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 [1] 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 [8]. 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 [1]. 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*,

---

[1] Our *balloon* (and also *fresh*) concept is similar to the externally unique object [5] except that we prohibit all (but immutable) objects inside a balloon from pointing to the outside of the 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 [1] using the object types and heap kinds from BI-Java:

I_1 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 this balloon reference (if it exists) is external to the reachable object graph from the balloon object itself;

I_3 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 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.

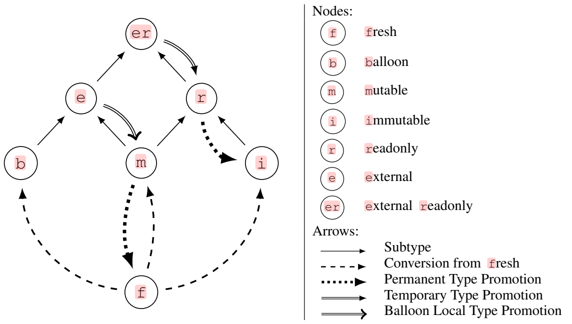

Figure 1. Object and reference kinds and possible relationships

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:

```
int h(b C x1, b C x2, i D x3, f G x4) {
  int x = x3.m(x1, x4);
  int y = new H().k(x2, x3);
  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 `D.m` and `H.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 parallisation 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 [8], 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 readonly reference. However, this does not prevent observational exposure [4] as readonly references can be stored in any (type compatible) field that has a readonly or external readonly type and thus any changes to the object can be *observed* by the referrer. The following example shows how `d` in `client` can observe a change to a readonly field `myC` because of the alias created inside the `action` method:

```
class C {int i;...}
class D {
  er C myC=new C(8);
  int inspect()er{return this.myC.i;}
  void action(r C c)m{
    int x=c.i*2;
    this.myC=c;
    //c.i=0;//Wrong since c is readonly
  }
  void client(m C c,m D d)m{
    c.i=1;
    d.action(c);//cannot modify c
    //here c.i==1 and d.inspect()==1
    c.i=2;
    /*here d.inspect()==2*/ }}
```

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 obtaining 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:

```
...
  void action(e C c)m{
    int x=c.i*2;
    //this.myC=c;//Wrong since c is external
    c.i=0;
  }
  void client(m C c, m D d)m{
    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:

```
...
  void action(er C c)m{
    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)m{
    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 external readonly 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 objected. 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.

```
class BarReader{
  m Bar read1(e Scanner s)er{...}
  f Bar usage1(m Scanner s)m{
    return this.read1(s);/*promotion happens here*/ }
  r Bar read2(e Scanner s)er{...}
  i Bar usage2(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.[2]

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:

---

[2] Conversions and permanent type promotions combined allow transparent field access and easy field update over fresh references.

```
void m(f Foo f1, f Foo f2)m{
  //NB! We are statically sure that f1!=f2
  f Foo f3=f1; //f1 is not accessible anymore
  i Foo f4=f3; //ok: conversion fresh ->immutable
  i Foo f5=f3; //ok: f3 is now Immutable
  b 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 4.

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

```
class Bar{//Shows how immutable works
  i Foo f;
  Bar(i Foo f) {this.f=f;}
  void m(i Foo f1,i Foo f2)m {
    m Bar b1=new Bar(f1);
    f1=f2;       //variable f1 Immutable, not final
    b1.f=f2;     //field f Immutable, not final
    i 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
  b Foo f;...
  void m(m Beer b1, f Foo f1)m{
    b1.f=f1;//ok conversion fresh->balloon
    e Beer b2=b1;//implicit cast mutable->external
    b Foo f2=b1.f;//mutable b1 exposes f as balloon
    e Foo f2=b2.f;//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{
  r Bar queryMethod(r Bar b1,r Bar b2)er{...}
  er Bar usage(b Bar b1,er Bar b2)er{
    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 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 loosely 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{
  m Bar mutatorMethod(m Bar b)er{ ... }
  e Bar usage(e Bar b)er{
    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; float y;
  Point(float x,float y){this.x=x;this.y=y;}
  boolean isNear(er Point p, float d)er{...}
}
class Triangle{
  m Point p1;  m Point p2;  m Point p3;
  Triangle(m Point p1,m Point p2,m Point p3){
    this.p1=p1;this.p2=p2;this.p3=p3; }
  boolean isOverlap(er Triangle t)er{...}
}
```

As you can see, methods `isNear` and `isOverlap` 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 fresh or an immutable one.

3.1 `Mesh`, and Two Ways to Add a `Triangle`

We assume vectors for mutable points (`VectorMutableP`) and triangles (`VectorMutableT`):

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

As you can see, methods `seekPt1` and `addTriangle1`, `seekPt2` 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`):

```
class Workbench{
  b  VectorBalloonM meshes;
  Workbench(f VectorBalloonM m){ this.meshes=m;}
  void addMesh() e { meshes.add(new Mesh());}

  void addTriangleGui(int mIndex,e Gui gui) e{
    e Mesh mExt=this.meshes.get(mIndex);
    m Mesh m=mExt;//balloon local type promotion
    m.addTriangle1(gui.tolerance(),gui.selectPoint(),
      gui.selectPoint(), gui.selectPoint());
  }
  void mergeMeshes(int mIndex1, int mIndex2)e{
    e Mesh m2=this.meshes.get(mIndex2);
    m Mesh m1=this.meshes.get(mIndex1);//balloon local
    for(er Triangle t: m2.triangles)
      m1.addTriangle2(0.15f,t.p1,t.p2,t.p3);
  }
```

```
  boolean isTriangleOverlap()er{
    f VectorReadonlyT v= new VectorReadonlyT();
    for(er m:this.meshes)
      for(er Triangle t: m.triangles)
        v.add(t);//v converted to mutable
        //t was promoted to readonly (temp promotion)
    while(!v.isEmpty()){
      er Triangle t1=v.pop();
      for(er Triangle t2:v)
        if(t1.isOverlap(t2)) return true;
    }
    return false;
  } }
```

The constructor shows how fresh 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 fresh 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 fresh.

```
class Reader {
  f Point readPoint(e Scanner s)er{
    return new Point(s.nextFloat(),s.nextFloat());}
  f Shape readMesh(e Scanner s)er{
    m Mesh m=new Mesh(
      new VectorMutableP(), new VectorMutableT());
    while(m.hasNextFloat())
      m.addTri1(0.15f,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]), fresh analysis, conventional typing and promotion rules. After defining the observable behaviour of BI-JAVA, in Section 5 we show how to transparently optimise a program using parallelisation.

4.1 Syntax

In Figure 2 we show the syntax of BI-JAVA. We assume countably infinite sets of variables x, class or interface names C, method

p	$::=$	$\overline{cd}\,\overline{id}$	program
cd	$::=$	$\textbf{class}\ C\ \textbf{implements}\ \overline{C}\ \texttt{\{}\ \overline{fd}\ \overline{md}\ \texttt{\}}$	class declaration
id	$::=$	$\textbf{interface}\ C\ \textbf{extends}\ \overline{C}\ \texttt{\{}\ \overline{mh\texttt{;}}\ \texttt{\}}$	interface declaration
fd	$::=$	$T\,f\texttt{;}$	field declaration
mh	$::=$	$T\,m\,\texttt{(}\overline{T\,x}\texttt{)}\,\mathcal{M}$	method header
md	$::=$	$mh\ e$	method declaration
e	$::=$	$x \mid e.m\,\texttt{(}\overline{e}\texttt{)} \mid \textbf{new}\ C\,\texttt{(}\overline{e}\texttt{)}$	
		$\mid e_0.f \mid e_0.f\texttt{=}e_1 \mid T\,x = e_0\texttt{;}\ e_1$	expression
T	$::=$	$\mathcal{M}C$	type
$\mathcal{M}$	$::=$	$\texttt{f} \mid \texttt{b} \mid \texttt{m} \mid \texttt{i} \mid \texttt{e} \mid \texttt{r} \mid \texttt{er}$	type modifiers

Figure 2. Syntax

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 $T\,x = e_0\texttt{;}\,e_1$; 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 $\mathcal{M}$ and a class or interface name C. A modifier $\mathcal{M}$ have seven possible values: `f` (for fresh), `b` (for balloon), `m` (for mutable), `i` (for immutable), `e` (for external), `r` (for readonly) and `er` (for external readonly).

4.2 Reduction

Figure 3 shows reduction rules for BI-JAVA (without parallelism). In order to show reduction, we have to enrich the expressions with object identifiers ι 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 $\mathcal{M}$ are completely ignored.

Rule (CTX) is standard. Rule (F-ACCESS) extracts the value of field f. We use the notation $\mu(\iota).f$ for the value of field f of object ι. Rule (CONSTRUCTOR) reduces constructor invocations.

