Automatic Parallelisation in OO Languages with Balloons and Immutable Objects

Marco Servetto  Alex Potanin
Victoria University of Wellington, New Zealand
{servetto,alex}@ecs.vuw.ac.nz

Abstract
Automatically detecting when it is safe to execute program expressions in parallel is hard. In this paper we take a step towards this ultimate goal by extending the original Almeida’s balloons system with immutability and we utilise this information to figure out when the execution order of some expressions does not influence the result. Our type system for an imperative object-oriented language takes away the burden of parallelisation from the programmer and does not require the use of complex program analysis tools.

Categories and Subject Descriptors D.3 Programming Languages [D.3.3 Language Constructs and Features]

General Terms Languages

Keywords Full encapsulation, immutability, auto parallelisation

1. Introduction
In this paper we present Balloon Immutable Java (BI-JAVA), a language that supports both immutability and aliasing restrictions that can aid parallelisation. Traditionally, full encapsulation was considered too strict to be expressive enough for general purpose programming [3]. Further work chose to extend it with either more flexible encapsulation guarantees, such as ownership types [13], or some form of immutability, such as readonly references in Universes [11]. To be useful for parallelisation, we are concerned with reachability of object references and thus we consider more flexible encapsulation schemes to be not as useful for our guarantees. BI-JAVA chooses to keep a full encapsulation scheme and couple it with immutability support as the basis of our approach. We start with a tour of BI-JAVA concepts.

1.1 Immutable Objects
There is a significant recent body of work on immutable objects in Java-like languages [6][14]. An immutable object can never be mutated via any reference to it. Immutability can be either deep or shallow property and most approaches distinguish between mutable and immutable objects. Additionally, references to objects can be treated as readonly to prevent changes via this particular reference to an object. A typical approach is to introduce types that can distinguish readonly references, references to mutable objects, and references to immutable objects. Commonly a readonly reference type is treated as a supertype of both reference to a mutable and an immutable object [18][19].

BI-JAVA offers transitively immutable objects, as typically found in functional programming languages. They provide capability to share and offer closed semantics. They are referentially transparent: it is irrelevant if two references point to the same immutable object or to different but equivalent immutable object graphs. Indeed a language offering immutable values can provide useful memory optimisations under the hood.

One problem with introducing immutable objects is safe initialisation of immutable (and potentially cyclic [19]) data structures. Our solution is to introduce fresh objects — these objects dominate their reachable object graph and only allow heap references to such objects to come from inside their reachable object graph. Moreover, references from inside a fresh reachable object graph to the outside are only allowed if they are to immutable objects. Thanks to these strong limitations, fresh objects can safely be converted both to immutable objects and to mutable objects.

So far, we have introduced three kinds of objects: fresh, mutable, immutable; and four kinds of references: fresh, mutable, immutable, readonly.

The first main contribution of this paper is a type system that can detect if a mutable object can be converted to fresh. We call this mechanism permanent type promotion.

1.2 Balloon Objects
Almeida [1] introduced a concept of balloon objects. The original balloons were class based and maintained a unique reference to the “balloon object” (thus no objects, including those “inside” the balloon, could refer to it). We propose a fourth and final kind of object in our language: balloon that extends Almeida’s concept by allowing references from inside the balloon to the “balloon object”.

The balloon reference is somewhat similar to the concept of a unique reference in a sense that it is the only reference to a balloon object from outside of the balloon. Thus, having a balloon reference guarantees that you are the only holder of a balloon reference to this balloon object. In our language balloons are object based and can be nested inside other balloons.

Just like readonly reference can point at both mutable and immutable objects, we introduce an external reference type that can point both at mutable and balloon objects. An external reference to an object can be passed to methods and stored in stack variables but can never be put inside a balloon. We generalise both external and readonly reference types as external readonly which are references to objects that neither can be put inside balloons nor mutated.

Figure 1 shows all four kinds of objects (balloon, mutable, immutable, and fresh) and seven kinds of reference types (balloon,

[Copyright notice will appear here once ‘preprint’ option is removed.]
mutable, immutable, fresh, as well as readonly, external, and external readonly) present in BI-JAVA.

1.3 Four Kinds of Heaps and Balloon Invariant

Almeida distinguishes two kinds of heaps: static (or normal) heap prohibits any references to objects inside a balloon that bypass the entry object; dynamic heap on the other hand allows references to objects inside balloons stored in the static heap, however no references from static heap to dynamic heap are allowed. We extend the Almeida’s heap concepts by introducing four kinds of heaps:

1. immutable heap contains all immutable objects;
2. fresh heap contains all fresh objects and their reachable object graphs, but not the objects in the immutable heap;
3. shared heap (similar to Almeida’s static heap) can contain mutable objects that are referred to using mutable references from the stack and their reachable mutable object graphs. Notably such graphs can refer to balloon objects (see Section 4.6);
4. temporary heap (similar to Almeida’s dynamic heap) contains all the other objects; that is objects referred to using balloon, external, readonly or external readonly references from the stack and their reachable object graphs (as long as such objects are not already captured by the above three heap definitions).

During the program execution objects inside the fresh heap can migrate to any other kind of heap; objects inside the shared heap can migrate to the temporary heap; while objects inside the temporary heap can never migrate to any other heap.

Now we can clearly re-state the balloon invariant by Almeida [11] using the object types and heap kinds from BI-JAVA:

\[
I_1 \text{ at most one object can refer to a balloon object using a field of balloon type. Other objects can refer to a balloon if the field is of mutable type (or a supertype thereof);} \\
I_2 \text{ this balloon reference (if it exists) is external to the reachable object graph from the balloon object itself;} \\
I_3 \text{ for any mutable object on the stack, its reachable mutable object graph can refer to the internal objects of at most a single balloon;} \\
I_4 \text{ balloons referred to from the shared heap or directly referred to from the stack refer to disjoint object graphs, and all direct sub-balloons contained in a balloon refer to disjoint object graphs as well.}
\]

Note how if some balloon, external, readonly or external readonly references refer to disjoint object graphs, any operation over these “disjoint” references will not establish any connection between these disjoint object graphs. That is, any mutable object created during such operation is: (1) injected into the reachable object graph of some balloon or external reference; (2) is in the reachable object graph of the result of the operation; or (3) can be safely garbage collected at the end of the operation itself.

Our second main contributions is a type system that is able to recognise expressions returning a value not injected in any balloon or external reference. Inside such expressions, any non fresh reference can be seen as readonly and thus stored inside conventional data structures. We call this mechanism temporary type promotion.

1.4 Field Update Operation

The limitations that are imposed by immutable (or readonly) and balloons (or external) references find its origins in the typical field update operation:

\[ e_0.f = e_1 \]

There are two parts to a field update: on the left, the field of \( e_0 \) is modified, and on the right, the result of \( e_1 \) is being stored.

Field update operation is prohibited if

- receiver \((e_0)\) is immutable, readonly or external readonly.
- assigned value \((e_1)\) is balloon, external or external readonly.

Moreover, if the receiver \((e_0)\) is balloon or external, the assigned value has to be fresh or immutable.

The original proposal for Balloons uses a copy assignment. Our system clearly subsumes that proposal, since a (deep-)clone method would produce a fresh result.

Our third main contribution is a type system that is able to recognise expressions that are executed “as inside” of a balloon: inside such expression an external reference to the inside of this balloon can be seen as mutable, thus allowing free field assignment. We call this mechanism balloon local type promotion.

Figure 1 shows our object and reference types together with subtype relationship, conversion from fresh and the three kinds of promotion.

1.5 Why Balloons Are Useful for Parallelisation

Balloons allow modular reasoning since they offer closed semantics: if two balloon references are different, the corresponding balloons are disjoint. Firstly, this helps the programmer by simplifying reasoning. Secondly, it is possible to use this information for automatic parallelisation. Consider the following example:

\[
\text{int } h(b \ C \ x1, \ D \ x3, \ F \ x4) \{ \\
\text{int } x = x3.m(x1, \ x4); \\
\text{int } y = new H() \cdot k(x2, \ x3); \\
\text{return } x+y; \}
\]

In this simple setting, there are two expressions that have to be executed before being able to compute a result. Thanks to the property of balloon (and fresh and immutable) references, no matter what computation will be executed by methods \( g \cdot n \) and \( h \cdot k \): if the reference \( x1 \) is not equal to the reference \( x2 \) the two expressions can be computed in parallel, preserving the semantics of the program. That is, in many cases a simple dynamic test on pointers \((x1 != x2)\) is enough to verify the correctness of this optimisation, opening the door for a completely safe speculative parallelism.

