Our billion dollar fix

Safe modular circular initialisation without `null`

Marco Servetto
Victoria University of Wellington, New Zealand
servetto@ecs.vuw.ac.nz

Alex Potanin
Victoria University of Wellington, New Zealand
alex@ecs.vuw.ac.nz

Abstract

"The introduction of null references was my billion-dollar mistake." *Tony Hoare*

The above quote highlights the well-known problem of null references. However, to simply remove **null** from the language is too restrictive: **null** (or its variants) is essential for any type of circular initialisation. Many approaches [6, 14, 16, 17] have tried to address the problems resulting from having **null** in a programming language. However, they either require dynamic checks [17] or do not support *modular* creation of circular structures [6, 14, 16].

In this paper we present FJ′, a Java-like language simpler than other approaches. Our contributions are as follows:

1. FJ′ avoids the need for **null** or any other default initialisation concept;
2. FJ′ provides a new language construct (*placeholders*) to support safe *modular* circular initialisation. In particular, FJ′ is the only language that can safely express multiple circular initialisation using a Factory pattern [8] in a *modular* fashion;
3. FJ′ solves the problem of lazy modular circular initialisation present in lazy languages, by statically preventing reducing initialisation expressions to the form `x=x`.

Categories and Subject Descriptors D.3 Programming Languages [*D.3.3 Language Constructs and Features*]: Classes and objects

General Terms Languages

Keywords Java, circular initialization, null, factories

1. Introduction

At QCon 2009 Tony Hoare called the introduction of null references "my billion-dollar mistake", however he recognised that not having **null** means that: [1]

- when optional values are needed, they must be explicitly defined by programmers (as done in functional languages);
- a dedicated sub-language is required to properly initialise circular object graphs.

[1] `http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare` (around minute 26 onwards)

While defining optional values is a well established area [1, 13], only recent proposals have tried to provide sub languages allowing safe circular initialisation: Delayed Types [6], Masked Types [14], Freedom Before Commitment [16], and Value Recursion Challenge [17]. However, a common accepted practice [6, 14, 16] is to hard code a dependency between the construction of objects involved in a circular object graph. The main contribution of this paper is to propose a way to support independent construction (i.e. *modular*) of such objects.

Consider the following examples showing the default constructor for a circular doubly linked list in Freedom Before Commitment [16], Delayed Types [6], and Masked Types [14] — the current state of the art that supports circular initialisation while guaranteeing the absence of `NullPointerException`.

- F. Before Commitment -

```
class List { ...
  Node! sentinel ;
  List(){this.sentinel = new Node(this);} }
class Node { ...
  Node! prev; Node! next;
  List! parent; Object? data;
  Node([Free] List! parent){//sentinel construction
    this.parent = parent;
    this.prev = this; this.next = this; } }
```

This example uses a `Node!` notation to denote non **null** reference to `Node` and `Object?` to denote a potentially **null** reference to `Object` as well as additional annotations to statically guarantee that non **null** fields are not used without having been initialised. This code is statically checked for the absence of `NullPointerException`. Note that the identical code with annotations removed would result in a Java version of the circularly initialised doubly linked list. Most importantly, note that the construction of the list has to create the node internally.

The following two code samples show the same example using Delayed Types [6] and Masked Types [14]:

- Delayed types -

```
List.sentinel:Node!
void List.ctor[t,t>Now](this:List!◊t)
{ this.sentinel }
{ let tmp:Node!◊t = alloc Node[t]//allocate
    in tmp.ctor1[t](this);//call constructor
  this.sentinel = tmp; }
void Node.ctor1[t,t>Now]
  (this:Node!◊t,parent:List!◊t)
{ this.parent, this.prev, this.next }
{ this.parent = parent;
  this.prev = this; this.next = this; }
```

- Masked types -

```
header = createHeader(); ...
private static Entry createHeader(){
  Entry\(*-Entry.sub)! h = new Entry();
  h.element=dummyElement;
  h.next=h; h.previous=h;
  return h; }
```

Both of the above approaches introduce significant additional annotations to guarantee the absence of `NullPointerException`. Furthermore, note that the construction of the list in Delayed Types creates the node internally and Masked Types completely separate the creation and initialisation stages.

FJ′ utilises *placeholders* (a variable name prefixed with a single quote as defined in the next section) to provide a safe modular circular initialisation construct that allows the construction of the list to be independent from the construction of the node:

```
FJ′
class List{ Node sentinel; }
class Node{
  Node prev; Node next;
  List parent; Object data;
}
class ListFactory{
  List newList(){
   return {
     List l=new List('n);
     Node n=new Node('n,'l,new Object());
   } in l;} }
```

We present a Java-like language[2] called FJ′, allowing safe circular initialisation. FJ′ has a sound modular type system, where modular circular initialisation is possible: objects can always be initialised using the Factory pattern.

The rest of the paper is organised as follows: in Section 2 we provide an informal presentation of FJ′ and the typing issues. In Section 3 we formally define reduction and typing rules. In Section 4 we discuss our work. In Section 5 we compare FJ′ with related work and in Section 6 we summarise the contributions.

2. A Tour of the FJ′ Language

Consider the following approach written in Java that uses **null**; current approaches [6, 14, 16] devote a significant amount of effort trying to ensure that the two step initialisation sequence above or variation thereof is safe.

```
Java
class A{ B myB; A(B myB){this.myB=myB;} }
class B{ A myA; B(A myA){this.myA=myA;} }
...//Two step initialisation sequence using null:
A a= new A(null); B b= new B(null);//Step 1
a.b= b; b.a= a;//Step 2
return a; ...
```

Imagine no **null**: a Java-like language with heap, references, field assignment but no **null** value. Every reference has to be fully initialised either at the point of declaration (for variables) or inside the constructor (for fields). We assume implicit *conventional* constructors similar to Featherweight Java(FJ)[10]: a constructor is a simple sequence of field initialisation expressions of the form **this**.`f=f`, where every single field has to be given a value via a corresponding constructor argument. No other constructors are allowed.

On the one hand, the big advantage of such a language is that it is impossible to get a `NullPointerException` as there is no **null** to begin with. On the other hand, such a language is not suitable for general purposes as it would not be possible to initialise any circular dependencies such as the following without using **null** [13]:

```
class A{
  B myB;
  /* FJ' assumes implicit constructors that initialise
  all fileds, e.g.: A(B myB){this.myB=myB;} */ }
class B{
  A myA; /* implicit: B(A myA){this.myA=myA;} */ }
// Cannot create an instance of A as it requires an instance
// of B that requires an instance of A ad infinum...
... return new A(new B(???)); ...
```

[2] That is, a statically typed, class-based object-oriented language, with nominal types and explicitly declared subtyping. Classes contain methods and the core language operation is the single-dispatch dynamic method call.

The core insight behind FJ′ is a *multiple initialisation* expression that removes the need for using **null** or any **null**-like values and thus avoids any possibility of a `NullPointerException` but still enables the programmer to perform circular initialisation. For example, the code above in FJ′ would look like this:

```
class A{ B myB; } class B{ A myA; }
...
//multiple initialisation:
return {
  A a = new A('b); B b = new B('a);
} in a;
...
```

This example (using conventional constructors only) is easy to understand and does not require a complex initialisation pattern (e.g. a special constructor coordinating other constructors [16]). In fact, we believe that none of the current approaches [6, 14, 16, 20] can express this minimal example without resorting to **null**.

The key contribution of FJ′ language is a full formalisation and proof of soundness in the presence of circular initialisation with no need for **null** or **null**-like values.[3] Furthermore, every primitive type in FJ′ has a sensible value without any need for defaults (e.g. `0` for integers in Java etc). In fact, we believe that in many cases `NullPointerException`s are a symptom of "language enforced default initialisation".

2.1 Placeholders

Consider a commonly used **let** expression: **let** $T\ x = e$ **in** e'
Multiple initialisation is a generalisation of **let**: $\{\overline{T\ x = e}\}$ **in** e'
where a sequence of variables initialised in the curly brackets can be used in the expression e' (after **in**), that is called *conclusive expression*. For example, consider the code from the previous section:

```
return {
  A a = new A('b); B b = new B('a);
} in a;
```

we can observe that `'a` and `'b` are used in place of variables that might not have been initialised. We prefix such variables with a single quote and call them *placeholders*.

Placeholders can only appear inside multiple initialisation expression as part of a variable initialisation. Placeholders can be used to initialise object fields and passed as method arguments, however they cannot be used as receivers since they represent values that still do not exist. Placeholders are *introduced* and *consumed* by multiple initialisation expressions. Our example above *introduced* placeholders `'a` and `'b` by declaring variables `a` and `b` respectively.
In general, multiple initialisation expressions containing a variable declaration `T x =...` can contain placeholder `'x` inside initialisation expressions, and the variable `x` inside conclusive expressions.

When the control reaches the **in** keyword, all the placeholders declared in the multiple initialisation expression are *consumed*; that is, every occurrence of such placeholder on the heap is replaced with the corresponding value.

To summarise, placeholders are entities that temporarily lie in place of objects that still do not exist. Placeholders are not objects, and thus there is no references in the heap pointing directly to placeholders. From the types point of view, we use (′) to indicate placeholder types, for example `'b` is a simple expression of type `B′`.

2.2 Partially Initialised and Fully Initialised Objects

FJ′ is a language that manipulates placeholders and objects. We distinguish two kinds of objects: *partially initialised objects* and *fully initialised objects*. In a simple setting, where all the fields are

[3] One other use of **null** is to help model optional values (e.g. `Option` type in ML or Scala). In our opinion, using a Proxy design pattern that returns an appropriate value depending on whether it is appropriate at any given point is a much safer way than relying on a reference containing **null**.

of a fully initialised type, an object is fully initialised if no placeholders are contained in its reachable object graph, and partially initialised otherwise. To make our language more expressive, we extend this intuitive concept, allowing classes to declare fields of placeholder and partially initialised types:

- Fully initialised objects can contain fully initialised objects in their fully initialised fields, but not partially initialised objects or placeholders.
- Fully initialised objects can contain partially or fully initialised object in partially initialised fields, but not placeholders.
- Fully initialised objects can contain placeholders, partially or fully initialised objects in their placeholder fields.

Otherwise, an object is considered *partially initialised*.

Partially initialised objects are conventional objects stored in the heap. We introduce a special partially initialised type (marked with a `^`) to denote such objects. For example, a variable of type `Data^` can refer to an instance of class `Data` that currently contains a placeholder instead of a fully initialised instance in one of its fields of a fully initialised type.

Partially initialised types and placeholder types can be used everywhere a type is expected. For example a method can take placeholders and produce a partially initialised object.

2.3 Constructors

Consider the constructors calls used in our example:

```
class A { B myB; } class B { A myA; }
//multiple initialisation:
return { A a = new A('b); B b = new B('a); } in a;
```

Observe that we invoked the constructors of `A` and `B` by passing a placeholder where a fully initialised object was expected.

A constructor for `A` produces a fully initialised instance of `A` when a fully initialised instance of `B` is provided.

FJ′ extends this behaviour so that a *partially* initialised object is returned instead of a *fully* initialised object when either a partially initialised or placeholder argument is supplied to a constructor call in place of a fully initialised argument. This extension is safe since FJ′ constructors have no code: they perform only fields initialisation.[4] For example, consider the following well typed methods:

```
A  m1( B  b){return new A(b);}
A^ m2( B' b){return new A(b);}
A^ m3( B^ b){return new A(b);}
```

The constructor of `A` expects a `B`, but in FJ′ `B` is *not* a supertype of either `B'` or `B^` [5] and thus if these three were method calls instead of constructor calls, both `m2` and `m3` would not type check as the actual argument's type is not a subtype. However, for constructors, FJ′ allows a partially initialised object or a placeholder to be used in place of the expected fully initialised object, *but the result of the constructor call would no longer be a fully initialised object*. Finally, note that in the body of `m2` we use `b` to denote a variable of placeholder type. One cannot write `'b` here as the placeholders can only be introduced inside the multiple initialisation expressions.

2.4 Reduction Example

To put together what we have seen so far, consider what happens during the evaluation (reduction) of the multiple initialisation expression used as our example:

```
{ A a = new A('b); B b = new B('a); } in a
```

We evaluate this expression in the empty heap. The *heap* is a finite map from object identifiers to records annotated with a class name.

[0] $\emptyset$ | `{A a=new A('b); B b=new B('a);} in a`

[1] $\iota_1 \mapsto$ `A(myB='b)` | `{A a=`ι_1`; B b=new B('a);} in a`

[2] $\iota_1 \mapsto$ `A(myB='b)`, $\iota_2 \mapsto$ `B(myA='a)` | `{A a=`ι_1`;B b=`ι_2`;} in a`

[3] $\iota_1 \mapsto$ `A(myB=`ι_2`)`, $\iota_2 \mapsto$ `B(myA=`ι_1`)` | ι_1

- We start in state 0, and in the first step the instance of class `A` is created. Indeed in state 1 the heap contains a partially initialised object of type `A` containing placeholder `'b` instead of a reference to an object of type `B`. The reference corresponding to the placeholder `'b` is still unknown.