e	$::=$	$\ldots \mid \iota$	expressions (also run time)
μ	$::=$	$\overline{\iota \mapsto C(f_1 = \iota_1, \ldots, f_n = \iota_n)}$	memory
$\mathcal{E}$	$::=$	$\Box \mid \mathcal{E}.m\,\texttt{(}\overline{e}\texttt{)} \mid \iota.m\,\texttt{(}\overline{\iota}, \mathcal{E}, \overline{e}\texttt{)}$	
		$\mid \textbf{new}\ C\,\texttt{(}\overline{\iota}, \mathcal{E}, \overline{e}\texttt{)} \mid T\,x = \mathcal{E}\texttt{;}\,e$	
		$\mid \mathcal{E}.f \mid \mathcal{E}.f\texttt{=}e \mid \iota.f\texttt{=}\mathcal{E}$	context

$$\boxed{\mu_1 \mid e_1 \ \rightarrow\ \mu_2 \mid e_2}$$

$$(\textsc{ctx})\quad \frac{\mu_1 \mid e_1 \ \rightarrow\ \mu_2 \mid e_2}{\mu_1, \mu \mid \mathcal{E}[e_1] \ \rightarrow\ \mu_2, \mu \mid \mathcal{E}[e_2]}$$

$$(\textsc{f-access})\quad \frac{}{\mu \mid \iota_0.f \ \rightarrow\ \mu \mid \iota_1}\quad \text{with } \mu(\iota_0).f = \iota_1$$

$$(\textsc{constructor})\quad \frac{}{\mu \mid \textbf{new}\ C\,\texttt{(}\iota_1\texttt{,}\ \ldots\texttt{,}\,\iota_n\texttt{)} \ \rightarrow\ \mu, \iota \mapsto C(f_1 = \iota_1, \ldots, f_n = \iota_n) \mid \iota}$$
with
$$\iota \notin \mathsf{dom}(\mu)$$
$$p(C) = \textbf{class}\ C\ \textbf{implements}\,\_\ \texttt{\{}\,\_f_1\texttt{;}\ \ldots\texttt{;}\_f_n\texttt{;}\ \overline{md}\ \texttt{\}}$$

$$(\textsc{m-invk})\quad \frac{}{\mu \mid \iota_0.m\,\texttt{(}\iota_1\texttt{,}\ \ldots\texttt{,}\,\iota_n\texttt{)} \ \rightarrow\ \mu \mid e[x_1 = \iota_1, \ldots, x_n = \iota_n, \textbf{this} = \iota_0]}$$
with
$$\mu(\iota) = C(\_)$$
$$p(C).m = \_\,m\,\texttt{(}\_x_1\texttt{,}\ \ldots\texttt{,}\_x_n\texttt{)}\,\_\ e$$

$$(\textsc{v-dec})\quad \frac{}{\mu \mid T\,x = \iota\texttt{;}\,e \ \rightarrow\ \mu \mid e[x = \iota]}$$

$$(\textsc{f-update})\quad \frac{}{\mu \mid \iota_0.f\texttt{=}\iota_1 \ \rightarrow\ \mu[\iota_0.f = \iota_1] \mid \iota_1}$$

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 $e[x_1 = \iota_1, \ldots, x_n = \iota_n]$ for variable substitution, that is, we simultaneously replace all the occurrences of x_i in e with ι_i. Finally, Rules (V-DEC) and (F-UPDATE) are straightforward. We use notation $\mu[\iota_0.f = \iota_1]$ to override the heap location $\iota_0.f$ with the value ι_1.

4.3 Fresh Analysis and Γ

Figure 4 shows syntax for extended types, type environments and the split relation $\Gamma = \Gamma_1 \circ \ldots \circ \Gamma_n$ as explained below.

Free variables are typed using a variable environment Γ. The treatment is conventional for all variable kinds but fresh ones. If a 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 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 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 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 $\texttt{m}\langle\overline{\textit{TP}}\rangle$ (where the metavariable $\textit{TP}$ ranges over the three different promotion kinds) with the following meaning: if $\Gamma(x) = \texttt{m}\langle\overline{\textit{TP}}\rangle\ C$ 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 $\overline{\textit{TP}}$. Thus `m` is just a syntactic sugar for $\texttt{m}\langle\emptyset\rangle$.

The programmer cannot use types of the form $\texttt{m}\langle\overline{\textit{TP}}\rangle\ C$ with non empty $\overline{\textit{TP}}$. They are only used internally by the type system for typing expressions with many nested promotions.

$$
\begin{array}{lcll}
\mathcal{M} & ::= & \ldots \mid \mathtt{m}\langle\overline{\mathit{TP}}\rangle & \text{modifiers (also tracing fresh)}\\
\mathit{TP} & ::= & \textsc{ptp} \mid \textsc{ttp} \mid \textsc{bltp} & \text{promotion kinds}\\
\Gamma & ::= & \overline{x{:}T} & \text{variable environment}
\end{array}
$$

$$\boxed{\Gamma = \Gamma_0 \circ \Gamma_1}$$

$$
(\textsc{unrestricted})\quad \frac{\Gamma = \Gamma_0 \circ \Gamma_1}{\Gamma, x{:}T = \Gamma_0, x{:}T \circ \Gamma_1, x{:}T}\ \text{with } \mathsf{unrestricted}(T)
\qquad
(\textsc{empty})\quad \frac{}{\emptyset = \emptyset \circ \emptyset}
$$

$$
(\textsc{swap})\quad \frac{\Gamma = \Gamma_0 \circ \Gamma_1}{\Gamma = \Gamma_1 \circ \Gamma_0}
\qquad
(\textsc{affine})\quad \frac{\Gamma = \Gamma_0 \circ \Gamma_1}{\Gamma, x{:}T = \Gamma_0, x{:}T \circ \Gamma_1}\ \text{with } \mathsf{affine}(T)
$$

$$
(\textsc{affine-ext})\quad \frac{\Gamma = \Gamma_0 \circ \Gamma_1}{\Gamma, x{:}\mathcal{M}\,C = \Gamma_0, x{:}\mathcal{M}\,C \circ \Gamma_1, x{:}\mathtt{e}\,C}\ \text{with } \mathsf{affine\text{-}external}(\mathcal{M}\,C)
$$

Figure 4. Variable environment split

$$\boxed{\vdash cd : \mathrm{ok} \qquad \vdash id : \mathrm{ok}}$$

$$
(\textsc{class})\quad \frac{C_0 \vdash md_i : \mathrm{ok} \quad \forall i \in 1..n}{\vdash \textbf{class}\ C_0\ \textbf{implements}\ \overline{C}\ \texttt{\{}\ \overline{fd}\ md_1 \ldots md_n\ \texttt{\}} : \mathrm{ok}}
$$
$$
\begin{array}{l}
\text{with}\\
\quad \forall C \in \overline{C}, m \text{ s.t. } p(C).m = T\,m\,\texttt{(}\overline{T\,x}\texttt{)}\,\mathcal{M}\texttt{;},\\
\qquad T\,m\,\texttt{(}\overline{T\,x}\texttt{)}\,\mathcal{M}\_ \in md_1 \ldots md_n \text{ holds}
\end{array}
$$

$$
(\textsc{interface})\quad \frac{}{\vdash \textbf{interface}\ C_0\ \textbf{extends}\ \overline{C}\ \texttt{\{}\ mh_1 \ldots mh_n\ \texttt{\}} : \mathrm{ok}}
$$
$$
\begin{array}{l}
\text{with}\\
\quad \forall C \in \overline{C}, m \text{ s.t. } p(C).m = T\,m\,\texttt{(}\overline{T\,x}\texttt{)}\,\mathcal{M}\texttt{;},\\
\qquad T\,m\,\texttt{(}\overline{T\,x}\texttt{)}\,\mathcal{M}\texttt{;} \in mh_1 \ldots mh_n \text{ holds}
\end{array}
$$

$$\boxed{C \vdash mh\_ : \mathrm{ok}}$$

$$
(\textsc{meth-t})\quad \frac{\textbf{this}{:}T'_0, x_1{:}T'_1, \ldots, x_n{:}T'_n \vdash e : \_ \leq T}{C \vdash T\,m\,\texttt{(}T_1\,x_1\texttt{,}\ \ldots\texttt{,}\ T_n\,x_n\texttt{)}\,\mathcal{M}\,e : \mathrm{ok}}
$$
$$
\begin{array}{l}
\text{with}\\
\quad T_0 = \mathcal{M}C\\
\quad T'_i = T_i \text{ or } T_i = \mathtt{f}\,C_i \text{ and } T'_i \in \{\mathtt{b}\,C_i, \mathtt{i}\,C_i, \mathtt{m}\langle\overline{\mathit{TP}_i}\rangle\,C_i\} \quad \forall i \in 0..n
\end{array}
$$

Figure 5. Typing for Classes and Methods

In order to formalise the precise way fresh and $\mathtt{m}\langle\overline{\mathit{TP}}\rangle$ references have to be 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 mutable inside only temporary type promotions ($\mathtt{m}\langle\overline{\textsc{ttp}}\rangle$) are also unrestricted.
  Formally $\mathsf{unrestricted}(\mathcal{M}\ C)$ holds iff $\mathcal{M} \in \{\mathtt{b}, \mathtt{m}\langle\overline{\textsc{ttp}}\rangle, \mathtt{i}, \mathtt{e}, \mathtt{r}, \mathtt{er}\}$;
- affine variables can be used at most one time in an expression; fresh variables are affine, fresh variable converted in mutable inside at least one permanent type promotions are also affine.
  Formally $\mathsf{affine}(\mathcal{M}\ C)$ holds iff $\mathcal{M} = \mathtt{f}$ or $\mathcal{M} = \mathtt{m}\langle\overline{\mathit{TP}}\rangle$ and $\textsc{ptp} \in \overline{\mathit{TP}}$;
- 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.
  Formally $\mathsf{affine\text{-}external}(\mathcal{M}\ C)$ holds iff $\mathcal{M} = \mathtt{m}\langle\overline{\textsc{ttp}}\rangle$ and $\textsc{bltp} \in \overline{\mathit{TP}}$ but $\textsc{ptp} \notin \overline{\mathit{TP}}$.

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 5 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 $\overline{C}$ 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 $\overline{C}$ are correctly implemented; that is, for all the 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 $\mathcal{M}$. Note how this rule introduces the fresh variables inside Γ: 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 $T_1 \leq T_2$.

Figure 6 shows conventional typing rules for expressions. Rule (VAR-T) is conventional.