Our fourth main contribution is a compilation process able to infer simple logic expressions over pointer equalities and to dynamically enable fork-join parallelisation according to the results of these logic expressions, preserving the original semantics.

Outline. The rest of the paper is organised as follows: Section 2 and Section 3 presents the BI-JAVA language in detail, with examples showing the flexibility of our typing discipline. Section 4 provides semantics and type system rules for the BI-JAVA language without any parallelisation constructs. Section 5 adds a fork-join construct and describes a compilation process able to inject such construct in a sequential code preserving the semantics. Finally, Section 6 discusses related work and Section 7 concludes.
2. A Tour of the BI-JAVA Language

In this section we use a series of examples written in BI-JAVA to give a feel for all major aspects of our language.

2.1 Mutable and Readonly References

Mutable references are the closest kind of reference in our language to the default reference in Java. Mutable references provide capability to share and to write an object. As with usual Java, because of aliasing [3], any change to any object can potentially have an effect on the object referred to by a mutable reference.

A mutable reference can be safely cast to a readonly reference but once treated as readonly, it can never be cast back to mutable again. This guarantees that the object (and its reachable object graph) will not be modified via such a readonly reference. However, this does not prevent observational exposure [4] as readonly references can be stored in any (type compatible) field that has a external or readonly type and thus any changes to the object can be observed by the referer. The following example shows how a client can observe a change to a readonly field myC because of the alias created inside the action method:

```java
class C { int i;... }
class D {
    C myC=new C();
    int inspect() { return this.myC.i; }
    void action(e C c) { int x=c.i+2;
        this.myC=c;
        //c.i=0;//Wrong since c is readonly
    }
    void client(m C c, m D d) { c.i=1;
        d.action(c);
        //here c.i=-1 and d.inspect()=-1
        c.i=2;
        //here d.inspect()=-8/+ }
    }
}
```

2.2 External References

A mutable reference can be safely cast to an external reference but once treated as external, it can never be cast back to mutable again. This guarantees that the object (and its reachable object graph) will not be stored anywhere on the heap using this reference. In other words, no aliases can be created to an object using an external reference to it. However, an external reference does not prevent the object from mutating. If one were to guarantee that the object graph obtained by this external reference is fully disjoint from the current object graph, then none of these mutations can be observed in the current object graph. In fact, BI-JAVA provides such a guarantee in some circumstances. The following example shows how action can modify c but cannot store a reference to it, thus not affecting the state of d in client method:

```java
void action(e C c) { int x=c.i+2;
    //this.myC=c;//Wrong since c is external
    c.i=0;
}
void client(m C c, m D d) { c.i=1;
    d.action(c);
    //here c.i=-0 and d.inspect()=-8
    c.i=2;
    //here d.inspect()=-8/+ }
```

2.3 External Readonly References

As this is the common supertype of all the reference types in BI-JAVA, it only allows reading the data from the object but not storing it in fields nor modifying it as the following example shows:

```java
void action(e C c) { int x=c.i+2;
    //this.myC=c;//Wrong since c is external
    //c.i=0;//Wrong since c is readonly
}
void client(m C c, m D d) { c.i=1;
    d.action(c);
    //here c.i=-1 and d.inspect()=-8
    c.i=2;
    //here d.inspect()=-8/+ }
```

2.4 Loosely Closed Expressions

A closed expression is an expression without free variables. A fundamental concept in BI-JAVA is loosely closed expression — which is any expression where no free variables are mutable or readonly references. There are seven kinds of references in BI-JAVA. The loosely closed expressions do not contain mutable or readonly references to begin with and our type system prevents the introduction of aliases to fresh, balloon, external, and externalreadonly references. We refer to all the reference kinds that are not mutable or readonly as loosely closed reference types.

2.5 Permanent Type Promotion: Mutable into Fresh

A mutable reference produced by a loosely closed expression can be promoted to a fresh one. Indeed, any expression taking as input balloon, external or external readonly references will not be able to produce a mutable reference referring to objects reachable by the original references inside its reachable object graph.

All objects inside the produced mutable reference reachable object graph are either: (1) created inside such expression; (2) immutable; or (3) inside the fresh heap at the start of the expression execution.

From the property of loosely closed expressions it is clear that objects existing before such expression started executing cannot refer to those newly created object. Hence, the produced mutable reference points to a fresh object.

Consequently, since a readonly reference refers to either a mutable object or an immutable one, and since fresh can be converted into immutable, readonly reference produced by a loosely closed expression can be promoted into an immutable one.

The following example shows how the mutable result of method read1 is promoted to fresh in usage1. This is possible since read1 only requires an external reference to a Scanner object. Symmetrically method usage2 converts a readonly reference to an immutable one.

```java
class BarReader{
    BarRead1(e Scanner s) {...}
    BarUsage1(m Scanner s)...m{
        return this.read1(s); /*promotion happens here*/ }
    BarRead2(m Scanner s)...m{
        return this.read2(s); /*promotion happens here*/ }
}
```

FRESH REFERENCES The promotion mechanism we have explained is the only way to be able to use a fresh object in a program. For simplicity of formalisation our language prevents field access and field update over fresh references. Fresh references can be converted to any of the other kinds once and only once. Moreover, classes cannot declare fields of fresh type. Consider the following example:

```java
class BarReader{
    BarRead1(e Scanner s)...e {...}
    BarUsage1(m Scanner s)...m{
        return this.read1(s); /*promotion happens here*/ }
    BarRead2(m Scanner s)...m{
        return this.read2(s); /*promotion happens here*/ }
}
```

2 Conversions and permanent type promotions combined allow transparent field access and easy field update over fresh references.
void m(Foo f1, Foo f2){
    //NB! We are statically sure that f1!=f2
    Foo f3=f1; //f1 is not accessible anymore
    Foo f4=f3; //ok: conversion fresh ->immutable
    Foo f5=f3; //ok: f3 is now Immutable
    Foo f6=f2; //conversion fresh ->balloon
}

Thanks to our conversion mechanism, fresh objects can be used to bring into existence immutable and balloon objects. For simplicity, all the constructors return mutable objects, and the promotion system and the conversions are used to create the other kinds of objects.

2.6 Creation and Manipulation of Objects in BI-JAVA

Constructors always create mutable objects. In our simplified setting we consider only conventional constructors a la Featherweight Java. As usual, every field has to be initialised with a value of the corresponding type, but we have three exceptions: (1) balloon type fields have to be initialised with a fresh value; (2) external type fields have to be initialised with a mutable value; and (3) external readonly type fields have to be initialised with a readonly value.

Field update follows the same convention (e.g. balloon fields are updated with fresh values etc). Method calls are allowed over all kinds of receivers if the method has the correct modifier. Field access has to be treated more carefully and is presented in Rule (F-ACCESS-T) in Section 3.

The following examples shows how fresh conversions, permanent promotions, constructors and field access and update cooperates:

class Bar{//Shows how immutable works
    Foo f;
    Bar(Foo f) { this.f=f; }
}

void m(Foo f1, Foo f2){
    Bar b1=new Bar(f1);
    f1=f2; //variable f1 Immutable, not final
    Bar b2=new Bar(f2); //field f Immutable, not final
    Bar b2=new Bar(f1); //b2.f=f1; //Wrong, b2 Immutable
}

Immutable References Method Bar.m shows that some instances of Foo can be mutable while others can be immutable, and a permanent type promotion inside a fresh conversion can be used to initialize a new immutable object b2.

class Beer{//Shows how balloon works
    Foo f;
}

void m(Beer b1, Foo f1){
    b1.f=f1; //ok conversion fresh->balloon
    Beer b2=b1;//implicit cast mutable->external
    Foo f2=f1; //mutable b1 exposes f as balloon
    Foo f2=f2; //external b2 exposes f as external
}

Balloon References Operation b1.f=f1 converts the fresh object referred to by f1 into a balloon, and thus the object and its whole reachable object graph moves from the fresh heap to the one of the current Beer instance.

2.7 Temporary Type Promotion

All external readonly references can refer to any object except for fresh, the main use of such reference is to make queries over such wide variety of objects in a uniform way. In order to perform such queries over many external readonly references, the creation of temporary data structures holding such references could be required.

Our type system allows any loosely closed expression to consider some of its open variables as readonly, if the result of this expression is loosely closed. This is safe because values containing such promoted references cannot be injected into pre existing objects.

