Balloon Types for Safe Parallelisation over Arbitrary Object Graphs

Marco Servetto, David J. Pearce, Lindsay Groves, and Alex Potanin
Victoria University of Wellington, New Zealand
{servetto|djp|lindsay|alex}@ecs.vuw.ac.nz

Abstract

Safe parallelisation of object-oriented programs requires static guarantees about the shape and/or intended usage of reachable objects. For example, transitively immutable objects lend themselves naturally to concurrent access. However, parallelising tasks which potentially mutate reachable objects is more challenging. One approach to avoiding race conditions is to ensure the reachable object graphs of any concurrently executing tasks are disjoint. Numerous systems have been developed which provide guarantees of this kind (e.g. ownership types, regions, balloons, etc).

In this paper, we build on the work of Almeida who developed *balloons* as a mechanism for providing strong encapsulation. Our approach is closely related to the recent work of Gordon *et al.* who developed a system for safe parallelisation based on isolation (i.e. balloons). Their system can safely parallelise code which mutates reachable objects. However, their system also relies on the use of destructive field reads, which go against the natural object-oriented style. Our system brings together the ideas of balloons, immutability and lent references to enable safe parallelisation over mutable data without requiring destructive field reads.

1. Introduction

Developing a system for parallelising object-oriented programs is a challenging problem. Naturally, one desires that such a system be *safe* — that is, concurrently executing tasks should be guaranteed to either: not access shared mutable state, or only access shared mutable state through appropriate protection mechanisms (e.g. locks, transactions, etc). The latter case is complicated significantly by the potential for *deadlock*. In this paper, we are concerned with the former case, and our approach employs type modifiers to statically guarantee disjointness of reachable object graphs. We leave open the question of whether parallelisation is done automatically or by hand and, instead, simply provide a mechanism which could plausibly underpin either approach.

One way to ensure concurrent tasks do not access shared mutable state is to require that they only access *immutable state*. A significant body of work exists on developing immutability systems for object-oriented languages (see e.g. [15, 30, 33, 34]). Such systems permit various modifiers on class declarations and/or reference types. For example, classes can be explicitly declared as *immutable*, thereby prohibiting their instances from being mutated at all. Likewise, object reference types can be annotated *readonly* to indicate that the referenced object cannot be mutated through *that* reference [5], or objects themselves can be marked as *immutable objects* prohibiting modification via *any* reference [33, 34].

Systems based on immutability are very restrictive as they do not permit concurrent tasks which modify mutable state. Alternative approaches, based on *ownership* (e.g. [8, 14, 20, 23, 26]) or *regions* [6, 17, 31], have been developed which, instead, allow concurrently executing tasks to modify mutable state provided that state is disjoint between tasks (i.e. not *shared* mutable state).

1.1 Ownership

Ownership types (e.g. [1, 8, 12, 14, 26, 34]), provide strong guarantees about when and where aliasing is permitted between objects, and have found use in areas such as: parallel and concurrent systems [7, 8], specification languages [4, 21], real-time systems [3, 25], and more. Such systems operate by annotating types of the underlying language. For example:

Ownership
```
1 public class Shape {
2   private @Owned List<Point> points;
3   ...
4 }
```

In an ownership system adopting the *ownership-as-dominators* protocol (e.g. [8, 12, 14, 26]), the `@Owned` qualifier above indicates that every `Shape` instance *owns* the `List<Point>` object referred to by `points` — so, the only references to the `List<Point>` object are from the `Shape` itself. The `@Owned` qualifier also applies transitively, meaning that objects owned by a `Point` in `points` are also owned by the `Shape`. Thus, we can be sure that the objects owned (either directly or indirectly) by two distinct `Shape` objects, s_1 and s_2, are disjoint. Unfortunately, we cannot say anything about objects which are not owned by s_1 and s_2 and, hence, their reachable object graphs may overlap.