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

$\mathsf{f\text{-}access}_{\mathtt{m}}(\mathcal{M}) = \mathcal{M} \quad \mathsf{f\text{-}access}_{\mathtt{i}}(\_) = \mathtt{i} \quad \mathsf{f\text{-}access}\_(\mathtt{i}) = \mathtt{i}$
$\mathsf{f\text{-}access}_{\mathtt{b}}(\mathcal{M}) = \mathsf{f\text{-}access}_{\mathtt{e}}(\mathcal{M})$
$\mathsf{f\text{-}access}_{\mathtt{e}}(\mathcal{M}) = \mathtt{e}$ with $\mathcal{M} \in \{\mathtt{b}, \mathtt{e}, \mathtt{m}\}$
$\mathsf{f\text{-}access}_{\mathtt{e}}(\mathcal{M}) = \mathtt{er}$ with $\mathcal{M} \in \{\mathtt{r}, \mathtt{er}\}$
$\mathsf{f\text{-}access}_{\mathtt{r}}(\mathcal{M}) = \mathtt{r}$ with $\mathcal{M} \in \{\mathtt{m}, \mathtt{r}\}$
$\mathsf{f\text{-}access}_{\mathtt{r}}(\mathcal{M}) = \mathtt{er}$ with $\mathcal{M} \in \{\mathtt{b}, \mathtt{e}, \mathtt{er}\}$
$\mathsf{f\text{-}access}_{\mathtt{er}}(\mathcal{M}) = \mathtt{er}$ with $\mathcal{M} \in \{\mathtt{b}, \mathtt{m}, \mathtt{e}, \mathtt{r}, \mathtt{er}\}$

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 balloon fields. In this case to force correct field update it is enough to state that the assigned value is not balloon, external or external readonly. Indeed, references of these kinds cannot be assigned. Rule (F-UPDATE-TB) covers the case of 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 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 $\mathsf{l\text{-}close}(\_)$ that returns a loosely closed type for any reference kind as follows:

$$\boxed{\Gamma \vdash e : T}$$

(VAR-T)
$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

(F-ACCESS-T)
$$\frac{\Gamma \vdash e_0 : \mathcal{M}_0\; C_0}{\Gamma \vdash e_0.f \;:\; \textsf{f-access}_{\mathcal{M}_0}(\mathcal{M})\; C}$$
with
$p(C_0) = \textbf{class}\; C_0\; \textbf{implements}\,\_\,\texttt{\{}\;\_\,\mathcal{M}\; C\; f\texttt{;}\;\_\;\texttt{\}}$

(M-INVK-T)
$$\frac{\begin{array}{c}\Gamma_0 \vdash e_0 : \_\; C_0 \le \mathcal{M}\; C_0\\ \Gamma_i \vdash e_i : \_ \le T_i \quad \forall i = 1..n\end{array}}{\Gamma_0 \circ \ldots \circ \Gamma_n \vdash e_0.m\texttt{(}e_1\texttt{,}\; \ldots\texttt{,}\; e_n\texttt{)} \;:\; T}$$
with
$p(C_0).m = T\; m\texttt{(}T_1\, x_1\texttt{,}\; \ldots\texttt{,}\; T_n\, x_n\texttt{)}\; \mathcal{M}\,\_$

(F-UPDATE-T)
$$\frac{\begin{array}{c}\Gamma_0 \vdash e_0 : \mathtt{m}\; C_0\\ \Gamma_1 \vdash e_1 : \mathcal{M}\; C \le \mathcal{M}_1\; C_1\end{array}}{\Gamma_0 \circ \Gamma_1 \vdash e_0.f\texttt{=}e_1 : \mathcal{M}\; C}$$
with
$p(C_0) = \textbf{class}\; C_0\; \textbf{implements}\,\_\,\texttt{\{}\;\_\,\mathcal{M}_1\; C_1\; f\texttt{;}\;\_\;\texttt{\}}$
$\mathcal{M} \notin \{\mathtt{b}, \mathtt{e}, \mathtt{er}\}$

(F-UPDATE-TB)
$$\frac{\begin{array}{c}\Gamma_0 \vdash e_0 : \mathtt{m}\; C_0\\ \Gamma_1 \vdash e_1 : \mathtt{f}\; C \le \mathtt{f}\; C_1\end{array}}{\Gamma_0 \circ \Gamma_1 \vdash e_0.f\texttt{=}e_1 : \mathcal{M}\; C}$$
with
$p(C_0) = \textbf{class}\; C_0\; \textbf{implements}\,\_\,\texttt{\{}\;\_\,\mathtt{b}\; C_1\; f\texttt{;}\;\_\;\texttt{\}}$

(NEW-T)
$$\frac{\begin{array}{c}\Gamma_i \vdash e_i : \mathcal{M}_i\; C_i \le \mathcal{M}'_i\; C'_i \quad \forall i = 1..n \text{ where } \mathcal{M}'_i \neq \mathtt{b}\\ \Gamma_i \vdash e_i : \mathtt{f}\; C_i \le \mathtt{f}\; C'_i \quad \forall i = 1..n \text{ where } \mathcal{M}'_i = \mathtt{b}\end{array}}{\Gamma_1 \circ \ldots \circ \Gamma_n \vdash \textbf{new}\; C\,\texttt{(}e_1 \ldots e_n\texttt{)} : \mathtt{m}\; C}$$
with
$p(C) = \textbf{class}\; C\; \textbf{implements}\,\_\,\texttt{\{}\; \mathcal{M}'_1\; C'_1\,\_\texttt{;}\; \ldots\texttt{;}\mathcal{M}'_n\; C'_n\,\_\texttt{;}\; \overline{md}\;\texttt{\}}$
$\mathcal{M}_i \notin \{\mathtt{b}, \mathtt{e}, \mathtt{er}\} \quad \forall i \in 1..n$

(V-DEC-T)
$$\frac{\begin{array}{c}\Gamma_0 \vdash e_0 : \mathcal{M}\; C_0 \le T_0\\ \Gamma_1, x{:}T \vdash e_1 : T_1\end{array}}{\Gamma_0 \circ \Gamma_1 \vdash T_0\; x\texttt{=}e_0\texttt{;}\; e_1 : T_1}$$
with
either $T = T_0$
or $T_0 = \mathtt{f}\; C$ and $T \in \{\mathtt{b}\; C, \mathtt{i}\; C, \mathtt{m}\langle\overline{TP}\rangle\; C\}$

Figure 6. Type system expressions

$\textsf{l-close}(\mathtt{m}\; C) = \mathtt{e}\; C$
$\textsf{l-close}(\mathtt{r}\; C) = \mathtt{er}\; C$
$\textsf{l-close}(T) = T$ otherwhise

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 $\textsf{l-close-p}(\Gamma)$, that behave like $\textsf{l-close}(\Gamma)$ but is extended in order to manage the types of the form $\mathtt{m}\langle{\scriptstyle\mathtt{PTP}}, \overline{TP}\rangle\; C$.
Formally: $\textsf{l-close-p}(\mathtt{m}\langle{\scriptstyle\mathtt{PTP}}, \overline{TP}\rangle\; C) = \mathtt{m}\langle\overline{TP}\rangle\; C$ and $\textsf{l-close-p}(T) = \textsf{l-close}(T)$ otherwise.

Rule (TEMPORARY-TP) models temporary type promotion. As you can see, many variable of (a supertype of) an external readonly type can be promoted to readonly type. We use function $\textsf{l-close-t}(\Gamma)$, that behave like $\textsf{l-close}(\Gamma)$ but is extended in order to manage the types of the form $\mathtt{m}\langle{\scriptstyle\mathtt{TTP}}, \overline{TP}\rangle\; C$.
Formally: $\textsf{l-close-t}(\mathtt{m}\langle{\scriptstyle\mathtt{TTP}}, \overline{TP}\rangle\; C) = \mathtt{m}\langle\overline{TP}\rangle\; C$ and $\textsf{l-close-t}(T) = \textsf{l-close}(T)$ otherwise.

Rule (BALLOON-L-TP) models balloon local type promotion. It promotes a variable x_0 to mutable if it is (a supertype of an) external reference. We use function $\textsf{l-close-b}(\Gamma)$, that behave like $\textsf{l-close}(\Gamma)$ but is extended in order to manage the types of the form $\mathtt{m}\langle{\scriptstyle\mathtt{BLTP}}, \overline{TP}\rangle\; C$, and to ensure that balloon references are converted in external ones.
Formally: $\textsf{l-close-b}(\mathtt{m}\langle{\scriptstyle\mathtt{BLTP}}, \overline{TP}\rangle\; C) = \mathtt{m}\langle\overline{TP}\rangle\; C$, $\textsf{l-close-b}(\mathtt{b}\; C) = \mathtt{e}\; C$ and $\textsf{l-close-b}(T) = \textsf{l-close}(T)$ otherwise.

$$\boxed{\Gamma \vdash e : T}$$

(PERMANENT1-TP)
$$\frac{\textsf{l-close-p}(\Gamma) \vdash e : \mathtt{m}\; C}{\Gamma \vdash e : \mathtt{f}\; C}$$

(PERMANENT2-TP)
$$\frac{\textsf{l-close-p}(\Gamma) \vdash e : \mathtt{r}\; C}{\Gamma \vdash e : \mathtt{i}\; C}$$

(TEMPORARY-TP)
$$\frac{\textsf{l-close-t}(\Gamma), x_1{:}\mathtt{r}\; C_1, \ldots x_n{:}\mathtt{r}\; C_n \vdash e : \mathcal{M}\; C}{\Gamma, x_1{:}\mathcal{M}_1\; C_1, \ldots x_n{:}\mathcal{M}_n\; C_n \vdash e : \textsf{l-close}(\mathcal{M})\; C}$$
with
$\mathcal{M}_i\; C_i \le \mathtt{er}\; C \quad \forall i \in 1..n$

(BALLOON-L-TP)
$$\frac{\textsf{l-close-b}(\Gamma), x_0{:}\mathtt{m}\; C_0 \vdash e : \mathcal{M}\; C}{\Gamma, x_0{:}\mathcal{M}_0\; C_0 \vdash e : \textsf{l-close}(\mathcal{M})\; C}$$
with
$\mathcal{M}_0\; C_0 \le \mathtt{e}\; C_0$

(SUB-EXP-T)
$$\frac{\begin{array}{c}\Gamma \vdash e_0 : T_0\\ \Gamma, x_0{:}T'_0 \vdash \mathcal{E}[x_0] : T\end{array}}{\Gamma \vdash \mathcal{E}[e_0] : T}$$
with
either $T'_0 = T_0$
or $T_0 = \mathtt{f}\; C_0$ and
$T'_0 \in \{\mathtt{b}\; C_0, \mathtt{i}\; C_0, \mathtt{m}\langle\overline{TP}\rangle\; C_0\}$

Figure 7. Type system special

Note how in all the three variation of $\textsf{l-close}(\_)$ the other kinds of modifiers of the form $\mathtt{m}\langle\overline{TP}\rangle$ 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.

4.6 Soundness

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 8 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.