For example, in the following code a queryMethod requires two readonly parameters, but the method usage only has a balloon and an external readonly parameter. Temporary type promotion can be applied, and the result of the method is external readonly.

class BarQuery{
    Bar queryMethod(Bar b1, Bar b2){
        return this.queryMethod(b1,b2); //promotion here
    }
}

2.8 Balloon Local Type Promotion

All external values can be modified. However, simply allowing field update will break the balloon invariant. Some operations are clearly valid e.g. removing elements from an external list should be allowed. On the other hand, only fresh or immutable elements could be added. Such operations cannot break the balloon invariant.

In general, in order to modify an external variable, we need to consider it in the scope of its own balloon. Which is, in any loosely closed expression, one external variable can be considered mutable iff all the other balloon variables are considered external and the result is loosely closed. Indeed in BI-JAVA a top level expression is executed as outside of all the balloons, but subexpression where Balloon local type promotion is applied over variable x are executed “as inside of” the specific balloon that contains the object referred to by x.

This allows the programmer to modify in a very natural way the content of such balloon, without breaking the balloon invariant. Thanks to the property of locally closed expressions, nothing from the outside can be injected inside that balloon. The result of this expression has to be loosely closed too, in order to avoid elements from the inside of the balloon to be injected in other balloons.

This typing discipline subsumes allowing fresh and immutable objects to be used for field update over external receivers. Indeed fresh and immutable are loosely closed reference types. Note that this mechanism applies also to the balloon object itself: such object is seen as mutable inside the balloon reachable object graph.

The following example shows how method mutatorMethod requires a mutable b, however the usage method only provides an external b. Thanks to balloon local type promotion, the method mutatorMethod can be called and the result is seen as external instead of mutable.

class BarMutator{
    Bar mutatorMethod(Bar b){
        return this.mutatorMethod(b); //promotion here
    }
}

3. Real World Example

We can now exploit BI-JAVA in a more complete real world example: a two dimensional mesh editor, where meshes are composed of triangles and can be edited using a GUI:

class Point{
    float x, y;
    Point(float x, float y){this.x=x;this.y=y;}
    boolean isNear(Point p, float d){
        ...}
}

class Triangle{
    Point p1, p2, p3;
    Triangle(Point p1, Point p2, Point p3){
        this.p1=p1;this.p2=p2;this.p3=p3;}
    boolean isOverlapping(Triangle t){
        ...}
}

As you can see, methods isNear and isOverlapping can be annotated as external readonly. Indeed any method that just takes parameters to compute a result can always be annotated in this way. If
BI-JAVA was to be extensively used, we would expect to see many such methods in real applications. Depending on the specific computation involved, the returned value could be either a primitive value, a `readPoint` or an `addTriangle1(ReadonlyT)` method instead.

3.1 Mesh, and Two Ways to Add a Triangle
We assume vectors for mutable points (VectorMutableP) and triangles (VectorMutableT):

```java
class Mesh {
    VectorMutableP points; VectorMutableT triangles;
    Mesh(VectorMutableP p, VectorMutableT t) {
        this.points=p; this.triangles=t;
    }
    Point seekPt1(float tolerance, Point p1){
        for(Point p2: this.points)
            if (p2.isNear(p1, tolerance)) return p2;
        this.points.add(p1);
        return p1;
    }
    void addTriangle1(float tolerance, Point p1, Point p2, Point p3){
        this.triangles.add(new Triangle(seekPt1(tolerance, p1), seekPt1(tolerance, p2), seekPt1(tolerance, p3)));
    }
    Point seekPt2(float tolerance, Point p1){
        for(Point p2: this.points)
            if (p2.isNear(p1, tolerance)) return p2;
        return new Point(p1.x, p1.y);
    }
    void addTriangle2(float tolerance, Point p1, Point p2, Point p3) {
        this.triangles.add(new Triangle(seekPt2(tolerance, p1), seekPt2(tolerance, p2), seekPt2(tolerance, p3)));
    }
}
```

As you can see, methods `seekPt1` and `addTriangle1` and `addTriangle2` show two different ways to add a triangle to a mesh: the first does not duplicate points objects, while the second one does. This shows how the type annotation of a method makes the aliasing contract of the method explicit: whenever a parameter is taken as `balloon`, external or external `readonly`, no new aliases to that parameter are created.

3.2 Adding Encapsulation
The Mesh class by itself does not provide any guarantee about how points and triangles are shared between different Mesh instances. It is possible to add useful restrictions while using such instances as the following example demonstrates. We assume a vector of balloon meshes (VectorBalloonM) and a vector of readonly triangles (VectorReadonlyT):

```java
class Workbench{
    VectorBalloonM meshes;
    Workbench(VectorBalloonM m) { this.meshes=m; }
    void addMesh() @ { meshes.add(new Mesh()); }
    void addTriangleGui(int mIndex, Gui gui) @{
        Mesh mExt=this.meshes.get(mIndex);
        Mesh m=mExt;/ balloon local type promotion
        m.addTriangle1(gui.tolerance(), gui.selectPoint(), gui.selectPoint(), gui.selectPoint());
    }
    void mergeMeshes(int mIndex1, int mIndex2)@{
        Mesh m1=this.meshes.get(mIndex1);/ balloon local
        for(Triangle t: m1.triangles)
            m1.addTriangle2(0.15f, t.p1, t.p2, t.p3);
    }
}
```

```java
boolean isTriangleOverlap()@{
    VectorReadonlyT v= new VectorReadonlyT();
    for(Triangle t: m.triangles)
        v.add(t); // v converted to mutable
    while(!v.isEmpty()){
        Triangle t1=v.pop();
        for(Triangle t2:v)
            if(t1.isOverlap(t2)) return true;
    }
    return false;
}
```

The constructor shows how `readPoint` parameters can be used to initialise `balloon` fields. This pattern allows to avoid the inefficient defensive cloning or copying of objects that is required in Java in order to correctly provide encapsulation. [Item 24)

The method `addTriangleGui` shows how to interact with a GUI to add a triangle. For clarity we show explicitly an `external` local variable `mExt` that is converted to a `mutable` one thanks to the balloon `local` type promotion. In method `mergeMeshes` we show that balloon `local` type promotion can be applied to an expression with no need for a local variable. As you can see, since the points of the triangle are provided by the `gui`, it is possible to use such points to construct the triangle, and thus method `Mesh.addTriangle1` is called. On the other hand, in `mergeMeshes` the points come from another mesh. Hence, it is not possible to use the same point and preserve the balloon invariant for the meshes, so the type system force us to call the `Mesh.addTriangle2` method instead.

Finally, method `isTriangleOverlap` checks if in the whole set of triangles, from all the meshes, there are two triangles that overlap. A natural way to do this check is to put all the triangles in a collection and then consume it while checking for a couple of triangles, `t1` and `t2` such that `t1.isOverlap(t2)`. All of those triangles come from different, individually encapsulated meshes. In order to put all of them into a single collection we have to use the temporary type promotion.

3.3 Reading Meshes from Files
The following code shows how to read a `readPoint` mesh using Java Scanner object. As you can see, a mesh is quite complex object, with aliasing involved, but our type system is able to verify that the produced objects are `readPoint`.

```java
class Reader {
    Point readPoint(Scanner s)@{
        return new Point(s.nextFloat(), s.nextFloat());
    }
    Shape readMesh(Scanner s)@{
        Mesh m=new Mesh{
            new VectorMutableP(), new VectorMutableT()};
        while(m.hasNextFloat())
            m.addTriangle(new Triangle(t1, this.readPoint(s),
                this.readPoint(s), this.readPoint(s)),
                this.readPoint(s));
        return m;
    }
}
```

4. Formalisation
First we present a formal definition for BI-JAVA without any support for parallelisation. We show syntax, reduction and typing. Typing is composed of conventional subtyping relation (found in the Technical Report [16]), freshness analysis, conventional typing and promotion rules. After defining the observable behaviour of BI-JAVA, in Section 3 we show how to transparently optimise a program using parallelisation.

4.1 Syntax
In Figure 1 we show the syntax of BI-JAVA. We assume countably infinite sets of variables \( x \), class or interface names \( C \), method
names $m$, and field names $f$. As in FJ [9] variables include the special variable this.

A program $p$ is a set of class and interface declarations. A class declaration consists of a class name followed by the set of implemented interfaces, the sequence of field declarations and the set of method declarations. We assume conventional constructors as in FJ to be implicitly declared. To keep the presentation focused on our type modifiers and their application to parallelisation, we do not consider class composition operators like the extends operator in Java. We showed in [7, 10] how expressive composition operators (subsuming inheritance) can be added to Java-like languages.

