**RC24196 (W0703-111) March 23, 2007**
**Computer Science**

# IBM Research Report

## Declarative Object Identity Using Relation Types

**Mandana Vaziri, Frank Tip, Stephen Fink, Julian Dolby**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

# Declarative Object Identity Using Relation Types[⋆]

Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby

IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
{mvaziri,ftip,sjfink,dolby}@us.ibm.com

**Abstract.** Object-oriented languages define the identity of an object to be an address-based object identifier. The programmer may customize the notion of object identity by overriding the `equals()` and `hashCode()` methods following a specified contract. This customization often introduces latent errors, since the contract is unenforced, and at times impossible to satisfy. Notably, `equals()` may refer to mutable state, which allows object identity to change during execution, breaking standard library invariants.

We propose a programming model based on a relational view of the heap which defines identity declaratively, obviating the need for `equals()` and `hashCode()` methods. Each element in the heap (called a *tuple*) belongs to a *relation type* and relates an immutable *identity* to mutable state. The model entails a stricter contract: identity never changes during an execution. Objects, values, and singletons arise as special cases of tuples. We formalize the model as an adaptation of Featherweight Java, and implement it by extending Java with relation types. Experiments on a set of Java programs show that the majority of classes that override `equals()` can be refactored into relation types, and that most of the remainder are buggy or fragile.

## 1 Introduction

> IX: That every individual substance expresses the whole universe in its own manner and that in its full concept is included all its experiences together with all the attendant circumstances and the whole sequence of exterior events.
> *G. W. Leibniz, Discourse on Metaphysics (1686)*

Object-oriented languages such as Java and C# support an address-based notion of identity for objects or reference types. By default, the language considers no two distinct object instances *equal*; Java's `java.lang.Object.equals()` tests object identity by comparing addresses. Since programmers often intend an alternative notion of equality based on forms of structural equivalence, one may override the `equals()` method, leaving the definition of identity to the programmer's imagination.

[⋆] This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

In order for standard library classes such as collections to function properly, Java mandates that an equals() method satisfy an informal contract. First, it must define an equivalence relation, meaning that equals() should encode a reflexive, symmetric, and transitive relation. Second, the contract states that "it must be consistent", i.e., two objects that are equal at some point in time must remain equal, unless the state of one or both changes. Third, no object must be equal to null. Furthermore, when a programmer overrides equals(), he must also override hashCode() to ensure that equal objects have identical hash-codes.

Programmer customization of identity semantics causes problems for several reasons. First, creating an equivalence relation is often non-trivial and, in some cases, impossible [10] (for details, see Section 2). Second, the language has no mechanism to enforce the contract either statically or dynamically, leaving plenty of rope by which programmers routinely hang themselves. We found buggy or fragile equals() methods in nearly every Java application that we examined (see Section 2). Third, programmer identity tests often comprise repetitive and error-prone boiler-plate code, which must be updated manually as the code evolves over time. Even more boiler-plate code arises in patterns such as caching via hash-consing [16].

To alleviate these problems, we propose a programming model in which object identity is specified declaratively, without tedious and error-prone equals() and hashCode() methods. The model features a new language construct called a *relation type*. Each relation type declares zero or more fields, and designates a (possibly empty) subset of these as immutable *key fields*. An instance of a relation type is called a *tuple*, and the identity of a tuple is fully determined by its type and the identities of the instances referred to by its key fields. Consequently, our programming model provides a relational view of the heap, as a map from identities to their associated mutable state. In other words, two tuples $a$ and $b$ are equal if and only if: (i) $a$ and $b$ are of the same type and, (ii) corresponding key fields in $a$ and $b$ are equal. Operationally, one can think of tuples with the same identity as pointing to the same heap location, and our model permits efficient implementations (e.g., the use of space-efficient shared representations in combination with pointer-equality for fast comparisons).

Our model enforces a stricter contract than Java's since object identity *never* changes, and tuples of different types *must* have different identities. Several existing concepts arise as special cases of relation types: (i) a *class* of objects is one with an address as its only key field, (ii) a *value-type* [5, 13] is one with *only* key fields, and (iii) an instance of the SINGLETON design pattern [15] is one with *no* key fields.

We formalize our programming model as an adaptation of Featherweight Java, and prove that hash-consing identities preserves semantics. We implemented relation types in a small extension of Java called RJ, and created an RJ-to-Java compiler. We examined the classes that define equals() methods in several Java applications and refactored these classes to use relation types instead. We found that the majority of classes that define equals() can be refactored with minimal effort into relation types, and that most of the remainder are buggy or fragile.

To summarize, this paper makes the following contributions:

1. We present a programming model in which object identity is defined declaratively using a new language construct called relation types. By construction, relation types satisfy a strict contract that prevents several categories of bugs, and admits efficient implementations. Objects, value types, and singletons arise as special cases of the model.
2. We formalize the model using an adaptation of Featherweight Java, and prove that hash-consing is a safe optimization in this model.
3. We extended Java with relation types (RJ), and created an RJ-to-Java compiler. Experiments indicate that the majority of classes that define `equals()` in several Java applications can be refactored into relation types, and that most of the remainder are buggy or fragile.

## 2 Overview of RJ

This section examines Java's equality contract and illustrates several motivating problems. We then informally present our new approach based on relation types.

### 2.1 Java's Equality Contract

The contract for the `equals()` method in `java.lang.Object` [1] states that:

> The `equals` method implements an equivalence relation on non-null object references:
>
> (1) It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return true.
> (2) It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
> (3) It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
> (4) It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified.
> (5) For any non-null reference value `x`, `x.equals(null)` should return `false`.

Furthermore, whenever one overrides `equals()`, one must also override `hashCode()`, to ensure that equal objects have identical hash-codes.

This contract has several problems. First, it is impossible to extend an instantiatable class with a new field, and have the subclass be comparable to its superclass, while preserving the equivalence relation. Consider the example shown in Figure 1 (taken from [10]). Here, the `equals()` method of `ColorPoint` must be written as such to preserve symmetry. However, this violates transitivity as indicated in [10]. If one defines three points as follows:

```
class Point {
    int x;
    int y;
    public boolean equals(Object o){
        if (!(o instanceof Point))
            return false;
        return ((Point)o).x == x
        && ((Point)o).y == y;
    }
}
```

```
class ColorPoint extends Point{
    Color color;
    public boolean equals(Object o){
        if (!(o instanceof Point))
            return false;
        if (!(o instanceof ColorPoint))
            return o.equals(this);
        return super.equals(o) &&
        ((ColorPoint)o).color == color;
    }
}
```

**Fig. 1.** A class `Point` and its subclass `ColorPoint`.

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1,2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

then `p1.equals(p2)` is true and so is `p2.equals(p3)`, but `p1.equals(p3)` is false since `color` is taken into account.

A second problem with the contract is that the consistency (non-)requirement allows the identity relation defined by the `equals()` method to change over time: `equals()` may refer to mutable state. If an object's identity relation changes while the object resides in a collection, the collection's operations (e.g. `add()` and `remove()`) will not function as intended.

Most importantly, neither the compiler nor the runtime system enforces the contract in any way. If the programmer mistakenly violates the contract, the problem can easily manifest as symptoms arbitrarily far from the bug source. A correct implementation involves nontrivial error-prone boilerplate code, and mistakes easily and commonly arise, as we shall see in Section 2.2.

Java's contract (but not C#'s) is also under-specified because it permits `equals()` and `hashCode()` to throw run-time exceptions that could be avoided.

### 2.2 Examples

We carefully examined several applications, and found many problems in implementations of `equals()` and `hashCode()` methods, such as:

(a) **Dependence on mutable state.** Figure 2(a) shows a fragile code fragment from `org.hsqldb.GroupedResult.ResultGroup` in *hsqldb*, where `equals()` and `hashCode()` refer to a mutable field `row` of type `Object[]`, which is updated elsewhere. If the program modifies a `row` while a `ResultGroup` is stored in a collection, then subsequent attempts to retrieve elements from that collection may fail or produce inconsistent results[1]. While the `equals()` contract does not prohibit `equals()` and `hashCode()` from referring to mutable state, it "handles" these cases by declaring that "all bets are off" when the identity relation changes. The programmer must carefully maintain the non-local invariant that mutations do not overlap with relevant collection lifetimes, often resulting in buggy or hard-to-maintain code.