Rule (LOC-T) models typing of object identifiers and allows us to design the rest of the type system for the language without object identifiers, so that we can avoid tedious duplication of the definition of Γ split operator. Moreover, promotions are simpler in the language without object identifiers. In a conventional language rule (LOC-T) would make the proof non trivial, however in BI-JAVA the same issues are already caused by (SUB-EXP-T). A careful reader could notice how rule (LOC-T) is just a specialisation of a set of applications of (SUB-EXP-T) over subexpression of the form ι.

We introduce typing rule (MEM-T) stating when a memory is well formed under a stack. It uses judgement $\mu \vdash \iota$ to check all object identifiers to be well formed with respect to the memory. Formally $\mu \vdash \iota$ holds iff $\mu(\iota) = C(f_1\texttt{=}\iota_1 \ldots f_1\texttt{=}\iota_n)$, $\forall i \in 1..n$ $\mu(\iota_i) = C_i(\_)$, $p(C).f_i = \mathcal{M}\; C'_i$ and $\mathcal{M}\; C_i \le \mathcal{M}\; C'_i$
As you can see this judgement does not check that location ι actually refers to an object compatible with the type modifiers $\mathcal{M}$.

In order to define validity of immutable ($\textsf{valid-I}_{\mu}(\iota_0)$), fresh ($\textsf{valid-F}^{\Sigma}_{\mu}(\iota_0)$) and balloon ($\textsf{valid-B}^{\Sigma}_{\mu}(\iota_0)$) invariant we introduce the following *link* notation:

$\Sigma ::= \overline{\iota{:}T}$ stack

$$\boxed{\Gamma;\Sigma \vdash e : T}$$

(LOC-T)
$$\frac{\Gamma, x_1{:}T_1 \ldots x_n{:}T_n \vdash e : T}{\Gamma; \iota_1{:}T_1 \ldots \iota_n{:}T_n \vdash e[x_1 = \iota_1 \ldots x_n = \iota_n] : T}$$

$$\boxed{\Sigma \vdash \mu}$$

(MEM-T)
$$\frac{}{\Sigma \vdash \mu}$$
with
$$\begin{array}{l}
\Sigma = \iota_1{:}T_1 \ldots \iota_n{:}T_n\\
\mu \vdash \iota \quad \forall \iota \in \mathsf{dom}(\mu)\\
\textsf{valid-I}_\mu(\iota) \quad \forall \iota \in \{\iota_i \mid i \in 1..n,\ T_i = \mathtt{i}\_\}\\
\textsf{valid-F}^\Sigma_\mu(\iota) \quad \forall \iota \in \{\iota_i \mid i \in 1..n, \textsf{affine}(T_i)\\
\textsf{valid-B}_\mu(\iota) \quad \forall \iota \in \mathsf{BC}^\Sigma_\mu\\
\textsf{rog-I}_\mu(\iota_1) \cap \textsf{rog-I}_\mu(\iota_2) = \emptyset \quad \forall \iota_1, \iota_2 \in \mathsf{RB}_\mu(\Sigma), \iota_1 \neq \iota_2\\
\textsf{rog-I}_\mu(\iota_1) \cap \mathsf{SH}^\Sigma_\mu \neq \emptyset \text{ implies } \textsf{rog-I}_\mu(\iota_2) \cap \mathsf{SH}^\Sigma_\mu = \emptyset \quad \forall \iota_1, \iota_2 \in \mathsf{BC}^\Sigma_\mu, \iota_1 \neq \iota_2
\end{array}$$

$$\boxed{\Sigma_0 \rightsquigarrow \Sigma_1 \vdash \mu_0 \rightsquigarrow \mu_1}$$

(MEM-MEM-OK)
$$\frac{\Sigma, \Sigma_0 \vdash \mu_0 \qquad \Sigma, \Sigma_1 \vdash \mu_1}{\Sigma, \Sigma_0 \rightsquigarrow \Sigma, \Sigma_1 \vdash \mu_0 \rightsquigarrow \mu_1}$$
with
$$\begin{array}{l}
\mathsf{dom}(\Sigma_0) \cap \mathsf{dom}(\Sigma_1) = \emptyset\\
\mathsf{IH}_{\mu_0} \subseteq \mathsf{IH}_{\mu_1}\\
\mu_0(\iota) = \mu_1(\iota) \quad \forall \iota \in \mathsf{IH}_{\mu_0}\\
\textsf{rog-I}_{\mu_0}(\iota) \cap \textsf{rog-I}_{\mu_0}(\iota') = \emptyset\\
\quad \text{implies } \textsf{rog-I}_{\mu_1}(\iota) \cap (\textsf{rog-I}_{\mu_1}(\iota') \cup \textsf{rog-I}_{\mu_0}(\iota')) = \emptyset \quad \forall \iota, \iota' \in \mathsf{BC}^\Sigma_{\mu_0}
\end{array}$$

Figure 8. Typing memory and locations

- $\iota_1.f \xrightarrow[\mu]{\overline{\mathcal{M}}} \iota_2$ iff $\mu(\iota_1) = C(\_ f = \iota_2)$ and $p(C).f = \mathcal{M}\_$ that reads as "there is a link named f of kind $\mathcal{M} \in \overline{\mathcal{M}}$ between ι_1 and ι_2" For example $\iota_1.f \xrightarrow[\mu]{\mathtt{b}} \iota_2$ means that ι_1 have a field f of kind balloon referring to ι_2.

We generalise the link notion in the following way:

- If we omit the field name, we refer of a generic reference without keeping in account the field name, as in $\iota_1 \xrightarrow[\mu]{\mathtt{i},\mathtt{r}} \iota_2$ that reads as "there is a link of kind $\mathtt{i}$ or $\mathtt{r}$ between ι_1 and ι_2"
- If we omit $\overline{\mathcal{M}}$ then we refer to a generic reference without keeping in account the kind of the reference, as in $\iota_1 \xrightarrow[\mu]{} \iota_2$ iff $\mu(\iota_1) = C(\_ f = \iota_2)$ that reads as "there is a link between ι_1 and ι_2".
- We use $*$ to indicate reflexive and transitive closure, as usual. For example $\iota_1 \xrightarrow[\mu]{*\mathtt{m},\mathtt{r}} \iota_2$ reads as "there is a path composed by links of kind $\mathtt{m}$ or $\mathtt{r}$ between ι_1 and ι_2"

Now we can introduce the immutable invariant:
$\textsf{valid-I}_\mu(\iota_0)$ iff $\forall f, \iota_1 \in \textsf{rog}_\mu(\iota_0), \iota_2 \in \mathsf{dom}(\mu) \setminus \textsf{rog}_\mu(\iota_0)$,
$(\iota_2.f \xrightarrow[\mu]{} \iota_1$ implies $\iota_2.f \xrightarrow[\mu]{\mathtt{i},\mathtt{r},\mathtt{er}} \iota_1\ )$
where $\textsf{rog}_\mu(\iota)$ denotes the reachable object graph of ι.
As you can see, an object is immutable if all the reference to it in the heap are a supertype of immutable. The type of circular references inside the immutable object graph are not relevant ($\iota_2 \in \mathsf{dom}(\mu) \setminus \textsf{rog}_\mu(\iota_0)$).
The immutable heap IH_μ is simply composed by all the valid immutable objects: $\mathsf{IH}_\mu = \{\iota \in \mathsf{dom}(\mu) \mid \textsf{valid-I}_\mu(\iota)\}$
Now we can introduce the fresh invariant:
$\textsf{valid-F}^\Sigma_\mu(\iota_0)$ iff $\forall f, \iota_1 \in \textsf{rog-I}_\mu(\iota_0)$ $(\iota_1.f \xrightarrow[\mu]{} \iota_0$ implies $\iota_1.f \xrightarrow[\mu]{\mathtt{m},\mathtt{e},\mathtt{r},\mathtt{er}} \iota_0)$ and $\forall \iota_2 \in \textsf{rog-I}_\mu(\iota_0), \iota_3 \in \mathsf{dom}(\mu)$ $(\iota_3 \xrightarrow[\mu]{} \iota_2$ implies $\iota_3 \in \textsf{rog-I}_\mu(\iota_0)$ or $\textsf{gc}^\Sigma_\mu(\iota_3))$
with $\textsf{rog-I}_\mu(\iota_0) = \{\iota \in \textsf{rog}_\mu(\iota_0) \mid \text{not } \textsf{valid-I}_\mu(\iota)\}$ and predicate $\textsf{gc}^\Sigma_\mu(\iota)$ holds if ι is ready to be garbage collected.
As you can see, an object is fresh if all the reachable object graph (using $\textsf{rog-I}_\mu(\iota_0)$ to avoid keeping in consideration immutable objects) is only internally aliased ($\iota_3 \in \textsf{rog-I}_\mu(\iota_0)$), and all the (internal) links to the fresh object itself are of mutable kind or subtype thereof.

The fresh heap FH^Σ_μ is composed by all the fresh objects and their reachable object graphs
$\mathsf{FH}^\Sigma_\mu = \{\iota_0 \in \textsf{rog-I}_\mu(\iota_1) \mid \iota_1 \in \mathsf{dom}(\mu), \textsf{valid-F}^\Sigma_\mu(\iota_1)\}$
An object ι_0 is a valid ballon, that is $\textsf{valid-B}^\Sigma_\mu(\iota_0)$ iff

- at most one balloon reference exists to the balloon object, and is outside the balloon itself: $\forall f_1, f_2, \iota_1, \iota_2 \in \mathsf{dom}(\mu) \setminus \mathsf{IH}_\mu$ $(\iota_1.f_1 \xrightarrow[\mu]{\mathtt{b}} \iota_0$ and $\iota_2.f_2 \xrightarrow[\mu]{\mathtt{b}} \iota_0$ implies $\iota_1 = \iota_2$, $f_1 = f_2$ and $\iota_1 \notin \textsf{rog}_\mu(\iota_0))$
- all the internal aliasing of the balloon object itself are of a mutable kind or supertype thereof: $\forall f, \iota_2 \in \textsf{rog}_\mu(\iota_0)$ $(\iota_2.f \xrightarrow[\mu]{} \iota_0$ implies $\iota_2.f \xrightarrow[\mu]{\mathtt{m},\mathtt{e},\mathtt{r},\mathtt{er}} \iota_0)$
- finally, every balloon has a tree shape: that is, all the direct subballoons are disjoint $\forall \iota_1, \iota_2 \in \mathsf{RB}_\mu(\iota_0)$ such that $\iota_1 \neq \iota_2$, $\textsf{rog-I}_\mu(\iota_1) \cap \textsf{rog-I}_\mu(\iota_2) = \emptyset$ holds.

