Modular Reasoning About Object Relations

Micha Greutmann

Marco Eilers

Peter Müller

ETH zürich

A Scary Sight

```
> s = new Set()
> s.add(e1)
> s.add(e2)
> print(s.contains(e2))
False
```

$\exists e \in s.\, e.equals(e2)$

No concurrency!

Correct Set implementation!

Incorrect equality implementation: **Symmetry violated**

Binary Object Relations

Equality:	Strict partial order:	Partial order:	Containment:
o1 == o2	o1 < o2	o1 <= o2	o1 contains o2

Our setting (language-independent):

- Relations implemented via pure boolean methods
- Subtyping via subclasses, interfaces, traits, ...
- Implementation chosen based on dynamic type of first argument

$$o_1 == o_2 = \begin{cases} o_1.equals_{T_1}(o_2) & \text{if } type(o_1) = T_1 \\ o_1.equals_{T_2}(o_2) & \text{if } type(o_1) = T_2 \\ \dots & \\ o_1.equals_{T_n}(o_2) & \text{if } type(o_1) = T_n \end{cases}$$

Algebraic Properties

Reflexivity of $=$:	$\forall o_1.$	$o_1 = o_1$
Symmetry of $=$:	$\forall o_1, o_2.$	$o_1 = o_2 \Leftrightarrow o_2 = o_1$
Transitivity of $=$:	$\forall o_1, o_2, o_3.$	$o_1 = o_2 \wedge o_2 = o_3 \Rightarrow o_1 = o_3$
Irreflexivity of $<$:	$\forall o_1.$	$\neg o_1 < o_1$
Asymmetry of $<$:	$\forall o_1, o_2.$	$o_1 < o_2 \Leftrightarrow \neg o_2 < o_1$
Transitivity of $<$:	$\forall o_1, o_2, o_3.$	$o_1 < o_2 \wedge o_2 < o_3 \Rightarrow o_1 < o_3$
...		

These (except (ir)reflexivity) are properties ***between*** implementations in potentially ***different*** types:

Symmetry of $=$:	$\forall o_1 : T_1, o_2 : T_2.$	$o_1.equals_{T_1}(o_2) \Leftrightarrow o_2.equals_{T_2}(o_1)$
Transitivity of $=$:	$\forall o_1 : T_1, o_2 : T_2, o_3 : T_3.$	$o_1.equals_{T_1}(o_2) \wedge o_2.equals_{T_2}(o_3) \Rightarrow o_1.equals_{T_1}(o_3)$

Binary Object Relations: Example

```
class Point extends Object {
  private int x;
  private int y;
  override boolean equals(Object o)
  {
    if (o instanceof Point) {
      Point p = ((Point) o);
      return this.x ==  p.x &&
             this.y ==  p.y;
    }
    return false;
  }
}
```

```
class ColorPoint extends Point {
  private Color c;
  override boolean equals(Object o)
  {
    if (o instanceof ColorPoint) {
      ColorPoint p = ((ColorPoint) o);
      return this.x == p.x &&
             this.y == p.y &&
             this.c.equals(p.c);
    }
    return false;
  }
}
```

```
> p = new Point(1, 2)
> cp = new ColorPoint(1, 2, RED)
```

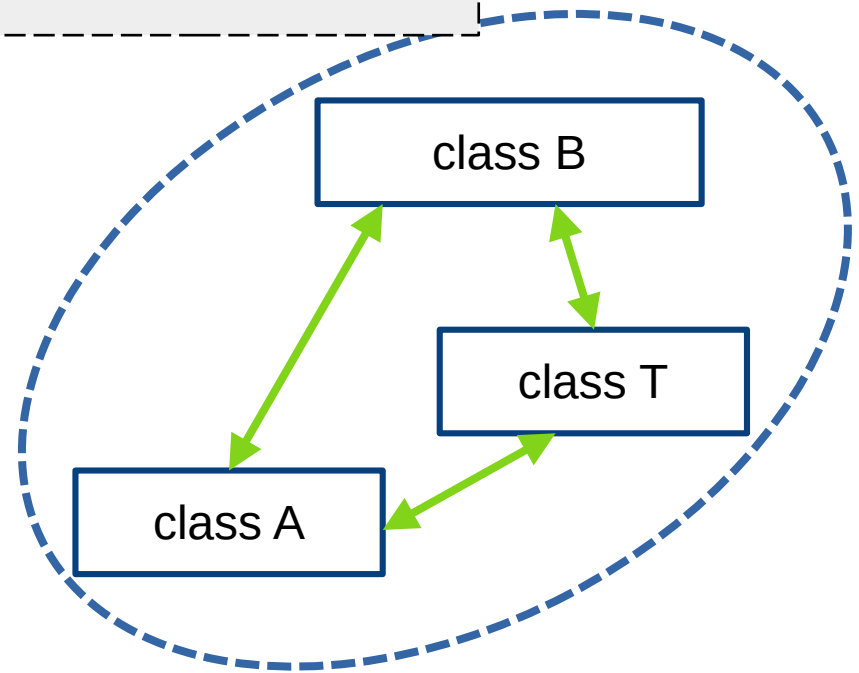
```
> print(p == cp)
True
> print(cp == p)
False
```

