The Billion-Dollar Fix

Safe Modular Circular Initialisation with Placeholders and Placeholder Types

Marco Servetto, Julian Mackay, Alex Potanin, and James Noble

Victoria University of Wellington
School of Engineering and Computer Science
{servetto,mackayjuli,alex,kjx}@ecs.vuw.ac.nz

Abstract. Programmers often need to initialise circular structures of objects. Initialisation should be safe (so that programs can never suffer null pointer exceptions or otherwise observe uninitialised values) and modular (so that each part of the circular structure can be written and compiled separately). Unfortunately, existing languages do not support modular circular initialisation: programmers in practical languages resort to Tony Hoare's "Billion Dollar Mistake": initialising variables with nulls, and then hoping to fix them up afterward. While recent research languages have offered some solutions, none fully support safe modular circular initialisation.

We present *placeholders*, a straightforward extension to object-oriented languages that describes circular structures simply, directly, and modularly. In typed languages, placeholders can be described by *placeholder types* that ensure placeholders are used safely. We define an operational semantics for placeholders, a type system for placeholder types, and prove soundness. Incorporating placeholders into object-oriented languages should make programs simultaneously simpler to write, and easier to write correctly.

1 Introduction

Imagine writing the top level of a simple web application, with a database access `DBA` component, a `Security` component, an `SMS` component able to send SMS text messages and finally a `GUI` component for user interaction. The security component needs to access the database (to retrieve user authorisation records), while the database needs to access the security component (to ensure users only access data they are authorised to view); the `GUI` component needs access to the `SMS` component (to send messages) and the `SMS` component needs access to the database to log received/sent messages; finally the database needs to update the `GUI` component whenever a change happens (for example to display received SMS messages). This system has a number of circular dependencies: the database needs the security system, which needs the database, which needs the security system; the `GUI` component needs the `SMS` component, which needs the database access, which needs (to update) the `GUI` component; and on *ad infinitum*.

How can programs initialise such a structure? The obvious code is obviously wrong:

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 205–229, 2013.

first attempt
```
Security s=new Security(dba);
DBA dba=new DBA(s,gui);
SMS sms=new SMS(s,dba);
GUI gui=new GUI(sms,dba,s);
```

attempting to initialise the security system with the database before the database itself is constructed. In languages designed in the last twenty years or so, this "uninitialised variable" error should be caught statically or dynamically; in older languages, this will be caught as a null pointer exception (if we are lucky) or by initialising the security system with the contents of the uninitialised `dba` variable (if we are not).

The traditional solution relies critically on Tony Hoare's "Billion Dollar Mistake": null pointers [12]. We initialise the security system with a null pointer, instead of a database, then initialise the database with the now-extant security subsystem, and then use a setter method to link the database back to the security system:

nulls and setters
```
Security s=new Security(null);
DBA dba=new DBA(s,null);
s.setDatabase(dba);
SMS sms=new SMS(s,dba);
GUI gui=new GUI(sms,dba,s);
sms.setGUI(gui);
dba.setGUI(gui);
```

More sophisticated versions of this approach may use dependency injection frameworks, writing XML configuration files rather than code, decoupling configuration from the code itself; or (monadic) option types rather than raw nulls, avoiding null pointer errors at the cost of enforcing tests of every access to every potentially uninitialised variable throughout the program.

1.1 Placeholders

We address this problem by introducing *placeholders*. A placeholder, as the name suggests, is a proxy or a stand-in for an uninitialised, as yet nonexistent object. Placeholders are created in *placeholder declarations*. A single placeholder declaration can declare and initialise the entire GUI/Database/SMS system:

placeholders
```
Security s=new Security(dba),
DBA dba=new DBA(s,gui),
SMS sms=new SMS(s,dba),
GUI gui=new GUI(sms,dba,s);
```

Placeholder declarations differ from the Java-style declarations shown earlier in two ways. Syntactically, a series of initialisation clauses are separated with commas (`,`). Programs can pass variables — placeholders, in fact — declared anywhere in the same placeholder assignment statement as arguments to any constructors within the same statement. In the code above, these placeholders are shown with a red background.

Semantically, placeholder declarations are evaluated in three phases (see Fig.1). First, a placeholder is created and bound to each declared placeholder variable. Second, the right-hand sides of each declaration (the initialisers) are run in turn, top to bottom, creating the actual objects that will be used in the rest of the program. Third, all pointers

to placeholders are redirected to point to the objects whose places they are holding. This redirection includes all variable bindings, so at the end of the third phase, all the variables are bound to the actual objects created in the second stage. At this point, execution of the placeholder declaration is complete, and the program can continue.

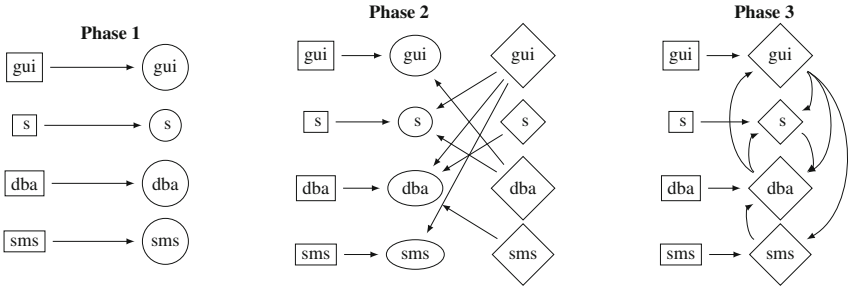

Fig. 1. Initialisation with placeholders

A key feature of placeholders and placeholder declarations is that they have minimal impact on the rest of the program. Placeholders are lexically scoped, limited in extent to the execution of the enclosing placeholder declaration. Placeholder declarations as a whole have well defined imperative semantics — important if any of the initialisers have side effects — as they are simply executed in the order in which they are written.

Placeholders are also flexible. For example, to avoid hard wiring classes into client code, programmers and languages are adopting the object-algebra style, where a *platform* object provides a single point of configuration [15,3]. Client code requests classes from the platform object, and then instantiates those objects indirectly:

object algebra

```
Platform p = ...; // p is a factory aka object algebra

Security s=p.makeSecurity(dba),
DBA dba=p.makeDBA(s,gui),
SMS sms=p.makeSMS(s,dba),
GUI gui=p.makeGUI(sms,dba,s);
```

This more complex creation style also works well with placeholders — the only difference is that placeholders are passed into factory methods [7], rather than directly into constructors. Overall, placeholders support circular initialisation of independent, encapsulated, modules, without any null pointers — the components to be created and initialised need only to present a factory interface where placeholders may be passed into some arguments. More than this: because placeholders can displace nulls for field initialisation, a programming language with placeholders can do without traditional Hoare-style null values entirely.

Unfortunately, placeholders are not by themselves a complete solution to the circular initialisation problem. Placeholders can replace nulls, allowing programs to create circular structures idiomatically. Placeholders are not accessible *outside* the lexical scope of their placeholder declaration, but placeholders have to be accessible *inside* the scope of their declarations, so that they can be used to configure other components. The problem

is that placeholders are placeholders, not actual objects, indeed the object to which they will refer may not have been created when the placeholder is used. Here's a slightly modified example, where the security system is configured last, and where it attempts to retrieve the DBA component via the GUI component, rather than from the dba variable.

placeholders

```
DBA dba=new DBA(s,gui),
SMS sms=new SMS(s,dba),
GUI gui=new GUI(sms,dba,s),
Security s=new Security(gui.getDBA());
```

To a first approximation, this code could reasonably be expected to work: when the GUI component is created, in phase 2 of the execution of the placeholder declaration, it will have received the placeholder for the DBA component, so a getDBA accessor method called on the DBA *object* should just return that placeholder. Indeed, a call on the DBA object would do just that. Unfortunately, at phase 2, all references are to *placeholders*, not to objects — note that the first three lines happily initialise the DBA, SMS, and GUI components with s, which must be a placeholder for the security component as the security component has not yet been created.