Ownership systems have been used successfully in the past for safe parallelisation. Boyapati *et al.* developed Safe Java [8] which itself extended their earlier system, PRFJ [7]. Safe Java employs an ownership type system to statically eliminate dead-locks and data-races. Dead-locks are prevented by requiring that locks are obtained in the order implied by the ownership tree, whilst data-races are prevented by ensuring that all mutated data is thread-local. In the latter, a special mechanism is provided for safely transferring data between threads. From our perspective, the critical feature of Safe Java is that it allows functions to safely operate on distinct instances (e.g. of `Shape`) in parallel, *even if they mutate them*. However, at the same time, it has some signification limitations: firstly, since parallelism can be expressed only at the thread level, it is rather coarse-grained; secondly, the use of ownership in this manner mandates a strict tree structure on the object graph and, hence, code executing over *arbitrary* object graphs cannot be parallelised.

1.2 Balloons

Whilst ownership systems provide important guarantees needed for safe parallelisation, it is still not sufficient[1]. The key difficulty is that deep ownership systems provide no mechanism for determining whether two objects are *distinct*. Existing systems which employ *unique types* (e.g. [1, 9–11, 13, 24]) do, however, provide such guarantees. For example:

Uniqueness
```
1 class Tree {
2     private @Unique Tree left;
3     private @Unique Tree right;
4     int data;
5 }
```

The `@Unique` modifier provides strong guarantees: firstly, that the objects referred to by `left` and `right` are distinct (i.e. `left != right`); and, secondly, that no other heap reference exists to the object referred to by `left` (resp. `right`). These guarantees, taken together, are sufficient to allow code executing on the `left` tree to be safely executed in parallel with code executing on the `right` tree. Unfortunately, whilst unique types can enable safe parallelisation [29] they are, like immutability, also very restrictive. In particular, they enforce a strict tree structure on the object graph and, hence, do not support safe parallelisation across *arbitrary* object graphs.

The early work of Almeida [2] introduced the concept of *balloons* and, since then, others have used for solving a variety of problems [15, 16, 19, 29]. We can think of balloons as bringing uniqueness and ownership together to provide strong encapsulation guarantees which are, as it turns out, ideally suited for safe parallelisation. Having a balloon reference guarantees that you are the only holder of a balloon reference to this object (although other non-balloon references may exist). The object referred to by a balloon reference, called the *root*, uniquely identifies the balloon. Specifically, all mutable objects reachable from the root are "inside" the balloon, whilst all others (except the root) are "outside". The root, then, acts as the unique gateway to the balloon and references from objects "outside" to those "inside" are prohibited.

Whilst Almeida did not consider safe parallelisation as a use-case for his balloons, others have since explored this connection. In particular, recent work of Gordon *et al.* [16] is strongly related to our approach. Their system combines `isolated` types (a.k.a balloons) with `readable` (i.e. read-only) and `immutable` reference types. These types enable safe parallelism and essentially bring the concepts of deep ownership, unique types and immutability together. Their system is both elegant and powerful and can, for example, safely describe the construction of cyclic immutable object structures. Furthermore, `isolated` types can be safely converted to `immutable` or `writeable` and back again if required. One important aspect of this approach is that, in order to preserve the strong encapsulation property that `isolated` provides, their system employs *destructive field reads*. That is, in order to read an `isolated` value from an object field into a stack location, the destructive field read operation automatically assigns **null** to the field in order to prevent two references to the same object.

1.3 Contributions

In this paper we introduce Balloon Immutable Java (BI-JAVA), a language that supports aliasing and mutability restrictions that aid parallelisation. Like the work of Gordon *et al.* [16], this brings ideas from deep ownership, uniqueness and immutability together into a coherent framework. Like them, we support *immutable*, *balloon* and *read-only references*, as well as *type promotion* (referred to as *recovery* in [16]). Our system is comparable to their work, but provides additional flexibility by incorporating the well-known idea of *lent* [1] (or *borrowed* [19, 22]) references into the balloon system.

Importantly, BI-JAVA does not employ destructive field reads to maintain the strong encapsulation property which are extreme and, we argue, lead to a non-intuitive system; BI-JAVA leverages `@Lent` modifiers to allow flexible usage of balloons.

The aim of this paper is to present the main ideas behind BI-JAVA — we do not present any formalisation or proof of soundness. A preliminary formalisation and soundness proof for BI-JAVA, along with additional discussion, is available as a technical report [28].

Organisation. The remainder of this paper is organised as follows. In §2, we introduce the main concepts in BI-JAVA and, in particular, highlight our novel contributions. In §3, we discuss in more detail the meaning of the various type modifiers used in BI-JAVA, and clarify some of the more complex issues. In §4 we discuss how our work can be used for automatic parallelisation. In §5, we briefly discuss related work and, finally, in §6 we conclude.