How to make sure all relations are correct in a program?

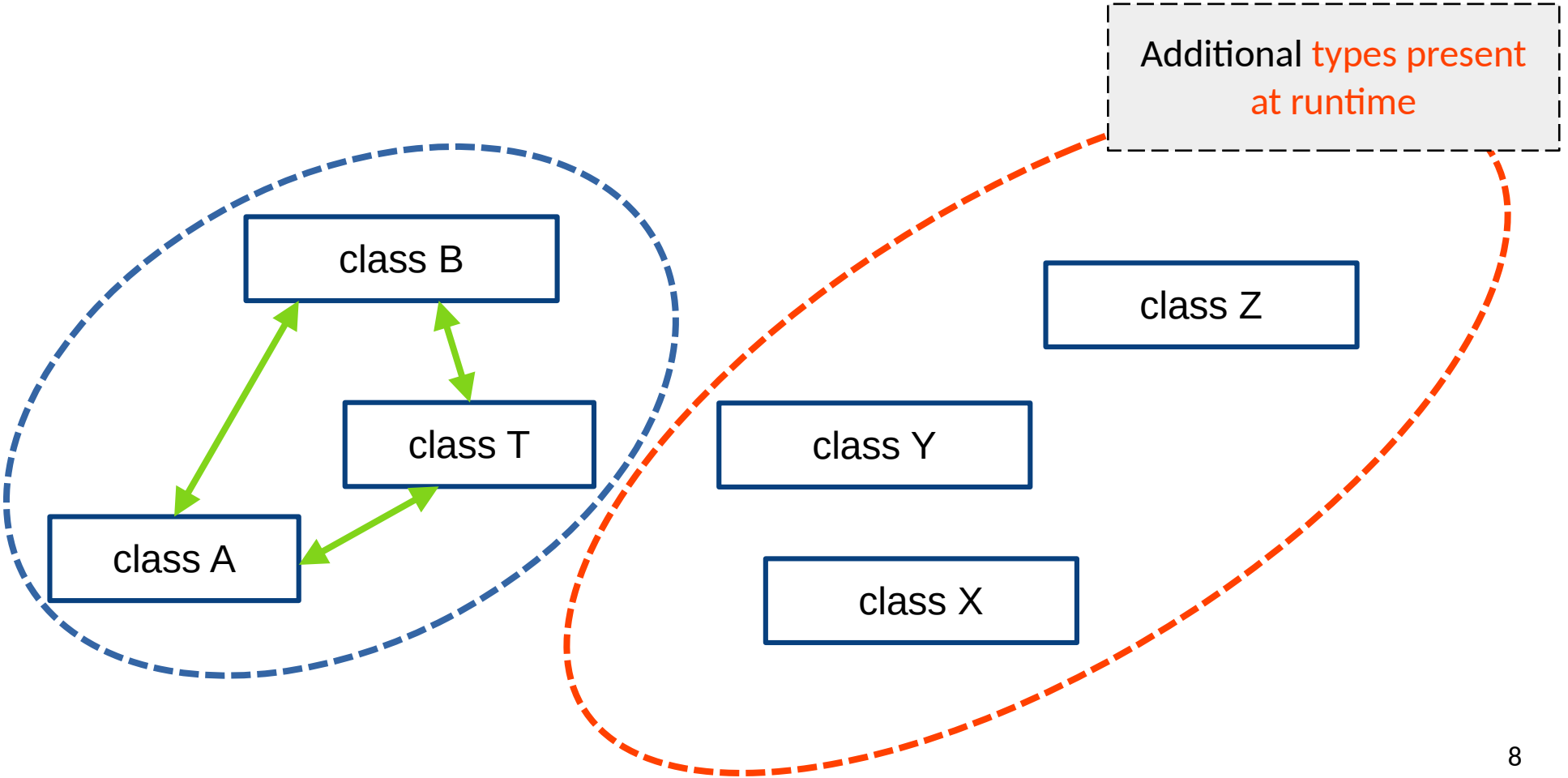
Question for developers:
How to make sure my type is correct?