  You can now see that values are object identifiers ι or placeholders $'x$. Note that values does not include **null**.
- In the second step the instance of class `B` is created. In state 2 ι_1 and ι_2 are partially initialised objects. Note how placeholders inside the heap are bound by the local variables declared inside the multiple initialisation expression.
- The third step concludes the initialisation, and in state 3 ι_1 and ι_2 are fully initialised objects. As you can see, placeholders are replaced by objects identifiers when multiple initialisation expressions are reduced. We expect final values to be only object identifiers.

2.5 Attempting to Write a Factory

Consider a more complex example closer to the real world that uses factories to create circularly initialised objects that conform to the interfaces `IA` and `IB`. This example demonstrates how FJ′ can safely detect at type checking time whether an object is truly fully initialised while allowing circular initialisation and avoiding the need for **null**. We provide two possible implementations for these interfaces (classes `A` and `B`) and we add a `data` field to class `B` for the sake of example.

```
interface IA{IB getIB();} interface IB{IA getIA();}
class Data{...}

class A implements IA{
  IB myB; IB getIB(){return this.myB;} }
class B implements IB{
  IA myA; IA getIA(){return this.myA;} Data data;}

class AF{
  IA^ makeIA(IB' myB){
    return new A(myB);} }
class BF{
  IB^ makeIB(IA' myA, Data data){
    return new B(myA, data);} }

class User{
  IA makeIAIB(){
    return {
      IA a=new AF().makeIA('b);
      IB b=new BF().makeIB('a,new Data(...));
    } in a;} }
```

Note that the factories can chose to use any implementation of `IA` or `IB` interfaces as the method signatures of `AF.makeIA` and `BF.makeIB` only refer to interface types. This example is still limited, since factories are embedded in the initialisation. A first attempt at providing factories as parameters can look as follows:

```
interface IAF{IA^ makeIA(IB' myB);}
interface IBF{IB^ makeIB(IA' myA, Data^ data);}
class AF implements IAF{/*as before*/}
class BF implements IBF{/*as before*/}

class User{//does not type check in FJ'
  IA makeIAIB(IAF af, IBF bf, Data data){
    return {
      IA a=af.makeIA('b);
      IB b=bf.makeIB('a,data);
    } in a; } }
```

[4] Our approach is not easily extensible to fully fledged constructors that are equivalent to normal methods (as in Java or C#). However, we believe that using Factory method is a better programming practice than writing a complex constructor body.

[5] `B` is indeed a *subtype* of both `B'` and `B^`

Using a multiple initialisation expression method `User.makeIAIB` declares variables `a` and `b` using expressions `af.makeIA('b)` and `bf.makeIB('a,data)`. We expect the code after the **in** keyword to be able to use the fully initialised objects denoted by variables `a` and `b`. However our assumption is wrong and FJ′ will reject the declaration of `makeIAIB` method above. The following code shows why and demonstrates how one can exploit this vulnerability:

```
class AF2 implements IAF{
 IA^ extIA;
 IA^ makeIA(IB' myB, Data data){
   return this.extIA;} }
...
  return {
    IB b2=new User().makeIAIB(
         new AF2(new A('b2)),
         new BF(),
         new Data(...)
         ).getIB();
  } in b2;
...
```

Here, at the point in which the method `getIB()` would be called, the resulting value of **new** `User().makeIAIB(...)` is still a partially initialised value (while `User.makeIAIB` promises to return a fully initialised object).[6] To better understand what is happening, we show the evaluation of the following multiple initialisation expression:

[0] $\emptyset$

```
{IB b2=new User().makeIAIB(
     new AF2(new A('b2)),
     new BF(),
     new Data(...)
     ).getIB();
} in b2
```

[1] $\iota_1 \mapsto$ User()
$\iota_2 \mapsto$ A(myB='b2)
$\iota_3 \mapsto$ AF2(extIA=ι_2)
$\iota_4 \mapsto$ BF()
$\iota_5 \mapsto$ Data(...)

```
{IB b2=ι1.makeIAIB(
    ι3,
    ι4,
    ι5).getIB();
} in b2
```

[2] $\iota_1 \mapsto$ User()
$\iota_2 \mapsto$ A(myB='b2)
$\iota_3 \mapsto$ AF2(extIA=ι_2)
$\iota_4 \mapsto$ BF()
$\iota_5 \mapsto$ Data(...)

```
{IB b2=({
  IA a=ι3.makeIA('b);
  IB b=ι4.makeIB('a,ι5);
  } in a).getIB();
 } in b2
```

[3] $\iota_1 \mapsto$ User()
$\iota_2 \mapsto$ A(myB='b2)
$\iota_3 \mapsto$ AF2(extIA=ι_2)
$\iota_4 \mapsto$ BF()
$\iota_5 \mapsto$ Data(...)

```
{IB b2=({
  IA a=ι3.extIA;
  IB b=new B('a,ι5);
  } in a).getIB();
 } in b2
```

[4] $\iota_1 \mapsto$ User()
$\iota_2 \mapsto$ A(myB='b2)
$\iota_3 \mapsto$ AF2(extIA=ι_2)
$\iota_4 \mapsto$ BF()
$\iota_5 \mapsto$ Data(...)
$\iota_6 \mapsto$ B(myA='a, data=ι_5)

```
{IB b2=({
  IA a=ι2;
  IB b=ι6;
  } in a).getIB();
} in b2
```

[5] $\iota_1 \mapsto$ User()
$\iota_2 \mapsto$ A(myB='b2)
$\iota_3 \mapsto$ AF2(extIA=ι_2)
$\iota_4 \mapsto$ BF()
$\iota_5 \mapsto$ Data(...)
$\iota_6 \mapsto$ B(myA='a, data=ι_5)

```
{IB b2=ι2.getIB();
} in b2
```

[6] $\iota_1 \mapsto$ User()
$\iota_2 \mapsto$ A(myB='b2)
$\iota_3 \mapsto$ AF2(extIA=ι_2)
$\iota_4 \mapsto$ BF()
$\iota_5 \mapsto$ Data(...)
$\iota_6 \mapsto$ B(myA=ι_2, data=ι_5)

```
{IB b2='b2;} in b2
```

[6] Lazy languages using variables to "encode" placeholders usually go into an infinite loop when the result of this expression has to be evaluated, see Section 5.1.

- We start in state 0 and the first step collapses five straightforward object creation steps, resulting in state 1.
- The second step simply reduces the method `makeIAIB`. In state 2 we obtain two nested multiple initialisation expressions and the nature of the error starts to emerge: `makeIA` will ignore the parameter and instead will return a partially initialised value (ι_2) generated in the outer scope.
- The third step collapses two function calls.
- The fourth step collapses a field access and an object creation. In state 4 the inner multiple initialisation expression is ready to be reduced.
- The fifth step reduces the inner multiple initialisation expression: variable `b` is no more present in state 5 and both ι_5 and ι_6 are no longer referred to in the program and therefore are lost.
- In state 6 it is clear that something has gone wrong. Our approach is stuck at this execution point. Indeed `{IB b2='b2;}` **in** `b2` is trying create a fully initialised object that was initialised with itself - which is clearly not a well defined operation.

2.6 Working Factories that Use Shared References

The example above contains two nested multiple initialisation expressions both introducing placeholder variables. We recall that a partially initialised object becomes fully initialised object if and only if all the reachable fully initialised fields no longer contain placeholders. This is supposed to happen at the end of the multiple initialisation expression where the placeholders were introduced. The reason our example so dramatically failed was that in-between the two nested multiple initialisation expressions we had a fully initialised object incorrectly containing a placeholder that would not be properly replaced until we complete the outer multiple initialisation expression. Thus we were able to attempt to initialise such placeholder with itself.

This problem could be prevented if we can distinguish between the references that are *shared* and *non shared* as follows:

- a *shared reference* can be assigned to a field, as is the case in Java and in most languages;
- a *non shared reference* cannot be assigned to a field, however, a non shared reference can be passed as a parameter to a method call, and local variables can be used to store non shared references. Moreover a non shared reference can be passed as parameter to a (conventional) constructor call that will in turn produce a non shared reference. Accessing a field using a non shared reference produces another non shared reference. In this way the non shared status is a transitive property: non shared status propagates to the whole reachable object graph.

From the type system point of view, we use the non shared (**ns**) modifier in order to denote non shared references. In order to stay closer to Java, for the scope of this article we will keep shared annotation as the default, that is, no (explicit) modifier. Any shared reference can be used in place of a non shared one, that is, shared references are a subtype of non shared ones.

Non shared references allow us to prevent the problem shown in the previous section. FJ′ type system guarantees that a partially initialised object becomes fully initialised only if the execution of the multiple initialisation expression can be shown not to inject placeholders declared by outer multiple initialisation expressions into the non placeholder fields of the freshly created objects.

One simple way to ensure this, is to require free variables to be non shared. That is, with the current declaration of the factories interfaces the only way to write the `User` class would be as follows:

```
class User{
  IA^ makeIAIB(IAF af, IBF bf,Data data){
    return {
      IA^ a=af.makeIA('b);
      IB^ b=bf.makeIB('a,data);
    } in a;} }
```

This forces us to produce a partially initialised object as a result of `makeIAIB` method. In order to ensure that a fully initialised object is produced we have to use the non shared modifier as follows:

```
interface IAF{IA^ makeIA(IB' myB) ns;}
interface IBF{IB^ makeIB(IA' myA, Data^ data)ns;}

class AF implements IAF{
  IA^ makeIA(IB' myB)ns{
    return new A(myB);} }
class BF implements IBF{
  IB^ makeIB(IA' myA, Data^ data)ns{
    return new B(myA, data);} }

class User{
  IA makeIAIB(ns IAF af, ns IBF bf, Data data){
    return {
      IA a=af.makeIA('b);
      IB b=bf.makeIB('a,data);
    } in a;} }
```

where we use the C++ convention for type modifiers over **this** for method headers.

As you can see we have added **ns** annotations to methods in `IAF`, `IBF`, `AF`, `BF` and `User`. With this new method signature class `AF2` would not type check.

Fully initialised objects created before the evaluation of a given multiple initialisation expression can be used inside any of the initialisation expressions in it. However they must be treated as partially initialised objects, or the reference to them must be treated as non shared. This is because they may contain placeholders in reachable fields. If we are treating them as partially initialised, it would prevent us from using such objects for initialisation. Hence, in order to integrate the externally generated `Data` inside the freshly created object we have changed the type of the `data` parameter of method `makeIB` from `Data` to `Data^`.

2.7 When does an Object become Fully Initialised?

Any multiple initialisation expression declares a number of local variables that can have one of three kinds of type: fully initialised, partially initialised, or placeholder type:

```
{ T1  v1 = e1; T2^ v2 = e2; T3' v3 = e3; } in e;
```

Here we have three variables declared as part of this multiple initialisation expression: `v1` is declared as fully initialised, `v2` is declared as partially initialised, and `v3` is declared as placeholder. For the fully initialised and partially initialised variables `v1` and `v2`, placeholders `'v1` and `'v2` are available to use as part of the initialisation expressions inside the same block. Since `v3` is declared as placeholder type to begin with, it makes no sense to refer to `'v3` as `v3` is a placeholder.

The important observation is that only those variables that are declared as fully initialised (in our example: only `v1`) will be guaranteed to be fully initialised by FJ′ type system at the end of the initialisation expressions in curly brackets, just before evaluating e on the right of the **in** keyword. Theorem 4 (Initialisation Guarantee) proves exactly this for the FJ′ type system.

To be able to achieve this, consider the possible types the expression e_1 can have to be able to guarantee that `v1` is fully initialised. Clearly, if e_1 has a fully initialised type, then there is no problem. However, what if e_1 has a partially initialised type or a placeholder type?

FJ′ allows e_1 that has a partially initialised type to be associated to a `v1` of a fully initialised type, thus effectively converting a partially initialised type to a fully initialised type, if and only if both of the following conditions are met:

1. e_1 has to have a *shared* type (i.e. not annotated with **ns**);
2. every free variable used inside the multiple initialisation expression, has to either have *non-shared* (**ns**) type or have a fully initialised type but be treated as partially initialised inside the multiple initialisation expression.

Our condition requires e_1 to be of shared type, but no requirement is needed for the types of the local variables (e.g. `v1` to `v3` in the example above), that can be either of shared or non shared (**ns**) type.

For simplicity, FJ′ does not allow expressions of placeholder types to be used to initialise variables of fully initialised type. However, as long as there is no circular dependency between any placeholders involved in the multiple initialisation expression and as long as the above two conditions for partially initialised types are also met for placeholder types, then the placeholder will be replaced with a fully initialised type after the **in** keyword. This can be accommodated by FJ′ in future work.