An interface declaration consists of an interface name followed by the sets of extended interfaces and method headers. Field declarations are as in FJ. Method declarations are composed by a method header and a body. The method header is as in FJ. However since here types are richer, we complete the method header with the type modifier for the implicit parameter this. We write the type modifier for this after the parameters list, following the syntactic convention of C++ const modifier.

As in FJ, method bodies are simply ordinary expressions. Expressions can be variables, method or constructor calls, field access, field update and local variable declarations.

Variables can be declared in method headers or inside expressions of the form $T\ x = e_0; e_1$; that is equivalent to a let in functional languages. For simplicity, an expression is well-formed only if the same variable is not declared twice.

In method bodies we omit curly brackets and the keyword return inside the formalisation. In the examples we place the keyword inside the top level expression, but after any local variable declaration.

In the sequences of field declaration, parameter declaration and method or constructor calls the order is relevant. In all the other sequences the order is irrelevant, and thus, they can be considered as sets (or maps).

Types are composed of two components: a type modifier $M$ and a class or interface name $C$. A modifier $M$ have seven possible values: $\text{e}$ (for external), $\text{x}$ (for mutable), $\text{f}$ (for fresh), $\text{c}$ (for constant), $\text{f}$ (for final), $\text{i}$ (for interface), $\text{s}$ (for string), $\text{m}$ (for map), $\text{a}$ (for array)

4.3 Fresh Analysis and $\Gamma$

Figure 4. Syntax

As in FJ, method bodies are simply ordinary expressions. Expressions can be variables, method or constructor calls, field access, field update and local variable declarations.

In the sequences of field declaration, parameter declaration and method or constructor calls the order is relevant. In all the other sequences the order is irrelevant, and thus, they can be considered as sets (or maps).

4.2 Reduction

Figure 3 shows reduction rules for $\text{BI-JAVA}$ (without parallelism). In order to show reduction, we have to enrich the expressions with object identifiers as run time expressions. As usual, the heap is modelled as a finite map from object identifiers to records annotated with a class name.

Reduction is an arrow over pairs consisting of a heap and an expression. To improve readability we will mark in grey the heap part. During reduction modifiers $M$ are completely ignored.

Rule (CTX) is standard. Rule (F-ACCESS) extracts the value of field $f$. We use the notation $\mu_{\{f\}}$ for the value of field $f$ of object $i$. Rule (CONSTRUCTOR) reduces constructor invocations.

```
Figure 3. Reduction
```

Rule (M-INVK) models a conventional method call. We assume a fixed program $p$ and we use notation $p(C)\ .\ m$ to extract the method declaration. We use the notation $e[x_1 = i_1, \ldots, x_n = i_n]$ for variable substitution, that is, we simultaneously replace all the occurrences of $x_i$ in $e$ with $i_i$. Finally, Rules (V-DEC) and (F-UPDATE) are straightforward. We use notation $\mu_{\{f\}}$ to override the heap location $i_0, f$ with the value $i_1$.

Free variables are typed using a variable environment $\Gamma$. The treatment is conventional for all variable kinds but $\text{fresh}$ ones. If a $\text{fresh}$ reference is not converted to another kind of reference, it can be used only one time: indeed typing rules propagate that variable to only one sub expression. If a $\text{fresh}$ reference is converted to an immutable or balloon one, then it can be used in many points in the expression, and thus it is treated as a normal variable of that kind. In many cases, if a $\text{fresh}$ reference is converted to a mutable one, it is simply considered mutable, and it can be used at many points in the expression. However, if a $\text{fresh}$ reference is converted to a mutable one inside a promotion expression, the expression outside that promotion can or cannot see that variable depending on the kind of promotion.

We add to the type modifiers the production $\text{m}(\tau)$ (where the metavariable $\tau$ ranges over the three different promotion kinds) with the following meaning: if $\Gamma(x) = \text{m}(\tau)$, $C$ then $x$ refers to a fresh object that is going to be converted into mutable after entering the sequence (where order is relevant) of nested promotion $\tau$. Thus $\text{m}$ is just a syntactic sugar for $\text{m}(\emptyset)$.

The programmer cannot use types of the form $\text{m}(\tau)$ $C$ with non empty $\tau$. They are only used internally by the type system for typing expressions with many nested promotions.
In order to formalise the precise way fresh and \( \bar{\phi} \) refer to used, we employ a substructural type system \([17]\), very similar to an affine type system. In detail, we define three different treatments for variables:

- unrestricted variables can be used any number of times in an expression; all the non fresh variables are unrestricted; fresh variable converted in \( \text{mutable} \) inside only temporary type promotions \( \bar{\phi}(\bar{\phi}) \) are also unrestricted.
- affine variables can be used at most one time in an expression; fresh variables are affine, fresh variable converted in \( \text{mutable} \) inside at most one permanent type promotions are also affine.
- last, we introduce the concept of affine external variables, that can be used at most one time with their type, but many time with the corresponding external type.

Now we can provide a generic Context split relation \( \Gamma = \Gamma_0 \circ \Gamma_1 \) as in \([17]\), mapping a single variable environment into an unordered pair (see Rule (SWAP)) of variable environments.

Rule (UNRESTRICTED) states that an unrestricted variable can be used many times. Rule (AFFINE) states that an affine variable will be contained in only one of the two environments.

Finally, rule (AFFINE-EXT) states that an affine external variable will be contained in one of the two environments with its type, while the other one will contain that variable with the corresponding external type.

The non-deterministic splitting relation \( \Gamma = \Gamma_0 \circ \Gamma_1 \) does not provide a direct guide for an efficient implementation of our type system. We believe that the implementation approach described in \([17]\) can be easily adapted to our case.

4.4 Conventional Typing

Figure 4 shows straightforward rules for well formedness of classes and interfaces. For any well-typed program all classes and interfaces are valid. Rule (CLASS) validates a class if all methods are valid and if the interfaces \( \overline{\text{C}} \) are correctly implemented; that is, for all methods of all the implemented interfaces, a method with an analogous header is declared in the class \( C_0 \). Note how methods are validated in the context of their class. Similarly, rule (INTERFACE) validates an interface if the interfaces \( \overline{\text{C}} \) are correctly implemented; that is, for all method headers of all the implemented interfaces, an analogous method header is declared in the interface \( C_0 \).

Rule (METH-T) derives the type for this from the class name \( C \) and the type modifier \( M \). Note how this rule introduces the fresh variables inside \( \Gamma \): they can be non-deterministically introduced as fresh or as any of the possible conversion for a fresh variable. That is, the conversion is executed non-deterministically when the variable enters in scope. In the premise we use notation \( \Gamma \vdash e : T_1 \leq T_2 \) as a shortcut for \( \Gamma \vdash e : T_1 \) and \( \Gamma \vdash e : T_2 \).

Figure 5 shows conventional typing rules for expressions. Rule (VAR-T) is conventional.

Rule (F-ACCESS-T) is conventional but uses function \( \text{f-access}(\_\_\_\_) \) to provide the correct modifier to the resulting value. Formally:

\[
\begin{align*}
\text{f-access}(M)(\_\_\_\_) & = M \\
\text{f-access}(\_\_\_\_)(\_\_\_\_) & = M \\
\text{f-access}(\_\_\_\_)(M) & = M \\
\text{f-access}(\_\_\_\_)(\_\_\_\_) & = M \\
\text{f-access}(\_\_\_\_)(\_\_\_\_) & = M \\
\text{f-access}(\_\_\_\_)(\_\_\_\_) & = M \\
\end{align*}
\]

Rule (M-INVK-T) is conventional. Note how here and in the following we use relation \( \Gamma = \Gamma_0 \circ \ldots \circ \Gamma_n \).

Rules (F-UPDATE-T) and (F-UPDATE-TB) manage field update. Rule (F-UPDATE-T) covers the case of non \( \text{balloon} \) fields. In this case to force correct field update it is enough to state that the assigned value is not \( \text{balloon} \), external or external readonly. Indeed, references of these kinds cannot be assigned. Rule (F-UPDATE-TB) covers the case of \( \text{balloon} \) fields. It requires the assigned value to be fresh. In any case the modified value has to be mutable. Promotion can be used to modify external values.

Rule (NEW-T) is conventional, but, just like the field update, has to ensure that all the \( \text{balloon} \) fields are initialised with fresh values, and that non assignable values are not assigned.

Rule (V-DEC) is conventional but, as for (METH-T), takes care of the fresh conversion.

