Just-in-Time Inheritance

A Dynamic and Implicit Multiple Inheritance Mechanism

Mattias De Wael
Software Languages Lab
Vrije Universiteit Brussel (Belgium)
madewael@vub.ac.be

Janwillem Swalens
Software Languages Lab
Vrije Universiteit Brussel (Belgium)
jswalens@vub.ac.be

Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel (Belgium)
wdmeuter@vub.ac.be

Abstract

Multiple inheritance is often criticised for the ambiguity that arises when multiple parents want to pass on a feature with the same name to their offspring. A survey of programming languages reveals that no programming language has an inherently implicit and dynamic approach to resolve this ambiguity. This paper identifies just-in-time inheritance as the first implicit and dynamic inheritance mechanism. The key idea of just-in-time inheritance is that one of the parents is favoured over the others, which resolves the ambiguity, and that the favoured parent can change at runtime. However, just-in-time inheritance is not the silver bullet to solve all ambiguity problems heir to multiple inheritance, because it is not applicable in all scenarios. We conclude that the applicability of just-in-time inheritance is to be found in systems where multiple inheritance is used to model an "is-a *OR* is-a"-relation, rather than the more traditional "is-a *AND* is-a"-relation.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features: Inheritance; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs: Object-oriented constructs

Keywords multiple inheritance, just-in-time data structures

1. Introduction

Inheritance is one of the key features of object-oriented languages. It allows for code reuse and modularisation. Multiple inheritance is a feature that only some object-oriented languages support. However, multiple inheritance is often criticised due to the ambiguity introduced when inheriting features with colliding names from multiple parents [3]. Different solutions to resolve the ambiguity have been proposed over the years, ranging from rejecting ambiguous programs to putting the burden to resolve the ambiguity fully on the developer. A survey of the existing solutions (see section 2) reveals that most programming languages propose a static approach (*e. g.*, rejection at compile time) and that only a few allow the developer to express dynamic resolution strategies. Based on this survey, we conclude that no programming language exists today that proposes a *dynamic and implicit* multiple inheritance mechanism. We identify that such a mechanism is favourable when multiple inheritance is used to model an "is-a *OR* is-a" relation instead of the more conventional use of multiple inheritance, which is to model an "is-a *AND* is-a" relation.

The core contributions of this paper are the identification and the description of the first *dynamic and implicit* multiple inheritance mechanism, which we call *just-in-time inheritance*. The key idea behind just-in-time inheritance is that one of the multiple parents is favoured over the others and that the *favoured parent* can change at runtime.

Just-in-time inheritance has been implemented in the programming language JitDS [6]. The goal of this paper is to study just-in-time inheritance in isolation and in comparison to other multiple inheritance mechanisms. In earlier work, we already extensively motivated, discussed, and evaluated the language JitDS in general [6]. Here, the focus and argumentation is different: we study how other languages deal with the ambiguity introduced by multiple inheritance (section 2); we taxonomize just-in-time inheritance according to the Treaty of Orlando (section 2.1); we motivate the need for a *dynamic and implicit* multiple inheritance mechanism (section 3); we formally describe the semantics of a method invocation in just-in-time inheritance (section 3.2); we discuss in which scenarios just-in-time inheritance is applicable (section 4); and we compare just-in-time inheritance with other software engineering techniques (section 4.1). Finally, in section 5, we compare the idea of just-in-time inheritance with other programming languages and frameworks that are not necessarily related to multiple inheritance and section 6 summarises our findings.