2. Overview of BI-JAVA

We now introduce the main ideas behind BI-JAVA through a sequence of examples, and consider how they could be safely parallelised. Our goal is to present the novel contributions of BI-JAVA over related work (notably that of Gordon *et al.* [16]).

2.1 Running Example

In BI-JAVA, a `@Balloon` reference is guaranteed to be the only `@Balloon` reference to that object. Consider the following example which consists of a `Tree` class implementing a binary tree, and a `sum()` function which traverses a `Tree`, summing the `data` value at each node and caching the sum for each subtree to avoid recomputation:

BI-JAVA
```
1  class Tree {
2    int data;
3    Integer cache
4    @Balloon Tree left;
5    @Balloon Tree right;
6    ...
7  }
8
9  int sum(Tree t) {
10   if(t == null) { return 0; }
11   if(t.cache == null) {
12    t.cache = t.data + sum(t.left) + sum(t.right);
13   }
14   return t.cache;
15 }
```

Now, suppose we wish to parallelise this code by allowing subtrees to be summed in parallel. Observe that, since `sum()` mutates the object graph, an approach to safe parallelisation based on immutability is not applicable.

The calls to `sum(t.left)` and `sum(t.right)` can be safely executed in parallel, provided that the reachable object graphs for the `left` and `right` trees are disjoint. This is exactly the property enforced by the `@Balloon` modifier, so the above calls can be safely parallelised in our balloon system (and that of Gordon *et al.*). At this stage, it should be noted that we are considering a subset of Java where **static** (i.e. global) variables are prohibited — this helps ensure that a function can only access parts of the heap which are reachable through the given parameters (including via the implicit **this** parameter).

A subtle aspect of our example is that, by using e.g. `t.left` as a parameter to `sum()`, *we are placing a reference to a balloon on the stack*. This may appear to break the invariant stated at the beginning of §2.1 — namely, that a `@Balloon` reference is guaranteed to be the only `@Balloon` reference to that object. However, this is not

[1] Indeed [8] does not rely on ownership alone, but leverages other parallelisation constructs.

the case because a reference which happens to refer a balloon root is not the same as a `@Balloon` reference. More concretely, since the parameter for `sum()` is unqualified, the reference placed on the stack is not considered a `@Balloon` reference (rather, it is a general *mutable* reference). In contrast, if the method was declared as `sum(@Balloon Tree)`, then there would be a violation and our system would report an error.

2.2 Lent References

Reading `@Balloon` references from the heap onto the stack presents a challenge for any balloon-oriented system, such as that presented here or the work of Gordon *et al.* [16]. Let us consider a method for performing an in-place subtraction of one `Tree` from another, presented in the system of Gordon *et al.*:

```
1 void sub(isolated Tree t1, isolated Tree t2) {
2   if(t1 == null || t2 == null) { return; }
3   t1.data -= t2.data;
4   sub(t1.left,t2.left);
5   sub(t1.right,t2.right);
6 }
```

Recall the `isolated` qualifier is equivalent to our `@Balloon` qualifier. Now, suppose we want to execute the two recursive calls to `sub()` in parallel. The system of Gordon *et al.* enables this although, due to the requirement for a *destructive field read*, this behaves in an unexpected manner. That is, since e.g. parameter `t1` must be a `isolated` reference, there can be no other `isolated` references to that object at that point — *including in the parent node*. Therefore, on executing `sub(t1.left,t2.left)` an `isolated` reference is read from field `t1.left` (resp. `t2.left`) and that field is then implicitly assigned **null** to preserve the `isolated` invariant (i.e. that no two `isolated` references exist to the same object). In other words, traversing `isolated` structures also destroys them — meaning after calling `sub()` above, `left`==**null** and `right`==**null** for every node that was reachable from either `t1` or `t2`. To fix this, we can manually rebuild the structure as the recursion unwinds:

```
1 isolated Tree sub(isolated Tree t1,
2                   isolated Tree t2) {
3   if(t1 == null || t2 == null) { return t1; }
4   t1.data -= t2.data;
5   t1.left = sub(t1.left,t2.left);
6   t1.right = sub(t1.right,t2.right);
7   return t1;
8 }
```

This now correctly computes the subtraction of `t2` from `t1`. However, the side-effect that `t2` is destroyed remains (i.e. `left`==**null** and `right`==**null** for all nodes reachable from `t2`, including `t2`) and, to fix this issue properly, we must additionally rebuild `t2` as the stack unwinds (e.g. by returning a *pair* of tree nodes).