So what can we do when a method is requested on a placeholder — effectively a nonexistent object — rather than the object that has yet to come into being? Well, in the absence of time-travel, we treat this as a programmer error, and throw a *PlaceholderException*. Placeholders, after all, are not objects, they are just placeholders for objects. This rationale is also why we do not replace references to placeholders until all the object are created and initialised: we wish to avoid partially initialised objects as much as possible.

This means we have jumped out of the frying pan and into the fire. Rather than having just one distinguished "null" pseudo-object that can be used to manage object initialisation (and for many other purposes) we have created a vast number of placeholder pseudo-objects, access to any one of which terminates program execution with a PlaceholderException just as surely as a null terminates a program with a NullPointerException.

1.2 Placeholder Types

To solve the problem of the PlaceholderExceptions (that replace NullPointerExceptions), we provide a static type system with support for *Placeholder Types*. This type system guarantees that references to (potential) placeholders may only be accessed once they have been replaced with actual objects.

Placeholder types are a small addition to standard Java-like type systems. A reference of placeholder type may refer either to objects or to placeholders of the underlying object type — we will write C′ for the placeholder type corresponding to an object type C. The types in the remainder of the system (we term them "object types" where it is necessary to distinguish) are essentially standard Java types: they accept objects, but do not accept placeholders; nor do they accept **null** pointers — indeed, there are no **null** values at all in our formal system. A reference of object type always refers to an instance of the declared type, never to a placeholder.

Because of the limited scope and lifetimes of the placeholders themselves, placeholder types are required only in a very small part of most programs — the placeholder declara-

tions, constructors, and factory methods used to create and initialise components. Within the actual body of a placeholder declaration, the declared variables are all interpreted as having placeholder types, while after the end of the placeholder declaration — but while those variables are still in scope — the declared variables have object types. This follows directly from the semantics of the placeholder declarations: in phase 1, placeholders are bound to all the declared variables, while at the end of phase 3, all those placeholders have been replaced by actual objects, so that variables that were holding placeholders now refer to those actual objects.

placeholder types

```
DBA dba=new DBA(s,gui),
SMS sms=new SMS(s,dba),
GUI gui=new GUI(sms,dba,s),
  // ''gui'' has placeholder type ''GUI''  within
  // placeholder declaration statement
Security s=new Security(gui.getDBA());
  // gui.getDBA() here is a type error,
  // cannot request methods on placeholders
gui.getDBA();
  // after the end of the placeholder declaration
  // gui has object type ''GUI'' --- it cannot be
  // a placeholder, so method requests are permitted
```

To prevent PlaceholderExceptions from being raised when methods are requested from placeholders (instead of objects) we forbid placeholder types in the receiver position ("before the dot") of any method requests. This rule prevents calls like "`gui.getDBA())`" within the body of the placeholder declaration itself (before the first semicolon "`;`"), when "`gui`" has placeholder type "`GUI'`". After the end of the placeholder declaration — but within the block where the declared names are visible — "`gui`" has object type "`GUI`" and may receive method requests.

To keep the overall system simple, we further restrict placeholder types. Placeholder types may only appear as parameters or return values from methods, or parameters to constructors: they may not appear as types of objects' fields. The aim here is to ensure that placeholders do not escape from the lexical context of their placeholder declarations, so that all the objects created with those placeholders are fully initialised at the end of their placeholder declaration. These restrictions are liberal enough to permit many kinds of factory methods and object algebras. For example, here is code implementing the `makeSecurity` method in the `Platform` object factory (object algebra) discussed earlier:

object algebra

```
class Platform {//...
  Security makeSecurity(DBA' dba) {
    return new Security(dba);
  }
}
```

Here the `makeSecurity` factory method takes a `DBA'` placeholder type and passes it as an argument to the underlying constructor. This is permitted by the rules on the use of placeholder types, while invoking a method on `dba`, or storing it into a field is not permitted, as that could lead to a PlaceholderException, sooner or later.

These rules are necessarily subtle, especially when objects are created that are initialised by placeholders. These partially initialised objects will only become fully initialised when those placeholders themselves become fully initialised. Partially initialised objects do not have placeholder types — in particular, they may be returned as object types — but they are treated as if they were placeholders until they are returned. Consider the following version of the Platform class:

partial initialisation

```
class DifferentPlatform {
  Security makeSecurity(DBA' dba) {
    ExternalSecurity x = new ExternalSecurity(dba);
    x.validate();
    //x.validate here is a type error
    //cannot request methods on partially initialised objects
    if (/*high security needed*/)return highSecurity(x);
    else return x;
    //this is not a type error
    //may return partially initialised objects
  }
}
```

Here we create a new object, initialised with a placeholder, and store that in the local variable `x`. This variable is not a placeholder type — we can return it as an object type — even though it refers to a partially initialised object. We know this object is partially initialised because its value comes from an expression that has a placeholder (`dba`) passed in as an actual argument. Within this method, we treat `x` as a placeholder, so we can pass this partially initialised object as a placeholder typed argument to another method (as `Security highSecurity(Security' s)`), but we cannot call methods upon it.

Partially initialised objects are safe because they are treated as if they were placeholders until they are fully initialised. Partially initialised objects can only be passed as placeholder typed arguments, and so they will be treated as full placeholders by any methods into which they are passed. When partially initialised objects are returned from methods (as object types) if that method was called with placeholder arguments, then that returned value will also be treated as partially initialised.

Finally, in order to be able to actually initialise fields of objects, we have to treat placeholder types in constructors slightly differently from elsewhere (of course, constructors already have special privileges in Java-like languages, notably to be able to initialise final fields. The special case within constructors is that we can initialise a field of object type `T` with an expression of placeholder type `T'` — because of our restrictions on placeholder types, any such placeholder must have been passed in as an argument to that constructor.

class decl

```
class Security {
  DBA dbaComponent;
  Security(DBA' dba) {this.dbaComponent=dba;}
}
```

To maintain soundness, we cannot read values back from fields in constructors, otherwise we could attempt to request a method from a placeholder stored in a field with

p	$::= \overline{cd}\ \overline{id}$	program
cd	$::= \texttt{class}\ C\ \texttt{implements}\ \overline{C}\ \texttt{\{}\ \overline{fd}\ k\ \overline{md}\ \texttt{\}}$	class declaration
id	$::= \texttt{interface}\ C\ \texttt{extends}\ \overline{C}\ \texttt{\{}\ \overline{mh;}\ \texttt{\}}$	interface declaration
fd	$::= C\ f\texttt{;}$	field declaration
k	$::= C\texttt{(}\ C'_1\ f_1\texttt{,}\ \ldots\texttt{,}\ C'_n\ f_n\texttt{)\{ this.}f_1\texttt{=}f_1\texttt{; }\ldots\texttt{; this.}f_n\texttt{=}f_n\texttt{;\}}$	constructor
mh	$::= T\ m\texttt{(}\overline{T\ x}\texttt{)}$	method header
md	$::= mh\ e$	method declaration
e	$::= x \mid e\texttt{.}m\texttt{(}\overline{e}\texttt{)} \mid \texttt{new}\ C\texttt{(}\overline{e}\texttt{)} \mid e\texttt{.}f \mid e_1\texttt{.}f\texttt{=}e_2 \mid \iota$	expressions
	$\mid xe_1\texttt{,}\ \ldots\texttt{,}\ xe_n\texttt{;}\ e$	place holder declaration
xe	$::= T\ x = e$	variable initialisation
v	$::= \iota \mid x$	value
μ	$::= \overline{\iota \mapsto C(f_1 = v_1, \ldots, f_n = v_n)}$	memory
T	$::= C \mid C'$	type

Fig. 2. Syntax

an object type which ostensibly does not permit placeholders. On the other hand, since placeholder types can refer to objects (of the appropriate type) as well as placeholders, this constructor and the above object factory can always be called passing in actual objects, rather than placeholders, if the programmer has them to hand.