Our technique allows to manage two principal roles that variables coming from outer scope usually cover:

- they have to be integrated inside the newly created objects;
- they provide interesting behaviour useful for that computation, but are not going to be memorised.

Factory objects fall in the latter category: they perform the initialisation but are not integrated inside the newly created objects. In order to make these two roles explicit at the type level we use partially initialised types for the first role, and non-shared types for the second role.

2.8 Putting It All Together: Circular List

We finish this section by presenting an example to provide a better feel for programming using FJ′. This example shows how to create a circular list of arbitrary length, without relying of any field update operation.

```
class List{int e;  List next;}
class ListProducer{
  List^ mkAll(int e, int n, List' l)ns{
    if(n==1) return new List(e, l);
    return new List(e, this.mkAll(e+1, n-1, l));
    }
  List make(int e,int n){
    return {
      List x = this.mkAll(e, n, 'x);
      }in x; } }
```

A circular list `List` has a field `e` containing a value and a field `next` containing a `List`.

Method `mkAll` takes three parameters: a value `e`, a length `n` and a list `l`. `mkAll` creates a list of length `n` containing values starting from `e` as list elements, ending with list `l`.

Observe how all three different annotations (`^`, `'`, **ns**) are carefully utilised in the declaration of `mkAll` method on line 3.

Finally, Method `make` takes a value `e`, a length `n` and creates a circular list of length `n`. Method `make` performs the circular initialisation and returns a fully initialised `List`. Note how `x` has a fully initialised type even if method `mkAll` is declared with a partially initialised return type.

3. Formalisation

The FJ′ syntax is shown in Figure 1. We assume countably infinite sets of variables x, object identifiers ι, class or interface names C, method names m, and field names f. As in FJ 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. To keep the presentation focused on the problem of circular initialisation, we do not consider class composition operators like the *extends* operator in Java. We showed in [3, 11] how expressive composition operators (subsuming inheritance) can be added to Java-like languages.

p	::=	$\overline{cd}\,\overline{id}$	program
cd	::=	**class** C **implements** $\overline{C}$ { $\overline{fd}\,\overline{md}$ }	class declaration
id	::=	**interface** C **extends** $\overline{C}$ { $\overline{mh;}$ }	interface declaration
fd	::=	$T\,f$;	field declaration
mh	::=	$T\,m(\overline{T\,x})\mathcal{M}^{\mathcal{L}}$	method header
md	::=	mh {**return** e;}	method declaration
e	::=	$x \mid e.m\,(\overline{e}) \mid$ **new** $C\,(\overline{e}) \mid e.f$	
		$\mid$ {$\overline{xe}$} **in** $e \mid {'}x \mid \iota$	expression
xe	::=	$T\,x = e$;	variable initialization
T	::=	$\mathcal{M}C^{\mathcal{S}}$	type
$\mathcal{M}$	::=	**ns** $\mid \epsilon$	type modifiers
$\mathcal{S}$	::=	$' \mid \hat{} \mid \epsilon$	superscript
$\mathcal{L}$	::=	$\hat{} \mid \epsilon$	object initialization status
μ	::=	$\iota \mapsto C(f_1 = v_1, \ldots, f_n = v_n)$	memory
v	::=	$\iota \mid {'}x$	value

Figure 1. Syntax

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 and superscript for the implicit parameter **this**. We write the type modifier and superscript for **this** after the parameters list, following the syntactic convention of C++ **const** modifier.
As in FJ, method bodies are simply ordinary expressions.

For the purpose of circular initialisation, the field update operation is not needed. Moreover admitting field updates will make our claim less meaningful: we are showing that some kinds of heaps are reachable through simple initialisation. In Section 3.7 we explicitly show how to add state modification (i.e. the field update operation). So, oddly enough, we define a language with state, explicitly modelled with heap and object identifiers, but no state modification.

Expressions are variables, method or constructor calls, multiple initialisations, placeholders and (within run time expressions) object identifiers.

Variables can be declared in method headers or inside expressions. An expression is well-formed only if the same variable is not declared twice. Method calls are exactly as in FJ.

Multiple initialisation 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 order of execution for the sub-expressions. Also 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).

A placeholder is an entity that temporarily lies in place of an object that still does not exists. In a closed expression ${'}x$ can occur only inside a multiple initialisation declaring (also) local variable x, and before the **in** keyword. That is, any multiple initialisation expression containing a variable declaration $T\,x = ...$ can contain placeholder ${'}x$ inside its initialisation expressions, and (as usual) the variable x inside its conclusive expression.

Types are composed of three components: a type modifier $\mathcal{M}$, a class name C and a superscript $\mathcal{S}$. A modifier $\mathcal{M}$ have two possible values: non shared **ns** or shared (default value). A superscript $\mathcal{S}$ has three possible values: placeholder ($'$), partially initialised ($\hat{}$) or fully initialised (default value). We use the metavariable $\mathcal{L}$ to denote the last two possibilities: partially initialised and fully initialised.
An heap is a finite map from object identifiers to records annotated with a class name.

Values are object identifiers ι or placeholders ${'}x$. Note that values do not include **null** and our language has no concept of default initialisation.

xv ::= $T\,x = v$; assignment value

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

(CTX)
$$\frac{\mu_1 \mid e_1 \;\to\; \mu_2 \mid e_2}{\mu_1 \mid \mathcal{E}[e_1] \;\to\; \mu_2 \mid \mathcal{E}[e_2]}$$

(FACCESS)
$$\frac{}{\mu \mid \iota.f \;\to\; \mu \mid v} \quad \text{with } \mu(\iota).f = v$$

(CONSTRUCTOR)
$$\frac{}{\mu \mid \textbf{new}\; C\,(v_1, \ldots, v_n) \;\to\; \mu, \iota \mapsto C(f_1 = v_1, \ldots, f_n = v_n) \mid \iota}$$
with
$\iota \notin \mathsf{dom}(\mu)$
$p(C) = \textbf{class}\; C\; \textbf{implements}\; \_\; \{\; \_f_1; \ldots; \_f_n; \overline{md}\; \}$

(METH-INVK)
$$\frac{}{\mu \mid \iota.m\,(v_1 \ldots v_n) \;\to\; \mu \mid e[x_1 = v_1, \ldots, x_n = v_n, \textbf{this} = \iota]}$$
with
$\mu(\iota) = C\,(\_)$
$p(C).m = \_\,m(\_x_1, \ldots, \_x_n)\_^{\sim}\; \{\textbf{return}\; e;\}$
$v_i \notin \mathsf{dom}(\mathsf{ph}(e)) \quad \forall i \in 1..n$

(INIT)
$$\frac{}{\mu \mid \{\overline{xv}\}\; \textbf{in}\; e \;\to\; \mu[\overline{xv}] \mid e[\overline{xv}]}$$

Figure 2. Reduction rules

Placeholders are replaced by object identifiers when multiple initialisation expressions are reduced. Thus we expect final values to be only object identifiers.
For example, given as a program **class** `C{C f;}`, the heap $\iota_1 \mapsto \mathtt{C(f='y)}$, $\iota_2 \mapsto \mathtt{C(f=}\iota_1\mathtt{)}$, $\iota_3 \mapsto \mathtt{C(f=}\iota_3\mathtt{)}$ contains three object identifiers for objects of class `C`:

- ι_1 refers to an object containing a single field named `f` pointing to the placeholder `'y`,
- ι_2 refers to an object containing a single field named `f` pointing to ι_1. Thus, ι_1 and ι_2 denote two *partially initialised objects*: objects containing a placeholder or another partially initialised object where a *fully initialised object* is expected.
- ι_3 denotes an object containing a single field named `f`, referencing to ι_3 itself. Thus ι_3 is a fully initialised object.

In order to keep the formalization of FJ′ simple, it is illegal to access any fields of partially initialised objects. We plan to address this limitation in future work, as discussed in [15].

The heap is well-formed with respect to a program if the fields of the records are exactly the fields declared inside the corresponding classes.

3.1 Subtyping

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

- $\mathcal{M}\,C \leq \mathcal{M}\,C^{\hat{}} \leq \mathcal{M}\,C'$
- $T \leq T$
- $C^{\mathcal{S}} \leq$ **ns** $C^{\mathcal{S}}$
- $\mathcal{M}\,C_1^{\mathcal{S}} \leq \mathcal{M}\,C_2^{\mathcal{S}}$ if
  $p(C_1) =$ **class** C_1 **implements** $C, \_\,\{\,\_\,\}$ and $\mathcal{M}\,C^{\mathcal{S}} \leq \mathcal{M}\,C_2^{\mathcal{S}}$
- $\mathcal{M}\,C_1^{\mathcal{S}} \leq \mathcal{M}\,C_2^{\mathcal{S}}$ if
  $p(C_1) =$ **interface** C_1 **extends** $C, \_\,\{\,\_\,\}$ and $\mathcal{M}\,C^{\mathcal{S}} \leq \mathcal{M}\,C_2^{\mathcal{S}}$

Implemented interfaces induce subtyping as in Java, subtyping induced by the modifier and the superscript is straightforward.

3.2 Reduction

Reduction is an arrow over pairs consisting of an heap and an expression. To improve readability we will mark in grey the heap part. A pair $\mu \mid e$ is well-formed only if all the placeholders contained in the heap μ are bound by the local variables declared inside the expression e.
We omit the formal definition of the evaluation context, assuming a standard deterministic left-to-right call-by-value reduction strategy. We do not define a language with lazy evaluation, where expressions are evaluated when the result is really needed.

$\vdash cd : \text{ok} \qquad \vdash id : \text{ok}$

(CLASS)
$$\frac{C_0 \vdash md_i : \text{ok} \quad \forall i \in 1..n}{\vdash \textbf{class}\ C_0\ \textbf{implements}\ \overline{C}\ \{\ \overline{fd}\ md_1 \dots md_n\ \} : \text{ok}}$$
with
$\forall C \in \overline{C}, m$ s.t. $p(C).m = T\ m(\overline{T\ x})\mathcal{M}^{\mathcal{S}}{}_-$,
$T\ m(\overline{T\ x})\mathcal{M}^{\mathcal{S}}{}_- \in md_1 \dots md_n$

(INTERFACE)
$$\frac{}{\vdash \textbf{interface}\ C_0\ \textbf{extends}\ \overline{C}\ \{\ mh_1\texttt{;}\ \dots\ mh_n\texttt{;}\ \} : \text{ok}}$$
with
$\forall C \in \overline{C}, m$ s.t. $p(C).m = T\ m(\overline{T\ x})\mathcal{M}^{\mathcal{S}}{}_-$,
$T\ m(\overline{T\ x})\mathcal{M}^{\mathcal{S}}{}_- \in mh_1 \dots mh_n$

$C \vdash mh\ \{\textbf{return}\ e\texttt{;}\ \} : \text{ok}$

(METH-T)
$$\frac{\textbf{this}{:}\mathcal{M}\,C^{\mathcal{L}}, x_1{:}T_1, \dots, x_n{:}T_n; \emptyset; \emptyset \vdash e : \_ \leq T}{C \vdash T\ m(T_1\ x_1\texttt{,}\ \dots\texttt{,}\ T_n\ x_n)\mathcal{M}^{\mathcal{L}}\ \{\textbf{return}\ e\texttt{;}\ \} : \text{ok}}$$

Figure 3. Typing rules for classes, interfaces and methods

Figure 2 defines the reduction arrow.

Rule (CTX) is standard, however note that alpha-substitution conversion can be needed to ensure the well-formedness of the resulting expression.

Rule (FACCESS) extracts the value of field f. We use the notation $\mu(\iota).f$ for the value of field f of object ι. We assume a fixed program p and we use notation $p(C).m$ to extract the method declaration.

Rule (CONSTRUCTOR) reduces constructor invocations,

Rule (METH-INVK) models a conventional method call. Methods where the receiver is a partially initialised object can never (even transitively) access the fields of the receiver. However, such methods are more expressive with respect to Java static methods, since they provide a concept of dynamic type and, thus, dynamic method dispatch. We use the notation $e[x_1 = v_1, \dots, x_n = v_n]$ for variable substitution, that is, we simultaneously replace all the occurrences of x_i in e with v_i. The last side condition ensures that no placeholder inside the set of values $v_1 \dots v_n$ is accidentally captured when injected inside the expression e. Alpha-substitution can be used to satisfy this side condition. From any expression it is possible to extract a map from placeholders to their declared type, denoted by $\mathsf{ph}(e)$. Formally:

$\mathsf{ph}(\iota) = \emptyset, \quad \mathsf{ph}(x) = \emptyset, \quad \mathsf{ph}(e.m\,(\overline{e})) = \mathsf{ph}(e), \mathsf{ph}(\overline{e})$,
and $\mathsf{ph}(\{\ T_1\ x_1 = e_1\texttt{;}\ \dots\texttt{;}\ T_n\ x_n = e_n\texttt{;}\ \}\ \textbf{in}\ e_0) =$
$'x_1{:}T_1, \dots, 'x_n{:}T_n, \mathsf{ph}(e_0), \dots, \mathsf{ph}(e_n)$.