We believe that the use of destructive field reads does not yield an intuitive system. Unfortunately, this is the only mechanism provided for parallelising functions which mutate state (such as above) in the system of Gordon *et al*. To address this limitation, BI-JAVA adopts a more relaxed approach to avoid destructive field reads. In BI-JAVA, we can express the above example as follows:

BI-JAVA

```
1 void sub(@Lent Tree t1, @Lent Tree t2) {
2   if(t1 == null || t2 == null) { return; }
3   t1.data -= t2.data;
4   sub(t1.left,t2.left);
5   sub(t1.right,t2.right);
6 }
```

Here, the `@Lent` modifier is similar, in spirit, to the `@Balloon` modifier but with fewer guarantees. Specifically: multiple `@Lent` references may reference the same object, but this is guaranteed to be the root of a balloon; and, at most one `@Balloon` reference may exist which refers to the same object as one (or more) `@Lent` references. Intuitively, instead of providing the destructive field read as the mechanism for moving `@Balloon` references from the heap onto the stack, we provide a more permissive mechanism where `@Balloon` references can be lent temporarily. This is similar, in many ways, to the notion of *borrowed* [19, 22, 32] or *lent* [1] references.

In BI-JAVA, the above calls to `sub(t1.left,t2.left)` and `sub(t1.right,t2.right)` can be safely executed in parallel without using destructive field reads (i.e. without destroying them as happens in the system of Gordon *et al.*). This is the key contribution of our system over previous work. We will now describe in more detail the process for safely parallelising calls over `@Lent` references in BI-JAVA.

2.3 Safe Parallelisation over Lent References

Whilst `@Lent` references provide a more permissive solution than destructive field reads, they also provide fewer guarantees than `@Balloon` references. As such, care must be taken when parallelising code over them. In particular, we do not have a guarantee that two `@Lent` references are distinct, although we do know each points to the root of a balloon.

To be able to execute the calls `sub(t1.left,t2.left)` and `sub(t1.right,t2.right)` safely in parallel we must *dynamically* establish that their reachable objects are disjoint. Fortunately, however, we know both `t1` and `t2` are balloon roots (since this is guaranteed by `@Lent`). Therefore, we need only to check that they reference different objects (i.e. that `t1!=t2`) *to know that their entire reachable graphs are disjoint*. This is the key advantage of using a `@Lent` reference over a normal mutable (i.e. unqualified) reference, since the latter would require traversing the reachable object graphs to establish disjointness.

Reworking the `sub()` method to include this dynamic check yields the following:

BI-JAVA

```
1  void sub(@Lent Tree t1, @Lent Tree t2) {
2    if(t1 == null || t2 == null) { return; }
3    t1.data -= t2.data;
4    if(t1 == t2) {
5      // execute sequentially as before
6      sub(t1.left,t2.left);
7      sub(t1.right,t2.right);
8    } else {
9      // safe to execute in parallel here
10     sub(t1.left,t2.left) | sub(t1.right,t2.right);
11 } }
```

Here, we use the vertical bar (i.e. `|`) to indicate statements which can be executed in parallel. Note, however, that this is not part of the syntax for BI-JAVA and is only used here to highlight where it is safe to parallelise code. Furthermore, whilst we have hand-implemented the necessary dynamic check in this example, it could (in principle) be inserted automatically by a parallelisation system (see §4 for more discussion of this).

Finally, note that there are cases when one can avoid using a dynamic check altogether. In particular, when the types of the `@Lent` references are incompatible (e.g. `@Lent String` and `@Lent Integer`) then we know statically they reference distinct objects.

3. A Closer Look at BI-Java

BI-JAVA provides four type qualifiers for references: `@Balloon`, `@Lent`, `@Immutable` and `@Readonly`. In this section, we examine the semantics of these qualifiers in more detail by giving more precise, though still informal, descriptions. For a more formal discussion of their semantics, the reader is referred to our preliminary formalisation of BI-JAVA [28].

