Always-available static and dynamic feedback: Unifying static and dynamic typing

Michael Bayne
Richard Cook
Michael D. Ernst

University of Washington

Ductile

Static feedback helps programmers

- Correctness/consistency throughout the program
- Types are machine-checked documentation
- Supports other analyses (refactoring, …)

Dynamic feedback helps programmers

- Testing builds insight, reveals emergent behavior
- Checks properties that types do not capture
  - User satisfaction, algorithmic properties, …
- No false positive warnings

Complementary verification technologies

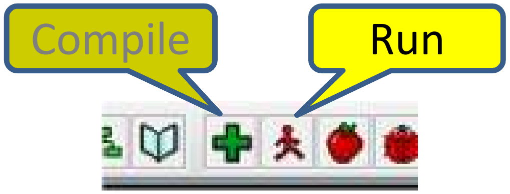
Static type-checking is not always the most important goal

Dynamic testing is not always the most important goal

Idea: let the programmer choose the best approach, at any moment during development

- Fast, flexible development, as with dynamic types
- Reliable, maintainable applications, as with static types

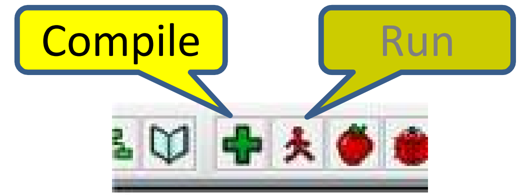
Dynamic languages inhibit reasoning



- Good support for testing, at any moment
- No possibility of static type checking

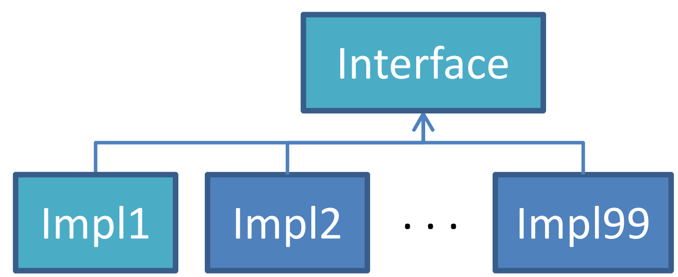
Example problem:
a field crash after hours of execution

Static languages inhibit testing



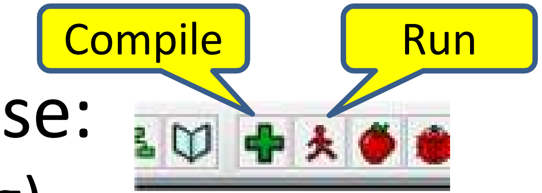
- Support *both* testing and type-checking
  - … in a specific order
- No tests are permitted until types are perfect
  - Delays learning from experimentation

Example problem:
cannot change an interface &
1 implementation, then test



Result: frustration, wasted effort, workarounds

Putting the developer in charge



At any moment, developer can choose:

- static feedback (sound type-checking)
- dynamic feedback (execution, testing)

The Ductile approach:

- Write types from the outset
  - Programmer has types in mind
  - Run the type-checker at any time
- Execute a type-erased program
  - Temporarily ignore types
  - Do all checks dynamically
  - Execute a slice of a correctly-typed program

Feedback vs. action

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

- tests
- lint
- theorem-proving
- code reviews
- performance tuning
- version control conflicts
- … but no choice about the type-checker

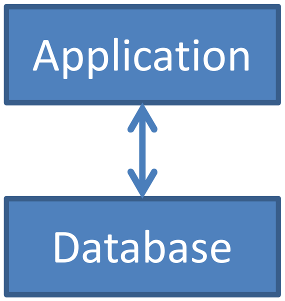
Need to separate when feedback is discovered and acted upon

Outline

- Motivation and approach
- Evaluation
  - **Prototyping**
  - Evolution (refactoring)
- Implementation
- Related work
- Conclusion

Prototyping case study

Address book architecture:



Goal: create an email address book

Tool: Ductile implementation for Java

Developers:

- >10 years commercial experience
- prefer statically-typed languages

Duck typing and access control

```
class AddressBook {
  …
  Database db = new InMemoryDB();
  db.getName(emailAddr);
  …
}
```

Detyped declaration

Call uses reflection

```
class InMemoryDB {
  getName(String s)  {…}
}
```

- When app is complete, define the interface
- Advantage: didn't have to keep interface up to date with rapidly evolving prototype
  - Experimental client code had used other methods

Checked exceptions

- For "checked exceptions", Java requires a **`try/catch`** block or a declaration
- Deferred writing these until design was stable
- Advantages:
  - Focus on main functionality while experimenting
  - Don't insert placeholder error code
  - No dummy constructs: **`try`**, **`catch`**, **`throws`**