[1] Note that, if an ordered collection such as a `TreeSet` is used, then the modification of an element after storing it into the collection may affect retrieval operations involving other elements.

```
class ResultGroup {
  Object[] row;
  int hashCode;
  private ResultGroup(Object[] row) {
    this.row = row;
    hashCode = 0;
    for (int i = groupBegin;
        i < groupEnd; i++) {
      if (row[i] != null) {
        hashCode += row[i].hashCode();
      }
    }
  }
  public int hashCode() { return hashCode; }
  public boolean equals(Object obj) {
    if (obj == this) { return true; }
    if (obj == null ||
      !(obj instanceof ResultGroup)) {
      return false;
    }
    ResultGroup group = (ResultGroup)obj;
    for (int i = groupBegin;
          i < groupEnd; i++) {
      if (!equals(row[i], group.row[i])) {
        return false;
      }
    }
    return true;
  }
  private boolean equals(Object o1,
        Object o2) {
    return (o1 == null) ? o2 == null
        : o1.equals(o2);
  }
}
```

**(a)** Program fragment taken from *hsqldb*

```
public class NumberAxis extends ValueAxis ... {
  private boolean autoRangeIncludesZero;
  public boolean equals(Object obj) {
    if (obj == this) { return true; }
    if (!(obj instanceof NumberAxis)) {
      return false;
    }
    if (!super.equals(obj)) { return false; }
    NumberAxis that = (NumberAxis) obj;
    if (this.autoRangeIncludesZero !=
        that.autoRangeIncludesZero) {
      return false; }
    ...
  }
public class CyclicNumberAxis
        extends NumberAxis {
  protected double period;
  public boolean equals(Object obj) {
    if (obj == this) { return true; }
    if (!(obj instanceof CyclicNumberAxis)) {
      return false;
    }
    ...
    ...
    if (!super.equals(obj)) { return false; }
    CyclicNumberAxis that =
      (CyclicNumberAxis) obj;
    if (this.period != that.period) {
      return false;
    }
    ...
  }
```

**(b)** Program fragment taken from *jfreechart*

```
public class LocalVariableGen implements ... {
  public int hashCode() {
    //If the user changes the name or type,
    //problems with the targeter hashmap
    //will occur
    int hc = index ^ name.hashCode()
      ^ type.hashCode();
    return hc;
  }
  public boolean equals( Object o ) {
    if (!(o instanceof LocalVariableGen)) {
      return false;
    }
    LocalVariableGen l = (LocalVariableGen) o;
    return (l.index == index)
        && (l.start == start)
        && (l.end == end);
  }
}
```

**(c)** Program fragment taken from *bcll*

```
public class MethodNameDeclaration
  public boolean equals(Object o) {
    MethodNameDeclaration other =
      (MethodNameDeclaration) o;
    if (!other.node.getImage().
      equals(node.getImage())) {
      return false;
    }
    ...
  }
```

**(d)** Program fragment taken from *pmd*

```
public class emit {
  protected static void emit_action_code( ... ){
    ...
    if (prod.action() != null &&
      prod.action().code_string() != null &&
      !prod.action().equals(""))
      out.println(prod.action().code_string());
    ...
  }
}
```

**(e)** Program fragment taken from *javacup*

**Fig. 2.** Problems with the equality contract encountered in practice.

**(b) Asymmetry.** Figure 2(b) shows excerpts from two classes from *jfreechart*, one a superclass of the other. These `equals()` implementations are asymmetric: it is easy to construct a `NumberAxis a` and a `CyclicNumberAxis b` such that `a.equals(b)` but `!b.equals(a)`. This violates the contract and may produce inconsistent results if a heterogeneous collection contains both types of objects.

**(c) Contract for `equals()/hashCode()`.** In Figure 2(c), from *bcel*, `equals()` and `hashCode()` refer to different subsets of the state, so two equal objects may have different hashcodes. The developers apparently knew of this problem as is evident from the comment "If the user changes the name or type, problems with the targeter hashmap will occur".

**(d) Exceptions and `null` values.** Figure 2(d) shows an `equals()` method from *pmd*, which has two immediate problems. First, if the parameter `o` is `null`, the method throws a `NullPointerException` rather than return *false* as per the contract. Second, the code will throw a `ClassCastException` if this object is ever used in a collection that compares it to an object of an incompatible type.

**(e) Inadvertent test of incomparable types.** Figure 2(e) shows a buggy code fragment taken from method `java_cup.emit.emit_action_code()` in *javacup*. Here, the last part of the condition, `!prod.action().equals("")`, compares an object of type `java_cup.action_part` with an object of type `String`. Such objects are never equal to each other, hence the condition trivially succeeds. This bug causes spurious blank lines in the parser that is generated by *javacup*. We confirmed with the developers [24] that they intended to write `!prod.action().code_string().equals("")`. More generally, the problem is that objects of arbitrary types are allowed to be equal to each other under the current contract, and as a result, it is not possible to check for inconsistent comparisons at compile-time.

### 2.3 Revised Equality Contract

Revised Equality Contract for $==_R$ identity relation, on non-null references:

(1') $==_R$ is *reflexive*: For any non-null reference value `x`, `x` $==_R$ `x` must return true.
(2') $==_R$ is *symmetric*: For any non-null reference values `x` and `y`, `x` $==_R$ `y` returns true if and only if `y` $==_R$ `x` returns true.
(3') $==_R$ is *transitive*: For any non-null reference values `x`, `y`, and `z`, if `x` $==_R$ `y` returns true and `y` $==_R$ `z` returns true, then `x` $==_R$ `z` must return true.
(4') For any non-null reference values `x` and `y`, multiple tests `x` $==_R$ `y` consistently return true, or consistently return false throughout the execution.
(5') For any non-null reference value `x`, `x` $==_R$ `null` must return false.

We propose a model that forces programmers to define object identity declaratively. In this model, programmers must explicitly indicate the fields in a type that comprise its identity, automatically inducing an equivalence relation $==_R$. Our programming model enforces a new equality contract as defined above.

Our model's revised contract differs from Java's as follows:

- It is *enforced*: The language implementation generates $==_R$ automatically and forbids the programmer from manipulating this relation explicitly.
- It is *more strict* than the original contract in item (4'); object identity cannot change throughout the execution.
- The problems with defining an equivalence relation in the presence of subclassing are resolved by making relation types and their subtypes incomparable, meaning that `x` $==_R$ `y` will always evaluate to false if `x` and `y` are not of exactly the same type.

### 2.4 Relation Types

Our programming model introduces a new notion of class called a *relation type*. We informally present the notion here; Section 3 defines the semantics formally.

A relation type resembles a class in Java, except a programmer may not override the `equals()` and `hashCode()` methods. Instead, the programmer must designate a (possibly empty) subset of instance fields as *key fields*, using the keyword `key`. Key fields are implicitly `final` and `private`. We call an instance of a relation type a *tuple*, and its identity is fully determined by its type and the identities of the instances referred to by its key fields.

The programmer does not explicitly allocate a tuple using `new`; instead, she calls a predefined `id()` method, whose formal parameters correspond exactly to the types of the `key` fields (including all those declared in its supertypes). Informally, the `id()` method does an associative lookup to find the tuple with the same identity. If no such tuple is found, `id()` creates a new tuple.

Figure 3 shows a simple example of pseudo-code with relation types. The relation type `Car` declares key fields `model`, `year`, and `plate`. This means that two cars with the same `model`, `year` and `plate` have the same identity and are indistinguishable. Since `Car` has no mutable state, it can be viewed as a value type [5].

Relation types are more general than value types because tuples may contain mutable state. Consider relation type `Person`, which has a key field `SSN` and a mutable field `Name`. This means that there exists at most one `Person` tuple with a given `SSN`, and that assignments to `SSN` are forbidden. So on the right side of the example, variables `p1` and `p2` refer to the same tuple (they are aliased). Assignments to the non-key field `name` are allowed (see line 10).

Inheritance among relation types resembles single inheritance for classes: subtypes may add (but not remove) additional key fields as well as other instance fields and methods. A subtype inherits methods and fields declared in a relation supertype. A relation type and its subtype are incomparable; subtype tuples have different identity from supertype tuples. Should the programmer want to compare a tuple to the corresponding subtuple of a subtype, the language provides predefined *coercion methods* to convert subtypes to supertypes.

Consider the relation type `Name` and its subtype `FullName` in the figure. Tuples of these relations have different identities (see line 11), and the predefined coercion operator `toName()` must be used to compare the corresponding key

```
relation Car {
    key String model;
    key int year;
    key String plate;
}

relation Person {
    key int SSN;
    Name name;
}

relation Name {
    key String first;
    key String last;
}

relation FullName extends Name {
    key String middle;
    String nickname;
}

class Policy { ... }

relation CarInsurance {
    key Person person;
    key Car car;
    Policy policy;
    int computePremium() { ... }
}

relation PolicyMgr {
  // no key fields
  public void addPolicy(Policy p){
    policies.add(p);
  }
    List<Policy> policies =
    new ArrayList<Policy>();
}
```

```
1.  public static void main(String[] args){
2.      Person p1 = Person.id(123);
3.      Person p2 = Person.id(123);
4.      Person p3 = Person.id(456);
5.      // p1 == p2
6.      // p1 != p3
7.      // p1.SSN = 789 --> compile error

8.      Name n1 = Name.id("Alice","Jones");
9.      Name n2 = FullName.id("Alice","Jones","Leah");
10.     p1.name = n1;
11.     // n1 != n2
12.     // n1 == ((FullName)n2).toName()
13.     // p2.name.first == "Alice"
14.     // p2.name.last == "Jones"

15.     Policy pol1 = new Policy();
16.     Policy pol2 = new Policy();
17.     // pol1 != pol2

18.     Car c1 = Car.id("Chevy",2004,"DZN-6767");
19.     CarInsurance cins = CarInsurance.id(p1,c1);
20.     cins.policy = pol1;
21.     cins.computePremium();

22.     PolicyMgr pm = PolicyMgr.id();
23.     pm.add(pol1);
24.     pm.add(pol2);

25.     Set<Person> people = new HashSet<Person>();
26.     people.add(p1);
27.     people.add(p3);
28.     // people.contains(p2) is true

29.     p2.name = n2;
30.     // ((FullName)p1.name).middle == "Leah";
31.     // people.contains(p2) is still true
32.   }
```

**Fig. 3.** Example of relation types.

fields of these relations (see line 12). The assertions shown on lines 13 and 14 follow from the fact that p1 and p2 refer to the same tuple.