3.1 Understanding Balloons

In BI-JAVA, every object can be classified as *immutable* or *mutable*, where immutable objects cannot be modified. This classification extends to references as follows:

- An **immutable reference** (identified with the `@Immutable` qualifier) is guaranteed to refer to an immutable object. That is, an object which henceforth cannot be modified. Note that, since a mutable object can be *promoted* to an immutable object, it is possible the object was modified at some point in the past (this is discussed further below).
- A **mutable reference** is any unqualified reference. Such references adopt the conventional meaning of a Java reference, whereby the referents may be mutated freely. Thus, a normal (i.e. completely unqualified) Java program is a valid BI-JAVA program, where every reference is mutable — and, hence, where we have no additional information to aid safe parallelisation.
- A **balloon reference** (identified by the `@Balloon` qualifier) refers to a mutable object `o` called the *root*. All mutable objects reachable from `r` are *inside* the balloon, whilst all others are *outside*. The *balloon invariant* states that there exists no other balloon reference to `o` and, furthermore, that no object inside the balloon refers to a mutable object outside of it.

As one can see, this distinction between immutable and mutable references is an important building block for BI-JAVA because a balloon only contains mutable objects, never immutable objects. Indeed, a balloon consists of all mutable objects reachable from the balloon reference and, thus, the balloon reference uniquely identifies the balloon.

We now clarify several subtle points arising from the description of a balloon reference:

1. *No two balloon references (either in the heap or on the stack) refer to the same object*. However, given a balloon reference, there may exist other (non-balloon) references to the same object. Whilst this may seem counter-intuitive, it is an important product of the balloon invariant. In particular, we can only be certain that the reachable object graphs for two references are disjoint *if they are both balloon references*.
2. *Objects inside the balloon may refer to immutable objects outside of it*. Since we can always safely parallelise across immutable data, there is no reason to prohibit two balloons from sharing immutable objects. Gordon et al. [16] also take this approach.
3. *A balloon may itself contain other balloons*. Whilst a balloon root cannot be the root of another balloon, other objects inside a balloon can be balloon roots. This yields a strict dominance tree of nested balloons, similar to that imposed by ownership systems.

Finally, we detail the remaining reference kinds in BI-JAVA:

- A **lent reference** (identified by the `@Lent` qualifier) refers to a balloon root. Observe that lent references do not break the balloon invariant, since that is only applied to `@Balloon` references. Multiple lent references may refer to the same balloon root, which means we cannot tell whether two lent references refer to disjoint graphs. However, as discussed in §2.3, a single dynamic check is sufficient to determine disjointness.
- A **readonly reference** (identified by the `@Readonly` qualifier) may refer to *any* object (either immutable or mutable). However, as expected, field updates are not permitted through readonly references. Readonly references are particularly useful for traversing the contents of a balloon. Additionally, a readonly reference cannot be assigned to a field.

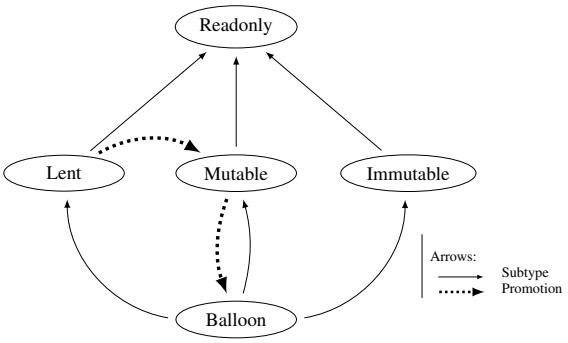

Figure 1. Object and reference kinds and possible relationships

We have now outlined the various qualifiers found in BI-JAVA. It remains, however, to consider what restrictions these place on field access and updates, and what subtyping relationships exist between them.

3.2 Type Promotion

As with the work of Gordon *et al.* [16], BI-JAVA offers the ability to *promote* a reference of one kind to another. This makes BI-JAVA a more flexible system. For example, we can create a cyclic data structure and obtain a balloon reference and then convert it to an immutable one. This offers a powerful solution to the problem of creating fully immutable (yet cyclic) object structures (see e.g. [18, 34]).