Partial implementations

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

Sufficient for use cases that exercise a subset of functionality

Alternative: IDE "automatic fixes"

An IDE could have made the code type-check

- Add methods to **`Database`** interface
- Set methods/fields to **`public`**
- Add **`try`**/**`catch`** blocks or declare more exceptions

This would have degraded the code

- May not indicate this is a temporary experiment
- Likely to be forgotten and left in final code

Prototyping case study conclusion

Key advantages:

- Avoid signature pollution, by deferring details until design is stable
  - Interfaces
  - Access control
  - Exception-handling
- Test with partially-defined code

Outline

- Motivation and approach
- Evaluation
  - Prototyping
  - **Evolution (refactoring)**
- Implementation
- Related work
- Conclusion

Evolution case study

- Proposed change in class Figure in JHotDraw:
  - containsPoint(int x, int y) $\Rightarrow$ containsPoint(Point p)
- Goal: fast evaluation of refactoring
  - Evaluate the change by running test TriangleFigureTest
  - After evaluating, decide whether to continue or undo

3 key required changes:

- Figure.containsPoint: change signature
- TriangleFigure.containsPoint: change signature and body
- TriangleFigureTest: change call to containsPoint

Comparison of refactoring approaches

- Manual: **24** edits
  - 14 definitions of containsPoint
  - 10 calls to containsPoint
- Eclipse: **1** refactoring **+ 16** manual edits
  - Used "Change Method Signature" refactoring
- Ductile: **3** edits
  - Developer only had to make the key edits to evaluate the refactoring

Refactoring case study conclusion

Ductile approach:

- Fast evaluation with few edits
- General approach
  - Many program transformation tasks lack tool support

Need both static *and* dynamic feedback
in *all stages* of software development

Late discovery of any problem is costly

Outline

- Motivation and approach
- Evaluation
- **Implementation**
- Related work
- Conclusion

Ductile implementation

DuctileJ is a dialect of Java

Transparent to use:

Add **`detyper.jar`** to your classpath

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

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

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

- Compilation always succeeds
  - Code must be syntactically correct
- Code can always be run
  - Run-time failures are possible

Challenges to dynamic interpretation

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

Preserve semantics of well-typed programs

Goal: an execution through well-typed code behaves exactly as under Java

Challenges:

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

More challenges: type resolution, arrays, `final`, primitive operators, control flow constructs, widening/narrowing, annotations/enums, outer `this`, anonymous inner classes, definite assignment, varargs, partially implemented interfaces, security manager, primitive vs. object equality, …

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

```
int readChar(InputStream in) {
  try {
    return in.read();
  } catch (IOException e) {
    return -1;
  }
}
```

```
Object readChar(Object in) {
  try {
    return RT.invoke("read", in);
  } catch (IOException e) {
    return -1;
  }
}
```

`RT.invoke` does not throw `IOException`

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

Detyper must operate on source code

- Because the code doesn't compile!

Bytecode transformation is possible for libraries

- But programmer's focus is not the library

Solution: untransformed code is treated like a primitive operation

- Signatures inherited from libraries remain un-transformed – e.g., **`hashCode()`**

Reflection and serialization

Cannot reflectively call:

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

Solution: Fight magic with more magic

Reflection and serialization observe the transformation

Solution: Un-transform signatures in results

[Tatsubori 2004, McGachey 2009]

Assessment: Preserving semantics

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

We edited 23 lines of code and 49 lines of tests
to work around DuctileJ's reflection/serialization limitations

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 enabled/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

Perform detyping even for 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:

- Check against static type and record the result
- Never halt the program because of a mismatch

If the program succeeds:

- User can choose to ignore or examine the log

If the program fails:

- Show relevant recorded type failures (true positives)

Innovation: Blame assignment as late as possible

Outline

- Motivation and approach
- Evaluation
- Implementation
- **Related work**
- Conclusion

