
- Slicing: what computations can affect a value
- Memory safety: Purify, LCLint
- Atomicity checking [Flanagan 03]
- Specification checking
- Specification generation [Cousot 77, ICSE 99]

Look for more analogies and gaps!

Lessons

- Put the programmer in control
  - Separate feedback from action
  - Programmer workarounds $\Rightarrow$ new solutions
- Scratch your own itches
- Complement existing analyses
  - Characterize strengths, needs
  - Blend two lines of research
- Power/transparency tradeoff
- Change tool design based on insights