Problem: Naive solution is not compositional!

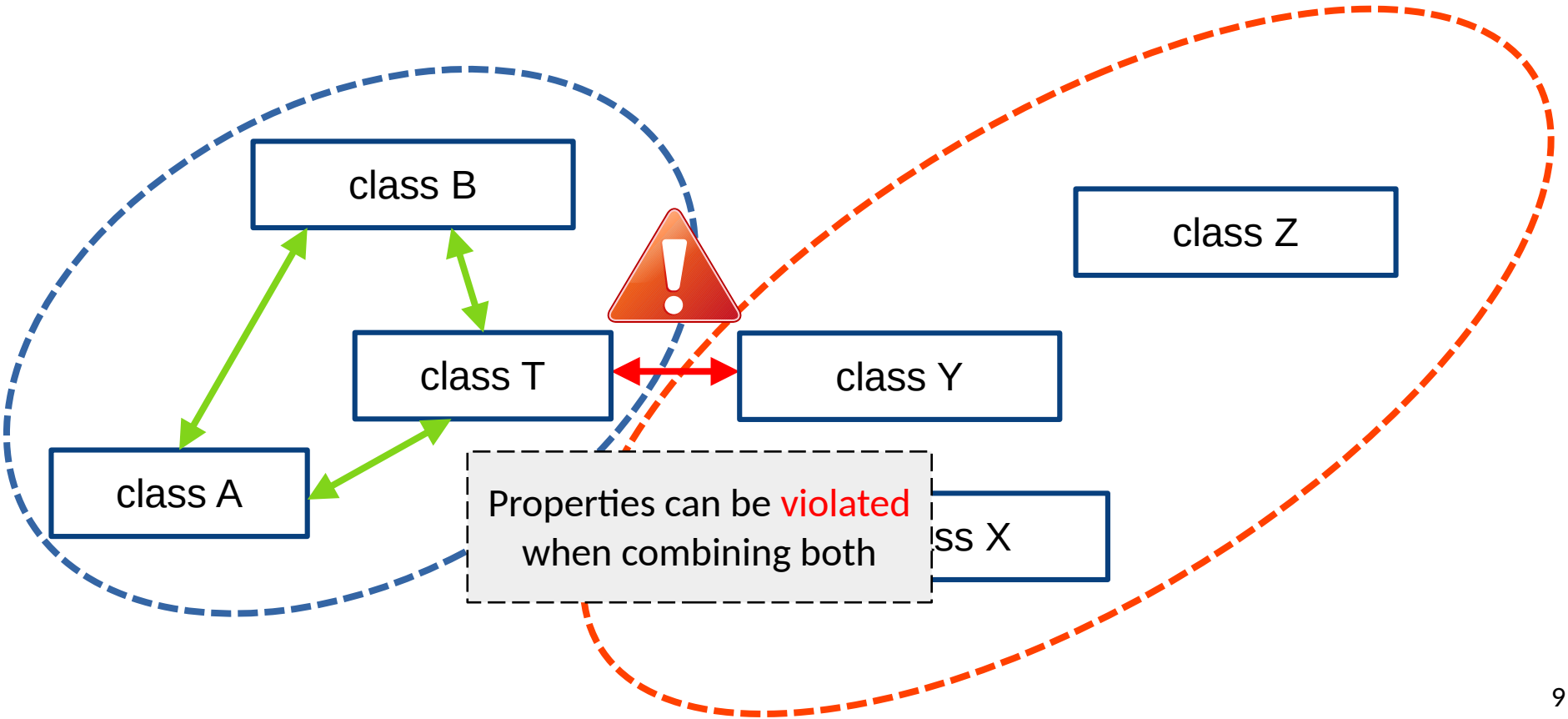
Properties hold for types known to developer of T



Problem: Naive solution is not compositional!