Rule (INIT) reduces multiple initialisation expressions. Of course $e[\overline{xv}]$ denotes simultaneous replacement of variables with values. We use the analogous notation $\mu[\overline{xv}]$ to denote simultaneous replacement of placeholders with values. Formally:

$[\overline{xv}] = \emptyset$
and $(\mu, \iota \mapsto C(x_1 = v_1, \dots, x_n = v_n))[\overline{xv}] =$
$\mu[\overline{xv}], \iota \mapsto C(x_1 = v_1[\overline{xv}], \dots, x_n = v_n[\overline{xv}])$.

3.3 Expressions Typing Rules

In the following we present the typing rules: rules for typing multiple initialisation expressions are in Figure 5 and actually show our technique, while rules for classes, interfaces and methods and conventional expressions are in Figure 3 and Figure 4 respectively. Such rules are standard, with the only additions being that **ns** modifiers have to be taken into account and a slight relaxation in the rules for the constructor call typing.

$\Gamma; \Gamma' ; \Sigma \vdash e : T$

(VAR-T)
$$\frac{}{\Gamma; \_; \_ \vdash x : T}$$
with
$\Gamma(x) = T$

(METH-INVK-T)
$$\frac{\Gamma; \Gamma' ; \Sigma \vdash e_0 : \_\ C_0^- \leq_{\mathcal{M}} C_0^{\mathcal{L}} \quad \Gamma; \Gamma' ; \Sigma \vdash e_i : \_ \leq T_i \quad \forall i = 1..n}{\Gamma; \Gamma' ; \Sigma \vdash e_0.m\,(e_1\texttt{,}\ \dots\texttt{,}\ e_n) : T}$$
with
$p(C_0).m = T\ m(T_1\ x_1\texttt{,}\ \dots\texttt{,}\ T_n\ x_n)\,\mathcal{M}^{\mathcal{L}}{}_-$

(NEW-T1)
$$\frac{\Gamma; \Gamma' ; \Sigma \vdash e_i : T_i \leq T_i' \quad \forall i = 1..n}{\Gamma; \Gamma' ; \Sigma \vdash \textbf{new}\ C\,(e_1 \dots e_n) : \mathcal{M}\ C}$$
with
$p(C) = \textbf{class}\ C\ \textbf{implements}\ \_\ \{\ T_1'\_\texttt{;}\ \dots\texttt{;}\ T_n'\_\texttt{;}\ \overline{md}\ \}$
$\mathcal{M} = \textbf{ns}$ if $\textbf{ns}\ \_^- \in T_1 \dots T_n$

(NEW-T2)
$$\frac{\Gamma; \Gamma' ; \Sigma \vdash e_i : T_i \quad \forall i = 1..n}{\Gamma; \Gamma' ; \Sigma \vdash \textbf{new}\ C\,(e_1 \dots e_n) : \mathcal{M}\ C^{\wedge}}$$
with
$p(C) = \textbf{class}\ C\ \textbf{implements}\ \_\ \{\ T_1'\_\texttt{;}\ \dots\texttt{;}\ T_n'\_\texttt{;}\ \overline{md}\ \}$
$\mathcal{M} = \textbf{ns}$ if $\textbf{ns}\ \_^- \in T_1 \dots T_n$
$\mathsf{fullyInit}(T_1 \dots T_n) \leq T_1' \dots T_n'$

(FACCESS-T1)
$$\frac{\Gamma; \Gamma' ; \Sigma \vdash e : C}{\Gamma; \Gamma' ; \Sigma \vdash e.f : T}$$
with
$p(C) = \textbf{class}\ C\ \textbf{implements}\ \_\ \{\ \_\ T\ f\texttt{;}\ \_\ \}$

(FACCESS-T2)
$$\frac{\Gamma; \Gamma' ; \Sigma \vdash e : \textbf{ns}\ C}{\Gamma; \Gamma' ; \Sigma \vdash e.f : \textbf{ns}\ C_1^{\mathcal{S}}}$$
with
$p(C) = \textbf{class}\ C\ \textbf{implements}\ \_\ \{\ \__{\mathcal{M}}\ C_1^{\mathcal{S}}\ f\texttt{;}\ \_\ \}$

Figure 4. Typing rules for expressions

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. 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.

Rule (METH-T) derives the type for **this** from the class name C, the type modifier $\mathcal{M}$ and the superscript $\mathcal{L}$.

Methods where **this** is of type $\_\ C^{\wedge}$ can be safely called over partially initialised objects. Such methods are restricted to not access any field: see rules (FACCESS-T1) and (FACCESS-T2). In the premise we use notation $\Gamma; \Gamma' ; \Sigma \vdash e : T_1 \leq T_2$ as a shortcut for $\Gamma; \Gamma' ; \Sigma \vdash e : T_1$ and $T_1 \leq T_2$.

The typing judgement for expression is presented in Figure 4 and uses the following environments:

Γ	::=	$\overline{x{:}T}$	variable environment
Γ'	::=	$\overline{'x{:}T}$	placeholder environment
Σ	::=	$\overline{\iota{:}C^{\mathcal{L}}\ x_1 \dots x_n}$	heap environment

Variable environment Γ is a map from variable names into types. Placeholder environment Γ' is a map from placeholders into types. Heap environment Σ is a map that for any location stores its class name C, its initialisation status $\mathcal{L}$, and the set of placeholders in its reachable object graph.

Type judgement is of the form $\Gamma; \Gamma' ; \Sigma \vdash e : T$, where Γ is needed to type expressions with free variables, Γ' is needed to type expressions with placeholders and Σ is needed to type expressions with locations. Only expressions inside multiple initialisation expressions are typed with a non empty Γ', representing the placeholders introduced by that multiple initialisation expressions.

$\Gamma; \Gamma' ; \Sigma \vdash e : T$

(PH-T)
$$\frac{}{\_; \Gamma' ; \_ \vdash {'x} : C'} \quad \text{with } \Gamma'('x) = C^{\mathcal{L}}$$

(INIT1)
$$\frac{\mathsf{lock}(\Gamma); \Gamma'_0; \Sigma \vdash e_i : T'_i \quad \forall i = 1..n \qquad \Gamma, x_1{:}T_1, \ldots x_n{:}T_n; \Gamma'_1; \Sigma \vdash e : T}{\Gamma; \Gamma'_1; \Sigma \vdash \{\, T_1\ x_1 = e_1; \ldots; T_n\ x_n = e_n; \,\}\ \mathbf{in}\ e : T}$$
with $\Gamma'_0 = {'x_1}{:}T_1, \ldots, {'x_n}{:}T_n$, $\mathsf{unlock}(T'_1, \ldots T'_n) \leq T_1 \ldots T_n$

(INIT2)
$$\frac{\Gamma; \Gamma'_0; \Sigma \vdash e_i : \_ \leq T_i \quad \forall i = 1..n \qquad \Gamma, x_1{:}T_1 \ldots x_n{:}T_n; \Gamma'_1; \Sigma \vdash e : T}{\Gamma; \Gamma'_1; \Sigma \vdash \{\, T_1\ x_1 = e_1; \ldots; T_n\ x_n = e_n; \,\}\ \mathbf{in}\ e : T}$$
with $\Gamma'_0 = \Gamma'_1, {'x_1}{:}T_1, \ldots, {'x_n}{:}T_n$

(SUB-EXP-T)
$$\frac{\Gamma; \Gamma' ; \Sigma \vdash e_0 : T_0 \qquad \Gamma, x_0{:}T_0; \Gamma' ; \Sigma \vdash \{\, \overline{x e}_1\ T\ x = \mathcal{E}[x_0]; \overline{x e}_2 \,\}\ \mathbf{in}\ e_1 : T_1}{\Gamma; \Gamma' ; \Sigma \vdash \{\, \overline{x e}_1\ T\ x = \mathcal{E}[e_0]; \overline{x e}_2 \,\}\ \mathbf{in}\ e_1 : T_1}$$

Figure 5. Type system for circular initialisation expressions

Rules (VAR-T) and (METH-INVK-T) are standard and straightforward. Note that the last _ in the side condition of (METH-INVK-T) could be either a method body or a semicolon, depending on C_0 being a class or an interface. Since (METH-INVK-T) is the only expression typing rule using the program p, and p is only used to recover the method header, there is no need to examine the method implementations during expression type checking.

Rules (NEW-T1) and (NEW-T2) types constructor calls. In both rules the resulting object is non shared if at least one of the method parameter is non shared.

In (NEW-T1) the parameters have to be subtypes of the object fields. The result is a fully initialised instance of C.

Rule (NEW-T2) is more liberal, allowing parameters to be placeholders or partially initialised objects, independently of the corresponding superscript declared in the class body. This is formally modelled with the function $\mathsf{fullyInit}(T)$, that marks all the parameters as fully initialised.
Formally $\mathsf{fullyInit}(\mathcal{M}\ C^{\mathcal{S}}) = \mathcal{M}\ C$.
In this case the result is a partially initialised instance of C.

Rule (FACCESS-T1) ensures that fields can be accessed only from fully initialised objects, while rule
(FACCESS-T2) ensures also the propagation of (**ns**) property over the whole reachable object graph.

3.4 Circular Initialisation Typing Rules

Rule (PH-T) types a placeholder literal using the placeholder environment. The placeholder literal can only have a shared placeholder type. Moreover the variable corresponding to the placeholder has to be declared with: shared and partially initialised type or shared and fully initialised type.
Rules (INIT1) and (INIT2) type multiple initialisation expressions.

Rule (INIT1) verifies multiple initialisation expressions where shared partially initialised object identifiers become fully initialised. Informally $\mathsf{lock}(\Gamma)$ ensures that placeholders declared into outer scopes are not injected in shared object identifiers, while $\mathsf{unlock}(\overline{T})$ unlocks shared partially initialised types; note that $\mathsf{unlock}(\overline{T})$ is used over types of the initialisation expressions.
For example, we can type check the following code:

```
class C{C myC;}
class User{
  C makeC(){
    return { C x= new C('x); } in x;} }
```

Class `User` has a `C makeC()` method performing circular initialisation. Of course placeholder `'x` has type `C'`.
new `C('x)` is of type `C^`, thanks to (NEW-T2); since
$\mathsf{unlock}(\texttt{C\^{}}) = \texttt{C}$, **new** `C('x)` can be used to initialise variable `C x`.

Function $\mathsf{unlock}(\Gamma)$ converts shared partially initialised ($\hat{}$) types to shared fully initialised ones (default superscript), and is the identity on other types. We define the $\mathsf{unlock}(\_)$ function as follows:

$\mathsf{unlock}(\mathcal{M}\ C) = \mathcal{M}\ C$
$\mathsf{unlock}(\mathcal{M}\ C') = \mathcal{M}\ C'$
$\mathsf{unlock}(\mathbf{ns}\ C\hat{}) = \mathbf{ns}\ C\hat{}$
$\mathsf{unlock}(C\hat{}) = C$

We define the $\mathsf{lock}(\_)$ relation as follows:

$\mathsf{lock}(\_\ C^{\mathcal{S}}) = \mathbf{ns}\ C^{\mathcal{S}}$
$\mathsf{lock}(\mathcal{M}\ C) = \mathcal{M}\ C\hat{}$

Even if the definition of $\mathsf{lock}(\Gamma)$ looks very simple, it is not trivial. Placeholders and partially initialised types are converted into non shared (**ns**) types using first clause. However, there are two possible conversion for fully initialised types: either using the first or the second clause.

If the first clause is used, the intended role of the referenced object is to provide useful behaviour: using non shared types we enforce that the object (and any object in its reachable object graph) are not reachable by the newly created objects with shared type.

If the second clause is used, the intended role of the referenced object is to be integrated in the newly created objects: Accessing the object using only a partially initialised type prevents any field from being accessed within the scope of the initialisation.

We abuse the notation and use $\mathsf{lock}(\Gamma)$ as a function, with the meaning that any possible output produces a valid premise for the meta-rule (INIT1).[7]
Rule (INIT2) is applied when shared partially initialised variables are not guaranteed to becomes fully initialised. Note the absence of $\mathsf{lock}(\_)$ and $\mathsf{unlock}(\_)$.
Consider the following example:

```
class C{C myC;}
class User{
  C^ makeC(C' y){
    return {//typed with init-2
      C^ x= new C(y);
      } in x;}
  ns C^ makeCNS(C' y){
    return {//typable with init-1 or init-2
      ns C^ x= new C(y);
      } in x;}
  ns C makeCNSWrong(C' y){
    return {//ill-typed
      ns C x= new C(y);
      } in x;} }
```

Method `C^ makeC(C' y)` takes a placeholder and returns a partially initialised object. Variable `y` inside **new** `C(y)` is a reference to an external object. Rule (INIT-1) would apply $\mathsf{lock}(\texttt{C'y})$. In this way `y` is seen of type **ns** `C'`. Thus **new** `C(y)` is of type **ns** `C^`. Since `x` is declared of type `C^` its initialisation expression have to be of shared type, (INIT-1) can not be applied. However, (INIT-2) can be smoothly applied.