Figure 1 presents diagrammatically the possible promotions, as well as the subtyping relationships. The following example illustrates the promotion from a mutable to an immutable reference:

BI-JAVA
```
class Link {
  Link next; int data;
  Link(Link n, int d) {
    this.next = n; this.data = d;
} }

@Immutable Link createCyclicList() {
  Link l = new Link(null,0);
  l.next = l;
  return l; // mutable reference promoted to immutable
}
```

In this example, a mutable `Link` is created which is then implicitly promoted first to a balloon reference and then converted to an immutable reference. It it safe to do this because the context in which the `Link` is created is restricted so that all state reachable within the function is either immutable or a balloon. In this case, since `createCyclicList()` accepts no parameters, there is, in fact, no state reachable to the function (recall that we ignore global variables at this stage). In contrast, the following attempt at promotion fails because the context is not sufficiently restricted:

BI-JAVA
```
@Immutable Link createCyclicList(Link l) {
  l.next = l;
  return l; // type error — cannot promote
}
```

The issue above is that, since parameter `l` is an arbitrary mutable reference, we cannot ensure that no other references to it exist which, subsequent to this call, could be used to modify it and, hence, break the invariant implied by `@Immutable`.

An interesting variant which further illustrates the power of promotion is the following:

```
BI-JAVA
1 Link createCyclicList(Link l) {
2   l.next = l;
3   return l;
4 }
5 @Immutable Link createImmutableCyclicList() {
6     // safe to promote from mutable to immutable below
7     return createCyclicList(new Link(null,0));
8 }
```

Whilst this example appears very similar to the previous one, there are some subtle differences. In particular, at the point of the call to `createCyclicList()` we know exactly what state is being made available to that function and, hence, what it can possibly modify. Thus, we can safely promote the result of `createCyclicList()` even though it is not safe for that function itself to return an `@Immutable` object.

Unfortunately, due to a lack of space, we are unable to further examine the various promotions given in Figure 1. The reader is referred to our preliminary formalisation of BI-JAVA [28] which describes this issue in more depth.

3.3 Value Flow & Assignment

To ensure that the invariants specified by the various qualifiers of BI-JAVA are properly maintained, care must be taken to properly restrict the flow of values. For example, value flow must adhere to the subtyping relationships identified in Figure 1. This implies that e.g. an `@Immutable` reference may only be assigned to a field which is annotated as `@Immutable` or `@Readonly`.

In addition to restricting value flow, we must additionally restrict field assignment as well. In particular, a field assignment $\mathtt{e_0.f} = \mathtt{e_1}$ is permitted if either:

1. the receiver $\mathtt{e_0}$ is mutable, and the assigned value is mutable, `@Balloon` or `@Immutable`; or,
2. the receiver $\mathtt{e_0}$ is `@Lent` or `@Balloon`, and the assigned value is `@Balloon` or `@Immutable`.

The last rule ensures that we are not adding arbitrary references into a balloon which may refer to things outside of our control (and, hence, which we cannot logically place into the balloon).

4. Automatic Parallelisation

With our type system, it is possible to automatically add parallelisation as a compiler optimisation. Informally, this requires that observable behaviour is not changed; to use the terminology of [6] we can obtain a parallel performance model without altering the well known sequential semantics model.

To illustrate this idea, we extend the language with fork-join blocks. These are only added during compilation, and thus are completely transparent from the point of view of the programmer. Formally this is a semantics preserving program transformation.

A fork-join block contains sequences of normal local variable declarations. We annotate each declaration with a list of dependencies of the form x **if** L, where x is a variable initialised earlier in the same fork-join block and L is a logic expression composed from the constants **true** and **false**, the logical *or* operation, and the identity check over variables. A dependency x **if** L means that the variable being initialised depends on the value of x if condition L holds.

Any expression with an empty dependency list can be executed immediately. An expression with a non empty dependency declaration can be executed either after the dependency is verified to not hold, or when the variable it depends on has been computed; that is the flow of parallel execution is regulated by the logic expressions inside the fork-join blocks. For example,

```
fork-join
1 fork-join {
2   int x=1+2; []
3   int y=1+2; []
4   int z=x*y; [x if true, y if true] }
```

Here, the values assigned to `x` and `y` do not depend on any other variables and can be computed in parallel, while `z` depends on `x` and `y`, so before computing `z` we have to wait for them to be computed.