2. Mitigating Ambiguity, *How Do They Do It*

When a class inherits from more than one super-class, one might wonder what happens with members that share the same name? This question identifies the ambiguity that arises when dealing with multiple inheritance. Language designers have tried to mitigate this ambiguity in various ways. Here, we provide a concise overview of these language designs:

Ambiguity Rejection A trivial solution to resolve ambiguity is to *not allow* it. A compiler that detects ambiguity usually generates an error or warning, which puts the burden on the developer to *work around* the problem. In C++, for instance, the implementation for `T`, which implements both `A` and `B`, will not be accepted by the compiler. GCC, for instance, gives the following error: `member 'foo' found in multiple base classes of different types`.

```
class A { virtual int foo(){ return 0;} };
class B { virtual int foo(){ return 1;} };
class T : public A, public B { };
```

Linearisation Another approach is to statically define, *e. g.*, as part of the language specification, how method and field lookup is going to be performed. To this end all features in the inheritance hierarchy, a graph, are *linearised* accordingly. The programming language Dylan, for instance, uses the C3 algorithm to turn the hierarchy graph into a linear structure [1]. The simplest, and often used, linearisation algorithm is "last man standing", where after textual inclusion of the super-classes, the last definition "wins". This is for instance the case in OCAML [19]. Lookups in a linearised hierarchy are similar to lookups in the context of single inheritance. Hence, there is no ambiguity.

Prioritised Multiple Inheritance also relies on some form of linearisation, but there the responsibility for resolving ambiguity lays explicitly with the developer [4]. It is the job of the developer to assign priorities to each of the parents to guide the linearisation. This strategy was first introduced in SELF [4].

Select and Rename is a technique to resolve ambiguity by explicitly renaming features with colliding names in the subclass. In Eiffel [14], to remove the ambiguity the version of `foo` from `A` could be renamed to `bar` and the version of `foo` from `B` could be renamed to `baz`.

```
class T inherit A rename foo as bar
                B rename foo as baz
```

Explicit Inheritance can be seen as a variant of "Select and Rename", where features with colliding names can only be used when they are explicitly qualified with the intended super-class' name. This is for instance allowed in C++, where the call `ptr->A::foo()` explicitly uses the method `foo` of `A` [8].

Protocols/Interfaces allow classes to inherit only the protocol they have to adhere to: only method signatures are inherited, without an implementation. Since this implies there is only a single implementation, namely that of the *implementing* subclass, no ambiguity on which code to execute is present. In Java, for instance, a class can inherit from only one super-class, but it can implement as many interfaces— this is how protocols are called in Java —as needed [12].

Method Combination is a technique where a developer can resolve ambiguity in the body of an overriding method, that needs access to a super-method, by combining all the super-methods. In Common Lisp (CLOS), for instance, a method can return the *sum* (*i. e.*, a combination) of the results of *all* its super-methods [22]. Besides +, eight other primitive method-combinators are available, but defining own method-combinators is also possible.

```
(defgeneric leaf-count (tree)
  (:documentation "Return leaf count.")
  (:method-combination +))
```

Most Specific Argument When a subclass overrides a method of one of its ancestors, it is in some languages allowed to *refine* the signature, *i. e.*, methods are co-variant in the return type and contra-variant in their argument types. In CLOS, for instance, the method that matches the argument types the most is invoked, *i. e.*, including dispatch on the argument type. Note that this technique only reduces the "chances" of ambiguity: when there is still ambiguity present CLOS resorts to other techniques to resolve the ambiguity.

Meta-object Protocol Some languages make the implicit behaviour (*e. g.*, member lookup) of objects (and classes) developer definable. A meta-object protocol (MOP) [17], for instance, allows a developer to redefine, on a per instance basis, how an object should behave when an attribute is requested. This meta-object protocol includes a definition of how an object should invoke a method. Thus, altering the meta-object of an object gives a programmer explicit control to implement his own feature resolution algorithm.

All techniques presented above[1] focus on *behaviour*. When *state* is also considered, there is another dimension that comes into play: replication versus unification. C++ [8], for instance, tackles this explicitly through *virtual inheritance*, which ensures that the state of a shared ancestor is only present once. In the remainder of this text, however, we focus on behaviour only.

[1] The attentive reader might be missing a discussion on mixins (or traits or shakeins) in this section. While mixins are arguably not multiple inheritance, they where introduced to resolve some problems that stem from multiple inheritance. When it comes to resolving ambiguity, however, mixins suffer from the same issues. Hence, languages that support mixins need to resort to one of the ambiguity mitigation strategies as presented in this survey, *e. g.*, linearisation as in Scala [20, Chapter 5].