1.3 Contributions

This paper makes two main contributions: first, placeholders and placeholder declarations, and second, placeholder types. Placeholders support idiomatic, modular, circular initialisations of complex object structures, while placeholder types ensure placeholders are only used safely, and so will never cause a PlaceholderException.

The rest of this paper is structured as follows. Section 2 presents the FJ′ language and its formalisation and Section 3 gives the details of the type rules and states the main soundness theorems. Section 4 demonstrates the expressiveness of FJ′ by presenting a selection of more complex examples, and then Section 5 discusses implementation considerations for supporting placeholders in a Java-like setting. Section 6 overviews related work and Section 7 concludes. The accompanying technical report presents the proofs for our formal system and a more extensive discussion of placeholders [17].

2 Syntax and Semantics of FJ′

Syntax

The syntax of FJ′ is shown in Figure 2. 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 [13] variables include the special variable **this**.

Classes and interfaces. 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, a conventional constructor 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.

An interface declaration consists of an interface name followed by the sets of extended interfaces and method headers. As one can see, for simplicity, we use interfaces to provide subtyping, i.e. there is no subtype relation between classes. Field declarations are as in FJ. To simplify the formalisation, constructors (k) are standard conventional FJ constructors, taking exactly one parameter for each field and only initialising fields. Method declarations are composed of a method header and a body. The method header is as in FJ. Since only fully initialised objects can be receivers, the implicit parameter **`this`** is always of a fully initialised object type. As in FJ, method bodies are simply expressions. We omit the return keyword in the formal definition but we insert it in the appropriate places in the examples to aid readability.

Expressions. Expressions are variables, method or constructor calls, field access, field update, object identifiers (within run time expressions), and finally, placeholder declarations. Variables can be declared in method headers or inside expressions. An expression is well-formed only if the same variable is not declared twice. Hence, we have no concept of variable hiding. Placeholder declarations are a sequence of variable initialisation clauses separated by comma (`,`) and an expression terminated by semicolon (`;`). Note that the order of variable initialisations is relevant since it induces the order of execution for the sub-expressions. The order is also relevant in sequences of field declarations and parameter declarations.

Syntax for placeholders. In a closed expression an occurrence of x is a placeholder only if it is inside a placeholder declaration declaring local variable x, and before the semicolon. That is, any placeholder declaration with a variable initialisation clause $T\ x$=... can contain the placeholder x inside its initialisation expressions, and (as usual) the variable x inside its terminating expression.

Values and memory. Values are object identifiers ι or placeholders x. Note that values do not include **`null`** and we have no default initialisation. Placeholders are replaced by object identifiers when placeholder declarations are reduced. Thus final values are only object identifiers.
A memory is a finite map from object identifiers to records annotated with a class name. For example, a computation over the program "**`class`** `C`**`{`**`C f;C(C' f)`**`{this.`**`f=f;}}`", can produce the memory "$\iota_1 \mapsto$ `C(f=y)`, $\iota_2 \mapsto$ `C(f=`ι_1`)`, $\iota_3 \mapsto$ `C(f=`ι_3`)`", containing 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.

A memory is well-formed with respect to a program if the fields of the records are exactly the fields declared inside the corresponding classes. (Note that a well formed memory may not be well typed, see rule (MEM-OK)).

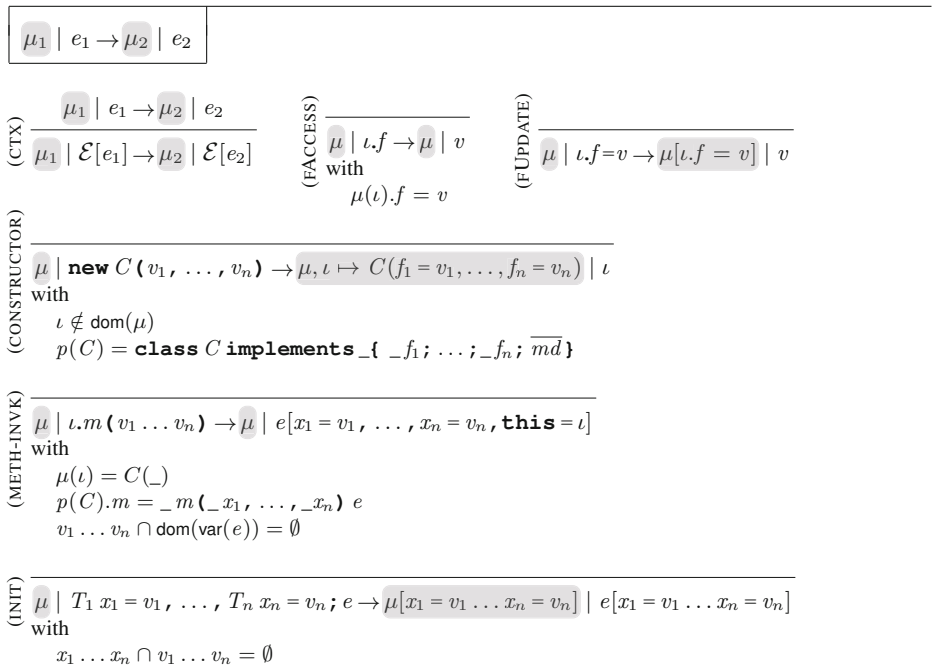

Fig. 3. Reduction rules using memory

Types. are composed of a class name C and an optional quote ($'$) sign. We assign type C' (*placeholder type*) to placeholders and type C (*object type*) to objects of class C. C is a subtype of C'.

Expressions of type C reduce to objects (either partially or fully initialised), while expressions of type C' can also reduce to placeholders. Fields can only be of object type C, and field access is always guaranteed to return a fully initialised object.

Subtyping

Our subtyping is the normal Java subtyping, with the addition that a placeholder type is a subtype of the corresponding object type. Formally:

- $C_1 \leq C_2$ if $p(C_1) = \mathtt{class}\ C_1\ \mathtt{implements}\ C, \mathtt{\_\{\ \_\}}$ and $C \leq C_2$
- $C_1 \leq C_2$ if $p(C_1) = \mathtt{interface}\ C_1\ \mathtt{extends}\ C, \mathtt{\_\{\ \_\}}$ and $C \leq C_2$
- $T \leq T$
- $C_1' \leq C_2'$ and $C_1 \leq C_2'$ if $C_1 \leq C_2$

Reduction

Reduction is defined in the conventional way as an arrow over pairs consisting of a memory and an expression. The only novelty is rule (INIT). To improve readability we will mark memory in grey. A pair $\mu \mid e$ is well-formed only if all the placeholders contained in the memory μ are bound by the local variables declared inside the expression e; that is, a memory with placeholders without an associated expression is meaningless.

We omit the formal definition of the evaluation context, assuming a standard deterministic left-to-right call-by-value reduction strategy.