Variable `x` in method `makeCNS` is of non shared type, so we can apply both (INIT-1) or (INIT-2) to type check method `makeCNS`. However, since function $\mathsf{unlock}(\_)$ works only over shared types, both (INIT-1) and (INIT-2) would be of no use to type method `makeCNSWrong`.

[7] $\mathsf{lock}(\_)$ could be defined as a function in a richer type language, with an additional type for the intersection of **ns** C and $C\hat{}$. However, we believe the benefit would be inferior compared to the cost of a more complex type language.

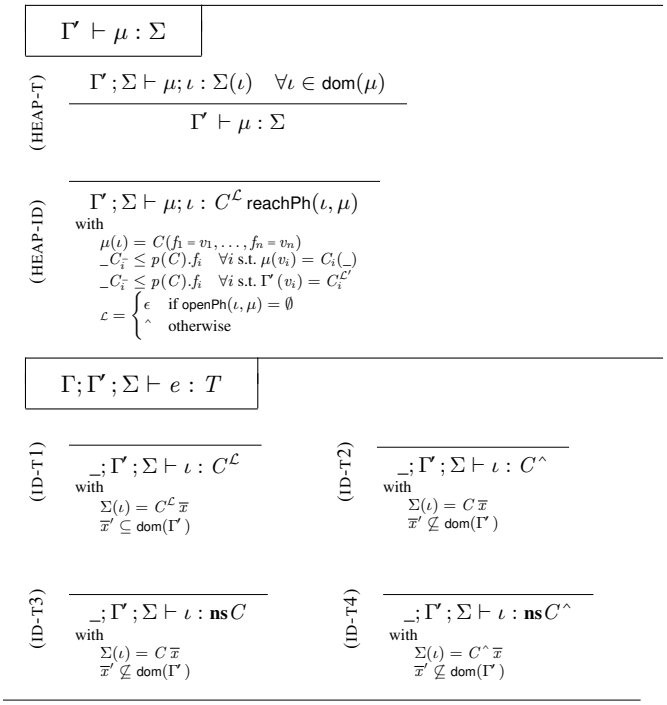

Figure 6. Typing rules for heaps and object identifiers

Rule (INIT-2) is very useful to save temporary results in a variable, as is usually done with the **let** construct. Moreover it allows to wrap placeholders and not fully initialised objects (coming from outer scopes) inside object fields. For example

```
class C{C x1; C x2;}
class User{
  C^ makePartialC(C' x1) ns{
    return {//typed with init-2
      C^ x2=new C(x1,'x2);
      } in x2;}
  C makeC(){
    return {//typed with init-1
      C x1=this.makePartialC('x1);
      } in x1;}}
```

Method `makePartialC` is typed using (INIT-2), while method `makeC` is typed using (INIT-1), typing the object created by `makePartialC` with a fully initialised type.

Note how (INIT2) covers all the cases where conventionally a **let** (or a local variable declaration) is used.

As you can see, the point where an object is ensured to be fully initialised is a type property, not a purely syntactical notion. This allow us to safely express initialisation patterns where the work of Syme [17] would have signalled a (false positive) dynamic error.

Since (INIT1) uses variables and variable locking to distinguish between internal and external references, rule (SUB-EXP-T) allows to see any sub-expression in a multiple initialisation expression as an externally declared variable. Without this rule our system would require the programmer to manually factorise such sub-expressions inside variables.

3.5 Heap Typing Rules

Figure 6 defines typing rules for heaps and object identifiers. We underline that rules for typing a non empty heap are needed in order to type run-time expressions. The program execution will always start from the empty heap, typed in the empty heap environment.

Rule (HEAP-T) types a heap μ in a heap environment Σ if all the object identifiers are well typed. We use Γ' in order to keep trace of the types of the placeholders contained in the heap.

A pair heap, object identifier is typed by (HEAP-ID). Second side condition checks that all the locations have a compatible type with respect to the one declared in the class C, while third side condition checks that all the placeholders have a compatible type with respect to the one declared in the class C. The location is typed with the class name and with the set of placeholders that are reachable from that location. Formally:

- $x \in \mathsf{reachPh}(\iota, \mu)$ iff $\mu(\iota).\_ = x$ or $\mu(\iota).\_ = \iota'$ and $x \in \mathsf{reachPh}(\iota', \mu)$

The last side condition check if the location denote a partially initialised object or not. $\mathsf{openPh}(\iota, \mu)$ denotes the set of placeholders that have to be replaced in order to consider initialisation of ι complete. Formally:

- with $\mu(\iota) = C(f_1 = v_1, \ldots, f_n = v_n)$, $'x \in \mathsf{openPh}(\iota, \mu)$ iff exist $i \in 1..n$ s.t. $p(C).f_i$ is of fully initialised type and $'x \in \mathsf{openPh}(v_i, \mu)$ or $p(C).f_i$ is of fully or partially initialised type and $v_i = 'x$.

Rules (ID-T1), (ID-T2), (ID-T3) and (ID-T4) type object identifiers inside expressions.

Rule (ID-T1) types object identifiers (transitively) referring only to the placeholders defined in Γ'. As you can see in rule (INIT-1) and (INIT-2), those are the placeholders that will be replaced when the directly enclosing multiple initialisation expression typed with rule (INIT-1) will be reduced.

Rules (ID-T2) and (ID-T3) type object identifiers ι for fully initialised objects (transitively) referring to placeholders not contained in Γ'. As for $\mathsf{lock}(\_)$, there are two possibilities: ι has a shared partially initialised type (ID-T2) or has a non shared fully initialised type (ID-T3). Finally rule (ID-T4) types object identifiers for partially initialised objects (transitively) referring to placeholders not contained in Γ'.

That is, the heap environment Σ maps object identifiers with their initialisation status. However, partially initialised types, identified by the ($\hat{}$) superscript, are a slightly different concept: all the partially initialised objects are typed with partially initialised types. However, according to the rules (ID-T1)...(ID-T4), fully initialised objects referring to placeholders declared in outer multiple initialisation expressions can be typed with either a partially initialised type or a non shared fully initialised type, but not with a shared fully initialised type.

3.6 Soundness

Now we can proceed with the statement of soundness:

Theorem 1 (Soundness). *For all well typed programs p and for all expressions e under p, if $\emptyset; \emptyset; \emptyset \vdash e : T$ and $\emptyset \mid e \xrightarrow{*} \mu \mid e'$, then either e' is of form ι or $\mu \mid e' \rightarrow \_ \mid \_$*

As usual, soundness can be derived from progress and subject reduction properties.

Theorem 2 (Progress). *For all well typed programs p and for all expressions e and heap μ under p, if $\mathsf{ph}(e) \vdash \mu : \Sigma$ and $\emptyset; \mathsf{ph}(e); \Sigma \vdash e : T$, then either e is of form ι, e is of form $'x$ with $'x \in \mathsf{dom}(\mathsf{ph}(e))$ or $\mu \mid e \rightarrow \_ \mid \_$*

Theorem 3 (Subject Reduction). *For all well typed programs p and for all expressions e and heap μ under p, if $\mathsf{ph}(e) \vdash \mu : \Sigma$, $\Gamma; \mathsf{ph}(e); \Sigma \vdash e : T$, and $\mu \mid e \rightarrow \mu' \mid e'$ then $\mathsf{ph}(e') \vdash \mu' : \Sigma'$, $\Gamma; \mathsf{ph}(e'); \Sigma' \vdash e' : T'$ and $T' \leq T$.*

Finally, Subject Reduction requires Initialisation Guarantee, that form the centrepiece of our proof of Soundness.

Theorem 4 (Initialisation Guarantee).
*If $\Gamma' \vdash \mu : \Sigma_0$ and $\emptyset; \Gamma'; \Sigma_0 \vdash \{\overline{xv}\}$ **in** $e : T$*
with $\overline{xv} = T_1\ x_1 = v_1 \ldots T_n\ x_n = v_n$
then $\Gamma' \vdash \mu[\overline{xv}] : \Sigma_1$ and $\emptyset; \Gamma'; \Sigma_1 \vdash v_i : \_ \leq T_i \quad \forall i \in 1..n$

The proofs can be found in the technical report [15].

3.7 Extension with field Update

We have defined our language without the field update operation, however it can be easily defined in the following way:

$$e \quad ::= \quad \ldots \mid e_0.f{=}e_1$$

(FUPDATE-T) (FUPDATE)

$$\frac{}{\mu \mid \iota.f{=}v \;\rightarrow\; \mu[\iota.f = v] \mid v}$$

$$\frac{\Gamma;\Gamma';\Sigma \vdash e_0 : \mathcal{M}\, C_0^{\mathcal{L}} \qquad \Gamma;\Gamma';\Sigma \vdash e_1 : C_1^{\mathcal{S}}}{\Gamma;\Gamma';\Sigma \vdash e_0.f{=}e_1 : C_1^{\mathcal{S}}}$$

with
$p(C_0) = \textbf{class}\ C_0\ \textbf{implements}\ \_\ \{\ \_\ T\ f_i\ \_\ \}$
$T_1 \leq T$
$T_1 = \epsilon\_^{-}$

Where we use notation $\mu[\iota.f = v]$ to override the heap location $\iota.f$ with the value v. As you can see, rule (FUPDATE-T) ensures that the assigned value is, indeed, shareable (default modifier).

4. Discussion

From a static point of view it is possible to resolve circularity thanks to circular type annotations between classes. Techniques like virtual classes [5], C++ templates, and redirect-operator [3] can be used to flexibly modify such dependencies in derived/specialised classes. However, where there is the need to delay the configuration up to instantiation time such approaches are useless. We hope that our solution can inspire new patterns in this context.

For example, in the field of actors systems there is a clear need for circular initialisation support. Any actor potentially needs to reference all the other ones in order to perform message delivery. Our declarative syntax would provide a very intuitive way to connect *input* and *output* sources for actors.

FJ′ only works with FJ-like simple constructors. Indeed, full Java constructors do not provide strong guarantees, and for this reason we cannot extend our approach over them. It is possible to mitigate this limitation: Gil and Shragai [9] propose HardHat constructors: constructors that do not expose "this" before the construction process has ended. Such HardHat constructors could potentially be supported by our system, but we prefer to stick over the minimality of our approach: we chose to model classes with only conventional constructors, forcing the use of factory methods for the more expressive construction strategies.

Summers and Müller [16] present an empirical study on the applicability of their proposal for circular initialisation. Their language has partially initialised objects (called free) as FJ′ has, but unlike FJ′ it relies on special constructors. Replacing constructors with factory/creation methods is suggested by Fowler [7] (p. 304) and further empirically shown as a usually good methodology [4]. The combination of these two empirical studies strongly suggests that our approach of moving away from full Java constructors is a good idea since it is practical and usable by developers.

In FJ′, circular initialisation is allowed because of the following features: multiple initialisation expression, placeholders, partially-initialised types, and non shared references. However, we believe that all the FJ′ features are desirable in a general purpose well-designed programming language as discussed below.

Type modifiers $\mathcal{M}$ do not have a direct connection to initialisation, moreover they allow to enforce in a simple way a minimal ownership capability [19]: if a non shared (**ns**) object is passed to a method, after such method invocation the owner(s) of such object is (are) not changed. Non shared (**ns**) fields do not provide to clients the right to change the owner of such object, while shared fields do. Languages providing rich ownership support could probably rely on their native features in order to ensure safe circular initialisation, instead of the *very minimal* $\mathcal{M}$ type modifiers.

Placeholders are values of the form $'x$, temporarily lying in place of real object that does not yet exist. A placeholder is *not* an object and is more like a future [18]: the corresponding object does not exist until the computation that eventually produces it terminates. As a direct consequence, a placeholder has no concept of dynamic type. The dynamic type of the object that will eventually replace some placeholder is independent from the placeholder itself. In all the contexts it is illegal to use a placeholder as a receiver. From the capability [2] point of view, placeholders do not provide capability to compare identities (as in `x==y`), to inspect the dynamic type (as in **instanceof** `x`), and to access fields. Placeholder types are identified by the (′) superscript. Allowing fields of placeholder type provides an important expressive power: it allows to declare data structures to maintain and reason over placeholders; for example many possible values to potentially complete a doubly linked list can be stored, and one can be finally chosen depending on user input.