Listing 1: `RowMajorMatrix`.

```
class RowMajorMatrix {

  var rows, col, data;

  RowMajorMatrix(rows, cols) {
    this.rows = rows;
    this.cols = cols;
    this.data = new Array();
  }

  getRows() { return rows; }
  getCols() { return cols; }

  get(row, col) {
    idx = row*getCols()+col;
    return data[idx];
  }


  set(row, col, val) {
    idx = row*getCols()+col;
    data[idx] = val;
  }


  transpose() {
    /* implementation omitted */
  }
}
```

Listing 2: `SparseMatrix`.

```
class SparseMatrix {

  var rows, col, data;

  SparseMatrix(rows, cols) {
    this.rows = rows;
    this.cols = cols;
    this.data = new Dictionary();
  }

  getRows() { return rows; }
  getCols() { return cols; }

  get(row, col) {
    idx = row*getCols()+col;
    if (data.contains(idx)) return data.get(idx);
    else                    return 0.0
  }

  set(row, col, val) {
    idx = row*getCols()+col;
    if ( val == 0.0) data.drop(idx);
    else             data.put(idx, val);
  }

  nonZeroElementsIterator() {
    return data.iterator();
  }
}
```

2.1 Treaty of Orlando

Stein, Lieberman, and Ungar identified three dimensions along which to categorise inheritance mechanisms [18]. The third dimension describes whether *objects* inherit from other objects, or whether *classes* inherit from other classes. This dimension is extensively discussed by Jones et al. [15], but falls out of the intended scope of this text.

> The first two dimensions have been described by Stein, Lieberman, and Ungar in their Treaty of Orlando as follows [18]:
>
> *First, whether* ***static*** *or* ***dynamic****: When does the system require that the patterns of sharing be fixed? Static systems require determining the sharing patterns by the time an object is created, while dynamic systems permit determination of sharing patterns during runtime, when an object actually receives a message.*
>
> *Second, whether* ***implicit*** *or* ***explicit****: Does the system have an operation that allows a programmer to explicitly direct the patterns of sharing between objects, or does the system do this automatically and uniformly? Explicit delegation (or inheritance) allows the ability to delegate only a single method, rather than "anything that can't be handled locally."*

If we categorise all the solutions from section 2 using the first two dimensions of the Treaty of Orlando [18], we identify that none of the existing solutions are *dynamic* (the system determines sharing patterns at runtime) and *implicit* (the system handles inheritance automatically and uniformly) at the same time. Concretely, most solutions are static in the sense that, *for a given inheritance hierarchy*, the lookup strategy is fixed at object creation time. Those few approaches that are dynamic require explicit intervention of the programmer to modify how lookup should take place, *e. g.*, changing the MOP or changing priorities.

3. Just-in-Time Inheritance

Just-in-time inheritance is a dynamic multiple inheritance mechanism where one of the parents is favoured over the others and where this *favoured parent* can change at runtime. This means that the just-in-time inheritance mechanism is both dynamic and implicit, which is rather unique according to section 2. In this section we first briefly sketch a scenario in which a dynamic and implicit multiple inheritance mechanism is favourable. Then, we explain how just-in-time inheritance is realised in JitDS.

Imagine two implementations of the mathematical concept "matrix". One implementation stores all the elements in a large array, while the other stores the non-zero elements in a dictionary with the row-column pairs as keys. As a result of the different memory layouts, the behaviour (*e. g.*, how are elements retrieved and stored) is also different in both implementations. The choice of how to store the elements can have a significant effect on the performance of an application. Thus, depending on how a matrix object is used and/or how many zero values it contains, a different implementation might be favourable. The favoured implementation can even change during the execution of the program, *e. g.*, when the number of zero-values increases. In such a scenario, *dynamically* changing from which implementation to inherit the behaviour (and state), can be desirable. Furthermore, having an *implicit* mechanism is be desirable over

Listing 3: The class `Matrix` combines two representations.

```
1 class Matrix
2 combines RowMajorMatrix, SparseMatrix {
3   ...
4 }
```

Listing 4: Using an instance of `Matrix`.

```
1 mA = new Matrix.RowMajorMatrix();
2 mA.set(0, 1, 123);
3 it = mA.nonZeroElementsIterator();
4 while( it.hasNext() ) { ... }
5 mA.get(0,1);
```

a mechanism where the intended implementation has to be specified explicitly every time.