Figure 3 defines the reduction arrow. Rule (CTX) is standard, however note that alpha-conversion can be needed to ensure the well-formedness of the resulting expression. Rule (FACCESS) models conventional field access. It extracts the value of field f from object ι using the notation $\mu(\iota).f$. Rule (CONSTRUCTOR) is the standard reduction for constructor invocations, and rule (METH-INVK) models a conventional method call. We assume a fixed program p and we use notation $p(C).m$ to extract the method declaration.

We use the notation $e[x_1 = v_1, \ldots, x_n = v_n]$ 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 $v_1 \ldots v_n$ is accidentally captured when injected inside the expression e. Alpha-conversion 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 $\mathsf{var}(e)$. Formally:

$\mathsf{var}(\iota) = \emptyset, \quad \mathsf{var}(x) = \emptyset, \quad \mathsf{var}(e.m(\overline{e})) = \mathsf{var}(e), \mathsf{var}(\overline{e}),$ and

$\mathsf{var}(T_1\ x_1 = e_1, \ldots, T_n\ x_n = e_n; e_0) = x_1 : T_1, \ldots, x_n : T_n, \mathsf{var}(e_0), \ldots, \mathsf{var}(e_n).$

Rule (INIT) reduces placeholder declarations. Just as $e[\overline{x = v}]$ denotes simultaneous replacement of variables with values, we use the analogous notation $\mu[\overline{x = v}]$ to denote simultaneous replacement of placeholders with values. Formally: $[\overline{x = v}] = \emptyset$ and $(\mu, \iota \mapsto C(f_1 = v_1, \ldots, f_n = v_n))[\overline{x = v}] = \mu[\overline{x = v}], \iota \mapsto C(f_1 = v_1[\overline{x = v}], \ldots, f_n = v_n[\overline{x = v}])$.

Note how a normal variable declaration is just a special case of our placeholder declaration, where only one local variable is declared, the placeholder is not used and, thus, the placeholder replacement is an empty operation. Rule (INIT)'s side condition verifies that meaningless terms like `T x=x;x` are stuck.

To understand the details of the semantics of placeholders, consider the following example:

FJ−
```
class A{B myB;A(B' myB){this.myB=myB;}}
class B{A myA;B(A' myA){this.myA=myA;}}
...
  A a=new A(b), B b=new B(a); a
```

We show how to evaluate that expression in the empty memory.

[0]	$\emptyset$	`A a=new A(b), B b=new B(a); a`	Starting point
[1]	$\iota_1 \mapsto$ `A(myB=b)`	`A a=`ι_1`, B b=new B(a); a`	(CTX) + (CONSTRUCTOR)
[2]	$\iota_1 \mapsto$ `A(myB=b)` $\iota_2 \mapsto$ `B(myA=a)`	`A a=`ι_1`,B b=`ι_2`; a`	(CTX) + (CONSTRUCTOR)
[3]	$\iota_1 \mapsto$ `A(myB=`ι_2`)` $\iota_2 \mapsto$ `B(myA=`ι_1`)`	ι_1	(INIT)

- We start in state [0], and in the first step an instance of class `A` is created.
- In state [1], the memory 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. Placeholders are not objects, and thus there is no reference in the memory pointing directly to a placeholder. You can now see that values are object identifiers ι or placeholders x.
- 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 memory are bound by the local variables declared inside the placeholder declaration.

– The third step concludes the initialisation, and in state [3] ι_1 and ι_2 are fully initialised objects. The placeholders are replaced by objects when placeholder declarations are reduced. That is, when the control reaches the semicolon, all the placeholders declared in that placeholder declaration are *consumed*; every occurrence of such a placeholder in the memory is replaced with the corresponding value.

3 Type System of FJ′

Our definition of reduction introduces two stuck situations that are novel in FJ′: (1) method call (or field access) over a placeholder receiver, and (2) declarations of the form `T x = x`.

Our placeholder type system must prevent both these situations. For the first case, while it is clear that we need to forbid method invocation or field access on placeholders, there could be different strategies to ensure safety while manipulating partially initialised instances (objects containing placeholders in their reachable object graph). We believe that in this case the simplest solution is also the right one: **to forbid any method invocation or field access over partially initialised objects**, as we do for placeholders.

The solution for the second case relies on the distinction of placeholder types and object types: it is not possible to initialise a local variable or method parameter of object type with an expression of placeholder type, while a constructor **can** initialise a field with a placeholder.

Limitations over Parameters and Conventional Expression Typing Rules
We permit method parameters to have placeholder type — any factory method allowing circular initialisation needs to have at least one parameter with placeholder type. To ensure that a placeholder is never dereferenced, a method invocation must provide a fully initialised object for any parameter of object type (including the receiver), while either fully initialised objects or placeholders may be provided to parameters of placeholder type.

In FJ′, any closed expression of type C is guaranteed to reduce to a fully initialised object, while any expression using a placeholder could reduce to a partially initialised object. We say that an expression using a placeholder *depends* on that placeholder. Local variable declarations makes the reasoning a little bit more involved: any expression using a variable that depends on a placeholder, also depends any other placeholders that their placeholder declaration depends on. Consider the following code:

```
class C{ C x; C(C' x){this.x=x;} void m(){...} }
void example(C' x){
  //new C(x).m();//wrong: receiver depends from placeholder x

  C y=new C(x);
  //y.m();//wrong: receiver depends from placeholder x through variable y

  C z=new C(z);
  z.m();//correct

  C z1=new C(z2), C z2=new C(x);
  //z1.m();//wrong: z1 is declared together with z2, both depend on x
  }
```

- FJ′ -

$\Gamma; \Sigma \vdash e : T$

(VAR-T) $\dfrac{}{\Gamma; \Sigma \vdash x : \Gamma(x)}$

(ADDR-T) $\dfrac{}{\Gamma; \Sigma \vdash \iota : \Sigma(\iota)}$

(METH-INVK-T) $\dfrac{\begin{array}{c}\mathsf{objectTypes}(\Gamma); \Sigma \vdash e_0 : C_0 \\ \forall i \in 1..n \text{ such that } T_i = C_i : \ \mathsf{objectTypes}(\Gamma); \Sigma \vdash e_i : \_ \leq C_i \\ \forall i \in 1..n \text{ such that } T_i = C_i' \ : \ \Gamma; \Sigma \vdash e_i : \_ \leq C_i'\end{array}}{\Gamma; \Sigma \vdash e_0.m(e_1, \ldots, e_n) : T}$
with
$p(C_0).m = T\ m(T_1\ x_1, \ldots, T_n\ x_n)\ \_$

(NEW-T) $\dfrac{\forall i = 1..n : \quad \Gamma; \Sigma \vdash e_i : \_ \leq C_i'}{\Gamma; \Sigma \vdash \mathbf{new}\ C(e_1 \ldots e_n) : C}$
with
$p(C) = \mathbf{class}\ C\ \mathbf{implements}\ \_\{\ C_1\ \_; \ldots; C_n\ \_; \overline{md}\}$

(FACCESS-T) $\dfrac{\mathsf{objectTypes}(\Gamma); \Sigma \vdash e_0 : C_0}{\Gamma; \Sigma \vdash e_0.f : C_1}$
with
$p(C_0) = \mathbf{class}\ C_0\ \mathbf{implements}\ \_\{\ \_\ C_1\ f; \_\}$

(FUPDATE-T) $\dfrac{\begin{array}{c}\Gamma; \Sigma \vdash e_0 : C_0 \\ \mathsf{objectTypes}(\Gamma); \Sigma \vdash e_1 : C_1\end{array}}{\Gamma; \Sigma \vdash e_0.f = e_1 : C_1}$
with
$p(C_0) = \mathbf{class}\ C_0\ \mathbf{implements}\ \_\{\ \_\ C_1\ f; \_\}$

Fig. 4. Typing rules for expressions

FJ′ constructors can be called using expressions of placeholder type as parameters, and then initialise fields of object type. Consider the following code:

-FJ′-
```
class A { B myB; A(B′ myB){this.myB=myB;} }
class B { A myA; B(A′ myA){this.myA=myA;} }
...
  A a = new A(b), B b = new B(a);
```

Observe that we invoked the constructors of A and B by passing placeholders that then initialise fields of object type.