Placeholders are replaced with object identifiers when the multiple initialisation expression in which are declared is reduced. The strong similarity between placeholders and futures, and the intuitive fork-join semantics of multiple initialisation suggests that it should be possible to use such features for parallelisms purposes and not only for initialisation concerns.

An object identifier is a value of the form ι, where the object it refers to can be partially initialised or fully initialised. From the capability point of view, object identifiers of partially initialised type provide capability to compare identities and to inspect the dynamic type, but not to access fields. While the ones of fully initialised type provide capability to compare identities, to inspect the dynamic type and to access fields.

5. Related Work

As some astutely observed [6, 17], recursive bindings in a functional language like OCaml [12] serve a purpose very similar to placeholders:

-OCaml-
```
type t = A of t * t | B of t * t
let rec x = A( x, y )
  and y = B( y, x )
```

Here, two variables (`x` and `y`) are circularly initialised using the **let rec** recursive binding expression. OCaml imposes heavy restrictions [6]: *" the right-hand side of recursive value definitions to be constructors or tuples, and all occurrences of the defined names must appear only as constructor or tuple arguments. "*
These restriction make it impossible to express any variations of the Factory pattern [8] as method calls are prohibited in such initialisation expressions. However, thanks to these restrictions, the following implementation for such recursive bindings is possible: first allocate memory for the values being constructed. Once all bindings are established, the constructed values can be initialised normally.

One main difference between [6] and FJ' is that in Delayed Types [6] the allocation is fixed before computation involving `x` can even start; as the syntax "**let** `x`=**alloc** `C[t]` **in** `e`" suggests, it is impossible to decide the final value of `x` using a computation over `x`. It is only possible to decide the value of `x`'s fields. In parallel to Java this means that the exact type of the `x` variable has to be fixed before the computation over `x` starts. In Haskell, this means that the type constructor has to be fixed. In particular, in the example in following subsection:

Haskell
```
let x=(
  let y=(LCons 42 x) in (fi y)
)
```

`fi` is able to decide everything about the `x` value: different `fi` can provide different values, created with different type constructors.

In an extreme setting, the (implementation details of) function `fi` could ignore `y` and provide any constant.

Languages with a lazy semantic like Haskell can use laziness to encode placeholders, thus reaching an expressive power similar to the one of FJ′; but execution of initialisation code is indefinitely delayed. We discuss this in Section 5.1.

We believe that the best initialisation pattern is the factory, taking in input all the needed dependencies. In the absence of laziness, the conventional Java-like semantic is not enough to "express" such dependencies, in the case of circular dependencies, making impossible to use the Factory pattern [8]. We discuss the resulting "antipatterns" in Section 5.2. Note that Fahndrich et al. [6] do not allow Factory since allocation and initialisation phases are separated, and factories encapsulate both allocation and initialisation.

5.1 Comparison with Lazy Languages

The main problem faced by lazy circular initialisation is non well guarded variable definitions, as we describe below.

The significance of this problem is highlighted in a recent work by Syme [17]. He uses a disciplined form of laziness in order to design a language which is very similar in spirit to our approach, with roughly as much power but an additional requirement for dynamic checks in order to avoid non well-guarded cases to be evaluated in the context of data recursion, i.e. circular initialisation.

Since we are defining a call-by-value language, the FJ′ type system rules out such degenerated cases from the start, avoiding the need for dynamic checks.

Those degenerate cases are not so uncommon as one can suppose. While it would be equally applicable to any language that supports laziness such as Scala, C#, F#, we use Haskell for our examples in this section as the most recognised lazy language. In Haskell, a trivially degenerate case is the follows:

Haskell
```
data L = Lnil | Lcons Int L
ltop :: L->Int
ltop Lnil=0
ltop (Lcons n a) = n
main=(
  let x=x
  in putStr(show(ltop x)))
```

Here type `L` denotes a list of integers, with the two conventional type constructors. The function `ltop` returns the top of the list, or `0` for the empty list. it is clear that the definition of the main makes no sense. This definition really define nothing: we call it a *non well-guarded definition*.

In order to explain the importance of this problem, we now show an example where a non well-guarded expression emerges from an apparently benign code. Indeed in the general case is very difficult to spot non well-guarded definitions.
Consider the following two functions `f1` and `f2`:

Haskell-
```
f1 :: L -> L
f1 (a) = a
```
```
f2 :: L -> L
f2 Lnil = Lnil
f2 (Lcons n a) = a
```

Function `main1` correctly initialises an infinite list containing only the number 42, and shows 42. However, function `main2` is a non-well guarded definition:

Haskell
```
main1=(
 let x=(
   let y=(LCons 42 x) in (f1 y)
   )
 in putStr(show(ltop x)))
```

Haskell
```
main2=(
 let x=(
   let y=(LCons 42 x) in (f2 y)
   )
 in putStr(show(ltop x)))
```

Since the only difference is the occurrence of `f2` instead of `f1`, you can see that in Haskell, without knowing the code of `f1` and `f2`, it is impossible to predict whether some code is well-guarded or not.

In fact, executing `main1` in Haskell results in printing `42` while executing `main2` results in an infinite loop that is not prevented by the Haskell compiler. Using the terminology of FJ′, in the Haskell example the "externally-declared placeholder `x`" is injected in the created value `(LCons 42 x)`.

In the following we show the code of the last example rephrased in FJ′; we encode as methods `f1`, `f2`, `main1` and `main2` of the last Haskell example. Of course `main1` is verified by our type system, while `main2` is ill-typed.

```
class List{
 int e;
 List next;
 List^ f1()^{ return this; }
 List f2(){ return this.next; }
 List main1(){
   return {//well-typed
     List x={
       List^ y=new List(42,'x);
       } in y.f1()
     }in x;}
 List main2(){
   return {//ill-typed
     List x={
       List^ y=new List(42,'x);
       } in y.f2()
     }in x;} }
```

FJ′ recognises non well-guarded expressions at the type system level, dividing the multiple initialisation expressions into two groups: the ones typed by (INIT-1) and the ones typed by (INIT-2). In this case our type system recognises that the multiple initialisation
`{List^ y=new List(42,x);}` **in** `y.fi()` (for $i \in 1..2$) can be typed only by rule (INIT-2). Indeed `'x` inside
new `List(42,'x)` comes from an outer scope (i.e. it is not declared in the directly enclosing multiple initialisation expression). However, `'x` can be only considered shared in that scope, since the variable `y` is required to be initialised with a shared reference.

That is, `y` has to be typed as a `List^`. At this point `y.f1()` is a valid method call over `y`, since `f1` declares **this** with a partially initialised (`^`) type. Note that an alternative declaration `List f1(){return this;}` would have made the expression `y.f1()` ill-typed. On the other side `y.f2()` is not a valid method call over `y`, since `f2` declares **this** with a fully initialised type. Note that an alternative declaration `List^ f2()^{return this.next;}` would not type check, see (FACCESS-T1) and (FACCESS-T2). Finally, it is not possible to express this example using the existing non lazy approaches [6, 14, 16].

5.2 Circular Initialisation in Pure Java

Since without placeholders the Factory pattern is not applicable in the case of circular dependencies, the approaches that stick to Java semantics [6, 14, 16] focus their attention on the following antipatterns: Two Phase Initialisation and Chained Constructor Creation.

Two Phase Initialisation

There are two main variants for this pattern:
1. – Explicit null-initialisation and field update

```
Java
class A{ B myB; A(B myB){this.myB=myB;} }
class B{ A myA; B(A myA){this.myA=myA;} }
class User{
  A m(){
    A a=new A(null);
    B b=new B(a);
    a.myB=b;
    return a;} }
```

2. – Create and Init

```
Java
class A{ B myB; A(B myB){this.myB=myB;}
  static A makeA(){return new A(null);}
  void init(B b){myB=b;} }
class B{ A myA; B(A myA){this.myA=myA;}
  static B makeB(){return new B(null);}
  void init(A a){myA=a;} }
class User{
  A m(){
    A a=A.makeA();
    B b=B.makeB();
    a.init(b);
    b.init(a);
    return a;} }
```

The first kind of code is hard to maintain: when a modification in the internal implementation of `A` and `B` is needed, all the code that initialises new instances can be broken.

The second version uses the factory methods `makeA` and `makeB`, allowing a subtype of `A` and `B` to be produced. However the factory pattern is not properly applied since a method `init` must be called to complete initialisation.

This kind of solutions are proposed by [6, 14]. Their (complex) approaches ensure the absence of `NullPointerException`s in this kind of code.

Chained constructor creation

In this pattern concrete constructors provide initialisation strategies of their sub-components:

```
-Java-
class A{ B myB; A() { myB=new B(this);}}
class B{ A myA; B(A x){ myA=x;}}//here x.myB==null
```

The code inside the constructor of `B` cannot freely manipulate the parameter `x`, indeed if the control flow starts from expression **new** `A()`, the field `x.myB` is still **null** when variable `x` is visible. This solution is proposed by [16], and again the absence of `NullPointerException`s is proved. The code obtained when programming in this way is not flexible: the construction of `B` has to be fully controlled by the constructor of `A`.

6. Conclusion

Today writing correct and maintainable code involving circular initialisation is very difficult. This hampered the development of effective circular initialisation patterns in software engineering. Programming languages offering more declarative and safer approaches to circular initialisation could endorse many new programming patterns, or at least discourage "antipatterns".

Many approaches challenge the problem of circular data structures initialisation [6, 12, 14, 16]. In our opinion FJ′ offers a simpler type system compared to all the other approaches; moreover it allows simpler and cleaner programming patterns to be used. We can obtain such simplicity as we are attacking the problem from two different sides: first we define an operator with a very intuitive semantics, and then we develop a type system that supports it.

One way to interpret our work is that we are conducting an operational static analysis inside the type system. Our approach is straightforward, and critically, understandable to common programmers. This gives our system a good chance of adoption by real language implementations.

More details including proofs and implementation considerations can be found in the accompanying technical report [15]. We would like to conclude with another well known quote by Hoare: **"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."**

We have tried our best to make our approach falling in the first category, and the result looks pretty simple to us, however, is it simple enough to be a real fix for the billion dollar mistake?

References

[1] M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter. The spec# programming system: Challenges and directions. In *VSTTE 2005, Revised Selected Papers and Discussions*, volume 4171, pages 144–152. Springer, 2005.

[2] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP2001*, pages 2–27. Springer, 2001.

[3] A. Corradi, M. Servetto, and E. Zucca. DeepFJig - Modular composition of nested classes. In *PPPJ2011*, pages 101–110. ACM, 2011.

[4] S. Counsell, G. Loizou, , and R. Najjar. Evaluation of the 'replace constructors with creation methods' refactoring in Java systems. *IET Software*, 4(5):318–333, 2010.

[5] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In J. G. Morrisett and S. L. P. Jones, editors, *POPL2006*, volume 41, pages 270–282. ACM Press, 2006.

[6] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA2007*, pages 337–350. ACM Press, 2007.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[8] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[9] J. Gil and T. Shragai. Are we ready for a safer construction environment? In *ECOOP2009*, volume 5653, pages 495–519. Springer, 2009.

[10] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA1999*, pages 132–146. ACM Press, 1999. doi: 10.1145/320384.320395.

[] G. Lagorio and M. Servetto. Strong exception-safety for checked and unchecked exceptions. *Journal of Object Technology*, Volume 10(1): 1–20, 2011. doi: doi:10.5381/jot.2011.10.1.a1. URL `http://www.jot.fm/contents/issue_2011_01/article1.html`.

[11] G. Lagorio, M. Servetto, and E. Zucca. Featherweight Jigsaw - a minimal core calculus for modular composition of classes. In *ECOOP2009*, volume 5653, pages 244–268. Springer, 2009.

[12] X. Leroy. The Objective Caml system (release 2.00). Available at `http://paulliac.inria.fr/caml`, August 1998.

[13] B. Meyer. Avoid a void: the eradication of null dereferencing. White paper available at http://bertrandmeyer.com/tag/void-safety/, 2009. Submitted to a volume celebrating Hoare's 75th birthday.

[14] X. Qi and A. C. Myers. Masked types for sound object initialization. In Z. Shao and B. C. Pierce, editors, *ACM Symp. on Principles of Programming Languages 2009*, pages 53–65. ACM Press, 2009.

[15] M. Servetto and A. Potanin. Our billion dollar fix. Technical Report 12-19, ECS, VUW, 2012. `http://ecs.victoria.ac.nz/Main/TechnicalReportSeries`.

[16] A. J. Summers and P. Müller. Freedom before commitment - a lightweight type system for object initialisation. In *OOPSLA2011*. ACM Press, 2011.

[17] D. Syme. Initializing mutually referential abstract objects: The value recursion challenge. *Electronic Notes in Theoretical Computer Science*, 148(2):3–25, 2006.

[18] L. Zhang, C. Krintz, and P. Nagpurkar. Language and virtual machine support for efficient fine-grained futures in java. In *PACT2007*, 2007.