Conceptually, traditional Java classes (with address-based identity) correspond to relation types with an implicitly defined key field address, assigned a unique value by an object constructor. We use the class keyword to indicate a record type with an implicit address field (see class Policy in the example). The two tuples (objects) of type Policy created at lines 15 and 16 will not have the same identity (see line 17). Note that classes may not explicitly declare key fields or inherit from relation types that do. Our relation keyword indicates the absence of an address key field.

The relation type CarInsurance illustrates how relation types provide a relational view of the heap. The CarInsurance type maps distinct identities to mutable state stored in the policy field. By analogy to relational databases, the CarInsurance type resembles a relational table with three columns, two of which are keys. The type also defines methods such as computePremium() that may refer to all of all state of a particular CarInsurance tuple.

If a relation type has no key fields, then it corresponds to the Singleton design pattern [15], since its identity consists solely of the type. Figure 3 shows a (singleton) relation type `PolicyMgr` that provides access to a globally accessible list of insurance policies. Lines 22–24 access this list.

Finally, lines 25–31 illustrate what happens when we insert tuples into collections. Here, we define a set `people` and add `p1` and `p3` to it. Since `p1` and `p2` are equal, the test `people.contains(p2)` returns true. Now if we modify `p2` by changing its `name` field (line 26), `p2` remains in the set as expected (line 28). The result of the test remains unchanged because the identity of `p2` did not depend on mutable state, and `p2` was not removed from the set.

### 2.5 Lifetime Management and Namespaces

Thus far, we assumed that each relation type provides a single, global namespace for tuples of said type. Under this model, the program can support at most one tuple with a given identity. Now, consider the case where at least one of a tuple *t*'s key fields contains a reference to an object *o*. Then, there is no way to reconstruct *t*'s identity once *o* has been garbage-collected, and *t* itself becomes garbage at that point. However, if *t*'s identity is computable (i.e., all key fields refer to values that can be recomputed), then the implementation cannot know *a priori* whether the program will try to retrieve *t* in the future. In such cases, *t* is immortal and cannot be garbage-collected.

For a more flexible, practical model, the programmer can use *scopes* to provide separate namespaces for a type, and also to control tuple lifetime. Consider the pseudo-code of Figure 4(a). The code creates two `Persons`, each with the same identity (3), but which reside in different scopes. First, an object of an implicitly defined type `Scope<Person>` is created by calling a predefined method `Person.newScope()`. Type `Scope<Person>` declares an `id()` method with the same signature as that of `Person`. Then, instead of creating a tuple from global namespace via `Person.id()`, the progammer allocates a tuple from a particular named scope (e.g., `s1.id()`). Regarding garbage collection: a tuple becomes garbage when the program holds no references to its containing scope (provided all of its key fields have become garbage). In the example code, if `foo` returns `jack`, then `jane` may be garbage-collected when `foo` returns, since there will be no live references to `jane` nor its scope `s2`.

```
Person foo() {
  Scope<Person> s1 = Person.newScope();
  Person jack = s1.id(3);
  jack.setName(Name.id("Jack","Sprat"));
  Scope<Person> s2 = Person.newScope();
  Person jane = s2.id(3);
  jane.setName(Name.id("Jane","Sprat"));
  return (*) ? jack : jane;
}
```

**(a)**

```
Person foo() {
  Object s1 = new Object();
  Person jack = Person.id(3,s1);
  jack.setName(Name.id("Jack","Sprat"));
  Object s2 = new Object();
  Person jane = Person.id(3,s2);
  jane.setName(Name.id("Jane","Sprat"));
  return (*) ? jack : jane;
}
```

**(b)**

**Fig. 4.** Example of scopes.

The base programming model can emulate programming with scopes. We add to each record type an implicit key field called `scope`, whose type is an object, and whose reference is *weak* (does not preclude garbage collection). Figure 4(b) shows how the example of Figure 4(a) could be emulated. The language could include further syntactic sugar for scopes, such as lexical scope declarations, and utility functions to move tuples between different scopes via copying.

## 3 A Core Calculus for RJ

We formally define a core calculus for the RJ language as an adaptation of Featherweight Java [17] (FJ). For simplicity, we adopt the notation and structures of the model presented in [17]. RJ is analogous to Featherweight Java in its technical details, except that it has relation types instead of classes, and allows assignment.

### 3.1 Syntax

We use the notation $\bar{x}$ to denote a possibly empty sequence, indexed starting at 1, and $\bullet$ for the empty sequence. The size of the sequence is indicated by $\#\bar{x}$. We write $\bar{x} \in X$ to denote $x_1 \in X, \cdots, x_{\#\bar{x}} \in X$ (and similarly $\bar{x} \notin X$). For any partial function $F$, we write $\mathit{Dom}(F)$ for the domain of $F$. The notation $F(\bar{x})$ is short for the sequence $F(x_1), \cdots, F(x_n)$.

We use $L$ to denote a relation type declaration, and $M$ a method. We write $R$, $S$ and $T$ for relation type names; $f$, $g$ and $h$ for field names; $x$ for variables; $e$ and $c$ for expressions; $l$ and $a$ for memory locations, with subscripts as needed. The notation "$\bar{R}\ \bar{x}$" is shorthand for the sequence "$R_1\ x_1, \cdots, R_n\ x_n$" and similarly "$\bar{R}\ \bar{f}$;" is shorthand for the sequence declarations "$R_1\ f_1; \cdots R_n\ f_n$;" for some $n$. The declaration "`key` $\bar{R}\ \bar{f}$;" is similar with the annotation '`key`' preceding every declaration in the sequence.

The syntax of the RJ language is shown below:

$L$ ::= `relation` $R$ `extends` $R$ $\{$ `key` $\bar{R}\ \bar{f};\ \bar{R}\ \bar{f};\ \bar{M}\ \}$
$M$ ::= $R\ m(\bar{R}\ \bar{x})$ $\{$ `return` $e$; $\}$
$e$ ::= $x \mid e.f \mid e.m(\bar{e}) \mid e$ `==` $e \mid$ `if` $e$ `then` $e$ `else` $e \mid e.f \leftarrow e \mid R.$`id`$(\bar{e}) \mid l$

A relation type is declared to be a subtype of another using the `extends` keyword and consists of series of field declarations and a series of method declarations. Some field declarations are marked with the keyword `key` and represent the *key fields* of the relation type. Key fields are immutable, and non-key fields may or may not be mutable. We assume that there is an uninstantiatable relation type `Relation` at the top of the type hierarchy with no fields and no methods. As in Featherweight Java, a subtype inherits fields and methods, field declarations are not allowed to shadow fields of supertypes, and methods may override other methods with the same signature. A method declaration specifies a method's return type, name, formal parameters, and body. The body consists of a single statement which returns an expression $e$, which may refer to formal parameters and the variable `this`. Note that relation types do not have constructors. Instead, a tuple is constructed using a method `id()` with predefined functionality.

An expression consists of a variable, a field access, a method call, an equality test, or an `if–then–else` construct. Other forms are an assignment $e.f \leftarrow e$, which assigns to $e.f$ and evaluates to the value being assigned, or a relation construction expression $R$`.id(`$\bar{e}$`)`, which takes as many parameters as there are key fields in $R$ and its supertypes (in the same order). Informally, $R$`.id(`$\bar{e}$`)` refers to the tuple of type $R$ whose arguments are denoted by $\bar{e}$. If such a tuple already exists, then it is returned (with all existing non-key state), otherwise it is created with all non-key fields set to `null`. Finally, an expression may be a memory location $l$, which is used for defining semantics only and not by the programmer.

A relation table *RTable* is a mapping from a relation type name $R$ to a declaration of the form "`relation` $R$ `...`". *RTable* is assumed to contain an entry for every relation type except for the top level type `Relation`. The subtype relation (denoted <:) is obtained in a customary way [17]. An RJ program consists of a pair (*RTable*, $e$) of a relation table and an expression.

### 3.2 Semantics

Figure 5 shows some auxiliary functions needed to define the reduction rules of RJ. The function *keys*() returns the set of key fields of a relation type, while *nonKeys*() returns its non-key fields. Function *fields*() returns the sequence of all fields of a relation type. Partial function *mbody*($m$, $R$) looks up the body of method named $m$ for relation type $R$ and returns a pair of formal parameters and the expression that constitutes its body, written as $\bar{x}.e$. The notation $m \notin \bar{M}$ means that there is no method declaration for a method named $m$ in $\bar{M}$. We do not deal with method overloading as in Featherweight Java.

The *arity* of a relation type $R$ is the number of key fields in $R$ and all its supertypes, and is denoted by $|R|$.

Traditionally, a heap is defined as a map from memory locations and field names to memory locations. In our programming model, the identity of a tuple constitutes a high-level address for it. To reflect this, the heap in our model is comprised of two components: a map from locations to identities, and a map from identities and field names to locations. There is therefore a level of indirection to introduce these high-level addresses.