Assuming a class `C` with methods `C m()`, `C k()` and **int** `h(C)`, we can show a more complex example:

```
fork-join
1 C mm(@Lent D p1,@Lent D p2){
2   fork-join {
3     C x=p1.m(); []
4     C y=p2.k(); [x if p1==p2]
5   }
6   return x.h(y); }
```

Again, the value assigned to `x` does not depend on any other variables and can be computed immediately, while `y` depends on `x` if `p1` and `p2` point to the same object, so we have to check if `p1==p2` to determine whether initialisation of `y` has to await the initialisation of `x`:

- If `p1` and `p2` refer to different objects, we can start executing that expression in parallel,
- otherwise, we have to wait for the completion of the `x` expression.

As a further example, consider:

```
fork-join
1 C mm(@Lent D p1,@Lent D p2,@Lent D p3){
2   fork-join {
3     C x=p1.m();    []
4     C y=p2.k();    [x if p1==p2]
5     C z=p3.h(p1); [x if true, y if p3==p2 or p1==p2]
6   }
7   return z; }
```

Here, computation of `y` depends on the computation of `x` if `p1==p2`, while the computation of `z` always depends on `x` and depends on `y` if `p3==p2` or `p1==p2`. Thus, when `p1`, `p2` and `p3` are distinct objects, the computation of `x` and `y` can start in parallel, and as soon as the computation of `x` ends, the computation over `z` can start.

Whilst we have explicitly declared dependencies between variables in our examples above, the intention is that these be automatically determined by the compiler. Whilst the problem of automatically determining such dependencies has been well-studied in the literature, the BI-JAVA system presents some interesting twists. For example, immutable or balloon references cannot interfere with any other, so we only need to consider the other kinds of references. Similarly, if two expressions only use `@Readonly` variables, then there is no dependency between them. Finally, as discussed in §2.3, we can often avoid the dynamic test for variables with incompatible types (since they will never be equal). Our technical report [28] explores how one can automatically compute dependency relations between expressions, which we would like to investigate further.

Overall, our aim is develop a system where programmers specify the necessary `@Balloon` qualifiers, but where the compiler then infers appropriate fork-join blocks. Of course, the challenge then is to find the right balance when deciding which fork-join blocks to actually execute concurrently (i.e. between parallelism which is too fine-grained versus that which is too coarse-grained). Investigating this issue is an important direction for future work.

5. Related Work

BI-JAVA builds on the work by Gordon et al. [16] where our mutable reference is equivalent to their `writable`; `@Immutable` is equivalent to their `immutable`; `@Readonly` is equivalent to their `readable`. Additionally, we provide the extra guarantees that a readonly reference is not going to be stored; `@Balloon` is equivalent

to `isolated` and we introduce the `@Lent` modifiers that removes the need for destructive reads. Finally, Gordon et al. have the concept of recovering `isolation` while we have the concept of promotion.

Another very closely related work is Deterministic Parallel Java (DPJ) [6]. While our motivation is present in other recent works [27] few others have chosen to build on top a *full encapsulation* mechanism such as balloons as opposed to a less restrictive and more flexible ones such as *ownership types* [23]. DPJ [6] requires a programmer to think about the parallelisation explicitly and to manually insert the parallelisation constructs at the appropriate points in the program. DPJ uses an extended type checker with regions and effects, and requires explicit manipulation of memory regions. DPJ provides very strong guarantees, but their type system requires the programmer to manage many details. As such, we argue the required type annotations for DPJ are more complex than those for BI-JAVA. For example, the beginner's tutorial for DPJ proposes the following:

```
1 public static <region RN,RA | RN:* # RA:*>
2   Node<RN>  makeTree(DPJArray<Body,RA> arr,
3     double leftBound, double rightBound)
4       reads RA:* writes RN:* {...}
```

Srinivasan and Mycroft [29] define a language where actors can communicate using object trees. Their approach is similar to ours, but our `@Balloon`s provide more flexibility with respect to object trees, indeed Mycroft agrees that using balloons to relax the constraint of tree shaped messages would be a very interesting research direction. We believe our approach could inspire similar relaxation in other approaches allowing safe parallelism.