Where $\mathsf{RB}_\mu(\iota_0) = \{\iota \in \mathsf{dom}(\mu) \mid \iota_0 \xrightarrow[\mu]{*\mathtt{m}} \iota_1, \iota_1 \xrightarrow[\mu]{\mathtt{b}} \iota\}$
and BC^Σ_μ, balloon candidates are objects identifiers for witch a balloon reference exists in the memory or on the stack:
$\mathsf{BC}^\Sigma_\mu = \{\iota_0 \mid \iota_1 \in \mathsf{dom}(\mu) \setminus \mathsf{IH}_\mu, \iota_1 \xrightarrow[\mu]{\mathtt{b}} \iota_0\} \cup \{\iota \mid \Sigma(\iota) = \mathtt{b}\_\}$

Finally, in (MEM-T) 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 $\iota \in \mathsf{RB}_\mu(\Sigma)$ iff $\Sigma(\iota) = \mathtt{b}\_$ or $\iota \in \mathsf{RB}_\mu(\iota_0)$ with $\Sigma(\iota_0) = \mathtt{m}\_$, and with the following definition of shared heap:
$\mathsf{SH}^\Sigma_\mu = \{\iota \in \mathsf{dom}(\mu) \mid \Sigma(\iota_0) = \mathtt{m}\_, \iota_0 \xrightarrow[\mu]{*\mathtt{m}} \iota\}$.

Finally, rule (MEM-MEM-OK) verifies that the invariants are preserved during reduction: that is, a memory μ_1 can be the result of an operation over μ_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 μ_0 then the first balloon in μ_1 is disjoint with the second balloon, either if considered in μ_0 or μ_1.

Theorem 1 (soundness).
If p is well typed, $\emptyset \vdash e : T$ *and* $\emptyset \mid e \xrightarrow{*} \mu_0 \mid e_0$ *then either*

- $e_0 = \iota$, $\iota{:}T_0 \vdash \mu_0$ *and* $T_0 \leq T$, *or*
- $\mu_0 \mid e_0 \rightarrow \mu_1 \mid e_1$, $\quad \Sigma_0 \rightsquigarrow \Sigma_1 \vdash \mu_0 \rightsquigarrow \mu_1$,
  $\emptyset; \Sigma_0 \vdash e_0 : T_0 \leq T$ *and* $\emptyset; \Sigma_1 \vdash e_1 : T_1 \leq T_0$

As usual $\_ \xrightarrow{*} \_$ 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* $\emptyset \vdash \mu \mid e : T$
then either $e = \iota$ *or* $\mu \mid e \rightarrow \_ \mid \_$.

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

Theorem 3 (subject reduction).
If p is well typed, $\emptyset; \Sigma_0 \vdash e_0 : T_0$, $\Sigma_0 \vdash \mu_0$ *and* $\mu_0 \mid e_0 \rightarrow \mu_1 \mid e_1$
then $\Sigma_0 \rightsquigarrow \Sigma_1 \vdash \mu_0 \rightsquigarrow \mu_1$ *and* $\emptyset; \Sigma_1 \vdash e_1 : T_1 \leq T_0$

Proof. The proof is in the companion technical report [16].

$$
\begin{array}{llll}
e & ::= & \ldots \mid \mathtt{\{}\overline{J}\mathtt{\}}\ \textbf{in}\ e & \text{expressions (also fork-join)}\\
J & ::= & T\ x = e\mathtt{;}\ \mathtt{[}\overline{dep}\mathtt{]} & \text{Job}\\
dep & ::= & x\ \textbf{if}\ L & \text{dependencies}\\
L & ::= & \textbf{true} \mid \textbf{false} & \text{logic constants}\\
 & & \mid L_1 \vee L_2 \mid \pi_1 \equiv \pi_2 & \text{logic operations}\\
\pi & ::= & x \mid \iota & \\
\mathcal{E} & ::= & \Box \mid \mathcal{E}.m\,\mathtt{(}\overline{e}\mathtt{)} \mid \iota.m\,\mathtt{(}\overline{\iota},\mathcal{E},\overline{e}\mathtt{)} & \\
 & & \mid \textbf{new}\ C\,\mathtt{(}\overline{\iota},\mathcal{E},\overline{e}\mathtt{)} \mid T\ x = \mathcal{E}\mathtt{;}\ e & \\
 & & \mid \mathcal{E}.f \mid \mathcal{E}.f{=}e \mid \iota.f{=}\mathcal{E} & \\
 & & \mid \mathtt{\{}\overline{J}_1\ T\ x = \mathcal{E}\mathtt{;}\ \mathtt{[\,]}\ \overline{J}_2\mathtt{\}}\ \textbf{in}\ e & \text{context}
\end{array}
$$

$$\boxed{\mu_1 \mid e_1 \;\rightarrow\; \mu_2 \mid e_2}$$

$$\text{(OUT-E)}\quad \frac{}{\mu \mid \mathtt{\{\}}\ \textbf{in}\ e \;\rightarrow\; \mu \mid e}$$

$$\text{(COMMIT)}\quad \frac{}{\mu \mid \mathtt{\{}\overline{J}\ T\ x = \iota\mathtt{;}\ \mathtt{[\,]}\ \overline{J}'\mathtt{\}}\ \textbf{in}\ e \;\rightarrow\; \mu \mid \mathtt{\{}\overline{J}\ \overline{J}'[x = \iota]\mathtt{\}}\ \textbf{in}\ e[x = \iota]}$$

$$\boxed{\Gamma \vdash e : T}$$

$$\text{(FORK-JOIN-T)}\quad \frac{\Gamma'_i \vdash e_i : \_ \leq T_i \quad \forall i = 1..n \qquad \Gamma_0 \vdash e : T}{\Gamma \vdash \mathtt{\{}\ T_1\ x_1 = e_1\mathtt{;}\ \mathtt{[}\overline{dep}_1\mathtt{]}\ \ldots\mathtt{;}\ T_n\ x_n = e_n\mathtt{;}\ \mathtt{[}\overline{dep}_n\mathtt{]}\ \mathtt{\}}\ \textbf{in}\ e : T}$$

with

$T'_i = T_i$ or $T_i = \mathtt{f}\ C_i$ and $T'_i \in \{\mathtt{b}\ C_i, \mathtt{i}\ C_i, \mathtt{m}\langle\overline{TP_i}\rangle\ C_i\} \quad \forall i \in 1..n$

$\Gamma, x_1{:}T'_1 \ldots x_n{:}T'_n = \Gamma_0 \ldots \Gamma_n$

$\Gamma'_i = \Gamma_i$ restricted over free variables of $e_i \quad \forall i \in 1..n$

$dep_i = \text{compute-dep}(x_1 \ldots x_{i-1}, \Gamma_1 \ldots \Gamma_{i-1}, \Gamma_i) \quad \forall i \in 1..n$

Figure 9. Adding the Fork Join Expression

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.[3]

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 $\vee$ (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 **if** L means that a dependency exists with the expression that initialize x if L holds, that is, is equivalent to **true**. For example,

```
{ int x=1+2;[]
  int y=1+2;[]
  int z=x*y;[x if true, y if true]
} in z+1;
```

Means that `x` and `y` depends from nothing and can be computed in parallel, while before computing `z` we have *surely* to wait for the computation of `x` and `y`, since `z` depend from them.

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

```
{ m C x=ι1.m();[]
  m C y=ι2.k();[x if ι1 ≡ ι2]
} in x.h(y);
```

The expression associated with `x` can be computed, while in order to compute the one associated with `y` we have to check if $\iota_1 \equiv \iota_2$.

- if the two object identifiers are different, we can start executing that expression in parallel,
- otherwise, we have to wait for the completion 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 (OUT-E) is straightforward, while rule (COMMIT) replace occurrences of a local variable with its computed value. Dependency relation that are now equivalent to **false** are removed. Formally:

- substitution $J[x = \iota]$:

  $(T\ x = e\mathtt{;}\ \mathtt{[}\overline{dep}\mathtt{]})[x = \iota] = T\ x = e[x = \iota]\mathtt{;}\ \mathtt{[}\overline{dep}\mathtt{]}\,[x = \iota]$

- substitution $\mathtt{[}\overline{dep}\mathtt{]}\,[x = \iota]$:

  $\mathtt{[\,]}\,[x = \iota] = \mathtt{[\,]}$

  with $\mathtt{[}\overline{dep}\mathtt{]}\,[x = \iota] = \mathtt{[}\overline{dep}'\mathtt{]}$ and $x \neq x'$

  $\mathtt{[}\overline{dep}, x\ \textbf{if}\ L\mathtt{]}\,[x = \iota] = \mathtt{[}\overline{dep}'\mathtt{]}$

  $\mathtt{[}\overline{dep}, x'\ \textbf{if}\ L\mathtt{]}\,[x = \iota] = \mathtt{[}\overline{dep}'\mathtt{]}$ if $L[x = \iota]$ equivalent to **false**

  $\mathtt{[}\overline{dep}, x'\ \textbf{if}\ L\mathtt{]}\,[x = \iota] = \mathtt{[}\overline{dep}', x'\ \textbf{if}\ L[x = \iota]\mathtt{]}$ otherwise

- Substitution over logic expressions $L[x = \iota]$ is straightforward: distributed on $\vee$ 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:

1
```
{ C a=new C().m();[]   C b=ι1.m();[]
  C c=a.k();[a if true, b if ι1 ≡ a]} in b.h(c);
```

expressions associated with `a` and `b` can be computed in parallel,

2
```
{ C a=ι2;[]  C b=e1;[]
  C c=a.k();[a if true, b if ι1 ≡ a]} in b.h(c);
```

here we suppose the computation over `ι1.m()` to be quite long, so in the shown point `a` is already associated with the value ι_2, while `b` is of the form e_1.

If $\iota_2 \neq \iota_1$ then the reduction will proceed in the following way:

3
```
{ C b=e2;[]  C c=ι2.k();[]} in b.h(c);
```

The execution of `c` can start now, while `b` is transparently continuing its execution, and it is now in e_2.