Let Locs be a set of memory locations, Ids a set of *identities*, and Fields the set of field names. We use $k$ to denote an identity. Let a *heap* $\mathcal{H}$ be a partial function of type $(\textsc{Ids}, \textsc{Fields}) \to \textsc{Locs}$, and a *heap index* $\mathcal{L}$ a partial function of type $\textsc{Locs} \to \textsc{Ids}$. A heap index $\mathcal{L}$ *corresponds to* a heap $\mathcal{H}$ if and only if $\mathcal{L}$ is defined for every location in the range of $\mathcal{H}$ and every identity in the domain of $\mathcal{H}$ is in the range of $\mathcal{L}$. We use the notation $\mathcal{H}[(k, f) \to l]$ to denote the heap function identical to $\mathcal{H}$ except at $(k, f)$, which is mapped to $l$. Similarly, $\mathcal{L}[l \to k]$ denotes heap index $\mathcal{L}$ with $l$ mapped to $k$. We write $\mathcal{L}[\bar{l} \to \bar{k}]$ to denote $\mathcal{L}[l_1 \to k_1] \cdots [l_n \to k_n]$, where $\bar{l}$ and $\bar{k}$ have size $n$.

**Key lookup:**

$$\mathit{keys}(\texttt{Relation}) = \bullet \qquad \frac{\texttt{relation}\ R\ \texttt{extends}\ S\ \{\texttt{key}\ \bar{T}\ \bar{g};\ \bar{R}\ \bar{f};\ \bar{M}\} \quad \mathit{keys}(S) = \bar{S}\ \bar{h}}{\mathit{keys}(R) = \bar{S}\ \bar{h},\ \bar{T}\ \bar{g}}$$

**Non-Key lookup:**

$$\mathit{nonKeys}(\texttt{Relation}) = \bullet \qquad \frac{\begin{array}{c}\texttt{relation}\ R\ \texttt{extends}\ S\ \{\texttt{key}\ \bar{T}\ \bar{g};\ \bar{R}\ \bar{f};\ \bar{M}\} \\ \mathit{nonKeys}(S) = \bar{S}\ \bar{h}\end{array}}{\mathit{nonKeys}(R) = \bar{S}\ \bar{h},\ \bar{R}\ \bar{f}}$$

**Field lookup:**

$$\mathit{fields}(R) = \mathit{keys}(R),\ \mathit{nonKeys}(R)$$

**Method body lookup:**

$$\frac{\texttt{relation}\ R\ \texttt{extends}\ S\ \{\texttt{key}\ \bar{T}\ \bar{g};\ \bar{R}\ \bar{f};\ \bar{M}\} \quad T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mbody}(m,\ R) = \bar{x}.e}$$

$$\frac{\texttt{relation}\ R\ \texttt{extends}\ S\ \{\texttt{key}\ \bar{T}\ \bar{g};\ \bar{R}\ \bar{f};\ \bar{M}\} \quad m \notin \bar{M}}{\mathit{mbody}(m,\ R) = \mathit{mbody}(m,\ S)}$$

**Fig. 5.** Auxiliary functions for RJ

The function $\mathit{alloc}_n(\mathcal{L})$ allocates a sequence of $n$ fresh locations not mapped by $\mathcal{L}$. We write $\mathit{alloc}(\mathcal{L})$ as a shorthand for $\mathit{alloc}_1(\mathcal{L})$.

Finally, we introduce a function $\mathcal{I}$ that computes the identity of a tuple, given its type and the identities of its key fields. Formally, $\mathcal{I}$ is a partial injective function $\mathcal{I}$ that maps a relation type and a sequence of identities to an identity in Ids. $\mathcal{I}(R,\ \bar{k})$ is defined when $\#\bar{k} = |R|$, and undefined otherwise. Note that the base case for constructing identities is to construct the identity for a relation type without key fields (i.e., a singleton).

The typing rules for RJ are straightforward adaptations of those for Featherweight Java, and can be found in a forthcoming technical report [30].

Figure 6 shows reduction rules for RJ. A reduction step is of the form: $(\mathcal{L},\ \mathcal{H},\ e) \longrightarrow (\mathcal{L}',\ \mathcal{H}',\ e')$ and means that $e$ reduces in one step to $e'$ in the context of heap index $\mathcal{L}$ and heap $\mathcal{H}$, resulting in $\mathcal{L}'$ and $\mathcal{H}'$. We use the notation $\longrightarrow_{RJ}$ to denote one step of reduction in RJ, when it is not clear from context. We write $\longrightarrow^*_{RJ}$ to denote the reflexive, transitive closure of the $\longrightarrow_{RJ}$ relation.

Rule R-Field reduces expression $l.f_i$, where $l$ is a location, by looking up the identity that $l$ maps to in $\mathcal{L}$, and the location mapped to by this identity and field $f_i$ in $\mathcal{H}$, i.e. $\mathcal{H}(\mathcal{L}(l), f_i)$.

Rule R-Invk deals with method invocation and applies only after the receiver and parameters have been reduced to locations. The expression $[\bar{a}/\bar{x},\ l/\texttt{this}]e$ denotes $e$ in which formal parameters $\bar{x}$ and `this` have been replaced with actual parameters $\bar{a}$ and receiver $l$, respectively.

Rules R-Eq-True R-Eq-False show how to reduce an equality between two memory locations $l_1$ and $l_2$. These are equal if they hold the same identity, i.e., $\mathcal{L}(l_1) = \mathcal{L}(l_2)$.

Rules R-If-True and R-If-False show the reduction for the if-then-else expression in the obvious way.

$$\frac{\mathit{fields}(R) = \bar{R}\ \bar{f} \quad l \in \textsc{Locs} \quad l_i = \mathcal{H}(\mathcal{L}(l), f_i)}{(\mathcal{L},\ \mathcal{H},\ l.f_i) \ \longrightarrow\ (\mathcal{L},\ \mathcal{H},\ l_i)} \qquad (\textsc{R-Field})$$

$$\frac{\mathit{mbody}(m,\ R) = \bar{x}.e \quad l, \bar{a} \in \textsc{Locs}}{(\mathcal{L},\ \mathcal{H},\ l.m(\bar{a})) \ \longrightarrow\ (\mathcal{L},\ \mathcal{H},\ [\bar{a}/\bar{x},\ l/\mathit{this}]e)} \qquad (\textsc{R-Invk})$$

$$\frac{l_1, l_2 \in \textsc{Locs} \quad \mathcal{L}(l_1) = \mathcal{L}(l_2)}{(\mathcal{L},\ \mathcal{H},\ l_1 == l_2) \ \longrightarrow\ (\mathcal{L},\ \mathcal{H},\ \mathit{true})} \qquad (\textsc{R-Eq-True})$$

$$\frac{l_1, l_2 \in \textsc{Locs} \quad \mathcal{L}(l_1) \neq \mathcal{L}(l_2)}{(\mathcal{L},\ \mathcal{H},\ l_1 == l_2) \ \longrightarrow\ (\mathcal{L},\ \mathcal{H},\ \mathit{false})} \qquad (\textsc{R-Eq-False})$$

$$(\mathcal{L},\ \mathcal{H},\ \text{if } \mathit{true} \text{ then } e_T \text{ else } e_F) \ \longrightarrow\ (\mathcal{L},\ \mathcal{H},\ e_T) \qquad (\textsc{R-If-True})$$

$$(\mathcal{L},\ \mathcal{H},\ \text{if } \mathit{false} \text{ then } e_T \text{ else } e_F) \ \longrightarrow\ (\mathcal{L},\ \mathcal{H},\ e_F) \qquad (\textsc{R-If-False})$$

$$\frac{\mathit{nonKeys}(R) = \bar{R}\ \bar{f} \quad l, a \in \textsc{Locs}}{(\mathcal{L},\ \mathcal{H},\ l.f_i \leftarrow a) \ \longrightarrow\ (\mathcal{L},\ \mathcal{H}[(\mathcal{L}(l), f_i) \to a],\ a)} \qquad (\textsc{R-Assign})$$

$$\frac{k = \mathcal{I}(R) \quad l = \mathit{alloc}(\mathcal{L})}{(\mathcal{L},\ \mathcal{H},\ R.\mathrm{id}()) \ \longrightarrow\ (\mathcal{L}[l \to k],\ \mathcal{H},\ l)} \qquad (\textsc{R-Id-NoKey})$$

$$\frac{\begin{array}{c}\bar{a} \in \textsc{Locs} \quad k = \mathcal{I}(R,\ \mathcal{L}(\bar{a})) \quad \mathit{keys}(R) = \bar{R}\ \bar{g} \\ (k, \bar{g}) \notin \mathit{Dom}(\mathcal{H}) \quad l, \bar{l} = \mathit{alloc}_{\#\bar{a}+1}(\mathcal{L})\end{array}}{(\mathcal{L},\ \mathcal{H},\ R.\mathrm{id}(\bar{a})) \ \longrightarrow\ (\mathcal{L}[l \to k][\bar{l} \to \mathcal{L}(\bar{a})],\ \mathcal{H}[(k,\bar{g}) \to \bar{l}],\ l)} \qquad (\textsc{R-Id-Create})$$

$$\frac{\begin{array}{c}\bar{a} \in \textsc{Locs} \quad k = \mathcal{I}(R,\ \mathcal{L}(\bar{a})) \quad \mathit{keys}(R) = \bar{R}\ \bar{g} \\ (k, \bar{g}) \in \mathit{Dom}(\mathcal{H}) \quad l = \mathit{alloc}(\mathcal{L})\end{array}}{(\mathcal{L},\ \mathcal{H},\ R.\mathrm{id}(\bar{a})) \ \longrightarrow\ (\mathcal{L}[l \to k][\bar{l} \to \mathcal{L}(\bar{a})],\ \mathcal{H},\ l)} \qquad (\textsc{R-Id-Find})$$

