A Practical Type System and Language for Reference Immutability

Adrian Birka and Michael Ernst

MIT Computer Science & AI Lab

Goal: Reasoning about side effects

What code may modify a data structure?
Under what circumstances?

Currently, inadequate language support for:

- documenting immutability
- verifying immutability

Argument immutability

A library routine does not modify its argument

```
ElectionResults tabulate(Ballots votes) {
  ...
}
```

Argument immutability

A library routine does not modify its argument

```
ElectionResults tabulate(Ballots votes) {
  ...  // What does this code do?
}
```

Argument immutability

A library routine does not modify its argument

```
ElectionResults tabulate(readonly Ballots votes) {
  ...  // Does not tamper with the votes
}
```

Return value immutability

A client does not modify a return value

```
class Class {
  private Object[] signers;
  Object[] getSigners() {
    return signers;  // security hole in JDK 1.1.1
  }
}
```

Return value immutability

A client does not modify a return value

```
class Class {
  private Object[] signers;
  readonly Object[] getSigners() {
    return signers;
  }
}
```

Return value immutability

A client does not modify a return value

```
class Class {
  private Object[] signers;
  readonly Object[] getSigners() {
    return signers;
  }
}

class FileSystem {
  private Inode[] inodes;
  Inode[] getInodes() {
    ...  // Unrealistic to copy
  }
}
```

Return value immutability

A client does not modify a return value

```
class Class {
  private Object[] signers;
  readonly Object[] getSigners() {
    return signers;
  }
}

class FileSystem {
  private Inode[] inodes;
  readonly Inode[] getInodes() {
    return inodes;
  }
}
```

Object immutability

A value is never modified, via any reference

- Established if all references are read-only

```
readonly Date today = new Date();
```

```
Graph g1 = new Graph();
... construct cyclic graph g1 ...
// Suppose no aliases to g1 exist.
readonly Graph g = g1;
g1 = null;
```

Benefits of immutability constraints

Machine-checked documentation
Prevent/detect errors
Enable analyses and transformations

Immutability guarantees

Reference immutability

- A reference cannot be used to modify its referent
- Other references to the object may modify it
- Contrast: object immutability
  - Less expressive, requires aliasing information

Abstract state immutability

- All transitively reachable state (deep immutability)

Static type safety

- Can accommodate potentially unsafe casts via dynamic checking

Results

Type system achieving the above goals

Javari language implementation

- compatible with Java
- two new keywords: `readonly`, `mutable`

Experience with 160,000 lines of Javari code

Javari is practical and effective

- easy to use
- improved documentation
- revealed errors

Comparison to related work

C++: `const`, `mutable`, `const_cast`

- unsound: unchecked casts, many holes in type system
- not transitive: can modify contents of a `const` pointer
- no parameterization: leads to code duplication

JAC [Kniesel 2001]: `readonly`, `mutable`

- source-to-source translation into Java
- return type mutability = receiver mutability
- unsound: subtyping, arrays, casts, exceptions, ...

Type systems (e.g., ownership types)

- restrict programming language, less expressive
- typically object immutability, not reference immutability

Functional languages (e.g., OCaml): `mutable`

- no read-only types; must use interfaces and subtyping

Outline

Introduction

The Javari type system

Type casts

Experience

Conclusion

Language overview

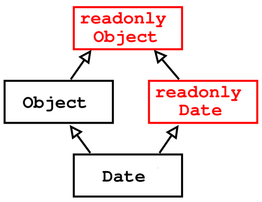
Two new keywords: `readonly` and `mutable`

`readonly` type modifier: immutable reference; cannot be used to modify its referent

`mutable` type modifier: excludes a field from abstract state

`(mutable)` type downcast: safe, dynamically checked type cast

`readonly` type modifier



For every reference type `T`,
 `readonly T` is a supertype of `T`

A read-only reference cannot be used to change the state of the object to which it refers.

`readonly` methods

- `readonly` applies to the receiver (`this`)
- `int size() readonly { ... }`

`readonly` is orthogonal to `final`

Type rules

```
class Link {
    int value;
    Link next;
}

readonly Link lk;
```

Fields of a read-only reference cannot be assigned

- `lk.value = 22;` // error

Fields of a read-only reference are themselves read-only references

- `lk.next` is read-only

A read-only reference cannot be copied to a non-read-only reference

- `Link lk2 = lk;` // error

`readonly` is orthogonal to `final`

- `lk = new Link();` // OK

mutable excludes a field from the abstract type

Read-only containers of non-read-only objects

```
class Link {
  mutable Object value;
  Link next;
}
```

mutable excludes a field from the abstract type

Cached results of read-only methods

```
class Link {
  int value;
  Link next;
  int length = -1;  // -1 means uninitialized
  int size() readonly {
    if (this.length == -1) {
      this.length = ...;   // error: size() is read-only
    }
    return this.length;
  }
}
```

mutable excludes a field from the abstract type

Cached results of read-only methods

```
class Link {
  int value;
  Link next;
  mutable int length = -1;  // -1 means uninitialized
  int size() readonly {
    if (this.length == -1) {
      this.length = ...;
    }
    return this.length;
  }
}
```