---

[3] Only object identifiers are nondeterministically chosen.

4 { C b=e_3;[] C c=e;[]} **in** b.h(c);

In this stage both b and c are associated with long standing, autonomous, parallel computations.

5 { C b=ι_3;[] C c=ι_4;[]} **in** b.h(c);

Now they have both concluded the associated computation, and the fork-join can be reduced to ι_4.h(ι_3).

5.3 New Typing Rule

The flow of parallel execution is regulated by the logic expressions inside the fork-join expressions, and the typing rule (FORK-JOIN-T) constraint the shape of such logic expression. Rule (FORK-JOIN-T) is quite complicated, since it performs different tasks:

- introduces local variables $x_1{:}T'_1, \dots x_1{:}T'_1$ taking care of fresh conversions;
- find $\Gamma_0 \dots \Gamma_n$ using the non deterministic splitting relation;
- compute $\Gamma'_0 \dots \Gamma'_n$ restricting the domain of $\Gamma_0 \dots \Gamma_n$ on the free variable of the corresponding expression. For $i \in 1..n$ we assume expression e_i to not have free variables $x_1 \dots x_i$; finally, dependency can be computed. Since dependencies $\overline{dep}_1 \dots \overline{dep}_n$ are completely calculated inside this rule, they can be automatically inferred during our program transformation. Formally:
  x_i **if true** $\in \mathsf{compute\text{-}dep}(x_1 \dots x_n, \Gamma_1 \dots \Gamma_n, \Gamma_0)$
  iff $x_i \in \mathsf{dom}(\Gamma_0)$ and $i \in 1..n$
  x_i **if** $\mathsf{dep}(x{:}T, x_0{:}T_0) \in \mathsf{compute\text{-}dep}(x_1 \dots x_n, \Gamma_1 \dots \Gamma_n, \Gamma_0)$
  iff $\Gamma_i(x) = T, \Gamma_0(x_0) = T_0, i \in 1..n$
  and $\mathsf{dep}(x{:}T, x_0{:}T_0)$ not equivalent to **false**.

  The definition of $\mathsf{dep}(x_1{:}T_1, x_2{:}T_2)$ is the subject of the next section.

5.4 Primitive Dependency $\mathsf{dep}(\pi^T{}_1, \pi^T{}_2)$

For convenience we will use meta variables of the form π^T to indicate pairs $\pi{:}T$.

Defining $\mathsf{dep}(\pi^T{}_1, \pi^T{}_2)$ is not straightforward, we have many possible design choices. A clearly correct solution is to define it as always **true**. This makes the assumptions of Theorem 4 and Theorem 5 trivially verified, since the program have to be executed in sequential fashion anyway. Of course our expressive type system allows us to do much more than that:
First of all, we know that if one of the two π^T is immutable or fresh, then there is no dependency at all:

$\mathsf{dep}(\pi^T{}_1, \pi^T{}_2) =$ **false**
if $\pi^T{}_1$ or $\pi^T{}_2$ have fresh or immutable type

If both the π^T are readonly (or external readonly), than no dependency is caused by this specific interaction.

$\mathsf{dep}(\pi^T{}_1, \pi^T{}_2) =$ **false**
if $\pi^T{}_1$ and $\pi^T{}_2$ have readonly or external readonly type

Moreover, if the two π^T both have a balloon type, we can simply check the pointer equivalence:

$\mathsf{dep}(\pi^T{}_1, \pi^T{}_2) = (\pi^T{}_1 \equiv \pi^T{}_2)$
if $\pi^T{}_1$ and $\pi^T{}_2$ have balloon type

where two equal variables will always refer to the same object:

$\pi{:}\mathcal{M}_1\ C_1 \equiv \pi{:}\mathcal{M}_2\ C_2 =$ **true**

and two variables with incompatible types will never be equal:

$\pi_1{:}T_1 \equiv \pi_2{:}T_2 =$ **false** if $T_1 \not\leq T_2$ and $T_2 \not\leq T_1$

otherwise we generate the dynamic test over pointer equivalence:

$\pi_1{:}\mathcal{M}_1\ C_1 \equiv \pi_2{:}\mathcal{M}_2\ C_2 = (\pi_1 \equiv \pi_2)$

For space reasons we stop here and declare that in all the other cases $\mathsf{dep}(\pi^T{}_1, \pi^T{}_2)$ is **true**.

However it is possible to extend the $\mathsf{dep}(\pi^T{}_1, \pi^T{}_2)$ function in a variety of ways, greatly increasing the set of cases where an 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

$$T_1\ x_1 = e_1\texttt{;} \dots T_n\ x_n = e_n\texttt{;}\ e_0$$

can be optimized in the form

$$\texttt{\{}\ T_1\ x_1 = e_1\texttt{;}\ [\overline{dep}_1] \dots T_n\ x_n = e_n\texttt{;}\ [\overline{dep}_n]\ \texttt{\}}\ \textbf{in}\ e_0$$

(where $\overline{dep}_1 \dots \overline{dep}_n$ are inferred by (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 $\simeq$).

Theorem 4 (Confluence).
A well typed program have a confluent reduction. Formally:
$\forall e$ *if* $\emptyset \vdash e : T,\ \ \emptyset \mid e \xrightarrow{*} \mu_0 \mid \iota_0$ *and* $\emptyset \mid e \xrightarrow{*} \mu_1 \mid \iota_1$
we have that $\mu_0 \mid \iota_0 \simeq \mu_1 \mid \iota_1$.

Theorem 5 (Safe Program Transformation).
For all well typed programs p_0 *containing a method declaration* md_0 *containing a sequence of local variable declaration* $T_1\ x_1 = e_1\texttt{;} \dots T_n\ x_n = e_n\texttt{;}$, *for any well typed program* p_1 *identical to* p_0 *but where such sequence of local variable declaration is replaced with a corresponding fork-join operation and for any expression* e *well typed under* p_0,
if such expression reduces to a pair $\mu_0 \mid \iota_0$ *under* p_0,
then such expression reduces to a pair $\mu_1 \mid \iota_1$ *equivalent to* $\mu_0 \mid \iota_0$ *modulo object identifiers alpha renaming.*
Formally:
$\forall p_0 =$ **class** C **implements** $\overline{C}$ **{** $\overline{fd}\ \overline{md}\ md_0$ **}** $\overline{cd}\ \overline{id}$, *where*
$md_0 = T\ m\,\texttt{(}T_1\ x_1 \dots T_n\ x_n\texttt{)}\,\mathcal{M}$
$\mathcal{E}[T_1\ x_1 = e_1\texttt{;} \dots T_n\ x_n = e_n\texttt{;}\ e_0]$ *and* p_0 *well typed,*
$\forall p_1 =$ **class** C **implements** $\overline{C}$ **{** $\overline{fd}\ \overline{md}\ md_1$ **}** $\overline{cd}\ \overline{id}$, *where*
$md_1 = T\ m\,\texttt{(}T_1\ x_1 \dots T_n\ x_n\texttt{)}\,\mathcal{M}$
$\mathcal{E}[\texttt{\{}\ T_1\ x_1 = e_1\texttt{;}\ [\overline{dep}_1] \dots T_n\ x_n = e_n\texttt{;}\ [\overline{dep}_n]\ \texttt{\}}\ \textbf{in}\ e_0]$
and p_1 *well typed,*
$\forall e$ *where* $\emptyset \vdash e : T$ *under* p_0 *and* $\emptyset \mid e \xrightarrow{*} \mu_0 \mid \iota_0$ *under* p_0
we have that $\emptyset \mid e \xrightarrow{*} \mu_1 \mid \iota_1$ *under* p_1 *and* $\mu_0 \mid \iota_0 \simeq \mu_1 \mid \iota_1$.

Thanks to these theorems, it is possible to start from a program with no fork-join expressions (and thus no parallelism) and to introduce automatic parallelisation in place of sequence of local variable declarations. Of course converting all the local variable declarations in fork-join expression could not be an optimal solution, investigating the optimal set of points to apply our transformation is future work. We are currently working on finalising the proof of these theorems and more discussion can be found in the companion technical report [16].

5.6 Application Example

There exists many well known operations transforming a mesh in some way, for example to provide a simplification. Thanks to our system, the following code can be transparently parallelised:

```
class Simpl{
  ...//some fields
  int simplify(m Mesh m)m{
    ...//returns number of removed triangles
} }
class Workbench{ ...//as before, plus
  int simplifyMeshes(int mIndex1, int mIndex2)e{
    Balloon m1=this.meshes.get(mIndex1);
```

```
    Balloon m2=this.meshes.get(mIndex2);
    int x1=new Simpl().simplify(m1);
    int x2=new Simpl().simplify(m2);
    return x1+x2; } }
```

`m1` and `m2` would be (rapidly) computed sequentially, then `x1` will start its execution. At the same time the run time check `m1`$\equiv$`m2` will be performed. If `m1`$\not\equiv$`m2` `x2` will instantaneously start its execution, otherwise if `m1`$\equiv$`m2` `x2` will start its execution only after `x1` is completed. This can happen if, for example, the user provides the same value for `mIndex1` and `mIndex2`.

Note how the programmer could naturally write this code without even ever wondering about the parallelisation concept itself. The code will be well typed thanks to a sequence of promotions of the expression **new** `Simpl().simplify(m1)`: **new** `Simpl()` is promoted to `f`resh using (PERMANENT1-TP); `m1` is promoted to `m`utable using (BALLOON-L-TP) thanks to (SUB-EXPR-T) where **new** `Simpl()` is extracted as a variable with type `m`$\langle$BLTP$\rangle$.

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.

BI-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 transferred to the preventative checks that BI-JAVA inserts on behalf of the programmer. Future work might include an empirical study comparing the cost of these two approaches.

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 BI-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 $\mathsf{dep}(\pi^{T}{}_1, \pi^{T}{}_2)$ 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.

References

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

[] J. Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley, 2001.

[2] R. Bocchino. *An Effect System and Language for Deterministic-by-Default Parallel Programming*. PhD thesis, University of Illinois at Urbana-Champaign, 2010.

[3] 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.