4.5 Promotions Rules

First, we define a function \( \text{f-close}(\_\_\_) \) that returns a loosely closed type for any reference kind as follows:
FIGURE 6. Type system expressions

In order to define Soundness, we need to define concept of stack, well formedness of the memory in a stack, and how to type a run time expression. Figure 7 defines a stack (Σ) as a map from object identifiers to types. The stack associated with a run time expression maps all the identifiers appearing inside such expression to a type. Rule (LOC-T) states that if an expression e with some free variable is well typed, then the same expression with some variable replaced with object identifier is well typed too, as long as the stack Σ is rich enough to provide the type information for all the object identifiers introduced inside e.

4.6 Soundness

Formally: \( \text{t-close-b}(\text{TP}(\text{C})) = \text{TP}(\text{C}) \) and \( \text{t-close-b}(\text{TP}(\text{T})) = \text{TP}(\text{T}) \) otherwise.

Note how in all the three variation of \( \text{t-close}(-) \) the other kinds of modifiers of the form \( \text{TP}(\text{C}) \) are undefined. This ensure that the rules are applicable only if such modifiers are not involved.

Finally, rule (SUB-EXP-T) allows to see any sub expression as a local variable, removing from the programmer the need of declare a local variable only to please the type system.

\begin{align*}
\Gamma \vdash e : T \\
\text{t-close}(\text{C}) = \text{C} \\
\text{t-close}(\text{C}) = \text{C} \\
\text{t-close}(\text{T}) = \text{T} \quad \text{otherwise}
\end{align*}

Rules (PERMANENT1-TP) and (PERMANENT2-TP) define permanent type promotions: as you can see mutable results can be promoted into fresh while readonly results can be promoted into immutable. We use function \( \text{t-close-p}(\Gamma) \), that behave like \( \text{t-close}(\Gamma) \) but is extended in order to manage the types of the form \( \text{TP}(\text{C}) \) and \( \text{TP}(\text{T}) \).

Formally: \( \text{t-close-p}(\text{TP}(\text{C})) = \text{TP}(\text{C}) \) and \( \text{t-close-p}(\text{TP}(\text{T})) = \text{TP}(\text{T}) \) otherwise.

\begin{align*}
\Gamma \vdash e \text{ : } T \\
\text{t-close-p}(\Gamma) \vdash e \text{ : } \text{C} \\
\text{t-close-p}(\Gamma) \vdash e \text{ : } \text{C} \\
\text{t-close-p}(\Gamma) \vdash e \text{ : } \text{T} \text{ otherwise}
\end{align*}
The fresh heap \( FH^\Sigma \) is composed by all the fresh objects and their reachable object graphs

\[ FH^\Sigma = \{ t_0 \in \text{rog}(\Sigma) \mid t_1 \in \text{dom}(\mu), \text{valid}(\Sigma, t_0) \} \]

An object \( t_0 \) is a valid ballon, that is \( \text{valid}(\Sigma, t_0) \) iff

- at most one balloon reference exists to the balloon object, and is outside the balloon itself: \( \forall f_1, f_2, t_1, t_2 \in \text{dom}(\mu) \) \( | H_B \) \( (t_1, f_1 \xrightarrow{\mu} t_0) \) and \( (t_2, f_2 \xrightarrow{\mu} t_0) \) implies \( t_1 = t_2, f_1 = f_2 \) and \( t_1 \notin \text{valid}(\Sigma) \)

- all the internal aliasing of the balloon object itself are of a mutable kind or subtype thereof: \( \forall f, t_2 \in \text{rog}(\Sigma) \)

\[ (t_2 \xrightarrow{\mu} t_0) \implies (t_2 \xrightarrow{\mu} \text{valid}(\Sigma), t_0) \]

- finally, every balloon has a tree shape: that is all the subballons are disjoint \( \forall t_1, t_2 \in RB_B(\Sigma) \) such that \( t_1 \neq t_2 \), \( \text{rog}(t_1) \cap \text{rog}(t_2) = \emptyset \) holds.

Where \( RB_B(\Sigma) = \{ t \in \text{dom}(\mu) | t \xrightarrow{\mu} t_1, t_1 \xrightarrow{\mu} t \} \) and \( B^\Sigma_C \), balloon candidates are objects identifiers for which a balloon reference exists in the memory or on the stack:

\[ B^\Sigma_C = \{ t_0 | t_1 \in \text{dom}(\mu) | H_B, t_1 \xrightarrow{\mu} t_0 \} \cup \{ t | \Sigma(t) = \emptyset \} \]

Finally, in (MEM-MEM-OK) the last two side conditions check that all the reachable balloons in the stack are disjoint and that all the balloon data stored in the shared heap belongs to a single balloon. Where \( \mu \in RB_B(\Sigma) \) iff \( \Sigma(\mu) = B_C \) or \( \mu \in RB_B(\Sigma) \) with \( \Sigma(\mu) = \emptyset \), and with the following definition of shared heap:

\[ SH^\Sigma = \{ t \in \text{dom}(\mu) | \Sigma(t) = B_C \} \cup \{ t | \Sigma(t) = \emptyset \} \]

Finally, rule (MEM-MEM-OK) verifies that the invariants are preserved during reduction: that is, a memory \( \mu_C \) can be the result of an operation over \( \mu_0 \) only if the two memories are well formed in their respective stacks, and:

- all the immutable objects are preserved in the same state (side conditions 2 and 3),
- if two balloons are disjoint in the common part of the stack in \( \mu_C \) then the first balloon in \( \mu_C \) is disjoint with the second balloon, either if considered in \( \mu_0 \) or \( \mu_1 \).

**Theorem 1 (soundness).**

If \( p \) is well typed, \( \emptyset \vdash e : T \) and \( \emptyset | e \rightarrow \emptyset \mid e_0 \) then either

- \( e_0 = \epsilon, i : T_0 \vdash \mu_0 \) and \( T_0 \leq T \), or
- \( \emptyset \mid \mu_0 \vdash e_0 \rightarrow \emptyset \mid e_1 \rightarrow \emptyset \mid e_0 \rightarrow \emptyset \mid e_1 \rightarrow \emptyset \mid e_0 \rightarrow \emptyset \mid e_1 : T_1 \rightarrow T_0 \)

As usual \( \rightarrow \) is the reflexive and transitive closure over \( \rightarrow \). It is possible to divide the proof into progress and subject reduction:

**Theorem 2 (progress).**

If \( p \) is well typed, \( \Sigma \vdash \mu \) and \( \emptyset \vdash i : T \) then either \( e = \epsilon \) or \( \emptyset \mid e \rightarrow \emptyset \mid \epsilon \)

**Proof.** Since reduction ignores type modifiers the proofs is equivalent to that of FJ.

**Theorem 3 (subject reduction).**

If \( p \) is well typed, \( \emptyset \mid \Sigma_0 \vdash e_0 : T_0, \Sigma_0 \vdash \mu_0 \) \( \emptyset \mid e_0 \rightarrow \emptyset \mid e_1 \). Then \( \Sigma_0 \rightarrow \Sigma_1 \vdash \mu_0 \rightarrow \mu_1 \) and \( \emptyset \mid \Sigma_1 : T_1 \rightarrow T_0 \)

**Proof.** The proof is in the companion technical report [16].
5. Adding Parallelisation

The language BI-JAVA as defined before offers no support for parallelisation. Formally, the reduction arrow is deterministic, and thus is trivially confluent\footnote{Only object identifiers are nondeterministically chosen.}

We now provide a way to automatically add parallelisation as optimisation. Informally, it means that no observable behaviour has to change, to use the terminology of [3] we want to obtain a parallel performance model without altering the well known sequential semantic model.

We extend the language with fork-join expressions. Such expressions are added during compilation, replacing sequences of normal local variable declaration expressions, and thus are completely transparent from the point of view of the programmer.

Formally, we extend the language with fork-join expressions, allowing non-deterministic reduction. However a well typed program have a confluent reduction. Then, we show a program transformation preserving the semantic, but replacing local variable declaration expressions with the new fork-join expressions.

In Figure 9 we show syntax for fork-join and logical expressions, context, reduction and typing rules for fork-join expressions.

5.1 New Syntax

Fork join expressions are composed by a sequence of variable initialisations and a conclusive expression. Note that the order of variable initialisations is relevant since it induces the (perceived) order of execution for the sub-expressions. Every variable initialisation is annotated with a logic expression encoding the dependency of such expression. Such logic expressions are composed by the constants true and false, the \( \lor \) (logical or) operation and the identity check over variables, or object identifiers. Any logic expression is related to a specific variable, declared inside the same fork-join block, the syntax \( x : L \) means that a dependency exists with the expression that initialize \( x \) if \( L \) holds, that is, is equivalent to true. For example,
\[
\begin{align*}
\{ & \text{int } \times \equiv 1+2; [\] } \\
& \text{int } \gamma \equiv 1; [\] \\
& \text{int } z \equiv x+y; [\text{if true, y if true}] \\
\text{in } z+1; \\
\end{align*}
\]
Meants that \( x \) and \( y \) depends from nothing and can be computed in parallel, while before computing \( x \) we have surely to wait for the computation of \( x \) and \( y \), since \( z \) depend from them.