Typically, 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 a placeholder argument is supplied to a constructor call.

Figure 4 shows the rules for expressions that capture the discussed discipline. The typing judgement for expressions uses the following environments:
$\Gamma ::= \overline{x{:}T}$ variable environment
$\Sigma ::= \overline{\iota{:}C}$ memory environment
Variable environment Γ is a map from variable names to types. Memory environment Σ is a map that for any location stores its class name C. A type judgement is of the form $\Gamma; \Sigma \vdash e : T$, where Γ is needed to type expressions with free variables or placeholders, and Σ is needed to type expressions with locations.

Rules (VAR-T) and (ADDR-T) are standard and straightforward. Rule (METH-INVK-T) uses notation $\mathsf{objectTypes}(\Gamma)$ to restrict the environment to the object types. Formally: $\mathsf{objectTypes}(x{:}C) = x{:}C$ and $\mathsf{objectTypes}(x{:}C') = \emptyset$. Every expression well typed in

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

$$\text{(V-DEC-N)}\ \frac{\begin{array}{c}\forall i = 1..n \text{ with } T_i = C_i : \quad \mathsf{objectTypes}(\Gamma); \Sigma \vdash e_i : \_ \leq T_i \\ \forall i = 1..n \text{ with } T_i = C_i' \ : \quad \Gamma; \Sigma \vdash e_i : \_ \leq T_i \\ \Gamma, x_1{:}T_1, \ldots x_n{:}T_n; \Sigma \vdash e : T\end{array}}{\Gamma; \Sigma \vdash T_1\ x_1 = e_1\texttt{,}\ \ldots\texttt{,}\ T_n\ x_n = e_n\texttt{;}\ e : T}$$

$$\text{(V-DEC-C)}\ \frac{\begin{array}{c}\forall i = 1..n : \quad \mathsf{objectTypes}(\Gamma), x_1{:}C_1' \ , \ldots x_n{:}C_n' \ ; \Sigma \vdash e_i : \_ \leq C_i \\ \Gamma, x_1{:}C_1, \ldots x_n{:}C_n; \Sigma \vdash e : T\end{array}}{\Gamma; \Sigma \vdash C_1\ x_1 = e_1\texttt{,}\ \ldots\texttt{,}\ C_n\ x_n = e_n\texttt{;}\ e : T}$$

$$\text{(V-DEC-O)}\ \frac{\begin{array}{c}\forall i = 1..n : \quad \Gamma, x_1{:}C_1' \ , \ldots x_n{:}C_n' \ ; \Sigma \vdash e_i : \_ \leq C_i \\ \Gamma, x_1{:}C_1, \ldots x_n{:}C_n; \Sigma \vdash e : T \\ \Gamma, x_1{:}C_1' \ , \ldots x_n{:}C_n' \ ; \Sigma \vdash e : \_\end{array}}{\Gamma; \Sigma \vdash C_1\ x_1 = e_1\texttt{,}\ \ldots\texttt{,}\ C_n\ x_n = e_n\texttt{;}\ e : T}$$

Fig. 5. Typing rules for placeholder declarations

$\mathsf{objectTypes}(\Gamma)$ denotes a fully initialised object. Thus, rule (METH-INVK-T) requires (premise one) the receiver and (premise two) all actual arguments to formal parameters of object type to be fully initialised objects, and (premise three) permits actual arguments to formal parameters of placeholder type to be placeholders or partially initialised objects and well as fully initialised objects. Note how we use notation $\Gamma; \Sigma \vdash e : T_1 \leq T_2$ as a shortcut for $\Gamma; \Sigma \vdash e : T_1$ and $T_1 \leq T_2$. 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. Rule (NEW-T) is conventional but accepts arguments of placeholder types. Rules (FACCESS-T) and (FUPDATE-T) ensure that fields can be accessed and updated only using fully initialised objects.

Placeholder Declaration Typing Rules

Figure 5 contains metarules for placeholder declarations. We classify variable declarations depending on what kind of variable is used in the initialisation expression:

- **neutral** (V-DEC-N) initialisation expressions $e_1 \ldots e_n$ do not use the introduced variables $x_1 \ldots x_n$. This type of variable declaration is completely equivalent to conventional Java, but when a local variable with object type is introduced, the corresponding initialisation variable has to denote a fully initialised object. See $\mathsf{objectTypes}(\Gamma)$ in the first premise.
- **closed** (V-DEC-C) initialisation expressions $e_1 \ldots e_n$ use variables $x_1 \ldots x_n$ with placeholder types $C_1' \ \ldots \ C_n'$ and no other placeholder declared outside this specific placeholder declaration. This result is obtained by the notation ($\mathsf{objectTypes}(\Gamma)$).
  At the end of the variable declaration clauses, the introduced variables denote fully initialised objects, that is, the terminating expression can see the declared variables as their object types $C_1 \ldots C_n$.

For example, we can type check the following code:

```
class C{C myC; C(C′ myC){this.myC=myC;}}
class User{
  C makeC(){
    C x= new C(x);
    return x;
  }}
```

-FJ′-

Class `User` has a `C makeC()` method performing circular initialisation. The placeholder `x` has type `C′`. **`new`** `C(x)` is of type `C`, thanks to (NEW-T) **`new`** `C(x)` can be used to initialise variable `C x`.

– **open** (V-DEC-O) initialisation expressions $e_1 \ldots e_n$ can see **both** introduced variables $x_1 \ldots x_n$ with placeholder types $C_1' \ldots C_n'$ **and** other placeholders from enclosing placeholder declarations (thus no usage of $\mathsf{objectTypes}(\Gamma)$). This rule is applied when partially initialised variables are not guaranteed to become fully initialised, that is, the terminating expression typing must take into account the possibility that introduced variables will denote partially initialised objects until any enclosing placeholder declarations complete phase 3 initialisation.
To ensure soundness we type the terminating expression twice: once in a context where the declared variables have their object types $C_1 \ldots C_n$, and again in a context where the declared variables have their placeholder types $C_1' \ldots C_n'$. In this way the result of the expression can be an object type, but we guarantee that the resulting value is never used as a receiver.
Consider the following example:

```
class C{C myC; C(C′ myC){this.myC=myC;}}
class User{
  C makeCPart(C′ y){
    //typed with v-dec-o
    C x= new C(y);
    return x;}
  C makeCAll(){
    //typed with v-dec-c
    C z=new C(this.makeCPart(z));
    return z;}
```

-FJ′-

Method `C makeCPart(C′ y)` takes a placeholder and returns a partially initialised object. Variable `y` inside **`new`** `C(y)` is a placeholder declared outside the initialisation of variable `x`. Rule (V-DEC-C) cannot be applied. Indeed (V-DECL-C) would apply $\mathsf{objectTypes}(\Gamma)$; in this way `y` would not be in scope in expression **`new`** `C(y)`, thus **`new`** `C(y)` would be not well typed. However, (V-DEC-O) can be applied smoothly.

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 [20] would have signaled a (false positive) dynamic error.