[4] J. Boyland. Why we should not add readonly to Java (yet). In *FTfJP*, Glasgow, Scotland, July 2005.

[5] D. Clarke and T. Wrigstad. External Uniqueness is Unique Enough. In *ECOOP*, volume 2473 of *LNCS*, pages 176–200, Darmstadt, Germany, July 2003. Springer-Verlag.

[6] D. Clarke, J. Noble, and T. Wrigstad, editors. *Aliasing in Object-oriented Programming*, volume In Print of *Lecture Notes in Computer Science*. Springer, 2012.

[7] A. Corradi, M. Servetto, and E. Zucca. DeepFJig - Modular composition of nested classes. In C. Wimmer and C. W. Probst, editors, *PPPJ'11 - Principles and Practice of Programming in Java*, ACM International Proceedings Series, pages 101–110. ACM Press, 2011.

[8] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. The Geneva convention of the treatment of object aliasing. *OOPS Messenger*, 3(2): 11–16, April 1992.

[9] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, May 2001.

[10] G. Lagorio, M. Servetto, and E. Zucca. Featherweight Jigsaw - a minimal core calculus for modular composition of classes. In *ECOOP'09 - Object-Oriented Programming*, volume 5653. Springer, 2009.

[11] P. Müller and A. Poetzsh-Heffter. Programming Languages and Fundamentals of Programming. Technical report, Fernuniversität Hagen, 2001. Poetzsh-Heffter, A. and Meyer, J. (editors).

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

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

[14] A. Potanin, J. Östlund, Y. Zibin, and M. D. Ernst. Immutability. In Clarke et al. [6].

[15] C. Reichenbach, Y. Smaragdakis, and N. Immerman. PQL: A purely-declarative Java extension for parallel programming. In *ECOOP*, volume 7313, pages 53–78. Springer-Verlag, 2012.

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

[17] D. Walker. *Advanced Topics in Types and Programming Languages (Edited by B. Pierce)*, chapter Substructural Type Systems. MIT Press, 2005.

[18] Y. Zibin, A. Potanin, S. Artzi, A. Kiezun, and M. D. Ernst. Object and reference immutability using Java generics. In *Foundations of Software Engineering*, 2007.

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

A. Subtyping Relation

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

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

- $\mathcal{M}\, C_1 \leq \mathcal{M}\, C_2$ if
  $p(C_1) =$ **interface** C_1 **extends** $C, \_$ **{** $\_$ **}** and $\mathcal{M}\, C \leq \mathcal{M}\, C_2$;
- $\mathcal{M}\, C \leq \mathtt{e}\, C$ iff $\mathcal{M} = \mathtt{b}$ or $\mathcal{M} = \mathtt{m}$;
- $\mathcal{M}\, C \leq \mathtt{r}\, C$ iff $\mathcal{M} = \mathtt{i}$ or $\mathcal{M} = \mathtt{m}$;
- $\mathcal{M}\, C \leq \mathtt{er}\, C$ iff $\mathcal{M} = \mathtt{e}$ or $\mathcal{M} = \mathtt{r}$;

B. Extending $\mathsf{dep}(\pi^{T}{}_1, \pi^{T}{}_2)$

We show one possible extension allowing to safelly parallelize also some mutable objects.

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

π ::= $x.\overline{f} \mid \iota.\overline{f}$ extended comparision

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 immutable or balloon fields are reached. We can inductively define a *predictable* class as a class where all the fields are of immutable, balloon or *predictable* type. Note that such definition rules out, for example, a linked list where the `next` field is mutable. We call mutable-predictable types composed by a mutable modifier and the name of a predictable class.

We can define the following two function over mutable predictable variables:

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

$\pi{:}\mathtt{m}\; C \in \mathsf{m\text{-}pred}(\pi{:}\mathtt{m}\; C)$

$\pi.\overline{f}.f{:}\mathtt{m}\; C_2 \in \mathsf{m\text{-}pred}(\pi{:}\mathtt{m}\; C_0)$

iff $\pi.\overline{f}{:}\mathtt{m}\; C_1 \in \mathsf{m\text{-}pred}(\pi{:}\mathtt{m}\; C_0)$

and $p(C_1) =$ **class** C_1 **implements** $\_$ **{** $\_\,\mathtt{m}\; C_2\, f$ **;** $\_$ **}**

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

$\pi.\overline{f}.f{:}\mathtt{m}\; C_2 \in \mathsf{b\text{-}pred}(\pi{:}\mathtt{m}\; C_0)$

iff $\pi.\overline{f}{:}\mathtt{m}\; C_1 \in \mathsf{m\text{-}pred}(\pi{:}\mathtt{m}\; C_0)$

and $p(C_1) =$ **class** C_1 **implements** $\_$ **{** $\_\,\mathtt{b}\; C_2\, f$ **;** $\_$ **}**

Path to immutable references are not relevant, and by our definition of predictable class other kinds of objects are simply not reachable.

Now is possible to define when $\mathsf{dep}(\pi^{T}{}_1, \pi^{T}{}_2)$ is true over variables with mutable-predictable type.

$\mathsf{dep}(\pi^{T}{}_1, \pi^{T}{}_2) = (\pi^{T}{}_1 \in \mathsf{b\text{-}pred}(\pi^{T}{}_2))$

if $\pi^{T}{}_1$ have balloon type and $\pi^{T}{}_2$ have mutable-predictable type

$\mathsf{dep}(\pi^{T}{}_1, \pi^{T}{}_2) =$

$\mathsf{m\text{-}pred}(\pi^{T}{}_1) \odot \mathsf{m\text{-}pred}(\pi^{T}{}_2))$

if $\pi^{T}{}_1$ and $\pi^{T}{}_2$ have mutable-predictable type

where[4]

non-disjointness: $\pi^{T}{}_1 \ldots \pi^{T}{}_n \odot \overline{\pi^{T}} = \pi^{T}{}_1 \in \overline{\pi^{T}} \vee \ldots \vee \pi^{T}{}_n \in \overline{\pi^{T}}$

inclusion: $\pi^{T} \in \pi^{T}{}_1 \ldots \pi^{T}{}_n = \pi^{T} \equiv \pi^{T}{}_1 \vee \ldots \vee \pi^{T} \equiv \pi^{T}{}_n$.

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

$\mathsf{dep}(\pi^{T}{}_1, \pi^{T}{}_2) =$ **true** otherwise

However, there are clearly many other cases in witch is possible, and convenient, to study a precise logic expression allowing safe parallelization. For example it would be possible to improve our study over 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 balloon fields and their reachable object graph[5]. For example

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

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

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

```
class Foo{
  private Balloon Bar bar;//no public getter method
  private boolean opInProgress=false;//no public setter
  void meth(Mutable Boh p1)Mutable{
    if(this.opInProgress)
      throw new Error("unespected indirect recursion");
    try{this.opInProgress=true;//this.bar only set in Foo
      //this.bar only accessed in Foo
      doStuff(this.bar,p1);}
    finally{opInProgress=false;}
  ...}
}
```

We use varible `opInProgress` to prevent undesired indirect multiple accesses to the `bar` data.[6] If the method have balloon parameters then a simple dynamic check (straigthforward identity test x!=y) can ensure full control. However if external or external readonly parameter are present, and they are of a type that can potentially reach a `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 guarancee that no other client will be able to see such uncoherent state.

C.2 Memory management optimisations

immutable objectss can have an ad hoc memory management. Whenever a fresh object is converted into immutable, 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 instantanely garbage collected. System generated hashcode and equals, supporting circularity, should be automatically introduced. Moreover, keys of a HashSet/HashMap should be only Immutable, such maps will be faster and with a simpler semantic.

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

This opens the door for a simple language extension with a safe *try-with-resource* statement, when the resource object is guaranteed to be collected at the end of the statement.

---

[4] No need to check $(\mathsf{b\text{-}pred}(\pi^{T}{}_1) \odot \mathsf{b\text{-}pred}(\pi^{T}{}_2)$ in the second case.

[5] 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 iphotesis. 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 $T\,x = \iota\texttt{;}\,e$ to be always applicable directly to the conclusive expression e.

$r \ ::= \ \iota.m\,\textbf{(}\bar{\iota}\textbf{)} \ \mid \textbf{new}\ C\,\textbf{(}\bar{\iota}\textbf{)}\,\iota_0.f \mid \iota_0.f{=}\iota_1 \mid T\,x = \iota\texttt{;}\ e$ redex

Case (CTX)

Choosing

(C1) $\iota \in \mathsf{dom}(\Sigma_0)$ iff ι is a subterm of r_0

(C2) $\mathsf{dom}(\mu_0) = \mathsf{rog}_{\mu_0,\mu}(\mathsf{dom}(\Sigma_0))$

Given

(G1) $\mu_0, \mu \mid \mathcal{E}[r_0] \ \rightarrow \ \mu_1, \mu \mid \mathcal{E}[e_1]$

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

(G3) $\emptyset; \Sigma_0, \Sigma \vdash \mathcal{E}[r_0] : T_0'$

Show

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

(R2) $\emptyset; \Sigma_1, \Sigma \vdash \mathcal{E}[e_1] : T_1' \leq T_0'$

we obtain:

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

(I2) $\Sigma_0^{\overline{TP}} \vdash \mu_0$ by (G2), (G3) and Lemma 7.i,

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

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

(IH2) $\emptyset; \Sigma_1^{\overline{TP}} \vdash e_1 : T_1 \leq T_0$ by (I1), (I2), (I3) and inductive iphotesis.

Now we can obtain our results: By (IH1), (G2) and Lemma 9.ii we obtain (R1) By (I3), (IH2), (G3), and Lemma 9.i we obtain (R2)

Lemma 6 (promotionPreservation).

$\Sigma_0 \vdash \mu_0$

implies

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

Proof. By induction over sequence of promotions:

Case empty set of promotion

Trivial

Case Promotion sequence ends with a permament type promotion

all the `m`utable or `r`eadonly $\iota\!:\!T$ in Σ_0 are `e`xternal or `e`xternal `r`eadonly, in $\Sigma_0^{\texttt{PTP}}$; respectivelly. This means that $\mathsf{RB}_\mu(\Sigma_0^{\texttt{PTP}}) \subseteq \mathsf{RB}_\mu(\Sigma_0)$. 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 `m`utable or `b`alloon variable in the stack are seen as `e`xternal.