[19] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and Immutability in Generic Java. In *OOPSLA2010*, pages 598–617. ACM, 2010.

[20] Y. Zibin, D. Cunningham, I. Peshansky, and V. Saraswat. Object Initialization in X10. In *ECOOP2012*, pages 207–231. Springer, 2012.

A. Implementation

When translating our language to a high level bytecode language (e.g. Java bytecode), the main issues are: (1) to find an adequate run-time representation for placeholders, and (2) to create a fast implementation for the placeholder replacement operation.

In order to satisfy the bytecode verifier, we need to instantiate ad-hoc dummy objects to serve as placeholders. For example any class could have a special subclass used to generate corresponding dummy placeholder objects. In a customised Virtual Machine it would be possible to reserve a little set of memory pages, and pointers to that space would be placeholders. We would never access such space, reserving a range for "special" pointers is enough.

One solution is to maintain a map `phMap` from dummy placeholder objects to objects containing such placeholders in one of their fields. If during initialisation a placeholder is placed inside a field, the map is correspondingly updated. As an optimisation, this check can be performed only during constructor execution for objects with placeholder fields, or while executing constructors typed by rule (NEW-T2).

In this way multiple initialisation expressions can be executed in the following way:

- initialise dummy placeholder objects,
- execute the initialisation expressions using the dummy placeholder objects. During such execution `phMap` is filled,
- for all the objects connected in `phMap` to one of the placeholders declared in the current multiple initialisation expression, replace such placeholder with the correct value obtained in the previous item.

B. Future Work

We now discuss five interesting extensions to FJ′.

B.1 Read Partially Initialised Objects Fields

Accessing a field of a partially initialised object is generally an unsafe operation. FJ′ and other approaches [20] choose to forbid any access to any fields of partially initialised objects. This is a simple solution, however some approaches try to provide a finer grained approach [14]. It is possible to extend FJ′ in order to allow field access over fields of partially initialised objects. Indeed if the receiver is of a shared partially initialised type and the field has a fully initialised type in the receiver class, then the field access operation is safe. In this case the read value can be seen with a shared placeholder type. For example, the following code would be well typed:

```
class C{ C myC; }
class User{ C′ m(C^x){ return x.myC; } }
```

B.2 Static methods

As we have already noticed, a method that requires **this** to be a partially initialised (non shared) variable is somehow similar to a Java static method. We can extend the language allowing the user to write class names C as valid expressions.[8] $e ::= \ldots \mid C$ Such expression would denote a partially initialised non shared object of class C that can never become fully initialised. In this way the field values of such object would be irrelevant.[9] This extension would allow, for example, to do the following:

```
class C {
  int f; C make()ns^{return new C(42);} }
... return C.make().f; ...
```

Note that even if this provides us de-facto static methods, we have no concept of static field. That is, partially initialised types can be leveraged to encode static methods, proving their utility outside the initialisation problem itself.

B.3 Runtime Support

Even if they can be abused, dynamic casts provide useful expressive power to Java-like languages. With the already proposed implementation, performing run-time checks to distinguish placeholders from partially initialised objects is very easy: dummy placeholders objects would have reserved types. On the other side, to check whether an object identifier has a fully initialised or shared type is not trivial. A naive, computationally expensive solution is to navigate the whole object reachable graph, emulating the typing rules for heap and locations. Finding more efficient implementation for this operation is future work.

B.4 Exceptions

The next example shows the possible look and feel of a version of FJ′ extended with exceptions:

```
class User{
  IA m(ns IAF af,ns IBF bf){
    return {
      IA a=af.makeIA('b)
        catch(Exc1 x)
          throw new Exc2("...",x);
      IB b=bf.makeIB('a)
        catch(Exc1 x)
          this.m(af,new BF());
    }in a;} }
```

In this example, if `IA` can not be initialised, we wrap the exception and produce a customised error message, while if `IB` can not be initialised, we try the initialisation providing a new instance of factory for `IB`.

Extending our language with exceptions is however not trivial: if an exception happens inside an initialisation block, no value will ever be used to replace the corresponding placeholders. It is possible to use exactly the same approach used in [**?**] in order to guarantee that whenever an exception is thrown, no modification is visible in the **catch** expression, an thus, no *unresolved* placeholder could leak. Formalising the composition of circular initialisation with exception safety is future work.

C. Hypertext Processing

Suppose we want to develop a library producing an in-memory object-oriented representation of an HTML hypertext,[10] where different implementation for the same HTML element are rendered in different styles. So, for example, `Div` will be an interface and `BoldBorderedDiv` (`BBDiv` for short) will be an implementation. Correspondingly we will have `DivF` and `BoldBorderedDivF` (`BBDivF` for short) as abstract and concrete factory, respectively. Those factories will have a method `fromString(String s)`, that behave like a parser, producing the in-memory representation of the

[8] Interface names would not be allowed as expressions.

[9] Since class names would be typed with non shared types, this extension have no interaction with the previous one.

[10] where, of course, links can provide circular dependencies between pages

corresponding tag, with such HTML string inside. For example:

```
new BBDivF().fromString("foo<div>bar</div>");
```

will produce an instance of class `BBDiv` containing the text `foo` and an inner `BBDiv` instance containing the text `bar`.

`Body` tag is commonly used to represent the content of a web page, while a `Div` tag represents an area on a page (that in modern HTML can contain sub pages by utilizing an `iframe` tag). As a simplification, suppose to have only `Body`s, and `Div`s, where `Body`s can contains `Div`s, and `Div`s can contains `Body`s.

Furthermore, we assume that a page content (`Body`) is either text (created using `fromString`) or a list of links (created using `fromList` method). As an additional requirement, any tag of type `Body` has to contain a link to the homepage (in turn represented by a `Body` object).

```
interface Div{...}
interface DivF{...
  Div fromString(String s);}
interface Body{...}
interface BodyF{...
  Body^ fromString(String s);
  Body^ fromList(Body^ ... links);
  }
class Link{ Body' home;}
//BoldBorderedDiv
class BBDiv implements Div{...}
class BBDivF implements DivF{...
  BodyF bf;}

//WhiteBackgroundBody
class WBBody implements Body{...
  Body'home;}
class WBBodyF implements BodyF{...
  DivF df; Link link;
  Body^ fromList(Body^ ... links){
    return new WBBody(this.link.home,...);
    } }
  Body^fromString(String s){...
    return new WBBody(this.link.home,...);
    } }

class MyCss{
  BodyF style(Body'home)ns {
    return
      {Link l=new Link(home);}in
      {BodyF bf=new WBBodyF(df,l);//Link^ l
       DivF df =new BBDiv(bf);
      }in bf; }  }
```

Interfaces `Div`, `DivF`, `Body` and `BodyF` provide abstract access to the functionality of the library. Class `Link` is nothing more than a wrapper class containing an `Body'`.

Classes `BoldBorderedDiv` and `BoldBorderedDivF`, `WhiteBackgroundBody` and `WhiteBackgroundBodyF` (`BBDiv`, `BBDivF`, `WBBody` `WBBodyF` for short) are concrete implementations and concrete factories for the HTML elements used. We omit most of the code, however we expand the code of `WBBodyF`, that contains a `Link`, a `DivF` and a `make` method. Method `fromString` is of course not trivial: it would parse the string and potentially call the `fromString` method of `df`. At the end `fromString` will use `link` to provide the `home` to the created `WBBody` instance.

Class `MyCss` provides a method `style` producing a `Body` factory initialised to produce pages with white backgrounds `Body`s and bold bordered `Div`s.

Note that `style` produces a fully initialised `BodyF` even if the `home` is still unknown (i.e. is still a placeholder). This is one example of a non trivial data structure that manages placeholders. Note also the use of the auxiliary class `Link`: `l` denotes a fully initialised object, coming from an outer scope with respect to the multiple initialisation expression defining `bf` and `df`. This is well typed code since `l` is a fully initialised value coming from outer scope that can be seen as a partially initialised one. Using the simple class `Link` in this way we tame an expressive power that Fahndrich [6] obtains using the nontrivial concept of existentially quantified temporal variables.

Circular initialisation and dependency injection

Dependency injection (DI) is a design pattern aiming to reduce coupling between components. Traditionally when a component needs to use another one the dependency is hard-coded, for example, explicitly calling the constructor of a class. Alternatively a component can just list the necessary components and a DI framework supplies these. At runtime, an independent component will instantiate and configure (in our context) factory classes for all the required components. Normally such DI framework makes use of reflection and XML configuration files, requiring an ad-hoc IDE support for, as an example, refactoring.

In FJ′, for example, `MyPages.myPages` is a method using `MyCss.style` to produce fully initialised `Body` instances. A local variable `bf` is initialised, and used for the initialisation of `p1`, `p2` and `home`, which is returned as a result.

```
  //initialise DI framework and create objects
class MyPages{
  Body myPages(){
    return {
      Body home=
        {BodyF bf=new myCss().style(home);}in
        {Body^ p1=bf.fromString("....");
         Body^ p2=bf.fromString("....");
        }in bf.fromlist(p1,p2);
      }in home;} }
```

Our method `style` does provide the same functionality as an XML configuration, but since it is not an extra linguistic mechanism offers several advantages: it is statically typed, thanks to method parameters it allows full parametrisation, and is transparently supported by any IDE supporting the language itself.

We believe that one of the reasons for the success of extralinguistic configuration files in the context of DI is the difficulty of writing correct circular initialisations in the conventional languages.

D. Proofs

Proof of Theorem 2.
By induction on the typing rules.

Case (VAR-T)
This case is empty.

Case (METH-INVK-T)
From the premise by the inductive hypothesis we have two cases:

- $\mu \mid e_i \rightarrow \_ \mid \_$ for some $i \in 0..n$. We can apply (CTX) to show that the whole term reduces.
- $e_0 \ldots e_n$ are values. In this case e_0 is a well-typed value of form ι. This hold because second side condition of (METH-INVK-T) requires e_0 to not be a of a placeholder type.
  Hence, we have typed ι by one of the rules (ID-T1) . . . (ID-T4)
  Indeed by rule (HEAP-ID), first side condition, we know that $\mu(\iota) = C(\_)$, with $C \leq C_0$. By (METH-INVK-T) first side condition, (CLASS) and (INTERFACE), we know that $p(C).m$ is defined, and is a method requiring n arguments. We can apply rule(METH-INVK), since alpha renaming can

always be applied on e to ensure that last (METH-INVK) side condition holds.

Case (NEW-T1) or (NEW-T2)

From the premise by the inductive hypothesis we have two cases:

- $\mu \mid e_i \rightarrow \_ \mid \_$ for some $i \in 1..n$. We can apply (CTX) to show that the whole term reduces.
- $e_1 \dots e_n$ are values. From (NEW-T1) or (NEW-T2) we know the class C have n fields. We can apply rule (CONSTRUCTOR).

Case (FACCESS-T1) or (FACCESS-T2)

From the premise by the inductive hypothesis we have two cases:

- $\mu \mid e \rightarrow \_ \mid \_$. We can apply (CTX) to show that the whole term reduces.
- e is a value. In this case e is a well-typed value of form ι. This hold because premise of (FACCESS-T1) or (FACCESS-T2) requires e to not be a of a placeholder type.
  Hence, we have typed ι by one of the rules (ID-T1) ... (ID-T4)
  Indeed by rule (HEAP-ID), first side condition, we know that $\mu(\iota) = C(\_)$. from (FACCESS-T1) or (FACCESS-T2) side condition we know that class C have a field f. From (HEAP-ID) we know that $\mu(\iota).f$ is defined. We can apply (FACCESS).

Case (INIT1) or (INIT2)

From the first premise, by the inductive hypothesis we have two cases:

- $\mu \mid e_i \rightarrow \_ \mid \_$ for some $i \in 1..n$. We can apply (CTX) to show that the whole term reduces.
- $e_1 \dots e_n$ are values. We can apply rule (INIT).

Case (SUB-EXP-T)

Assume $xe_1 = T_1\, x_1 = e_1 \dots T_{j-1}\, x_{j-1} = e_{j-1}$ and $xe_2 = T_j\, x_j = e_j \dots T_n\, x_n = e_n$. From the premise by the inductive hypothesis we have three cases:

- $\mu \mid e_i \rightarrow \_ \mid \_$ for some $i \in 1..n$. We can apply (CTX) to show that the whole term reduces.
- $\mu \mid e_0 \rightarrow \_ \mid \_$ We can apply (CTX) to show that the whole term reduces.
- e_0 is a value. In this case the whole term can be typed also with either (INTI1) or (INIT2).

Case (PH-T)

By rule side condition this case is empty.□

The following lemmas are needed to prove subject reduction.

Lemma 5. *If* $\mathcal{M}_1\ C_1^{\mathcal{S}_1} \leq \mathcal{M}_2\ C_2^{\mathcal{S}_2}$ *and* $p(C_2).m = mh\_$ *then* $p(C_1).m = mh\_$.