---

$$\frac{(\mathcal{L},\ \mathcal{H},\ e) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e')}{(\mathcal{L},\ \mathcal{H},\ e.f) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e'.f)} \qquad (\textsc{RC-Field})$$

$$\frac{(\mathcal{L},\ \mathcal{H},\ e) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e')}{(\mathcal{L},\ \mathcal{H},\ e.m(\bar{c})) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e'.m(\bar{c}))} \qquad (\textsc{RC-Invk-Recv})$$

$$\frac{(\mathcal{L},\ \mathcal{H},\ e) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e') \quad l, \bar{a} \in \textsc{Locs}}{(\mathcal{L},\ \mathcal{H},\ l.m(\bar{a}, e, \bar{c})) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ l.m(\bar{a}, e', \bar{c}))} \qquad (\textsc{RC-Invk-Arg})$$

$$\frac{(\mathcal{L},\ \mathcal{H},\ e) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e')}{(\mathcal{L},\ \mathcal{H},\ e == c) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e' == c)} \qquad (\textsc{RC-Eq-1})$$

$$\frac{(\mathcal{L},\ \mathcal{H},\ e) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e') \quad l \in \textsc{Locs}}{(\mathcal{L},\ \mathcal{H},\ l == e) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ l == e')} \qquad (\textsc{RC-Eq-2})$$

$$\frac{(\mathcal{L},\ \mathcal{H},\ e) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e')}{(\mathcal{L},\ \mathcal{H},\ e.f \leftarrow c) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e'.f \leftarrow c)} \qquad (\textsc{RC-Assign-1})$$

$$\frac{(\mathcal{L},\ \mathcal{H},\ e) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e') \quad l \in \textsc{Locs}}{(\mathcal{L},\ \mathcal{H},\ l.f \leftarrow e) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ l.f \leftarrow e')} \qquad (\textsc{RC-Assign-2})$$

$$\frac{(\mathcal{L},\ \mathcal{H},\ e) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ e') \quad \bar{a} \in \textsc{Locs}}{(\mathcal{L},\ \mathcal{H},\ R.\mathit{id}(\bar{a}, e, \bar{c})) \ \longrightarrow\ (\mathcal{L}',\ \mathcal{H}',\ R.\mathit{id}(\bar{a}, e', \bar{c}))} \qquad (\textsc{RC-Id-Arg})$$

**Fig. 6.** Computation and Congruence Rules for RJ

Rule R-Assign shows how to reduce an assignment expression $l.f_i \leftarrow a$. Field $f$ must be non-key, and $l$ and $a$ locations. The expression reduces to $a$ and the heap $\mathcal{H}$ is replaced with one that is identical except at $(\mathcal{L}(l),f_i)$, which is now mapped to $a$.

Rule R-Id-NoKey shows the reduction of a constructor expression for a relation type with no key fields $R$.`id()`. This expression is reduced to a fresh memory location $l$, which is mapped to the corresponding identity $k = \mathcal{I}(R)$ in the new heap index.

Rule R-Id-Create shows the reduction of a constructor expression $R$.`id`$(\bar{a})$ for the case where the identified tuple has not been created yet. For this rule to apply, the arguments must have been already reduced to locations $\bar{a}$. The identity $k$ of the tuple is computed using $\mathcal{I}(R,\ \mathcal{L}(\bar{a}))$. The expression $l, \bar{l} = alloc_{\#\bar{a}+1}(\mathcal{L})$ allocates $\#\bar{a}+1$ fresh memory locations from $\mathcal{L}$, one for the tuple itself, and $\#\bar{a}$ for each of its key fields. The constructor expression reduces to location $l$, which is mapped to $k$ in the new heap index. The heap itself is also updated at $(k,g)$ for each key field $g$ in $\bar{g}$ of $R$ to a fresh memory location. The notation $\mathcal{H}[(k,\ \bar{g}) \rightarrow \bar{l}]$ represents $\mathcal{H}[(k, g_1) \rightarrow l_1] \cdots [(k, g_n) \rightarrow l_n]$, where $\bar{g}$ and $\bar{l}$ have size $n$. The typing rules [30] guarantee that the constructor for $R$ has as many arguments as its number of key fields, which means that sequences $\bar{a}$ and $\bar{g}$ have the same length. Rule R-Id-Create applies when $(k, \bar{g})$ is not in the domain of $\mathcal{H}$ and the tuple is therefore created.

Rule R-Id-Find is similar, except that the tuple exists in the heap and it is therefore not updated. The constructor expression still reduces to a fresh memory location $l$, which is mapped to the identity $k$ in the new heap index.

The rest of the reduction rules (Figure 6) ensure that an expression is reduced in a deterministic fixed order.

RJ's computation on a program $(RTable, e)$ starts in a state $(\mathcal{L},\ \mathcal{H},\ e)$, where $\mathcal{L}$ and $\mathcal{H}$ have empty domains, and consists of a sequence of states obtained by applying the reduction rules, until none applies.

### 3.3 The RJ-HC Language

In this section, we prove that the hash-consing optimization [16], which consists of storing equal values at the same memory location, preserves semantics for RJ. To this end, we present RJ-HC, a version of the core calculus of RJ with a hash-consing operational semantics.

RJ-HC has the same syntax and auxiliary functions as RJ, except for memory allocation, which performs hash-consing. The auxiliary function $alloc_{\text{RJ-HC}}(\mathcal{L},\ k)$ in RJ-HC returns a location $l$ that maps to $k$ in $\mathcal{L}$, if such a location exists, and a fresh location otherwise. We write $alloc_{\text{RJ-HC}}(\mathcal{L},\ \bar{k})$ to denote the sequence $alloc_{\text{RJ-HC}}(\mathcal{L},\ k_1), \cdots, alloc_{\text{RJ-HC}}(\mathcal{L},\ k_n)$, where $n$ is the length of $\bar{k}$.

The reduction rules of RJ-HC are identical to RJ, except for R-Id-NoKey, R-Id-Create, and R-Id-Find. Figure 7 shows these new rules for RJ-HC. Rule R-Id-NoKey-HC is similar to that in RJ, except that it uses the new allocation function. There is a single rule R-Id-HC for the constructor expression, which also uses the hash-consing allocation function. It always updates the heap $\mathcal{H}$, possibly rewriting it with existing values.

We use the notation $\longrightarrow_{\textit{RJ-HC}}$ to denote one step of reduction in RJ-HC, when it is not clear from context. We write $\longrightarrow^{*}_{\textit{RJ-HC}}$ to denote the reflexive, transitive closure of the $\longrightarrow_{\textit{RJ-HC}}$ relation.

$$\frac{k = \mathcal{I}(R) \quad l = alloc_{\textit{RJ-HC}}(\mathcal{L},\ k)}{(\mathcal{L},\ \mathcal{H},\ R.\mathrm{id}()) \ \longrightarrow\ (\mathcal{L}[l \to k],\ \mathcal{H},\ l)} \qquad (\text{R-I{\scriptsize D}-N{\scriptsize O}K{\scriptsize EY}-HC})$$

$$\frac{\bar{a} \in \textsc{Locs} \quad k = \mathcal{I}(R,\ \mathcal{L}(\bar{a})) \quad keys(R) = \bar{R}\ \bar{g} \\ l = alloc_{\textit{RJ-HC}}(\mathcal{L},\ k) \quad \bar{l} = alloc_{\textit{RJ-HC}}(\mathcal{L},\ \mathcal{L}(\bar{a}))}{(\mathcal{L},\ \mathcal{H},\ R.\mathrm{id}(\bar{a})) \ \longrightarrow\ (\mathcal{L}[l \to k][\bar{l} \to \mathcal{L}(\bar{a})],\ \mathcal{H}[(k,\bar{g}) \to \bar{l}]\ ,\ l)} \qquad (\text{R-I{\scriptsize D}-HC})$$

**Fig. 7.** New Computation Rules for RJ-HC

We now show that RJ and RJ-HC have the same behavior on the same program (*RTable*, $e$). First, some definitions:

**Definition 1. (Well-Formed State)** *A state ($\mathcal{L}$, $\mathcal{H}$, $e$) in a computation of RJ (RJ-HC) is* well-formed *if $\mathcal{L}$ corresponds to $\mathcal{H}$ and for every location $l$ appearing in $e$, $\mathcal{L}$ is defined at $l$.*

It is easy to show that reduction preserves well-formedness both in RJ and RJ-HC.

**Definition 2. (Structural Equivalence)** *We say that two well-formed states ($\mathcal{L}$, $\mathcal{H}$, $e$) and ($\mathcal{L}'$, $\mathcal{H}'$, $e'$) are* structurally equivalent *if:*

1. *$\mathcal{H}$ and $\mathcal{H}'$ have the same domain and for all $(k,f)$ in that domain: $\mathcal{L}(\mathcal{H}(k,f)) = \mathcal{L}'(\mathcal{H}'(k,f))$*
2. *$[\mathcal{L}(\bar{l})/\bar{l}]e = [\mathcal{L}'(\bar{l'})/\bar{l'}]e'$, where $\bar{l}$ and $\bar{l'}$ are sequences of locations appearing in $e$ and $e'$, respectively. To denote this condition we write $e \equiv e'$ when $\mathcal{L}$ and $\mathcal{L}'$ are clear from the context.*

In Definition 2, item 1 states that the heaps and heap indices must have the same structure. Item 2 states that expressions $e$ and $e'$ where all locations are substituted with their corresponding identities are syntactically identical.