In the remainder of this section we explain just-in-time inheritance by implementing the scenario sketched above. This scenario formed the motivating example when designing the recent programming language JitDS. The code we show here, uses the syntax from [6], but is stripped from its static types.

Listings 1 and 2 define two *simple* classes that mostly implement the same methods: `getRows`, `getCols`, `get`, and `set`. This is the set of getters and setters one would typically expect in the implementation of a matrix ADT. Further, `RowMajorMatrix` implements the method `transpose` (actual implementation is omitted) and `SparseMatrix` implements the method `nonZeroElementsIterator`, an accessor that only makes sense for "sparse matrixes".

In JitDS, it is possible to define a just-in-time class which *combines* multiple representations, which are simple classes. Instances of such a class, *just-in-time objects*, inherit the members of all parents and thus make use of some form of multiple inheritance.

An example is the just-in-time class `Matrix` that combines the two classes `RowMajorMatrix` and `SparseMatrix`, and which is shown in listing 3.

Listing 4 shows how to create a new just-in-time object with `RowMajorMatrix` as *initial representation*. This means that `RowMajorMatrix` is the favoured parent of the object referenced by `mA` and that when `set` is called (line 2), the method as it is found in `RowMajorMatrix` is invoked.

Calling the method `nonZeroElementsIterator` (line 3) also works as expected: no viable method is found in the favoured parent, hence the method as it is found in `SparseMatrix` is executed. However, when this happens, the favoured parent is implicitly changed to the *specialist* representation, *i. e.*, the parent that does support the requested method. In JitDS this is called a *specialised swap*. In this example, the favoured parent changes from `RowMajorMatrix` to `SparseMatrix` and thus when `get` is called on line 5, the method as it is found in `SparseMatrix` is invoked.

Listing 5: The class `Matrix` combines three representations.

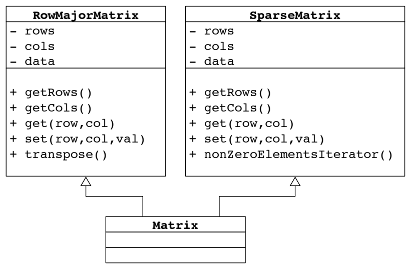
```
1  class Matrix
2  combines RowMajorMatrix,
3           ColMajorMatrix,
4           SparseMatrix  {
5
6    RowMajorMatrix to ColMajorMatrix {
7      target.rows = source.cols;
8      target.cols = source.rows;
9      target.data = source.data;
10     target.transpose();
11   }
12
13   ColMajorMatrix to RowMajorMatrix {
14     target.rows = source.cols;
15     target.cols = source.rows;
16     target.data = source.data;
17     target.transpose();
18   }
19
20   RowMajorMatrix to SparseMatrix {
21     target.rows = source.rows;
22     target.cols = source.cols;
23     target.data = new Dictionary;
24     for ( row in target.rows )
25       for ( col in target.cols )
26         target.set(row, col, source.get(row, col));
27   }
28
29   SparseMatrix to RowMajorMatrix {
30     target.rows = source.rows;
31     target.cols = source.cols;
32     target.data = new Array;
33     it = source.nonZeroElementIterator();
34     while ( it.hasNext() ) {
35       key, val = it.next();
36       target.data[key] = val;
37     }
38   }
39
40 }
```

Figure 1 shows how having a favoured parent can resolve ambiguity: in figure 1a it is unclear which of the methods has to be selected upon invocation, whereas in figures 1b and 1c the favoured parent (thick line) is chosen. The call to `nonZeroElementsIterator` causes the favoured parent to change, *e. g.*, like a transition from figure 1b to figure 1c.