Problem: Naive solution is not compositional!



Goal:
Check algebraic properties **modularly** per type:

If every type is correct, the whole program is correct

Approach

1. Express properties in type-centric normal form for type T
2. Determine finite set of other known types relevant for T
3. Prove property between T and these known types
4. Handle subtyping

Step 1: Normal Form

We identify a normal form for algebraic properties:

$$\forall o_1, o_2, \ldots, o_n . R(o_1, o_2) \Rightarrow P$$

May refer to R, other relations **only** with **receiver** (first argument) o_1 or o_2

Symmetry:

$$\forall o_1, o_2 . o_1 = o_2 \Leftrightarrow o_2 = o_1$$

$$\forall o_1, o_2 . o_1 = o_2 \Rightarrow o_2 = o_1$$

Transitivity:

$$\forall o_1, o_2, o_3 . o_1 = o_2 \Rightarrow (o_2 = o_3 \Rightarrow o_1 = o_3)$$

$$\forall o_1, o_2, o_3 . o_1 = o_2 \Rightarrow (o_2 = o_3 \Rightarrow o_3 = o_1)$$

Step 1: Normal Form

Normal form supports ***all*** standard properties except (ir)reflexivity

R	Name	n	Property
$=$	Symmetry	2	$o_1 = o_2 \Rightarrow o_2 = o_1$
$=$	Transitivity	3	$o_1 = o_2 \Rightarrow (o_2 = o_3 \Rightarrow o_1 = o_3)$
$=$	Hash-Consistency	2	$o_1 = o_2 \Rightarrow o_1.hash() = o_2.hash()$
$=$	Contains-Consistency	3	$o_1 = o_2 \Rightarrow (o_1\ contains\ o_3 \Rightarrow o_2\ contains\ o_3)$
$<$	Asymmetry	2	$o_1 < o_2 \Rightarrow \neg(o_2 < o_1)$
$<$	Transitivity	3	$o_1 < o_2 \Rightarrow (o_2 < o_3 \Rightarrow o_1 < o_3)$
$<$	Converse	2	$o_1 < o_2 \Rightarrow o_2 > o_1$
$contains$	Equals-Consistency	3	$o_1\ contains\ o_2 \Rightarrow (o_2 = o_3 \Rightarrow o_1\ contains\ o_3)$

Step 1: Normal Form

Idea: Prove the property separately for each possible type T of o_1.

$$\forall o_1, o_2 \;.\; o_1 = o_2 \Rightarrow o_2 = o_1$$

becomes

$$\forall o_1 : T, o_2 \;.\; o_1.equals_T(o_2) \Rightarrow o_2 = o_1$$

where $equals_T$ is the (known!) implementation of equals in T!

Question: Which implementation can $o_2 = o_1$ bind to if $o_1.equals_T(o_2)$?

Step 2: Constraining Sets

Which implementation can $o_2 = o_1$ bind to if $o_1 : Point$ and $o_1.equals_{Point}(o_2)$?

```
override boolean equals(Object o2)
{
  if (o2 instanceof Point) {
    Point p = ((Point) o2);
    return this.x ==  p.x &&
           this.y ==  p.y;
  }
  return false;
}
```

Observation:
Reasonable implementations of relations return true only if o_2 has specific types!

Constraining set C_T^R: A set of types s.t. $o_1.R_T(o_2)$ returns true only if o_2 has some type in the set (or a subtype).

$C_{Point}^{equals} = \{ Point \}$

$C_{ColorPoint}^{equals} = \{ ColorPoint \}$

Step 3: Modular Correctness Check

Check

$$\forall o_1 : T, o_2, \ldots, o_n \;.\; o_1.R_T(o_2) \Rightarrow P$$

separately for each type $T' \in C_T^R$ of o_2!

$$\forall o_1 : Point, o_2 \;.\; o_1.equals_{Point}(o_2) \Rightarrow o_2 = o_1 \qquad C_{Point}^{equals} = \{ Point \}$$

$$T' = Point$$

$$\forall o_1 : Point, o_2 \;.\; o_1.equals_{Point}(o_2) \Rightarrow o_2.equals_{Point}(o_1)$$

Step 4: Behavioral subtyping

Constraining set C_T^R: A set of types s.t. $R_T(o1, o2)$ returns true only if o2 has some type in the set ***(or a subtype)***.