Finally, recent work by Naden et al. [22] extends Plaid with a similar set of access permissions to ours so that they can support borrowing of unique, shared, and immutable objects. Their work has a much wider scope with respect to BI-JAVA; still one of their motivations is an ability to detect noninterference of concurrency, which we believe BI-JAVA achieves.

6. Conclusion

In this paper we have demonstrated how balloons and immutable objects can be utilised to guarantee when expressions can be executed in parallel. Our system employs a combination of static and efficient dynamic checks (e.g. a simple pointer equality). This is a step towards removing the burden of "thinking in parallel" from day to day programmers who want to take advantage of modern multicore architectures.

In the future we plan to extend our work to support a richer subset of a Java-like language; we are also developing a prototype implementation of our approach and are planning to perform a user study evaluating its ease of use in day-to-day programming.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, pages 311–330. ACM Press, Nov. 2002.

[2] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP*. Springer-Verlag, June 1997.

[3] C. Andreae, J. Noble, Y. Coady, C. Gibbs, J. Vitek, and T. Zhao. Stars: Scoped types and aspects for real-time systems. In *ECOOP*, 2006.

[4] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6): 27–56, 2004.

[5] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, 2004.

[6] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA*, pages 97–116. ACM Press, 2009.

[7] C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, pages 56–69, 2001.

[8] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.

[9] J. Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 31(6):533–553, May 2001.

[10] J. Boyland. Connecting effects and uniqueness with adoption. In D. Clarke, editor, *IWACO*, number 2003-030 in UU-CS, pages 42 – 57. Utrecht University, July 2003.

[11] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP'01*, pages 2–27, 2001.

[12] D. Clarke and S. Drossopoulou. Ownership, Encapsulation, and the Disjointness of Type and Effect. In *OOPSLA*, pages 292–310, 2002.

[13] D. Clarke and T. Wrigstad. External Uniqueness is Unique Enough. In *ECOOP*, pages 176–200, Darmstadt, Germany, 2003.

[14] D. Clarke, J. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, pages 48–64, 1998.

[15] D. Clarke, J. Noble, and T. Wrigstad, editors. *Aliasing in Object-oriented Programming*, volume In Print of *LNCS*. Springer, 2012.

[16] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *OOPSLA*, pages 21–40. ACM, 2012.

[17] D. Grossman, J. G. Morrisett, T. Jim, M. W. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *PLDI*, pages 282–293, NY, USA, 2002. ACM Press.

[18] C. Haack and E. Poll. Type-based object immutability with flexible initialization. In *ECOOP*, pages 520–545, 2009.

[19] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP'10*, pages 354–378. Springer, 2010.

[20] Y. Lu and J. Potter. Ownership and accessibility. In *ECOOP*. Springer-Verlag, 2006.

[21] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. 2002.

[22] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. In *POPL*, pages 557–570. ACM Press, 2012.

[23] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *ECOOP*, volume 1445 of *LNCS*, pages 158–185. Springer-Verlag, July 1998.

[24] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness and immutability. In *TOOLS Europe 2008*, 2008.

[25] A. Potanin, J. Noble, T. Zhao, and J. Vitek. A high integrity profile for memory safe programming in real-time Java. In *Proc JTRES*, 2005.

[26] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership. In *OOPSLA*, Portland, OR, USA, 2006. ACM Press.

[27] C. Reichenbach, Y. Smaragdakis, and N. Immerman. PQL: A purely-declarative Java extension for parallel programming. In *ECOOP*, 2012.

[28] M. Servetto and A. Potanin. Balloon immutable java. Technical Report 12-18, ECS, VUW, 2012. `http://ecs.victoria.ac.nz/Main/TechnicalReportSeries`.

[29] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP'08*, pages 104–128, 2008.

[30] M. S. Tschantz and M. D. Ernst. Javari: adding reference immutability to Java. In *OOPSLA'05*, pages 211–230, 2005.

[31] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson. Inferring method effect summaries for nested heap regions. In *ASE*, pages 421–432, 2009.

[32] E. Westbrook, J. Zhao, Z. Budimlimić, and V. Sarkar. Practical permissions for race-free parallelism. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[33] Y. Zibin, A. Potanin, S. Artzi, A. Kiezun, and M. D. Ernst. Object and reference immutability using Java generics. In *FSE*, 2007.

[34] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and immutability in generic java. In *OOPSLA*, pages 598–617, 2010.