Memory Typing

(MEM-OK) in Figure 6 defines a well typed memory. Note how this judgement requires a

(MEM-OK)
$$\frac{}{\Gamma \vdash \mu : \text{ok}}$$
with
$\mu = \iota_1 \mapsto C_1(\overline{v}_1) \ldots \iota_n \mapsto C_n(\overline{v}_n)$
$\forall \iota \mapsto C(v_1 \ldots v_k) \in \mu :$
$p(C) = \texttt{class}\ C\ \texttt{implements\_\{}\ C_1\, f_1\texttt{;}\ \ldots\ C_k\, f_k\texttt{;}\ \overline{md}\ \texttt{\}}$
$\forall j \in 1..k : \Gamma(v_k) \leq C_k$ or $\mu(v_k) = C'_k(\_)$ and $C'_k \leq C_k$

Fig. 6. Typing rule for memory

variable environment for the placeholders. The memory is well typed if for all the objects in the memory, all fields contain either an object of the right type or a placeholder of the right type. From a well typed memory we can extract the memory environment with the following two notations:

- A memory environment typing all the objects in their corresponding object type: $\Sigma^{\mu}(\iota) = C$ iff $\mu(\iota) = C(\_)$
- A memory environment typing all fully initialised objects using their corresponding object type and all the partially initialised objects using the corresponding placeholder type:
  $\Sigma'^{\mu}(\iota) = C$ iff $\mu(\iota) = C(\_)$ and $\mathsf{reachPh}(\iota, \mu) = \emptyset$
  $\Sigma'^{\mu}(\iota) = C'$ iff $\mu(\iota) = C(\_)$ and $\mathsf{reachPh}(\iota, \mu) \neq \emptyset$
  Where $\mathsf{reachPh}(\iota, \mu)$ denotes the set of reachable placeholders.

As for rule (V-DEC-O) a well typed expression has to be typed twice: first considering fully initialised objects with object types, and second considering fully initialised object with placeholder types.

Classes and Interfaces

In Figure 7 we present standard typing rules for classes, interfaces and methods.

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. Finally rule (METH-T) is straightforward.

3.1 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 $\emptyset;\emptyset;\vdash e : T$ 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.

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

$$\text{(CLASS)}\ \frac{\forall i \in 1..n : \quad C_0 \vdash md_i : \text{ok}}{\vdash \texttt{class}\ C_0\ \texttt{implements}\ \overline{C}\texttt{\{}\ \overline{fd}\ md_1 \ldots md_n\texttt{\}} : \text{ok}}$$
with
$\forall C \in \overline{C}, m$ such that $p(C).m = T\ m\texttt{(}\overline{T\ x}\texttt{);}$,
$T\ m\texttt{(}\overline{T\ x}\texttt{)}\ \_ \in md_1 \ldots md_n$

$$\text{(INTERFACE)}\ \frac{}{\vdash \texttt{interface}\ C_0\ \texttt{extends}\ \overline{C}\texttt{\{}\ mh_1\texttt{;} \ldots mh_n\texttt{;\}} : \text{ok}}$$
with
$\forall C \in \overline{C}, m$ such that $p(C).m = T\ m\texttt{(}\overline{T\ x}\texttt{);}$,
$T\ m\texttt{(}\overline{T\ x}\texttt{);} \in mh_1 \ldots mh_n$

$$\boxed{C \vdash mh\ e : \text{ok}}$$

$$\text{(METH-T)}\ \frac{\texttt{this}{:}C, x_1{:}T_1, \ldots, x_n{:}T_n; \emptyset \vdash e : \_ \leq T}{C \vdash T\ m\texttt{(}T_1\ x_1\texttt{,} \ldots\texttt{,}\ T_n\ x_n\texttt{)}\ e : \text{ok}}$$

Fig. 7. Typing rules for classes, interfaces and methods

Theorem 2 (Progress). *For all well typed programs p and for all expressions e and memory μ under p, if $\Gamma \vdash \mu : ok$, $\mathsf{objectTypes}(\Gamma); \Sigma^{\mu} \vdash e : C$, $\Gamma; \Sigma'^{\mu} \vdash e : \_$ and $\mathsf{objectTypes}(\Gamma) = \emptyset$, then either e is of form v, or $\mu \mid e \rightarrow \_ \mid \_$*

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

The proofs can be found in the accompanying technical report [17].

4 Expressive Power

We show now some examples of what can be achieved with FJ′Note that these examples never use the field update operation. This allows all the examples work with immutable data structures, and it would be easy to integrate placeholders with a type system offering immutability [1,22].

Circular linked list
A clearly interesting example of expressive power is an arbitrarily sized, immutable, circular list. Note how `ListProducer` can be a user class, and does not need to know implementation details of the `List`. (Other approaches [19] would require the production process to happen inside `List` class, encoded in the constructor.)

```
class List{
  final int e; final List next;
  List(int e, List' next){this.e=e; this.next=next;}
  }
class ListProducer{
  List' mkAll(int e, int n, List' head){
    if(n==1) return new List(e, head);
    return new List(e, this.mkAll(e+1, n-1, head));
    }
  List make(int e,int n){
    List x = this.mkAll(e, n, x);
    return x;
    }
  }
...
  new ListProducer().make(100,10)
```

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 `head`. Method `mkAll` creates a list of length `n` containing values starting from `e` as list elements, ending with list `head`. 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 `make(100,10)` can be directly used to make a circular list of numbers 100,101,...109.

Doubly linked list

We can convert the singly linked list example to implement an immutable doubly linked list:

```
class List{
  final int e; final List next; final List pred;
  List(int e, List' next,List' pred){
    this.e=e; this.next=next;this.pred=pred;
    }
  }
class ListProducer{
  List' mkAll(int e, int n, List' pred){
    if(n==1){
      List x=new List(e, x, pred);
      return x;
      }
    List x=new List(e, mkAll(e+1,n-1,x), pred);
    return x;
    }
  List make(int e,int n){
    List x = this.mkAll(e, n, x);
    return x;
    }
}
...
  new ListProducer().make(100,10)
```

Here, the produced list is a (non circular) doubly linked list, where termination is represented by having the same value for **this** and **this**.next or **this**.pred. If our List was mutable, we could make it circular after creation. To the best of our knowledge, in the type system presented in this paper, it is impossible to create an immutable doubly-linked list: we would need either multiple return values for a method or support for fields with placeholder types. A more complex version of FJ′, supporting placeholder fields, can be found in an associated technical report [18].

Parser Combinators

As Gilad Bracha suggests [2], it is possible to leverage the recursive nature of parsers in order to define classes that represent different typical operations in a BNF grammar. With overloading support for operators (|) and (,) (as in C++ or Newspeak) it is possible to obtain a syntax very near to conventional BNF. For example we could obtain the following:

parser combinator

```
Production operator|(Production left, Production right){
  return new OrProduction(left,right);}
Production operator,(Production left, Production right){
  return new SeqProduction(left,right);}
...
  Production number= term(1,9)| term(1,9), number0,
  Production number0= term(0,9) | term(0,9), number0,
  Production e= number | e,term("*"),e | e,term("+"),e;
```

where term is a static method. Thanks to placeholders, we can use number0 and e recursively.