Assuming a class \( C \) with methods \( m \leftarrow C() \), \( m \) \( k() \) and \( int \ h \leftarrow \left( m \right) c() \), we can show a more complex example:
\[
\begin{align*}
\{ & \text{c}\leftarrow x=1; [\] } \\
& \text{c}\leftarrow y=2; [\] \text{if } s_1 \equiv s_2 \\
& \text{in } x.h(y); \\
\end{align*}
\]
The expression associated with \( x \) can be computed, while in order to compute the one associated with \( y \) we have to check if \( s_1 \equiv s_2 \).

• if the two object identifiers are different, we can start executing that expression in parallel,

• otherwise, we have to wait for the composition of the \( x \) expression.

5.2 New Reduction Rules

Any expression with an empty dependency declaration can be reduced, see last case of the context declaration.

Rule (\text{OUT-E}) is straightforward, while rule (\text{COMMIT}) replace occurrences of a local variable with its computed value. Dependency relations that are now equivalent to false are removed. Formally:

• substitution \( J[x\leftarrow i] \):
  \[
  (T \leftarrow e_i; [\text{dep}]) \equiv T \leftarrow e \equiv T \leftarrow e_i; [\text{dep}][x\leftarrow i] \\
  \]

• substitution \( [\text{dep}][x\leftarrow i] \):
  \[
  [\text{dep}][x\leftarrow i] \equiv [\text{dep}][x\not\equiv x'] \\
  \]

Substitution over logic expressions \( [\text{dep}][x\leftarrow i] \) is straightforward: distributed on \( \lor \) operation, identity over constants and conventionally defined over variables.

In future work, it is clearly possible to perform many kinds of optimisations over dependency and logic expressions, either during the generation at compile time or during variable substitution. We show now a possible reduction sequence for the following example; we assume the same class \( C \) as before, we omit the memory and show only some relevant steps:

\[
\begin{align*}
1 \{ & \text{c}\leftarrow \text{new } C() \}; [\] \\
& \text{c}\leftarrow a.k(); [\text{if true, b if } t_1 \equiv a] \} \text{ in } b.h(c); \\
\end{align*}
\]
expressions associated with \( a \) and \( b \) can be computed in parallel,

\[
\begin{align*}
2 \{ & \text{c}\leftarrow e_2(); [\] \\
& \text{c}\leftarrow a.k(); [\text{if true, b if } t_1 \equiv a] \} \text{ in } b.h(c); \\
\end{align*}
\]
here we suppose the computation over \( e_1.m() \) to be quite long, so in the shown point \( a \) is already associated with the value \( e_2 \), while \( b \) is of the form \( e_1 \).

If \( e_2 \not\equiv t_1 \) then the reduction will proceed in the following way:

\[
\begin{align*}
3 \{ & \text{c}\leftarrow e_2(); [\] \\
& \text{c}\leftarrow e_2.k(); [\] \} \text{ in } b.h(c); \\
\end{align*}
\]
The execution of \( c \) can start now, while \( b \) is transparently continuing its execution, and it is now in \( e_2 \).
efficient test can allow safe parallel execution. One extension can be found in the Technical Report [16].

5.5 Parallelisation Guarantee

The main result of our work is that in any BI-JAVA program, any sequence of variable declaration of the form

\[ T_1 x_1 = e_1; \ldots; T_n x_n = e_n; e_0 \]

can be optimized in the form

\[ \{ T_1 x_1 = e_1; [\text{dep}_{p_1}] \ldots [\text{dep}_{p_n}] \} \text{ in } e_0 \]

(where \( \text{dep}_{p_1} \ldots \text{dep}_{p_n} \) are inferred by \( \text{FORK-JOIN-T} \)) and the semantic is not influenced in any way.

In the following theorems we express the preservation of the semantic in term of equivalence of the result modulo alpha conversions over object identifiers (symbol \( \equiv \)).

**Theorem 4 (Confluence).**

A well typed program have a confluent reduction. Formally:

\[ \forall p : e : T, [e] p_0 | t_0, \exists \text{ such that } [e] p_0 | t_0 \Rightarrow [e] \mu_1 | t_1 \]
6. Related Work

The most closely related work to our approach is Deterministic Parallel Java (DPJ) [2,3]. While our motivation is present in other recent works [15] we are among the few people who chose 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 [13].

DPJ [2,3] 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.

B1-JAVA takes the approach whereupon a straightforward reference comparison kind of check can be inserted automatically at the right points inside the program allowing safe parallel execution of expressions without the need for speculative parallelism (such as transactional memory etc). These safety checks capture the cases when speculative parallelism will succeed and thus avoid any need for potential roll backs. In other words, the cost of rolling back is when speculative parallelism will succeed and thus avoid any need for speculative parallelism etc). These safety checks capture the cases where expressions without the need for speculative parallelism (such as transactional memory etc) are executed in parallel. Our system employs a combination of a static checker with regions and effects and requires explicit manipulation of memory regions.

B1-JAVA takes the approach whereupon a straightforward reference comparison kind of check can be inserted automatically at the right points inside the program allowing safe parallel execution of expressions without the need for speculative parallelism (such as transactional memory etc). These safety checks capture the cases when speculative parallelism will succeed and thus avoid any need for potential roll backs. In other words, the cost of rolling back is when speculative parallelism will succeed and thus avoid any need for speculative parallelism (such as transactional memory etc). These safety checks capture the cases where expressions without the need for speculative parallelism (such as transactional memory etc) are executed in parallel. Our system employs a combination of a static checker with regions and effects and requires explicit manipulation of memory regions.

B1-JAVA takes the approach whereupon a straightforward reference comparison kind of check can be inserted automatically at the right points inside the program allowing safe parallel execution of expressions without the need for speculative parallelism (such as transactional memory etc). These safety checks capture the cases when speculative parallelism will succeed and thus avoid any need for potential roll backs. In other words, the cost of rolling back is when speculative parallelism will succeed and thus avoid any need for speculative parallelism (such as transactional memory etc). These safety checks capture the cases where expressions without the need for speculative parallelism (such as transactional memory etc) are executed in parallel. Our system employs a combination of a static checker with regions and effects and requires explicit manipulation of memory regions.

B1-JAVA takes the approach whereupon a straightforward reference comparison kind of check can be inserted automatically at the right points inside the program allowing safe parallel execution of expressions without the need for speculative parallelism (such as transactional memory etc). These safety checks capture the cases when speculative parallelism will succeed and thus avoid any need for potential roll backs. In other words, the cost of rolling back is when speculative parallelism will succeed and thus avoid any need for speculative parallelism (such as transactional memory etc). These safety checks capture the cases where expressions without the need for speculative parallelism (such as transactional memory etc) are executed in parallel. Our system employs a combination of a static checker with regions and effects and requires explicit manipulation of memory regions.

B1-JAVA takes the approach whereupon a straightforward reference comparison kind of check can be inserted automatically at the right points inside the program allowing safe parallel execution of expressions without the need for speculative parallelism (such as transactional memory etc). These safety checks capture the cases when speculative parallelism will succeed and thus avoid any need for potential roll backs. In other words, the cost of rolling back is when speculative parallelism will succeed and thus avoid any need for speculative parallelism (such as transactional memory etc). These safety checks capture the cases where expressions without the need for speculative parallelism (such as transactional memory etc) are executed in parallel. Our system employs a combination of a static checker with regions and effects and requires explicit manipulation of memory regions.

Finally, recent work by Naden et al. [12] extends Plaid with a similar set of access permissions to ours so that they can support borrowing of unique, shared, and immutable objects. One of their motivations is an ability to detect noninterference of concurrency, which we believe B1-JAVA achieves.

7. 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 a static and efficient dynamic check (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 formalism and proofs to support a richer subset of a Java-like language. In particular, our companion technical report [16] contains a section that extends the def(π1, π2) which we would like to investigate further. We also plan to complete a prototype implementation of our approach and perform a user study evaluating the ease of use of our approach in day to day programming.