Other features

Immutable classes (e.g., `String`)

- all fields and methods are `readonly` (and `final`)
- every reference is read-only

Mutability parameterization (generics)

- reduces source code duplication
- not implemented by type erasure

Interoperability with Java

- forward and backward compatible

Type-based analyses

Reference immutability is a useful, flexible property.
How can we provide even stronger guarantees?

- a type-based analysis can build upon it
- the analysis requires *partial* aliasing information

Return *value* non-mutation can be proved given

- reference immutability for returned reference
- escape analysis: no non-read-only reference escapes

Similarly for

- parameter non-mutation
- thread non-interference
- object immutability

Outline

Type systems reject safe programs

```
Object o = new Date();
Date d = o;
...
```

```
readonly JFrame jf = ...;
// Formal arg of JDialog is a mutable JFrame.
JDialog jd = new JDialog(jf);
// Use dialog box jd only to get a filename,
// without modifying the JFrame.
...
```

Working around type system limitations

Unsafe:

- Run the program without any guarantee

Safe:

- Forbid any program rejected by the type system
  - reduces expressiveness
- Create a more complex type system
  - popular with theoreticians, but not with programmers
- Insert run-time checks
  - static guarantee for the type-safe subset of the language

Downcasts

```
Object o = new Date();
Date d = (Date) o;   // Java inserts a check
...
```

```
readonly JFrame jf = ...;
// Formal arg of JDialog is a mutable JFrame.
JDialog jd = new JDialog((mutable) jf);
// Use dialog box jd only to get a filename,
// without modifying the JFrame.
...   // Javari inserts checks
```

Dynamic checking for downcasts

A readonly boolean for each *reference* (not object)

Mutability downcasts set the boolean

```
... (mutable) lk ...
lk_ro = true;
```

Mutations (field updates) check the boolean

```
if lk_ro then throw exception;
lk.value = 22;
```

Field dereferences preserve the boolean

```
... lk.next ...
(lk.next)_ro = lk_ro;
```

Assignments propagate the boolean

```
lk = lk2;
lk_ro = lk2_ro;
```

Runtime overhead is small

<10% for source-to-source translation

Many optimizations are possible

- no checks for
  - immutable classes
  - classes reached only via readonly references
  - classes not reached via downcasted references
- statically discharge checks
  - hoist checks out of loops
  - no extra booleans for intermediate results
- modify the JVM

Outline

Experience

160,000 lines of Javari code

Writing new code: easy

Annotating existing code: not hard

- Most effort was spent in reverse engineering

Frequency of new keywords:

- `readonly` modifiers: 1 per 10 lines
  - $\frac{1}{3}$ of possible locations
- generics (mutability parameters): 1 per 100 lines
- `mutable` fields: 1 per 1000 lines
- `(mutable)` downcasts: 1 per 1000 lines

Benefits of reference immutability

Improve documentation

- Make it consistent with the implementation

Enhance understanding of programs

Fix errors

- 3 in documentation
- 20 in implementation
  - representation exposure
  - unintentionally modify arguments
- ... plus 8 instances of bad style

Eliminate unnecessary copying

Lessons from using Javari

Importance of downcasts (and explicit syntax for them)
for application invariants and legacy libraries
Importance of mutability parameterization (generics)
Few uses of object immutability
Transitive immutability is a strong guarantee

Outline

Introduction

The Javari type system

Type casts

Experience

Conclusion

Contributions

Practical and effective combination of language features

- reference immutability
- transitive immutability

Type-based analyses to provide stronger guarantees

Combination of compile- and run-time type checking

- guarantee of safety

Type system for a full OO language (Java)

- non-trivial, but illuminating
- forward and backward compatible

Implementation and evaluation

- evidence of practicality
- improve documentation
- find errors, expose other problems

Advertisement

The Java platform: Tiger and beyond
Thursday, October 28
13:30-17:00
Meeting Room 13

Contributions

Practical and effective combination of language features

- reference immutability
- transitive immutability

Type-based analyses to provide stronger guarantees

Combination of compile- and run-time type checking

- guarantee of safety

Type system for a full OO language (Java)

- non-trivial, but illuminating
- forward and backward compatible

Implementation and evaluation

- evidence of practicality
- improve documentation
- find errors, expose other problems