The current implementation of parser combinators in Newspeak solves this problem in a much more ad-hoc solution, using reflection [2]. Designed concurrently with our placeholders, Newspeak 0.08 introduced "simultaneous slot definitions" that use futures:

" A simultaneous slot declaration with a right hand side expression e initialises the slot to the value of p computing: e, where p is the class Past'Future. The result is a future that will compute the expression e on demand. All these futures are resolved once the last slot declaration in the simultaneous slot definition clause has been executed. Past'Future implements a pipelined promise so that any well founded mutual recursion between simultaneous slots will resolve properly. "

Futures in Newspeak are objects forwarding all messages to the result of the computation. In this way, Newspeak can provide a similar expressive power to placeholders. Thanks to the dynamic nature of Newspeak, there is no type guarantee of well formedness of a circular initialisation using Newspeak futures. Newspeak requires an extra level of indirection (even if transparent in most of the cases), and the execution order of the different initialisation expression is "on demand" instead of sequential.

5 Implications for Implementation

In this section we discuss some options that could be used to implement placeholders.

Smalltalk offers a method called "become:" that changes object references. After "a become: b" all local variables and all object fields originally referring to the object

denoted by `a`, now refer to the object that was denoted by `b`, and vice versa. An implementation of placeholders over a virtual machine offering a "`become:`" method is very simple; for example, our initial placeholder declaration could be written as follows:

translation with become:

```
//placeholders initialisation
Security sP=new Security(null);
DBA dbaP=new DBA(null,null);
SMS sms=new SMS(null,null);
GUI guiP=new GUI(null,null,null);
//real initialisation
Security s=new Security(dbaP);
DBA dba=new DBA(sP,guiP);
SMS sms=new SMS(sP,dbaP);
GUI gui=new GUI(smsP,dbaP,sP);
//placeholders replacement
sP.become(s);  dbaP.become(dba);
smsP.become(sms);  guiP.become(gui);
```

Is it possible to emulate "`become:`" on a platform that does not support it natively? Of course, a general purpose "`become:`" comes with the prohibitive cost of the full heap scan; however the restricted usage of "`become:`" in our case can be implemented efficiently even in Java. The main idea is to produce a placeholder subclass (interface `Ph`) of each class that could be used as a placeholder, and to make each class that can be initialized using a placeholder implement the `ReplacePh` interface that defines a method to replace placeholders stored in fields with the placeholders' actual objects.

In this scheme, whenever an object is allocated using placeholders, we notify those placeholders that the newly allocated object refers to them. When a placeholder declaration completes initialisation, the introduced placeholders can be correctly and efficiently replaced. In the detail, such translation would:

- generate interfaces `ReplacePh`, `Ph` and class `PhAdd`

-Java-

```
interface ReplacePh{void replacePh(Ph ph, Object o);}
interface Ph{ List<ReplacePh> getList();}
class PhAdd{
public static<T> T add(T fresh, Object ... phs){
  for (Object o:phs)
    if(o instanceof Ph)
      ((Ph)o).getList().add((ReplacePh)fresh);
  return fresh;
  }}
```

  `ReplacePh` represents an instance whose fields can contain a placeholder `ph`, that can be replaced with `o` when the object is available;; `Ph` represents a placeholder, and can provide the list of all the objects whose fields point to that placeholder; `PhAdd.add` notifies placeholders in `phs` that the `fresh` object contains them in one of its fields.
- map both placeholder types and object types to the corresponding simple Java type; that is all the types of form C and C'' will be mapped to C.

– generate for all the classes and interfaces a "placeholder" class, extending the original one and the `Ph` interface, providing a no arguments constructor and a list of `Objects` containing all the objects whose fields refer to this "placeholder object",
– makes every class originally present in FJ′ implement `ReplacePh`. For example

FJ′
```
interface T1{...}
class C implements T1{
  T1 x; T2 y; C(T1' x,T2' y){this.x=x; this.y=y;}
  ...}
```

would be translated into

Java
```
interface T1{...}
class PhT1 implements T1,Ph{
  List<ReplacePh> list=new ArrayList<ReplacePh>();
  List<ReplacePh> getList(){return this.list;}
  .../*any method of T1 throws Error*/}

class C implements T1,ReplacePh{
  T1 x; T2 y; C(T1 x,T2 y){this.x=x; this.y=y;}
  void replacePh(Ph ph, Object o){
    if(this.x==ph)this.x=(T1)o;
    if(this.y==ph)this.y=(T2)o;
    }
  ...}
class PhC extends C implements Ph{
  List<ReplacePh> list=new ArrayList<ReplacePh>();
  List<ReplacePh> getList(){return this.list;}
  .../*any method of C throws Error*/}
```

– translate placeholder declarations into the three phase initialisation, so that:

FJ′
```
T1 x= new C().m(x,y),
T2 y= new D().m(x,y);
new K().m(x,y);
```

would be translated into

Java
```
//(1)initialise dummy placeholder objects
T1 _x= new PhT1();
T2 _y= new PhT2();
//(2) run normal initialisation code, using placeholder
T1 x= new C().m(_x,_y),
T2 y= new D().m(_x,_y);
//(3) replace placeholders with the correct value
for(ReplacePh o:_x.getList())  o.replacePh(_x,x);
for(ReplacePh o:_y.getList())  o.replacePh(_y,y);
new K().m(x,y);
```

– finally, for any constructor parameter that is statically a placeholder, we use static method `PhAdd.add` to insert the newly created object into the placeholder list; for example

FJ′
```
class C implements T1{
T1 m(T1′ x,T2′ y){ return new C(x,y);}
...}
```

would be translated into

Java
```
class C implements T1,ReplacePh{
T1 m(T1 x,T2 y){ return PhAdd.add(new C(x,y),x,y);}
...}
```

6 Related Work

Not embracing any sort of laziness
Many approaches in the area of circular initialisation do not support any form of laziness, and thus cannot support the initialisation example from our introduction in the obvious way. The construction process of one entity is interleaved with the initialisation process of other entities and they rely on explicit mutation to create circular object graphs.

Hardhat Constructors [8,21] restrict constructors to avoid the leak of partially initialised objects out of the constructor scope. Similar techniques can be employed by FJ′ to extend our calculus with more expressive constructors. OIGJ [23] and others [9,10] allow the creation of *immutable* object graphs by using the concept of "commitment points" (where the mutable object graph can be *promoted* to immutable). These approaches offer no guarantee against null pointer exceptions.

As Manuel Fahndrich, Songtao Xia, and Don Syme have astutely observed [5,20], recursive bindings in a functional language like OCaml [14] 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 [5]: *" 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 restrictions rule out any form of laziness, and makes it impossible to express any variations of the Factory pattern [7] 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 the bindings are established, the constructed values can be initialised normally.

Delayed Types [5] lift those limitations, but still are unable to provide good support for factory methods: indeed Delayed Types require factory methods to expose fields in their type annotations. In personal communication, Fahndrich agrees that this would break encapsulation for private fields and is in general not feasible in case of interfaces as return types. Moreover, both Delayed Types [5] and Masked types [16] require an explicit two step initialisation, where the values are created and the circular references are fixed explicitly afterward; that is, these two works try to verify code similar to the following:

```
-Java-  class C{ C myC; C(C myC){this.myC=myC;} }
        ... C c=new C(null);
            c.myC=c;
```

This kind of code is hard to maintain: when a modification to the internal implementation is needed, all the code that initialises new instances can be broken. Delayed Types and Masked Types [5,16] ensure the absence of `NullPointerException`s in this kind of code.

Freedom Before Commitment [19] proposes another approach, where constructors trigger the construction process 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 the expression **`new`** `A()`, the field `x.myB` is still **`null`** when variable `x` is visible. In [19] the absence of `NullPointerException`s is proved. This avoids an explicit two step initialisation, but leverages a sequence of recursive constructor calls where one of them triggers the initialisation of the other components.