A. Subtyping Relation

In our setting subtype $T_1 \leq T_2$ is defined as follows:

- $T \leq T$;
- $M.C_1 \leq M.C_2$ if $p(C_1) = \text{class } C_1 \text{ implements } C, \_ (\_)$ and $M.C \leq M.C_2$;

References

• \( M C_1 \leq M C_2 \) if \( \text{dep}(C_1) = \text{interface } C_1 \text{ extends } C, \ldots (\ldots) \) and \( M C \leq M C_2 \);
• \( M C \leq C \) if \( M = b \) or \( M = \infty \);
• \( M C \leq C \) if \( M = 0 \) or \( M = \infty \);
• \( M C \leq C \) if \( M = \infty \) or \( M = \infty \);

### B. Extending \( \text{dep}(\pi^1, \pi^2) \)

We show one possible extension allowing to safely parallelize also some \( \text{mutable} \) objects.

We extend the syntax to use \( \pi \) to indicate a variable or an object identifier possibly followed by a set of field accesses.

\[
\pi ::= x.J | i.J \quad \text{extended comparison}
\]

... is often happens to use objects as records containing some highly related data, and to use such object as parameter for complex computation. In those cases the object graph have a very simple structure, so simple that the shape of their object graph is statically predictable up to the point where \( \text{mutable} \) or \( \text{balloon} \) fields are reached. We can inductively define a \( \text{predictable} \) class as a class where all the fields are of \( \text{mutable} \), \( \text{balloon} \) or \( \text{predictable} \) type. Note that such definition rules out, for example, a linked list where the next field is \( \text{mutable} \). We call \( \text{mutable-predictable} \) types composed by a \( \text{mutable} \) modifier and the name of a \( \text{predictable} \) class.

We can define the following two function over \( \text{mutable-predictable} \) variables:

Function \( m-pred(\pi \pi C) \) denoting all the paths that brings to another \( \text{mutable} \) entity inside the reachable object graph of \( \pi \).

\[
\pi \pi C \in m-pred(\pi \pi C)
\]

\[
\pi \pi J \pi J \pi C_2 \in m-pred(\pi \pi C_0)
\]

iff \( \pi \pi J \pi C_1 \in m-pred(\pi \pi C_0) \)

and \( p(C_1) = \text{class } C_1 \text{ implements } \ldots (\ldots) \)

Function \( b-pred(\pi \pi C) \) denoting all the path that brings to a \( \text{balloon} \) entity inside the reachable object graph of \( \pi \).

\[
\pi \pi J \pi C_2 \in b-pred(\pi \pi C_0)
\]

iff \( \pi \pi J \pi C_1 \in b-pred(\pi \pi C_0) \)

and \( p(C_1) = \text{class } C_1 \text{ implements } \ldots (\ldots) \)

Path to \( \text{mutable} \) references are not relevant, and by our definition of \( \text{predictable} \) class other kinds of objects are simply not reachable.

Now is possible to define when \( \text{dep}(\pi^1, \pi^2) \) is true over variables with \( \text{mutable-predictable} \) type:

\[
\text{dep}(\pi^1, \pi^2) = \{ \pi^1 \in b-pred(\pi^2) \}
\]

iff \( \pi^1 \) have \( \text{balloon} \) type and \( \pi^2 \) have \( \text{mutable-predictable} \) type

\[
\text{dep}(\pi^1, \pi^2) = \text{m-pred(\pi^1) | m-pred(\pi^2)}
\]

iff \( \pi^1 \) and \( \pi^2 \) have \( \text{mutable-predictable} \) type

where\(^4\) non-disjointness: \( \pi^1 \ldots \pi^n \in \pi \) \( \pi^1 \in \pi \lor \ldots \lor \pi^n \in \pi \) inclusion: \( \pi \in \pi^1 \ldots \pi_n \Rightarrow \pi^1 \equiv \pi^1 \lor \ldots \lor \pi^n \equiv \pi^n \)

We now assume the dependency to hold for all the other cases:

\[
\text{dep}(\pi^1, \pi^2) = \text{true otherwise}
\]

However, there are clearly many other cases in which is possible, and convenient, to study a precise logic expression allowing safe parallelization. For example it would be possible to improve our study over \( \text{external} \) variables. This would require to enrich our logical expression with dynamic tests over the actually type of a variable.

### C. Discussion

#### C.1 Using Balloon invariant for encapsulation

In many cases an instance has full control of its \( \text{balloon} \) fields and their reachable object graph\(^5\).

For example a pattern like the following will be safe in any arbitrary complex (and potentially circular) object graph:

```java
class Foo{
    Balloon Bar bar;
    /* other fields */
    void meth(/*some parameters*/){...}
}
```

Independently of the other fields, if the parameters do not contains \( \text{balloon} \), \( \text{external} \) or \( \text{external} \) \( \text{readonly} \) elements the method execution will have full control of the \( \text{balloon} \) field.

For example a pattern like the following will be safe in any arbitrary complex (and potentially circular) object graph:

```java
class Foo{
    private Balloon Bar bar; // no public getter method
    private boolean opInProgress=false; // no public setter
    void meth(Mutable Balloon pi){
        if (this.opInProgress)
            throw new Error("unexpected indirect recursion");
        try{this.opInProgress=true;}/this.bar only set in Foo
        doStuff((this.bar,pi);)
        finally{opInProgress=false;}
    }
}
```

We use \( \text{opInProgress} \) to prevent undesired indirect multiple accesses to the \( \text{bar} \) data.\(^6\) If the method have \( \text{balloon} \) parameters then a simple dynamic check (straightforward identity test \( x=y \)) can ensure full control. However if \( \text{external} \) or \( \text{external} \) \( \text{readonly} \) parameter are present, and they are of a type that can potentially reach a \( \text{Bar} \) object in the reachable object graph, ensuring full control is not trivial. If an object points to a complex object graph that have non trivial consistency relations, and modification over this object graph requires to temporary break such relations, the pattern above allows is a way to encode such modification operation with the guarantee that no other client will be able to see such uncoherent state.

#### C.2 Memory management optimisations

\( \text{Immutable} \) objects can have an ad hoc memory management. Whenever a \( \text{fresh} \) object is converted into \( \text{mutable} \), the hash code is computed and the object is stored in a system Map. If an equivalent object is already present in the map, the new version can be instantaneously garbage collected. System generated hashcode and equals, supporting circularity, should be automatically introduced. Moreover, keys of a HashSet/HashMap should be only \( \text{Immutable} \), such maps will be faster and with a simpler semantic.

Moreover, from \( F_3 \) any expression that does not return a \( \text{balloon} \), \( \text{external} \) or \( \text{external} \) \( \text{readonly} \) reference can be safely garbage collect all the \( \text{balloon} \), \( \text{external} \) and \( \text{external} \) \( \text{readonly} \) created references at the end of its execution. Moreover, every \( \text{l-closed} \) expression that returns a \( \text{Immutable} \) value (or a primitive type) can safely garbage collect all the (non \( \text{Immutable} \)) created objects.

This opens the door for a simple language extension with a safe \( \text{try-with-resource} \) statement, when the resource object is guaranteed to be collected at the end of the statement.

\(^{4}\) No need to check \( \text{b-pred(\pi^1)} | \text{b-pred(\pi^2)} \) in the second case.

\(^{5}\) \( \text{Reachable objects are not taken into consideration by many approaches, so our guarantee is stronger, more useful and helps produce code that is maintainable w.r.t. the change of internal representation} \)

\(^{6}\) Other approaches replace this pattern with temporary borrowing facilities or destructive reads (i.e. the field returns null when accessed). We believe this pattern provides better error control, even if more verbose.
D. Proofs

Proof of Theorem 3: By induction over the reduction rules. All cases but rule (CTX) does not require the use of the inductive hypothesis. Since we have a call by value semantic all cases of direct reduction work over very simple expressions where there is little space for the application of promotions. In the case of (V-DEC) we expect promotions over the whole expression $T x \to T x \to e$ to be always applicable directly to the conclusive expression $e$.