What if o_2 is an instance of a **subtype** of Point?

Behavioral subtyping:
- Equip every type with a specification
- Every subtype must satisfy the specification of its supertype
- Check property relying **only** on specification of $o_2.equals_{Point}(o_1)$

Step 4: Behavioral subtyping

$C_{\neg}^{R}$

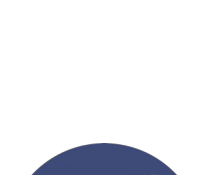
```
class Point {
  ...
  override boolean equals(Object o2)
    // ensures result <==>
    //   o2 instanceof Point &&
    //   o2.x == this.x &&
    //   o2.y == this.y;
  {
    ...
  }
}
```

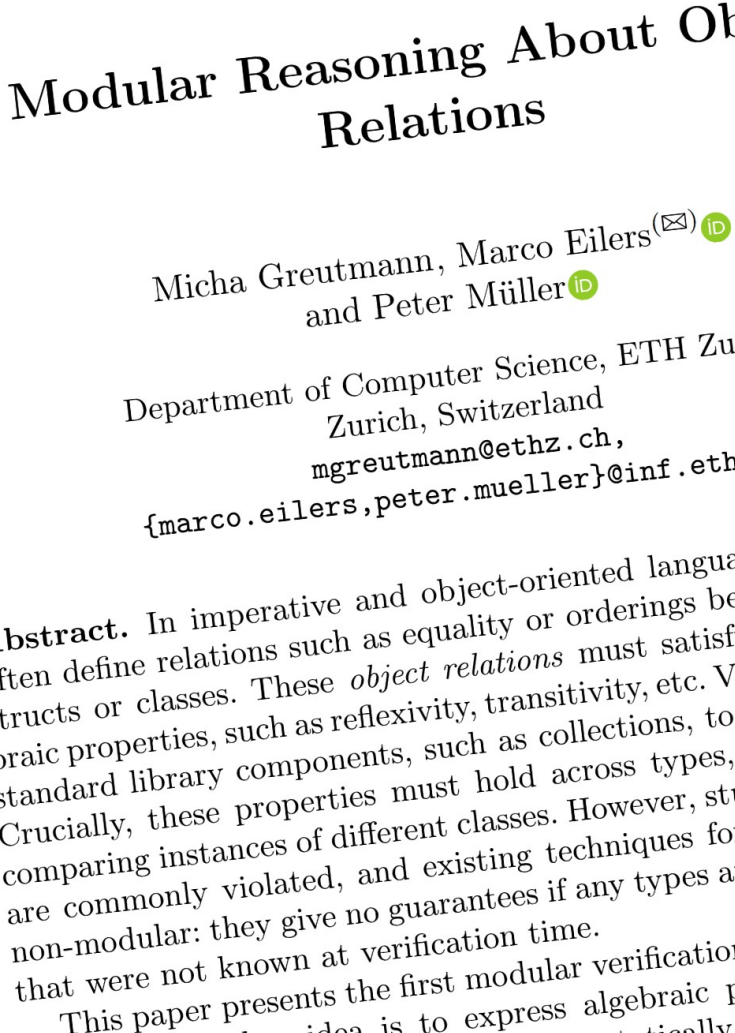
```
class ColorPoint extends Point {
  ...
  override boolean equals(Object o2)
    // ensures result <==>
    //   o2 instanceof Point &&
    //   o2.x == this.x &&
    //   o2.y == this.y;
  {
    ...
  }
}
```

$$\forall\, o_1 : Point\,, o_2 \;.\; o_1.equals_{Point}(o_2) \Rightarrow o_2.equals_{Point}(o_1)$$

Also in the Paper

- Proof rules as source-to-source encoding:
  - Usable with many verification tools
  - Logic-agnostic, tool-agnostic, language-agnostic
  - Requires only standard features (pure functions)
- Different levels of automation
- Soundness proof
  - Modular correctness of all types implies global correctness of all relations
- Implementation for equality in the Python verifier Nagini