**Lemma 1.** *Assume that $(\mathcal{L}_o,\ \mathcal{H}_o,\ e_o)$ and $(\mathcal{L},\ \mathcal{H},\ e)$ are structurally equivalent states resulting from the computation of RJ-HC and RJ, respectively, on the same program. If $(\mathcal{L}_o,\ \mathcal{H}_o,\ e_o) \longrightarrow_{\text{RJ-HC}} (\mathcal{L'}_o,\ \mathcal{H'}_o,\ e'_o)$ then there exists a state $(\mathcal{L}',\ \mathcal{H}',\ e')$, such that $(\mathcal{L},\ \mathcal{H},\ e) \longrightarrow_{\text{RJ}} (\mathcal{L}',\ \mathcal{H}',\ e')$ and $(\mathcal{L'}_o,\ \mathcal{H'}_o,\ e'_o)$ and $(\mathcal{L}',\ \mathcal{H}',\ e')$ are structurally equivalent.*

*Proof.* By induction on the derivation of $(\mathcal{L}_o,\ \mathcal{H}_o,\ e_o) \longrightarrow_{\textit{RJ-HC}} (\mathcal{L'}_o,\ \mathcal{H'}_o,\ e'_o)$ with a case analysis on the last reduction rule used. The proof can be found in [30].

**Theorem 1.** *Assume that* $(\mathcal{L}_o,\ \mathcal{H}_o,\ e_o)$ *and* $(\mathcal{L},\ \mathcal{H},\ e)$ *are structurally equivalent states resulting from the computation of RJ-HC and RJ, respectively, on the same program. If* $(\mathcal{L}_o,\ \mathcal{H}_o,\ e_o) \longrightarrow^*_{\text{RJ-HC}} (\mathcal{L}'_o,\ \mathcal{H}'_o,\ e'_o)$ *then there exists a state* $(\mathcal{L}',\ \mathcal{H}',\ e')$*, such that* $(\mathcal{L},\ \mathcal{H},\ e) \longrightarrow^*_{\text{RJ}} (\mathcal{L}',\ \mathcal{H}',\ e')$ *and* $(\mathcal{L}'_o,\ \mathcal{H}'_o,\ e'_o)$ *and* $(\mathcal{L}',\ \mathcal{H}',\ e')$ *are structurally equivalent.*

*Proof.* By induction on the length $n$ of reduction sequence $(\mathcal{L}_o,\ \mathcal{H}_o,\ e_o) \longrightarrow^*_{\textit{RJ-HC}} (\mathcal{L}'_o,\ \mathcal{H}'_o,\ e'_o)$.

**Case:** $n = 0$. Trivial.

**Case:** $(\mathcal{L}_o,\ \mathcal{H}_o,\ e_o) \longrightarrow_{\textit{RJ-HC}} (\mathcal{L}''_o,\ \mathcal{H}''_o,\ e''_o) \longrightarrow^*_{\textit{RJ-HC}} (\mathcal{L}'_o,\ \mathcal{H}'_o,\ e'_o)$.
By Lemma 1, we know that there exists a state $(\mathcal{L}'',\ \mathcal{H}'',\ e'')$ such that $(\mathcal{L},\ \mathcal{H},\ e) \longrightarrow_{\textit{RJ}} (\mathcal{L}'',\ \mathcal{H}'',\ e'')$ and $(\mathcal{L}''_o,\ \mathcal{H}''_o,\ e''_o)$ and $(\mathcal{L}'',\ \mathcal{H}'',\ e'')$ are structurally equivalent. By the induction hypothesis, there exists a state $(\mathcal{L}',\ \mathcal{H}',\ e')$ such that $(\mathcal{L}'',\ \mathcal{H}'',\ e'') \longrightarrow^*_{\textit{RJ}} (\mathcal{L}',\ \mathcal{H}',\ e')$, and $(\mathcal{L}',\ \mathcal{H}',\ e')$ and $(\mathcal{L}'_o,\ \mathcal{H}'_o,\ e'_o)$ are structurally equivalent. □

## 4 Implementation and Evaluation

To evaluate the utility of relation types, we extended Java with relation types and developed a compiler for translating programs written in the resulting RJ language to Java. We examined the classes that define `equals()` and `hashCode()` in a number of open-source Java applications. For each application, we determined if and how these classes could be rewritten with relation types.

### 4.1 Implementation

RJ adds a few minor extensions to Java syntax:

- The `relation` keyword indicates that a class or interface is a relation type.
- The `key` keyword indicates that a field in a relation type is a key field. A relation class may have zero or more key fields.
- For each relation class $R$, an `id()` method is implicitly defined with return type $R$ and argument types corresponding to the key fields in $R$ and its supertypes.

Conceptually, the hierarchy of relation types is completely distinct from the hierarchy of (non-relation) reference types. For pragmatic reasons, the implementation makes `java.lang.Object` the implicit supertype of a all relation types, but relation types cannot inherit explicitly from a reference types or vice versa.

We have implemented RJ using the Java 5.0 metadata facility. Embedding the RJ language in Java enabled us to leverage the Eclipse JDT refactoring framework as the basis for our compiler. Concretely, relation types are annotated with a `@Relation` annotation and key fields with a `@Key` annotation. Furthermore, we model the implicitly defined `id()` method as a constructor annotated with the `@Id` annotation[2]. Since our experiments target converting Java classes into relation types, our RJ implementation allows non-relation types and relations to co-exist. Specifically, we allow the declaration of `equals()` and `hashCode()` methods in non-relation Java classes.

We implemented a simple type checker for RJ that enforces the following constraints on relation types:

- Up-casts (implicit or explicit) from a relation type to `Object` are disallowed.
- Key fields must be `private` and `final`.
- Declaring `equals()` and `hashCode()` in a relation type is disallowed.
- In order to avoid programmer errors, the application of the `==` and `!=` operators to one operand of a relation type and another operand of a reference type results in a type error.
- Calling `equals()` on an expression of a relation type is a type error.

The RJ compiler translates RJ to Java using the AST rewriting infrastructure in Eclipse. The translation involves the following steps[3]: (i) generation of a nested `Key` class that contains the key fields declared in a relation type and that implements appropriate `equals()` and `hashCode()` methods, (ii) generation of a static map that contains the relation's tuples, (iii) generation of a constructor that initializes the key fields from corresponding formal parameters, (iv) generation of the `id()` method that returns a tuple with a given identity if it already exists, and creates such a tuple otherwise, and (v) updating the references to key fields (necessary because these fields are moved into the generated `Key` class). Figure 8 shows the annotated source and generated code for the `Person` class from Figure 3.

In the basic implementation approach discussed so far, tuples are never garbage collected. Therefore we have an alternative implementation based on weak references, which collects tuples when their identity becomes unreachable, as we discussed in Section 2.5. In this approach, the key fields use `WeakReference`s as pointers, and relation types use the `ReferenceQueue` notification mechanism to remove a tuple when any of its weak referents becomes dead. Additionally, the canonicalized tuple objects are cached using `SoftReference`s. If none of the key fields of a relation type are of reference types, the scope mechanism discussed in Section 2.5 can be used. A scope is a reference, so when the scope dies, so do its tuples.

### 4.2 Case Study: *javacup*

We now describe in detail one case study, investigating how *javacup* (version 11a), an LALR parser generator written in Java, can be refactored to use relation

[2] In a full language implementation, the programmer would not need to declare an `id()` method; our prototype implementation requires the explicit constructor as an expedient way to interoperate with the Eclipse Java development tools.

[3] This translation aims at the maximum amount of sharing. Several alternative translation approaches exist, e.g., one could directly include all key fields in the tuple but this is potentially inefficient because one cannot use `==` to compare tuples in the translated code.

```
@Relation public class Person {
  @Key private final int SSN;
  @Key private final Name name;
  @Id private Person(int SSN, Name name) {
    this.SSN = SSN;
    this.name = name;
  }
}
```

```
public class Person {
  protected final Key key;
  protected Person(Key key) {
    this.key = key;
  }
  public static Person id(int SSN, Name name) {
    Key k = new Key(SSN, name);
    Person c = m.get(k);
    if (c == null) {
      c = new Person(k);
      m.put(k, c);
    }
    return c;
  }
  private static Map<Key, Person> m =
    new HashMap<Key, Person>();

  protected static class Key {
    public Key(int SSN, Name name) {
      this.SSN = SSN;
      ... // continued on right column
      ...
      this.name = name;
    }
    public boolean equals(Object o) {
      if (o = null &&
        getClass().equals(o.getClass())) {
        Key other = (Key) o;
        return SSN == other.SSN &&
          (name == null) ? (other.name == null)
          : name.equals(other.name);
      }
      return false;
    }
    public int hashCode() {
      return 6079 * SSN + 6089 *
        ((name == null) ? 1 : name.hashCode());
    }
    private final int SSN;
    private final Name name;
  }
}
```

**Fig. 8.** RJ source code implemented with Java annotations (top), and generated Java implementation (bottom).

types. We examined each class that overrides `equals()`, identified the intended key fields by examining the `equals()` and `hashCode()` implementations, and then manually rewrote the class into a relation type. We then compiled the resulting RJ version of *javacup* into Java, ran both the original version and this generated version on a grammar for Java 5 and ensured that the resulting generated parsers are identical.