$r ::= i \cdot m \left( t \right) \mid \text{new} C \left( t \right) . f \mid t . f \to t . l | r \cdot m \left( t \right) \left[ \text{redex} \right]$

Case (CTX)

Choosing

(C1) $i \in \text{dom}(\Sigma_0)$ iff $i$ is a subterm of $r_0$

(C2) $\text{dom}(\mu_0) = \text{ren}_{\mu_0,0}(\text{dom}(\Sigma_0))$

Given

(G1) $\mu_0, \mu_1 \mid E[r_0] \to \mu_1, \mu_2 \mid E[e_1]$

(G2) $\Sigma_0, \Sigma \vdash \mu_0, \mu$

(G3) $\emptyset; \Sigma_0, \Sigma \vdash E[r_0] : T'_0$

Show

(R1) $\Sigma_0, \Sigma \vdash \Sigma_1, \Sigma \vdash \mu_0, \mu \vdash \mu_1, \mu$

(R2) $\emptyset; \Sigma_1, \Sigma \vdash E[e_1] : T_1 \leq T'_0$

we obtain:

(I1) $\mu_0 \mid r_0 \to \mu_1, \mu_2 \mid e_1$ by (G1),

(I2) $\Sigma_0^{\mu_0} \vdash \mu_0$ by (G2), (G3) and Lemma 7(ii),

(I3) $\emptyset; \Sigma_0^{\mu_0} \vdash r_0 : T_0$ by (G2), (G3) and Lemma 7(ii),

(IH1) $\Sigma_0^{\mu_0} \vdash \Sigma_1^{\mu_0} \vdash \mu_0 \vdash \mu_1$ and

(IH2) $\emptyset; \Sigma_0^{\mu_0} \vdash e_1 : T_1 \leq T_0$ by (I1), (I2), (I3) and inductive hypothesis.

Now we can obtain our results: By (IH1), (G2) and Lemma 7(ii) we obtain (R1) By (I3), (IH2), (G3), and Lemma 7(ii) we obtain (R2)

Lemma 6 (promotionPreservation).

$\Sigma_0 \vdash \mu_0$

implies

$\Sigma_0^{\mu_0} \vdash \mu_0$

Proof. By induction over sequence of promotions:

Case empty set of promotion

Trivial

Case Promotion sequence ends with a permanent type promotion

all the $\mu_0$ or $\text{ren}_{\mu_0,0}$ in $\Sigma_0$ are $\text{ren}_{\mu_0,0}$ or $\text{ren}_{\mu_0,0}$ $\text{ren}_{\mu_0,0}$, respectively. This means that $B_{\text{Ren}}(\Sigma_0^{\mu_0}) \subseteq B_{\text{Ren}}(\Sigma_0)$. Thus rule (MEM-T) is still applicable, since all side condition stay the same, but last is checked on less cases.

Case Promotion sequence ends with a balloon local type promotion

it can be shown correct in two steps:

- All the $\mu_0$ or $\text{ren}_{\mu_0,0}$ variable in the stack are seen as $\text{ren}_{\mu_0,0}$.

At this point rule (MEM-T) is still applicable, since all side condition stay the same, but last is checked on zero cases.

- Then we promote a single $\text{internal}$ in the stack to $\text{mutable}$.

This repopulate the set $B_{\text{Ren}}(\Sigma_0^{\mu_0})$ shared heap with (potentially) all the element inside the balloon containing such promoted object identifier.

Rule (MEM-T) is still applicable: the last two side conditions of (MEM-T) still holds:

- for the last clause of $\text{valid}\_B_{\text{Ren}}(\ell)$ we know that all the sub balloon are disjoint, and
- only elements inside a single balloon are added to the shared heap.

Case Promotion sequence ends with a temporary type promotion

As in the permanente type promotion case, we simply reduce the number of cases the last side condition of (MEM-T) is checked.

Lemma 7 (promotion application).

(G1) $\text{dom}(\mu_0) = \text{ren}_{\mu_0,0}(\text{dom}(\Sigma_0))$

(G2) $\Sigma_0, \Sigma \vdash \mu_0, \mu$

(G3) $\emptyset; \Sigma_0, \Sigma \vdash E[r_0] : T'_0$

implies

(i) $\Sigma_0^{\mu_0} \vdash \mu_0$

(ii) $\emptyset; \Sigma_0^{\mu_0} \vdash r_0 : T_0$

Proof.

(i) By (G2) and the property of the chosen division between $\Sigma_0, \Sigma$ and $\mu_0, \mu$ we know that $\Sigma_0 \vdash \mu_0$ We conclude by Lemma 6.

(ii) We choose $\Sigma_0^{\mu_0}$ to coincide with the set of promotion applied over subexpression $r_0$. By rule (LOC-T) $\emptyset; \Sigma_0^{\mu_0} \vdash r_0 : T_0$ iff $\Gamma_0^{\mu_0} \vdash r_0, t_1 \to t_2 \to \ldots t_n \to x_n : T_0$ with $\text{dom}(\Sigma) = t_1 \to t_2 \to \ldots t_n$, $\text{dom}(\Sigma) = x_1 \to x_2 \to \ldots x_n$, $\text{dom}(\Sigma) = x_1 \to x_2 \to \ldots x_n$, $\text{dom}(\Sigma) = x_1 \to x_2 \to \ldots x_n$

Thus, for our chosen $\Sigma_0^{\mu_0}$ we can conclude by straightforard application of the typing rules over $\Gamma_0, \Gamma \vdash E[r_0][t_1 \to x_1 \to \ldots t_n \to x_n] : T'_0$

Lemma 8 (substitution zero).

(G1) $\mu_0 \mid r_0 \to \mu_1, \mu_2 \mid e_1$

(G2) $\emptyset; \Sigma_0 \vdash r_0 : T_0$

(G3) $\emptyset; \Sigma_0, \Sigma \vdash E[r_0] : T'_0$

(G4) $\emptyset; \Sigma_1 \vdash e_1 : T_1 \leq T_0$

(G5) $\Sigma_0 \vdash \Sigma_1 \vdash \mu_0 \vdash \mu_1$

(G6) $\Sigma_0, \Sigma \vdash \mu_0, \mu$

exists a $\Sigma_1$ such that

(i) $\emptyset; \Sigma_1, \Sigma \vdash E[e_1] : T'_1 \leq T'_0$

(ii) $\Sigma_0, \Sigma \vdash \Sigma_1, \Sigma \vdash \mu_0, \mu \vdash \mu_1, \mu$

Proof. Induction over the typing rules for term (G3). For cases (F-ACCESS-T), (M-INVK-T) and (V-DEC-T) $\mu_0 = \mu_1$ and thus (ii) trivially holds. Cases for rules (F-ACCESS-T) and (M-INVK-T) are simple also with respect to (ii).

Case (F-UPDATE-T)

(i) holds by (F-UPDATE-T) second premise. The receiver is by construction inside the shared heap, in case the receiver is also inside a balloon, (ii) requires a case analysis over the possible kinds of the assigned value:

(ii) trivially holds if the assigned value is in the immutable heap.

(ii) holds, since readonly links does not contribute for the set of reachable ballons $B_{\text{Ren}}(\Sigma)$.
In this case we know that $r_0$ can be typed as mutable in $\text{l-close-p}(\Gamma_0) \vdash r_0[t_1 = x_1 \ldots t_n = x_n] : \mu C_0$ By cases depending on the kind of redex:

(1) $m \text{ (MEM-T)}$ also the expression $e_1$, body of the method $m$ can be typed with (PERMANENT1-TP) in this environment. (ii) holds since $\mu_0 = \mu_1$.

(2) $\text{new C (TP)}$ The created object clearly respects both the fresh invariants and the immutable invariants. So it can be considered inside the fresh heap.

The produced reference is inside the reachable object graph of $t_0$.

$t_0.f$ following the same process used for case (F-UPDATE-T) we can show that both $t_0$ and $t_1$ are in the shared heap, thus we are not violating the balloon invariants, thanks to (MEM-T) last side condition.

$T \xrightarrow{\tau} t_1 \ e$ also the expression $e$ can be typed with (BALLOON-L-TP) in this environment.

Case (PERMANENT2-TP)

In this case we know that $r_0$ can be typed as read-only in $\text{l-close-b}(\Gamma_0) \vdash r_0[t_1 = x_1 \ldots t_n = x_n] : \mu C_0$ By cases depending on the kind of redex, we omit details about (i) and (ii) since they are similar to the former point.

(1) $m \text{ (TP)}$ also the expression $e_1$, body of the method $m$ can be typed with (PERMANENT1-TP) or (PERMANENT2-TP) in this environment.

(2) $\text{new C (TP)}$ The created object clearly respects both the fresh invariants and the immutable invariants. So it can be considered inside the immutable heap. Indeed all reference in $\tau$ are either in the fresh heap or in the immutable heap.

The produced reference is inside the reachable object graph of $t_0$.

$t_0.f$ Following the same process used for case (F-UPDATE-T) we can show that both $t_0$ and $t_1$ are in the shared heap, thus we are not violating the balloon invariants, thanks to (MEM-T) last side condition.

$T \xrightarrow{\tau} t_1 \ e$ also the expression $e$ can be typed with (BALLOON-L-TP) in this environment.