  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 `e`xternal in the stack to `m`utable. This repopulate the set $\mathsf{RB}_\mu(\Sigma_0^{\texttt{BLTP}})$ 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 $\mathsf{valid\text{-}B}_\mu(\iota)$ 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) $\mathsf{dom}(\mu_0) = \mathsf{rog}_{\mu_0,\mu}(\mathsf{dom}(\Sigma_0))$
(G2) $\Sigma_0, \Sigma \vdash \mu_0, \mu$
(G3) $\emptyset; \Sigma_0, \Sigma \vdash \mathcal{E}[r_0] : T_0'$

implies

i $\Sigma_0^{\overline{TP}} \vdash \mu_0$
ii $\emptyset; \Sigma_0^{\overline{TP}} \vdash r_0 : T_0$

Proof.

i By (G2) and the property of the choosed division betwen Σ_0, Σ and μ_0, μ we know that $\Sigma_0 \vdash \mu_0$ We conclude by Lemma 6.

ii We choose $\overline{TP}$ to coincide with the set of promotion applied over subexpression r_0. By rule (LOC-T) $\emptyset; \Sigma_0^{\overline{TP}} \vdash r_0 : T_0$ iff $\Gamma_0^{\overline{TP}} \vdash r_0[\iota_1 = x_1 \ldots \iota_n = x_n] : T_0$ with $\mathsf{dom}(\Sigma) = \iota_1 \ldots \iota_n$, $\mathsf{dom}(\Gamma) = x_1 \ldots x_n$, $\Gamma_0(x_i) = \Sigma(\iota_i)$

Thus, for our chosen $\overline{TP}$ we can conclude by straigtforard application of the typing rules over
$\Gamma_0, \Gamma \vdash \mathcal{E}[r_0][\iota_1 = x_1 \ldots \iota_n = x_n] : T_0'$

Lemma 8 (substitution zero).

(G1) $\mu_0 \mid r_0 \ \rightarrow \ \mu_1 \mid e_1$
(G2) $\emptyset; \Sigma_0 \vdash r_0 : T_0$
(G3) $\emptyset; \Sigma_0, \Sigma \vdash \mathcal{E}[r_0] : T_0'$
(G4) $\emptyset; \Sigma_1 \vdash e_1 : T_1 \leq T_0$
(G5) $\Sigma_0 \rightsquigarrow \Sigma_1 \vdash \mu_0 \rightsquigarrow \mu_1$
(G6) $\Sigma_0, \Sigma \vdash \mu_0, \mu$

exists a Σ_1 such that

i $\emptyset; \Sigma_1, \Sigma \vdash \mathcal{E}[e_1] : T_1' \leq T_0'$
ii $\Sigma_0, \Sigma \rightsquigarrow \Sigma_1, \Sigma \vdash \mu_0, \mu \rightsquigarrow \mu_1, \mu$

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 analisis over the possible kinds of the assigned value:

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

`r` (ii) holds, since readonly links does not contribuite for the set of reachable ballons $\mathsf{RB}_\mu(\Sigma)$.

$\mathtt{m}$ (MEM-T) last side condition states that all the element in the shared heap belongs to at most one balloon. Since the receiver is also on the shared heap (ii) holds.

Case (F-UPDATE-TB)

(i) holds by (F-UPDATE-TB) second premise. (ii) trivially holds since the assigned value is inside the fresh heap in μ_0.

Case (NEW-T)

The application of this rule ignect a new element on the stack Σ_1, of kind $\mathtt{m}$utable. (i) holds by inductive hypotesis and subtype transitivity. Since this new element is inside the shared heap but outside any baloon, also (ii) holds.

Case (PERMANENT1-TP)

In this case we know that r_0 can be typed as mutable in $\textsf{l-close-p}(\Gamma_0) \vdash r_0[\iota_1 = x_1 \ldots \iota_n = x_n] : \mathtt{m}\ C_0$ By cases depending on the kind of redex:

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

new $C\,(\bar{\iota})$ The created object clearelly respect both the fresh invariants and the immutable invariants. So it can be considered inside the fresh heap. Indeed all refeence in $\bar{\iota}$ are either in the fresh heap or in the immutable heap.

$\iota_0.f$ The produced reference is inside the fresh heap, indeed ι_0 must be of a fresh kind $\mathtt{m}\langle\text{PTP}\rangle$ in Σ_0, that is $\Sigma_0(\iota_0) = \mathtt{m}\langle\text{PTP}\rangle\,\_$.

In the detail, (i) holds since $\iota_{n+1}[\iota_1 = x_1 \ldots \iota_{n+1} = x_{n+1}]$ can be typed with (VAR-T) as a fresh reference. (ii) holds since the newly created object, ι_{n+1}, can be considered inside the fresh heap.

$\iota_0.f{=}\iota_1$ clearelly $\Sigma_0(\iota_1) = \mathtt{m}\langle\text{PTP}\rangle\,\_$, thus ι_1 is already inside the fresh heap in Σ_0.

In the detail, (i) holds since $\iota_1[\iota_1 = x_1 \ldots \iota_n = x_n]$ can be typed with (VAR-T) as a fresh reference. (ii) holds since (G5).

$T\ x = \iota;\ e$ (i) holds since also the expression e can be typed with (PERMANENT1-TP) in this environment. (ii) holds since $\mu_0 = \mu_1$.

Case (PERMANENT2-TP)

In this case we know that r_0 can be typed as readonly in $\textsf{l-close-p}(\Gamma_0) \vdash r_0[\iota_1 = x_1 \ldots \iota_n = x_n] : \mathtt{r}\ C_0$ By cases depending on the kind of redex, we omit details about (i) and (ii) since they are similar to the former point.

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

new $C\,(\bar{\iota})$ The created object clearelly respect both the fresh invariants and the immutable invariants. So it can be considered inside the immutable heap.Indeed all refeence in $\bar{\iota}$ are either in the fresh heap or in the immutable heap.

$\iota_0.f$ The produced reference is inside the immutable heap, indeed ι_0 must be of a fresh kind $\mathtt{m}\langle\text{PTP}\rangle$ or of an immutable kind in Σ_0, that is $\Sigma_0(\iota_0) \in \{\mathtt{m}\langle\text{PTP}\rangle\,\_, \mathtt{i}\,\_\}$.

$\iota_0.f{=}\iota_1$ clearelly $\Sigma_0(\iota_1) \in \{\mathtt{m}\langle\text{PTP}\rangle\,\_, \mathtt{i}\,\_\}$, thus ι_1 is already inside the fresh or immutable heap in Σ_0.

$T\ x = \iota;\ e$ also the expression e can be typed with (PERMANENT1-TP) or (PERMANENT2-TP) in this environment.

Case (BALLOON-L-TP)

In this case we know that r_0 can be typed in $\textsf{l-close-b}(\Gamma), x_0{:}\mathtt{m}\ C_0 \vdash r_0[\iota_1 = x_1 \ldots \iota_n = x_n] : \mathcal{M}\ C$ and to proof (i) we have to show that $\Gamma, x_0{:}\mathcal{M}_0\ C_0 \vdash e_1[\iota_1 = x_1 \ldots \iota_n = x_n] : \textsf{l-close}(\mathcal{M})\ C$

That is, an $\mathtt{e}$xternal reference to ι_0 is seen as $\mathtt{m}$utable. By cases depending on the kind of redex:

$\iota.m\,(\bar{\iota})$ also the expression e_1, body of the method m can be typed with (BALLOON-L-TP) in this environment. (ii) holds since $\mu_0 = \mu_1$.

new $C\,(\bar{\iota})$ (i) holds since $\iota_{n+1}[\iota_1 = x_1 \ldots \iota_{n+1} = x_{n+1}]$ can be typed with (VAR-T) as a mutable reference. (ii) holds since the created object mutable reachable object graph clearelly refers only to elements inside the balloon that contains ι_0. So it can be considered inside the shared heap.

$\iota_0.f$ The produced reference is also inside the reachable object graph of ι_0.

$\iota_0.f{=}\iota_1$ Following the same process used for case (F-UPDATE-T) we can show that both ι_0 and ι_1 are in the shared heap, thus we are not violating the balloon invariants, thanks to (MEM-T) last side condition.

$T\ x = \iota;\ e$ also the expression e can be typed with (BALLOON-L-TP) in this environment.

Case (TEMPORARY-TP)

–

Case (SUB-EXP-T)

–

Lemma 9 (substitution).

(G1) $\boxed{\mu_0} \mid r_0 \ \rightarrow\ \boxed{\mu_1} \mid e_1$

(G2) $\emptyset; \Sigma_0^{\overline{TP}} \vdash r_0 : T_0$

(G3) $\emptyset; \Sigma_0, \Sigma \vdash \mathcal{E}[r_0] : T_0'$ *applying* $\overline{TP}$ *to* r_0

(G4) $\emptyset; \Sigma_1^{\overline{TP}} \vdash e_1 : T_1 \leq T_0$

(G5) $\Sigma_0^{\overline{TP}} \rightsquigarrow \Sigma_1^{\overline{TP}} \vdash \mu_0 \rightsquigarrow \mu_1$

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

exists a Σ_1 *such that*

i $\emptyset; \Sigma_1, \Sigma \vdash \mathcal{E}[e_1] : T_1' \leq T_0'$

ii $\Sigma_0, \Sigma \rightsquigarrow \Sigma_1, \Sigma \vdash \mu_0, \mu \rightsquigarrow \mu_1, \mu$

Proof. By induction over sequence of promotions:

Case empty set of promotion

Proved in Lemma 8

Case Promotion sequence ends with a permament type promotion

By cases over the different kinds of redex. In all cases but constructor invocation is enough to consider $\Sigma_0 = \Sigma_1$

The constructor invocation is thus the more intresting case.

Is required to conside the newly created object as fresh. This is possible since we are inside a perment type promotion: indeed outside of the promotion all the parameter of the constructed object will be either $\mathtt{i}$mmutable or $\mathtt{m}\langle\text{PTP}\rangle$.

Case Promotion sequence ends with a balloon local type promotion

–

Case Promotion sequence ends with a temporary type promotion

–