In the course of this exercise, we needed to apply a number of refactorings that preserve the behavior of *javacup*, but that ease the introduction of relation types. The most significant of these refactorings consisted of:

- Key fields were made `private` and `final`. In a few cases, methods that initialize these fields were inlined into a calling constructor, or eliminated as dead code. In a few cases, some minor code restructuring was needed to eliminate “spurious mutability”.
- Nontrivial constructors were replaced by a combination of (i) simple constructors that only initialize `key` fields, and (ii) factory methods [15] that contain the remaining initialization code for, e.g., initializing mutable fields.
- In a few cases, the code contained implicit up-casts to type `Object` because tuples were stored into collections. In such cases, we parameterized uses of collection types with parameters of the appropriate relation type in order to avoid the up-cast.

| **class** | **actions performed** |
|---|---|
| `java_cup.production_part` | Converted into relation type: 1 key field, 0 non-key fields |
| `java_cup.action_part` | Converted into relation type: 1 key field, 0 non-key fields |
| `java_cup.symbol_part` | Converted into relation type: 1 key field, 0 non-key fields |
| `java_cup.parse_action` | Converted into singleton relation type (0 key fields, 0 non-key fields) |
| `java_cup.nonassoc_action` | Converted into singleton relation type (0 key fields, 0 non-key fields) |
| `java_cup.shift_action` | Converted into relation type: 1 key field and 0 non-key fields |
| `java_cup.reduce_action` | Converted into relation type: 1 key field, 0 non-key fields. Error in use of `equals()` previously discussed in Section 2.2. |
| `java_cup.production` | Converted into relation type: 1 key field, 14 non-key fields |
| `java_cup.action_production` | Converted into relation type: 0 key field, 2 non-key fields |
| `java_cup.lr_item_core` | This class and its subclass `lalr_item` were refactored into a combination of 2 classes without `equals()/hashCode()` and one relation type with 2 key fields and 0 non-key fields. |
| `java_cup.lalr_item` | See comments for `java_cup.lr_item_core`. |
| `java_cup.lalr_state` | Converted into relation type: 1 key field, 2 non-key fields |
| `java_cup.symbol_set` | Not converted because `equals()` refers to mutable state. Note: `equals()` is dead, so could simply be removed. |
| `java_cup.terminal_set` | Not converted because `equals()` refers to mutable state. Note: `equals()` is dead, so could simply be removed. Does not declare `hashCode()`, hence `equals()/hashCode()` contract violated. |
| `java_cup.lalr_item_set` | Not converted because `equals()` refers to mutable state. |

**Table 1.** Summary of results for *javacup* case study.

After performing these steps, we deleted the `equals()` and `hashCode()` methods, added `@Relation`, `@Key`, and `@Id` annotations, and ensured that the resulting code could be compiled and executed successfully.

Interestingly, we found that the resulting version of *javacup* produced a parser with significantly different source text than the parser produced by the original *javacup*, but that these parsers *behave identically* when applied to a number of inputs. Further investigation revealed that the output of the original version depended on iterators whose order relied on hash-codes of the elements stored in hash-tables. The `hashCode()` methods in our generated code differ from those in the original *javacup*, which resulted in different (but equivalent) generated parsers. As a further experiment, we rewrote *javacup* to use `LinkedHashMaps`[4] instead of `Hashtables`, and repeated the entire experiment. The resulting *javacup* produced a parser that was syntactically identical to the one that was produced by the original *javacup*.

Table 1 shows, for each class in *javacup* with an application-defined `equals()` method, the outcome of this exercise. As the table shows, of 15 classes with application-defined `equals()` methods, 12 could be converted into relation types, and most of them with relatively little effort. Classes `lr_item_core` and `lalr_item` required a somewhat nontrivial transformation. The `equals()` methods in these classes do not reflect general object identity, but only apply within the context of an `lalr_item_set`. We therefore removed these `equals()` methods and rewrote `lalr_item_set` to appropriately manipulate these objects using a newly created relation type `ItemKey`. Another item of note was a bug in a use of

[4] A `LinkedHashMap` is a hash-table for which the iteration order is determined by the order in which elements are inserted instead of depending on the hash-codes of the elements.

reduce_action.equals() that we previously discussed in Section 2.2. Classes symbol_set, terminal_set and lalr_item_set could not be converted because their equals() methods refer to mutable collections. Interestingly, the equals() methods in symbol_set and terminal_set are dead, and could be removed. Furthermore, class terminal_set violates the equals()/hashCode()contract by not overriding Object.hashCode().

### 4.3 Other Benchmarks

We repeated the methodology of the case study on a number of open-source Java applications.

The benchmarks *ant*, *hsqldb*, *jfreechart*[5], *lucene*, and *pmd* are open-source codes; we used the versions collected in the DaCapo benchmarks [9], version dacapo-2006-10. *Bcel* is the Apache Bytecode Engineering Library [4], version 5.2. *Shrike* is the com.ibm.wala.shrike project from the T. J. Watson Libraries for Analysis (WALA) [2], version 1.0. We use *shrike* regularly, and chose it for consideration based on prior knowledge that it would suit relation types. *Shrike* also has sophisticated, hand-rolled hash-consing, which is now generated automatically by the RJ compiler. The other benchmarks were chosen based on their having a reasonable number of equals() methods, and based on the availability of some drivers to test for correct behavior.

As described for *javacup* earlier, we transformed each code by hand where necessary to make fields private and final, remove unnecessary mutable state, and similar local changes. While we believe our transformations were correct (modulo erroneous existing behavior), we have no mechanical proof that the changes are semantics-preserving. We ran a number of dynamic tests for each code, including unit tests where available, the DaCapo drivers, and other drivers we created, and verified that for each test the RJ implementation behaves identically to the original implementation. This methodology gives us some confidence that the RJ versions are correct.

Table 2 summarizes our findings. The columns of the table show, for each benchmark, from left to right:

1. The number of equals() methods originally declared.
2. The number of equals() methods eliminated by conversion to relation types.
3. The percentage of eliminated equals() methods.
4. The total number of relation types introduced.
5. The number of relation types that correspond to value types (i.e., all fields are key fields).
6. The number of relation types that correspond to singletons.
7. The number of relation types that have non-key fields.
8. A summary of the bugs and issues that we encountered, as explained in the legend of the table.

[5] *jfreechart* has more than 200 equals() methods—a daunting number to study by hand. So we looked only at the first two packages in lexicographic order: org.jfree.chart.annotations and org.jfree.chart.axis.

| **benchmark** | **#`equals()`** | | | **#relation types** | | | | **bugs/issues** |
|---|---|---|---|---|---|---|---|---|
| | **orig.** | **removed** | **%** | **total** | **value** | **sing.** | **non-value** | |
| *ant* | 12 | 9 | 75.0 | 9 | 3 | 0 | 6 | H,M |
| *bcel* | 20 | 14 | 70.0 | 22 | 11 | 5 | 11 | F,H,M |
| *hsqldb* | 12 | 2 | 16.7 | 2 | 0 | 0 | 2 | E,F,M,S |
| *javacup* | 14 | 11 | 78.6 | 11 | 8 | 2 | 3 | H,M,T |
| *jfreechart* | 46 | 33 | 71.7 | 40 | 40 | 1 | 0 | H,M,S |
| *lucene* | 27 | 27 | 100.0 | 27 | 23 | 0 | 4 | E,M |
| *pmd* | 12 | 5 | 41.7 | 5 | 3 | 2 | 2 | E,F,M,N,S |
| *shrike* | 32 | 32 | 100.0 | 61 | 55 | 3 | 6 | — |

Explanation of codes used for Bugs/Issues:

| | |
|---|---|
| E | `equals()` method throws exception if passed unanticipated argument |
| F | fragile (e.g., `equals()` defined in terms of `toString()`) |
| H | `equals()`/`hashCode()` contract violated |
| M | `equals()`/`hashCode()` depends on mutable state |
| N | violates contract for `equals(null)` |
| S | symmetry requirement in `equals()` contract violated |
| T | inadvertent test of incomparable types |

**Table 2.** Summary of results.

As the table reflects, during this exercise we were able to convert the majority of candidate classes to relation types with little program modification. Most of these types actually represent values with no mutable state. As is well known, programming in a functional style without mutation eliminates many classes of bugs and generally leads to more robust, maintainable code. Record types fit well into such a programming style.

The last column of the table shows that we found violations of the contract and other problems in every code except *shrike*. This reinforces our claim that the current unenforced contract leads to fragile and error-prone code. Record types encourage more robust code by enforcing a stricter contract and removing the need for tedious, error-prone boiler plate code.

Of the types which we did not convert to relation types, most fall into one of two categories. The first category comprises types where the programmer had already manually applied hash-consing or other caching and pooling optimizations. In such cases, the program complexity exceeded our threshold for rewriting in these experiments. Record types would obviate the need for such manual storage optimizations, since the compiler can implement hash-consing and related representation transformations automatically.