Combining static and dynamic typing

1. Add types to a dynamic language
2. Add `Dynamic` to a static language
3. Ad-hoc workarounds

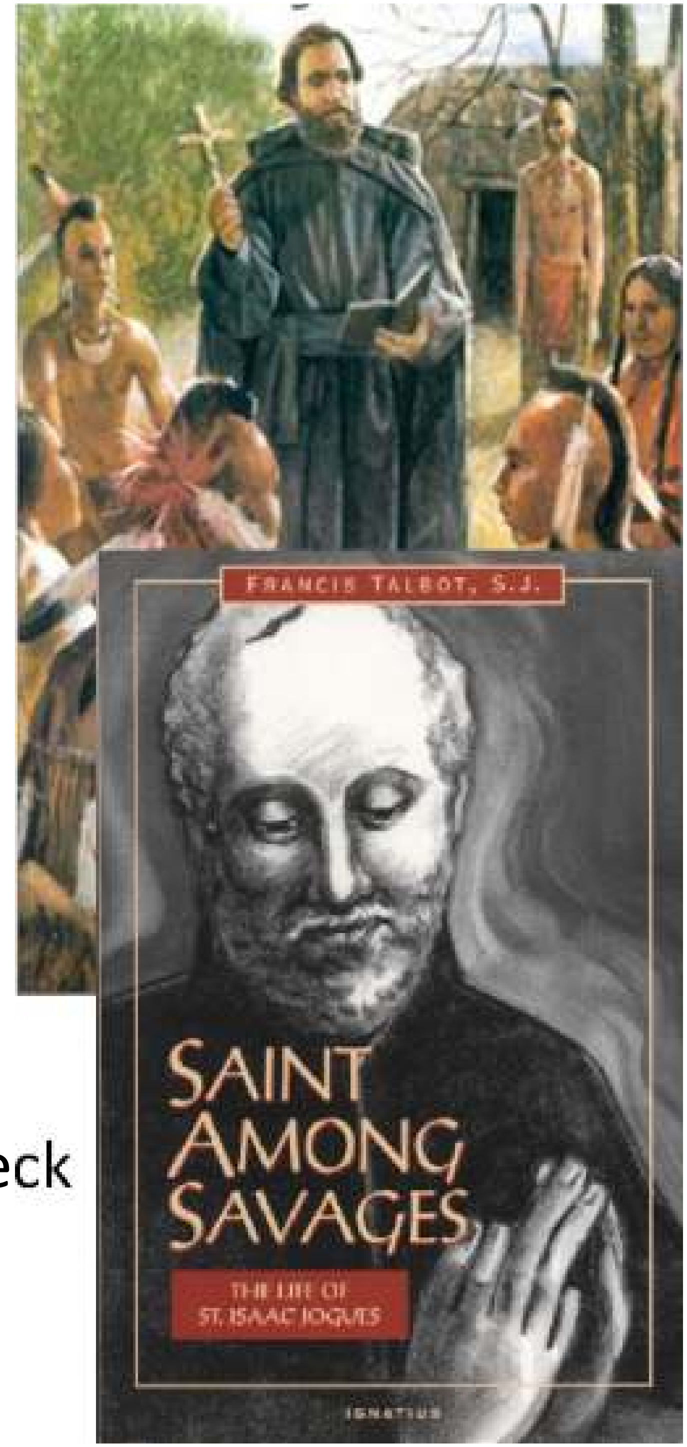
Add types to a dynamic language

Popular among academics

Scheme [Cartwright 91], Python [Aycock 00, Salib 04], Erlang [Nyström 03], Java [Lagorio 07, Ancona 07], PHP [Camphuijsen 09], Ruby [Furr 09], …

Not popular among practitioners

- Lack of guarantees: compiler warnings are advisory
- Realistic programs do not type-check
- Poor cost/benefit



Add Dynamic/Object/void* to a statically-typed language

Program is half-static, half-dynamic

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

"Incremental/gradual/hybrid typing"

Research challenge: behavior at the boundary

- Correctness [Ou 04, Flanagan 06; Siek 07, Herman 07; Findler 02, Gray 05]
- Blame assignment [Findler 01, Tobin-Hochstadt 06,08, Furr 09, Wadler 09]
- Efficiency [Herman 09, Siek 09,10]

Disadvantages of adding `Dynamic`

Reduced benefits:

- No type-checking guarantee
- Less encouragement to good design
- No documentation benefits (where `Dynamic` is used)

Increased costs:

- Reasoning burden
  - Identify boundary between typed & untyped
- Transformation burden
  - Represent the boundary to the type system
  - Later, undo work
- Boundary changes with time

Workarounds: Emulate static or dynamic typing

- Naming conventions
- Code analysis tools
- Partial execution
  - Don’t compile code with type errors
    - Comment out; modify build file
- Partial execution
  - Unexecuted casts
- Prototype in dynamic language, deliver in static
- IDE/editor tricks (Eclipse has several)
- … many more

Ductile provides a general mechanism

Outline

- Motivation and approach
- Evaluation
- Implementation
- Related work
- **Conclusion**

Why wasn’t this done before?

- Rigid attitudes about the “best” feedback
- Divide between static and dynamic research
- Aping of developer workarounds
- Choices made for the convenience of tools
- Difficult to design & implement

Contributions

- New approach unifies static and dynamic typing
  - View whole program through the lens of full static or full dynamic typing
  - Switch views seamlessly, on demand
- The programmer is in control
  - Separate feedback from action
- Implementation via detyping transformation
  - Case studies show correctness, utility

Try it! http://code.google.com/p/ductilej/