Modular Reasoning About Ob
Relations

Micha Greutmann, Marco Eilers(✉)
and Peter Müller

Department of Computer Science, ETH Zu
Zurich, Switzerland
mgreutmann@ethz.ch,
{marco.eilers,peter.mueller}@inf.eth

Abstract. In imperative and object-oriented langua
often define relations such as equality or orderings be
structs or classes. These *object relations* must satisf
braic properties, such as reflexivity, transitivity, etc. V
standard library components, such as collections, to
Crucially, these properties must hold across types,
comparing instances of different classes. However, stu
are commonly violated, and existing techniques fo
non-modular: they give no guarantees if any types a
that were not known at verification time.
This paper presents the first modular verificatio
idea is to express algebraic

Evaluation

Benchmark	R	LOC	Ann.	Enc.	Lang.	Time
Core 1	$=$	177	73	Auto	Python	4.42
Core 1F	$=$	24	13	Auto	Python	2.31
Core 2	$=$	233	112	Auto	Python	5.36
Core 2F	$=$	215	105	Auto	Python	5.09
Core 3	$=$	245	123	Auto	Python	5.59
Core 3F	$=$	233	120	Auto	Python	5.24
Birds	$=$	81	36	Auto	Python	5.59
Fractions	$=$	132	68	Auto	Python	3.56
Fractions	$=$	99	40	Manual	Dafny	1.27
Hash Consistency	$=$	109	58	Auto	Python	3.64
LessThan	$<$	148	56	Manual	Python	2.82
LessThan	$<$	42	7	Manual	Dafny	1.18
Map	$=$	51	30	Auto	Python	4.92
Points	$=$	91	45	Auto	Python	3.17
Point Patterns	$=$	178	95	Auto	Python	4.02
Stateless Tuples	$=$	17	10	Auto	Python	3.33
Strings	$=$	81	43	Auto	Python	3.17

Evaluation

Benchmark	R	LOC	Ann.	Enc.	Lang.	Time
Core 1	$=$	177	73	Auto	Python	4.42
Core 1F	$=$	24	13	Auto	Python	2.31
Core 2	$=$	233	112	Auto	Python	5.36
Core 2F	$=$	215	105	Auto	Python	5.09
Core 3	$=$	245	123	Auto	Python	5.59
Core 3F	$=$	233	120	Auto	Python	5.24
Birds	$=$	81	36	Auto	Python	5.59
Fractions	$=$	132	68	Auto	Python	3.56
Fractions	$=$	99	40	Manual	Dafny	1.27
Hash Consistency	$=$	109	58	Auto	Python	3.64
LessThan	$<$	148	56	Manual	Python	2.82
LessThan	$<$	42	7	Manual	Dafny	1.18
Map	$=$	51	30	Auto	Python	4.92
Points	$=$	91	45	Auto	Python	3.17
Point Patterns	$=$	178	95	Auto	Python	4.02
Stateless Tuples	$=$	17	10	Auto	Python	3.33
Strings	$=$	81	43	Auto	Python	3.17

Different relations

Evaluation

Benchmark	R	LOC	Ann.	Enc.	Lang.	Time
Core 1	=	177	73	Auto	Python	4.42
Core 1F	=	24	13	Auto	Python	2.31
Core 2	=	233	112	Auto	Python	5.36
Core 2F	=	215	105	Auto	Python	5.09
Core 3	=	245	123	Auto	Python	5.59
Core 3F	=	233	120	Auto	Python	5.24
Birds	=	81	36	Auto	Python	5.59
Fractions	=	132	68	Auto	Python	3.56
Fractions	=	99	40	Manual	Dafny	1.27
Hash Consistency	=	109	58	Auto	Python	3.64
LessThan	$<$	148	56	Manual	Python	2.82
LessThan	$<$	42	7	Manual	Dafny	1.18
Map	=	51	30	Auto	Python	4.92
Points	=	91	45	Auto	Python	3.17
Point Patterns	=	178	95	Auto	Python	4.02
Stateless Tuples	=	17	10	Auto	Python	3.33
Strings	=	81	43	Auto	Python	3.17

Different languages, different program logics

Conclusion

- First modular verification technique for algebraic properties of binary object relations
  - Proof obligations for implementer of each type
  - Supports all standard properties
- Expressed as source-to-source encoding
  - Applicable to many programming languages, program logics, verification tools
- Proved sound
- Implemented in Nagini for Python