The other category comprises types where identity depends on mutable state. Many instances of mutable identity appear spurious, and could be eliminated with a slightly more functional design. We also found a fairly large number of cases we call *piecemeal initialization*. In these cases, the program incrementally builds up an object's state piecemeal; for example, the program parses an XML document and mutates an object to represent the state as it parses. However, the object becomes logically immutable after initialization. To support such patterns, we plan to extend RJ with a facility to "freeze" a mutable object into an immutable relation tuple. Note that, in our current model, it is not possible to construct two tuples $t_1$ and $t_2$ such that the identity of $t_1$ is determined by $t_2$ and vice versa. The proposed extension would remedy this limitation.

# 5 Related Work

The C# language [22] supports both reference equality and value equality. As in Java, C# `Equals()` supports reference equality by default for reference types. The C# programmer can override `Equals` and `==` to support structural or value equivalence as desired, raising the same issues as when overriding `equals()` in Java. C# also supports built-in structural equality for C# *value types*, but restricts value types to structs and enumerations, with no inheritance.

A relation type's *key* annotation enforces an immutability constraint on the annotated field. Several other works have addressed language designs that incorporate immutability concepts.

Pechtchanski and Sarkar [23] propose a framework of immutability specification along three dimensions: lifetime, reachability, and context. Our *key* annotation provides a persistent and shallow immutability specification: a key field never mutates, but there is no constraint on mutability of state reached from a key field. In short, a key annotation behaves like "final" in Java. Of course, a key annotation conveys more information than simply immutability constraints; it also indicates what state constitutes the structural identity of a type.

Much other work defines analyses and expressive languages for immutability constraints (see [8, 11, 18, 26, 27]). Javari [27] provides an extension of Java that supports reference immutability, and enforces specifications expressing transitive immutability constraints. Javari also allows for the declaration of read-only methods that cannot modify the state of the receiver object, and read-only classes for which all instance fields are implicitly read-only. Our programming model could be combined with language extensions such as those in Javari, to support immutability constraints on non-key fields which do not contribute to the identity relation.

In our model, a relation type that has only key fields is a value type. Value types [5, 13, 22, 31] provide many benefits for the programmer. For example, they provide referential transparency: functions that manipulate only values have deterministic behavior. Since values are immutable, they eliminate aliasing issues and make code less error-prone. From an implementation viewpoint, value types simplify analyses that allow a number of aggressive compiler optimizations, such as unboxing [25], object inlining [14], memoization [21], data replication in distributed or cluster computing settings [13], and hash-consing [16].

Bacon's Kava language [5] is a variation on Java with a uniform object model that supports user-defined value types. Kava's notion of a value is that of an immutable object, with all fields pointing to other values. All value types are subclasses of a type `Value`, and they may inherit from other value types and from interfaces. In Kava, the programmer views values as objects down to the bitlevel, and the model provides a unification of objects and values in this sense. In our experience, Java programs commonly include "value-like" classes that define equality and hashcode based on an immutable subset of instance fields, but that also have some mutable state associated with them. Our relation types allow for such classes, and unify values and objects by providing a generalization of both as relations that map key fields to some possibly mutable state. Furthermore, due to this uniformity, we need not segregate type hierarchies for values and non-values, and a relation type may inherit from a value.

Our value-types are also more general than Titanium's [31] immutable classes, and C#'s value types [22], which are both implicitly final, and do not support "value-like" classes. Fortress's value objects [3] also do not support "value-like" classes, but they do allow fields of values to be set in order to allow piecemeal initialization.

Tuples have been added to object-oriented languages in various work (for example [19, 28, 20]). Our tuples differ in that they have keys, similar to primary keys in a row of a relational database, and relation types implicitly define a map from keys to non-keys. A relation type does not contain two tuples with equal keys but different non-key parts.

Some languages integrate object and relational data models to facilitate communication with a database (for example [20, 6]). The focus of our programming model is to view the heap itself as a relational database, and use concepts from databases such as primary keys to express identity. In future work, we plan to investigate the application of relation types to support data access integration.

Linda's [12] data model introduced an associative memory called a *tuplespace* as a model for sharing data in parallel programming. Relation types could perhaps be applied in this setting, providing a strong coupling between the object-oriented language and the distributed tuplespace. Relation types would also facilitate optimizations for data replication, as mentioned previously.

We presented our programming model using an adaptation of Featherweight Java (FJ), a functional language. Other extensions of FJ introduce mutation [7], using a heap that maps memory locations to mutable state. Our model provides a level of indirection in the heap, augmenting values with mutable state, thus providing a uniform framework for the functional and non-functional aspects of the language.

## 6 Summary and Future Work

We presented a programming model that provides a relational view of the heap. In this model, object identity is specified declaratively using a new language construct called relation types and programmers are relieved from the burden of having to write error-prone `equals()` and `hashCode()` methods. We formalized the model as an extension of Featherweight Java and implemented it as an extension of Java. Our experiments indicate that the majority of classes that override `equals()` can be refactored into relation types, and that most of the remainder are buggy or fragile.

We plan to extend the model with other features that borrow from database concepts (e.g., atomic sets [29]), and raise the level of abstraction for navigating the heap. Some of our ideas include a query language on top of relation types and features for pattern matching. We also plan to support delayed initialization of key fields, and to experiment with optimized representations for relation types.

**Acknowledgments.** We are grateful to David Bacon, Doug Lea, and Jan Vitek for their constructive feedback. Robert Fuhrer's help with the implementation was invaluable.

## References

1. http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html.
2. T. J. Watson Libraries for Analysis, December 2006. http://wala.sourceforge.net.
3. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele, G., and Tobin-Hochstadt, S. The Fortress language specification. http://research.sun.com/projects/plrg/fortress.pdf.
4. Apache Jakarta Project. BCEL, December 2006. http://jakarta.apache.org/bcel/.
5. Bacon, D. F. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency—Practice and Experience 15*, 3–5 (2003), 185–206.
6. Bierman, G., Meijer, E., and Schulte, W. The essence of data access in C$\omega$. In *European Conference on Object-Oriented Programming (ECOOP'05)* (2005).
7. Bierman, G., Parkinson, M. J., and Pitts, A. M. MJ: An imperative core calculus for Java and Java effects. Tech. Rep. 563, University of Cambridge, Computer Laboratory, April 2003.
8. Birka, A., and Ernst, M. D. A practical type system and language for reference immutability. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'04)* (2004), pp. 35–49.
9. Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (New York, NY, USA, Oct. 2006), ACM Press.
10. Bloch, J. *Effective Java, Programming Language Guide*. Addison-Wesley, 2001.
11. Boyland, J., Noble, J., and Retert, W. Capabilities for sharing: A generalisation of uniqueness and read-only. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)* (2001), pp. 2–27.
12. Carriero, N., and Gelernter, D. Linda in context. *Commun. ACM 32*, 4 (1989), 444–458.
13. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'05)* (2005), pp. 519–538.

14. Dolby, J., and Chien, A. An automatic object inlining optimization and its evaluation. *ACM SIGPLAN Notices 35*, 5 (2000), 345–357.

15. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

16. Goto, E. Monocopy and Associative Algorithms in an Extended Lisp. Tech. Rep. 74-03, Information Science Laboratory, University of Tokyo, 1974.

17. Igarashi, A., Pierce, B. C., and Wadler, P. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems 23*, 3 (2001), 396–450.

18. Kniesel, G., and Theisen, D. Jac – access right based encapsulation for Java. *Software: Practice and Experience 31*, 6 (2001), 555–576.

19. Krall, A., and Vitek, J. On extending Java. In *Joint Modular Languages Conference (JMLC'97)* (1997).

20. Meijer, E., and Shulte, W. Unifying tables, objects and documents. In *DB-COOL* (2003).

21. Michie, D. Memo functions and machine learning. *Nature*, 218, 19–22.

22. Microsoft. *C# Language Specification.* Microsoft Press, 2001.

23. Pechtchanski, I., and Sarkar, V. Immutability specification and its applications. In *Java Grande* (2002), pp. 202–211.

24. Petter, M. personal communication. October 2006.

25. Peyton-Jones, S., and Launchbury, J. Unboxed values as first class citizens. In *Functional Programming Languages and Computer Architecture: 5th ACM Conference* (1991).

26. Porat, S., Biberstein, M., Koved, L., and Mendelson, B. Automatic detection of immutable fields in Java. In *CASCON* (2000).

27. Tschantz, M. S., and Ernst, M. D. Javari: adding reference immutability to Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'05)* (2005), pp. 211–230.

28. van Reeuwijk, C., and Sips, H. J. Adding tuples to Java: a study in lightweight data structures. In *JGI'02* (2002).

29. Vaziri, M., Tip, F., and Dolby, J. Associating synchronization constraints with data in an object-oriented language. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2006), ACM Press, pp. 334–345.

30. Vaziri, M., Tip, F., Fink, S., and Dolby, J. Declarative object identity using relation types. Tech. rep., IBM Research, 2007. Forthcoming.

31. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., abd Paul Hilfinger, A. K., Graham, S., Gay, D., Colella, P., and Aiken, A. Titanium: A high-performance Java dialect. *Concurrency—Practice and Experience, Java Special Issue* (1998).