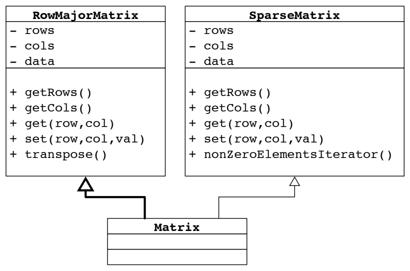
Transition Functions In practice, changing the favoured parent may be a less trivial transformation than assumed above. To express such transformations JitDS introduces a new kind of class member in the body of just-in-time classes: *transition functions*. Consider an extended version of `Matrix` (see listing 5) which this times also combines a third class `ColMajorMatrix`.[2]

The *transition function* on lines 6–11 in listing 5, for instance, defines the transition from a *source* object, an instance of `RowMajorMatrix`, to a *target* object, an instance `ColMajorMatrix`. The body of the transition function (between curly braces) is similar to a constructor's body, where `source` and `target` are pseudo-variables (*cf.* `this` and `super`) that denote the object as it was before

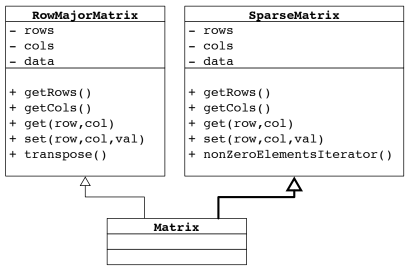
[2] The implementation is omitted, but is similar to the one found in listing 1 up to the computation of `idx`.



(a) A hierarchy where most method calls are ambiguous.



(b) A hierarchy where `RowMajorMatrix` is favoured.



(c) A hierarchy where `SparseMatrix` is favoured.

Figure 1: The hierarchy of `Matrix` with two super-classes `RowMajorMatrix` and `SparseMatrix`.

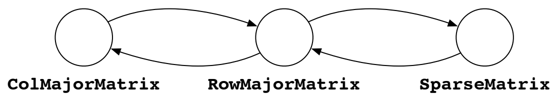


Figure 2: The transition graph for the just-in-time class `Matrix` from listing 5.

the transformation and the object as it supposed to be after the transformation, respectively. The set of four transition functions gives rise to the graph shown in figure 2. In JitDS, such a graph is called the *transition graph* of a just-in-time class.

When a new favoured parent has to be chosen, the transition graph resolves some ambiguity. The shortest path from the current favoured parent to another parent with the requested method is computed. The last node on this path becomes the new favoured parent. Note that just-in-time inheritance does not resolve *all* ambiguity. When the shortest path is not unique, the order in which the classes occur in the `combines`-clause is used as a tie-breaker. Relying on this tie-breaker is effectively a sort of *linearisation* of the transition graph, *cf.* section 2. In general, this ambiguity could be resolved using any of the solutions presented in section 2. When no path exists, a runtime error is issued.

Swap Statements Finally, JitDS also foresees a construct to explicitly change the favoured parent. To change the favoured parent of `mA` to `ColMajorMatrix`, one could use a *swap statement* and write `mA to ColMajorMatrix`.

3.1 Unconventional Class Hierarchy

For the sake of presentation, we opted to avoid a common ancestor for the matrix classes in the example programs above, hence the code duplication. The fields `rows` and `cols` and the methods `getRows` and `getCols`, for instance, could be implemented in a super class `VirtualMatrix` (*cf.* refactoring *Extract Superclass* [9]) to promote code reuse.