To conclude, these works [5,16,19] focus on statically preventing null pointer exceptions, supporting safe initialisation in OO languages as they are at the moment. Our goal is to change the language semantics to support the intuitive idea that the client code is responsible for knowing the dependencies between the different components in the system, so that factory methods can receive correct parameters.

From the type system point of view we share many similarities to Summers and Müller's Freedom Before Commitment. This system has partially initialised objects (called free), like FJ′, although we do not need to expose an explicit *partially initialised object type*. Unlike FJ′, Freedom Before Commitment relies on constructors, and does not support factory methods. Summers and Müller have conducted an extensive study of applicability of their approach, and they do consider factories instead of constructors. Replacing constructors with factory/creation methods is suggested by Fowler [6] and is empirically shown to be a good methodology [4].

Uses some form of laziness

Languages with lazy semantics, like Haskell, can use laziness/thunks to encode placeholders, thus reaching an expressive power similar to FJ′; but without the static guarantees offered by our type system (this style of programming is often called *tying the knot* in the Haskell community). In FJ′, dereferencing a placeholder is a dynamic error similar to a `NullPointerException` (a stuck state in the formal model) while in Haskell, accessing a thunk performs the associated computation on demand.
FJ′ and Haskell also differentiate when placeholders are replaced or thunks evaluated:

- In Haskell a thunk is evaluated when its value is needed and, as soon as the computation ends, all the occurrences of the thunk are replaced with the corresponding value. That is, the lifetime of a thunk is unbounded, and in some cases it is possible for a thunk to not be evaluated at all, saving computation time.

– In FJ′ initialisation expressions are executed in the conventional top down, left to right order; placeholders introduced in a placeholder declaration are replaced as soon as the semicolon is reached. The lifetime of a placeholder is lexically bounded: when the initialisation is concluded the placeholder is replaced. Placeholders retain the simplicity and predictability of the call by value semantics.

Both in Haskell and in FJ′ a term like `T x = x;` is meaningless; those degenerate cases are not so uncommon as one can suppose. In Haskell, a non trivially degenerate case is the following:

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

Here type `L` denotes a list of integers, with the two conventional type constructors. The function `head` returns the head of the list, or `0` for the empty list. It is clear that the definition of the `main` makes no sense: 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                    f2 :: L -> L
f1 a= (LCons 42 a)              f2 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=(f1 x) in putStr(show(head x)))
main2=(let x=(f2 x) in putStr(show(head 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.

In the following example we show the code of the last example rephrased in FJ′; encoding functions as methods on classes. `User1` is verified by our type system, while `User2` is ill-typed. Note how classes `User1` and `User2` differ only in the implementation of method `f`. `User1.main` produces an infinite list composed by a single cell with value 42, while `User2.main` is ill typed.

FJ′
```
class List{ int e; List next; }
class User1 {
  List f(List' x){ return new List(42,x);}
  List main(){ List x=f(x); return x; } } //safe
class User2{
  List f(List' x){ return x; }
  List main(){ List x=f(x); return x; } } //unsafe
```

The significance of this problem is highlighted in a proposal by Syme [20]. 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.

7 Conclusion

"I call it my billion-dollar mistake. It was the invention of the null reference"

Tony Hoare, "Null References: The Billion Dollar Mistake" [12]

Today, writing correct and maintainable code involving circular initialisation is very difficult: most practical solutions rely on null values — Tony Hoare's "Billion Dollar Mistake". This is an important problem since many software architectures and natural phenomena involve circular dependencies. As in the "chicken and egg" paradox, circular dependencies are hard for humans to understand and to reason about. This has hampered the development of effective techniques for supporting safe circular initialisation in software engineering, and their support in programming languages.

There have been many proposals to solve the problem of circular data structure initialisation [19,14,5,16]. In our opinion FJ′ offers a simpler type system compared to these approaches; moreover it allows simpler and cleaner programming patterns to be used. We can obtain such simplicity because we attack the problem from two different angles: first we define a new concept — placeholders — with intuitive semantics, and then we develop a type system to ensure placeholders are used safely. We hope this gives placeholders a good chance of adoption by real language implementations.

We 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."*

Tony Hoare, "The Emperor's Old Clothes" [11]

We have tried our best to follow the first way, and the result looks pretty simple to us. Whether is it simple enough to fix the billion dollar mistake, only time will tell.

Acknowledgments. Thanks to Gilad Bracha, Alex Summers, Peter Müller, Manuel Fahndrich, David Pearce, Elena Zucca and Jonathan Aldrich. This work is funded by RSNZ Marsden Fund.

References

1. Birka, A., Ernst, M.D.: A practical type system and language for reference immutability. In: OOPSLA, pp. 35–49 (2004)
2. Bracha, G.: Executable grammars in Newspeak. In: JAOO (2007)
3. Bracha, G., von der Ahé, P., Bykov, V., Kashai, Y., Maddox, W., Miranda, E.: Modules as objects in Newspeak. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 405–428. Springer, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1883978.1884007
4. Counsell, S., Loizou, G., Najjar, R.: Evaluation of the 'replace constructors with creation methods' refactoring in Java systems. IET Software 4(5), 318–333 (2010)
5. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: OOPSLA. pp. 337–350 (2007)
6. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (2000)
7. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley (1995)
8. Gil, J(Y.), Shragai, T.: Are we ready for a safer construction environment? In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 495–519. Springer, Heidelberg (2009)
9. Gordon, C.S., Parkinson, M.J., Parsons, J., Bromfield, A., Duffy, J.: Uniqueness and reference immutability for safe parallelism. In: OOPSLA, pp. 21–40 (2012)
10. Haack, C., Poll, E.: Type-based object immutability with flexible initialization. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 520–545. Springer, Heidelberg (2009)
11. Hoare, C.A.R.: The emperor's old clothes. Comm. ACM 24(2) (February 1981)
12. Hoare, C.: Null references: The billion dollar mistake (March 2009), abstract of QCon London Keynote qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake
13. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: OOPSLA, pp. 132–146 (1999)
14. Leroy, X.: The Objective Caml system (release 2.00) (August 1998), http://paulliac.inria.fr/caml
15. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses - practical extensibility with object algebras. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 2–27. Springer, Heidelberg (2012)
16. Qi, X., Myers, A.C.: Masked types for sound object initialization. In: POPL, pp. 53–65 (2009)
17. Servetto, M., Mackay, J., Potanin, A., Noble, J.: The billion dollar fix: Safe modular circular initialisation with placeholders and placeholder types. Tech. Rep. 12-25, ECS, VUW (2012), http://ecs.victoria.ac.nz/Main/TechnicalReportSeries
18. Servetto, M., Potanin, A.: Our billion dollar fix. Tech. Rep. 12-19, ECS, VUW (2012), http://ecs.victoria.ac.nz/Main/TechnicalReportSeries
19. Summers, A.J., Müller, P.: Freedom before commitment - a lightweight type system for object initialisation. In: OOPSLA, pp. 1013–1032 (2011)
20. Syme, D.: Initializing mutually referential abstract objects: The value recursion challenge. Electronic Notes in Theoretical Computer Science 148(2), 3–25 (2006)
21. Zibin, Y., Cunningham, D., Peshansky, I., Saraswat, V.: Object initialization in X10. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 207–231. Springer, Heidelberg (2012)
22. Zibin, Y., Potanin, A., Artzi, S., Kiezun, A., Ernst, M.D.: Object and reference immutability using Java generics. In: Foundations of Software Engineering (2007)
23. Zibin, Y., Potanin, A., Li, P., Ali, M., Ernst, M.D.: Ownership and immutability in generic Java. In: OOPSLA, pp. 598–617 (2010)