Proof. By straightforward induction over the definitions of $C_1 \leq C_2$.

Case $Base$

For the three base cases $\mathcal{M}\ C \leq \mathcal{M}\ C^{\wedge} \leq \mathcal{M}\ C'$, $T \leq T$ and $C^{\mathcal{S}} \leq$ **ns** $C^{\mathcal{S}}$ the thesis trivially holds since $p(C) = p(C)$.

Case $Inductive$

For the two base cases

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

and

$\mathcal{M}\ C_1^{\mathcal{S}} \leq \mathcal{M}\ C_2^{\mathcal{S}}$ if
$p(C_1) =$ **interface** C_1 **extends** $C, \_\ \{\ \_\ \}$ and $\mathcal{M}\ C^{\mathcal{S}} \leq \mathcal{M}\ C_2^{\mathcal{S}}$

We show the thesis hold using inductive hypothesis and (CLASS) or (INTERFACE), respectivelly. □

Lemma 6 (substitution lemma). *For the following, conventional, definition of context:*

$$\begin{aligned}\mathcal{E} \quad ::= \quad & \square \mid \mathcal{E}.m\ (\overline{e}) \mid \iota.m\ (\overline{\iota}, \mathcal{E}, \overline{e}) \mid \mathcal{E}.f \\ & \mid \textbf{new}\ C\ (\overline{\iota}, \mathcal{E}, \overline{e}) \mid \{\overline{xv} \dots T\ x = \mathcal{E}; \overline{xe}\}\ \textbf{in}\ e\end{aligned}$$

1. If
(G1) $\mu_1 \mid e_1 \rightarrow \mu_2 \mid e_2$,
(G2) $\Gamma; \Gamma'; \Sigma \vdash e_1 : T_1$,
(G3) $\Gamma; \Gamma'; \Sigma \vdash e_2 : T_2 \leq T_1$,
(G4) $\Gamma_0; \Gamma'_0; \Sigma \vdash \mathcal{E}[e_1] : T_1'$,
(G5) $\Gamma'_0 \vdash \mu_0 : \Sigma$, *and*
(G6) $\Gamma'_0 \vdash \mu_1 : \Sigma$,
then, for some T_2',
$\Gamma_0; \Gamma'_0; \Sigma \vdash \mathcal{E}[e_2] : T_2' \leq; T_1'$.

2. If $\Gamma_0, x_1{:}T_1, \dots x_n{:}T_n; \Gamma'_0; \Sigma \vdash e : T$,
$T_1' \dots T_n' \leq T_1 \dots T_n$ *and*
for all $i \in 1..n\ \Gamma_i; \Gamma'_i; \Sigma \vdash e_i : T_i'$
then, for some T',
$\Gamma_0; \Gamma'_0; \Sigma \vdash e[x_1{=}e_1 \dots x_n{=}e_n] : T'$ *and*
$T' \leq T$.

Proof.

1. By induction over the reduction arrow (G1); all cases but $\mathcal{E}$ are trivial since (G2)=(G4) and (G3) is the thesis.

   Case $\mathcal{E}$ is more complex, and is done by cases; Note that in all cases but multiple initialization, $\Gamma = \Gamma_0$ and $\Gamma' = \Gamma'_0$.

   Case □
   Also in this case, (G3) is the thesis.

   Case $\mathcal{E}.m\ (\overline{e})$
   In this case $\Gamma; \Gamma'; \Sigma \vdash \mathcal{E}[e_1] : T_1'$ is typed with rule (METH-INVK-T). Thanks to Lemma 5 and (METH-INVK-T) first premise, $\Gamma; \Gamma'; \Sigma \vdash \mathcal{E}[e_2] : T_1'$

   Case $\iota.m\ (\overline{\iota}, \mathcal{E}, \overline{e})$
   As in the precedent case, but thanks to (METH-INVK-T) second premise.

   Case $\mathcal{E}.f$
   In this case $\Gamma; \Gamma'; \Sigma \vdash \mathcal{E}[e_1] : T_1'$ is typed with rule (FACCESS-T1) or (FACCESS-T2). In both cases we know that the receiver type $\mathcal{M}\ C$ denotes a class (not an interface). Since in our system only interface and type modifiers induce subtyping, we know that either

   - $T_1 = T_2 = C$, using (FACCESS-T1) so the thesis trivially holds.
   - $T_1 = T_2 =$ **ns** C using (FACCESS-T2) and, again, the thesis trivially holds.

- Finally $T_1 = \mathbf{ns}\ C$ and $T_2 = C$ using (FACCESS-T2).

  In this case, $\Gamma; \Gamma' ; \Sigma \vdash \mathcal{E}[e_2] : T_2'$ will be typed using (FACCESS-T1). At this point it can either happen that $T_2' = T_1' = \mathbf{ns}\ C'^{\mathcal{S}'}$ or that $T_2' = C'^{\mathcal{S}'}$ and $T_1' = \mathbf{ns}\ C'^{\mathcal{S}'}$. In both cases $T_2' \leq T_1'$.

Case new $C\ (\overline{\iota}, \mathcal{E}, \overline{e})$

In this case $\Gamma; \Gamma' ; \Sigma \vdash \mathcal{E}[e_1] : T_1'$ is typed with rule (NEW-T1) or (NEW-T2). In both cases we know that C denotes a class (not an interface). We know that e_1 lies as one of the constructor parameters. We have two possiblities

- (NEW-T1) is applied to type $\Gamma; \Gamma' ; \Sigma \vdash \mathcal{E}[e_1] : T_1'$ by definition of subtyping we know that (NEW-T1) will be applied also to type $\Gamma; \Gamma' ; \Sigma \vdash \mathcal{E}[e_2] : T_2'$
- (NEW-T2) is applied to type $\Gamma; \Gamma' ; \Sigma \vdash \mathcal{E}[e_1] : T_2'$

  At this point, it can be shown that (NEW-T2) is also applicable to type $\Gamma; \Gamma' ; \Sigma \vdash \mathcal{E}[e_2] : T_2'$

  Moreover, either $T_2' = T_1'$ or $T_2' = C^{\mathcal{S}}$ and $T_1' = \mathbf{ns}\ C^{\mathcal{S}}$. In both cases $T_2' \leq T_1'$.

Case $\{\overline{xv} \ldots T\ x = \mathcal{E}\,;\, \overline{xe}\}$ **in** e

This is typed with a sequence of application of (SUB-EXP-T) followed by an application of either (INIT1) or (INIT2).

- Suppose we apply zero steps of (SUB-EXP-T) and an (INIT1). In this case we have that $\Gamma; \Gamma' ; \Sigma \vdash e_1 : T_1$ and $\mathsf{unlock}(T_1) \leq T_i$. By (G3), subtype transitivity and defintion of $\mathsf{unlock}(\_)$ we conclude $\mathsf{unlock}(T_2) \leq T_i$, and we can apply rule (INIT1) in the exact same way to type $\Gamma_0; \Gamma'_0; \Sigma \vdash \mathcal{E}[e_1] : T_1'$ or $\Gamma_0; \Gamma'_0; \Sigma \vdash \mathcal{E}[e_2] : T_1'$
- Suppose we apply zero steps of (SUB-EXP-T) and an (INIT2). In this case we have that $\Gamma; \Gamma' ; \Sigma \vdash e_1 : T_1 \leq T_i$ By (G3) and subtype transitivity we conclude $T_2 \leq T_i$, and we can apply rule (INIT2) in the exact same way to type $\Gamma_0; \Gamma'_0; \Sigma \vdash \mathcal{E}[e_1] : T_1'$ or $\Gamma_0; \Gamma'_0; \Sigma \vdash \mathcal{E}[e_2] : T_1'$
- We type the term using rule (SUB-EXP-T), and the first premise is of the form $\Gamma; \Gamma' ; \Sigma \vdash e_0 : T_0$ We have two cases:
  - $e_1 = \mathcal{E}'[e_0]$ and $e_2 = \mathcal{E}'[e_0']$. In this case, by inductive hypotesis we know that the first premise of (SUB-EXP-T) is still applicable, thus the whole term is typed in the same way.
  - Otherwise, we have applied (SUB-EXP-T) on a subterm that is not involved in the reduction. Thus (SUB-EXP-T) is still appliable in the same way.

  In both cases other application of (SUB-EXP-T) continue to holds inductivelly.□

2. Using alpha substitution we can ensure that $e_1 \ldots e_n$ do not use variables $x_1 \ldots x_n$. By induction over the number of variable and the number of occurency of the single variable itself.

   Case single variable; single occurency
   If there is only one occurency of a variable is possible to rewrite e as $\mathcal{E}[x_1]$, and thanks to (1) the case close.
   Case single variable; n occurencies
   Inductivelly, is possible to remove $n-1$ occurencies, and then the last one using (1).
   Case n variables
   Inductivelly, is possible to remove $n-1$ variables, and then the last one using the former case. □

Lemma 7. *If* $\Gamma' \vdash \mu : \Sigma$ *and* $\Gamma; \Gamma' ; \Sigma \vdash \iota : C\,\hat{}$ *then* $\mathsf{openPh}(\iota, \mu) \subseteq \mathsf{dom}(\Gamma')$

Proof. $\Gamma; \Gamma' ; \Sigma \vdash \iota : C\,\hat{}$ can be obtained only by applying rule (ID-T1) or (ID-T2).

Case (ID-T1)
In this case, by (HEAP-ID) we know that $\overline{x} = \mathsf{reachPh}(\iota, \mu)$, by and (ID-T1), second side condition,
$\mathsf{reachPh}(\iota, \mu) \subseteq \mathsf{dom}(\Gamma')$. We get the thesis since
$\mathsf{openPh}(\iota, \mu) \subseteq \mathsf{reachPh}(\iota, \mu)$ by definition of $\mathsf{openPh}(\_, \_)$ and $\mathsf{reachPh}(\_, \_)$.

Case (ID-T2)
Since $\Sigma(\iota) = C\,\overline{x}$ by (HEAP-ID) we know that
$\mathsf{openPh}(\iota, \mu) = \emptyset$.□

Proof of Theorem 4. A multiple initialisation expression can be typed by three rules.

Case (INIT1)
Where, of course, $e_1 \ldots e_n$ are of form $v_1 \ldots v_n$. The key point is to prove that function $\mathsf{unlock}(\_)$ behave correctly. Function $\mathsf{unlock}(\_)$ works only on shareable partially initialised types. Note that when T_i is a sharable partially initialised type, $\Gamma; x_1{:}\_ \ldots x_n{:}\_; \Sigma \vdash v_i : T_i'$ and T_i' is (a sub-type of) a sharable partially initialised type. Moreover v_i is of form ι_i. By Lemma 7 we know that only placeholders $x_1 \ldots x_n$ are open in ι_i. Since operation $\mu[xv]$ removes all the occurrences of placeholders $x_1 \ldots x_n$ form the heap, it is correct to consider reference ι_i a fully initialised object in e. In the detail, it is typed by (ID-T1) and (MEM-LOC) where $\mathsf{openPh}(\iota_i, \mu') = \emptyset$.

Case (INIT2)
We get the thesis by Lemma 6-(2) and subtyping transitivity.

Case (SUB-EXP-T)
If this rule is applied, since all the initialisation expressions are already reduced to value, one of the other two is also applicable.□

Proof of Theorem 3.
By induction on the reduction rules.

Case (CTX)
We get the thesis by Lemma 6-(1).

Case (CONSTRUCTOR)

typed by (NEW-T1) or (NEW-T2)
By (HEAP-ID) ι' is typed by one of the rules (ID-T1)...(ID-T4).
We get the thesis by subtyping transitivity.

Case (FACCESS)

typed by (FACCESS-T1) or (FACCESS-T2)

In this case, ι is typed by one of the rules (ID-T1)...(ID-T4).
By rule (HEAP-ID) second and third side condition, we know that all values inside $\mu(\iota)$ are typed with a subtype of the corresponding field type declared in class C. By (FACCESS-T1) or (FACCESS-T2) we have that $\Gamma; \Gamma' ; \Sigma \vdash v : \_ \leq T$ or $\Gamma; \Gamma' ; \Sigma \vdash v : \_ \leq \mathcal{M}\ C_1^{\mathcal{S}}$.

Case (METH-INVK)

typed by (METH-INVK-T)

In this case, ι is typed by one of the rules (ID-T1)...(ID-T4). By (METH-T) premise we obtain $\ldots; \emptyset; \emptyset \vdash e : \_ \leq T$ Finally, we get the thesis by Lemma 6-(2) and subtyping transitivity.

Case (INIT)

We can apply Theorem 4 to show that the variables $x_1 \ldots x_n$ are replaced by $e[\overline{xv}]$ with values $v_1 \ldots v_n$ of the correct type. Finally, we get the thesis by Lemma 6-(2) and subtyping transitivity.