The UML diagram of the resulting class hierarchy (as shown in figure 3) allows us to reflect on the unconventional class/type hierarchy when modelling a just-in-time class. First, `RowMajorMatrix` and `SparseMatrix` inherit the trait of being "something with rows and columns" from `VirtualMatrix`. This is a classic class/type hierarchy, found in many programs. `Matrix`, on the other hand, combines `RowMajorMatrix` and `SparseMatrix` into a just-in-time class. As a result, we expect instances of `Matrix` to support all the methods found in both representations. In other words, `Matrix` inherits the behaviour of both its parents, and this is exactly what multiple inheritance is about. The class/type hierarchy, however, is rather unconventional because `Matrix` is unwontedly a subtype and subclass of its representations. This is the case because just-in-time inheritance effectively models an "is-a *OR* is-a"-relation.[3] An instance of `Matrix`, for instance, behaves at any point in time as either a `RowMajorMatrix` or as a `SparseMatrix`. Traditionally, however, the "is-a *OR* is-a"-relation is modelled as multiple *specialisations* (*i. e.*, inverse of inheritance) of a single class. However, the goal of

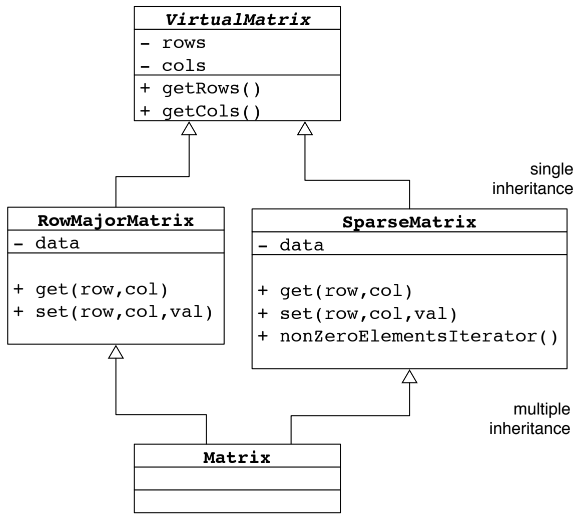


Figure 3: `VirtualMatrix` implements common behaviour, while the just-in-time class `Matrix` combines the behaviour of all its super classes.

[3] We revisit the "is-a *OR* is-a" relation in section 4 when we discuss the applicability of JitDS.

$$\frac{L(x) = \mathtt{\textbf{null}}}{(\langle \mathtt{vn\ =\ x.mn(\overline{a})\ \ ;}\ \overline{\mathtt{stmt}}, L\rangle : S, H, P) \longrightarrow (\langle \mathtt{Null\text{-}Pointer\text{-}Exception}, L\rangle : S, H, P)}\ \text{R-MI-NPE}$$

$$\frac{\begin{array}{c} L(x) = \mathit{objId} \qquad H(\mathit{objId}) = (\mathtt{Cn}_s, \mathtt{Cn}_d, F) \\ \textsf{find-class}(P, \mathtt{Cn}_d) = \mathtt{cd}_d \qquad \textsf{find-method}(\mathtt{cd}_d, \mathtt{mn}) = \mathtt{md} \qquad \mathtt{md} = \mathtt{mn}\,(\overline{\mathtt{pn}})\ \{\overline{\mathtt{stmt}_B}\ \mathtt{\textbf{return}\ x;}\ \} \\ \overline{L(a) = v} \qquad L_m = [\mathtt{\textbf{this}} \mapsto \mathit{objId}][\overline{\mathtt{pn} \mapsto v}] \qquad S' = \langle \mathtt{vn\ =\ x.mn(\overline{a});}\ \overline{\mathtt{stmt}}, L\rangle : S \end{array}}{(\langle \mathtt{vn\ =\ x.mn(\overline{a});}\ \overline{\mathtt{stmt}}, L\rangle : S, H, P) \longrightarrow \big(\langle \overline{\mathtt{stmt}_B}\mathtt{\textbf{return}\ x;}, L_m\rangle : S', H, P\big)}\ \text{R-MI-D}\textsc{irect}$$

$$\frac{\begin{array}{c} L(x) = \mathit{objId} \qquad H(\mathit{objId}) = (\mathtt{Cn}_s, \mathtt{Cn}_d, F) \qquad \textsf{find-class}(P, \mathtt{Cn}_d) = \mathtt{cd}_d \\ \neg\textsf{find-method}(\mathtt{cd}_d, \mathtt{mn}) \qquad \textsf{transition-path}(P, \mathtt{Cn}_s, \mathtt{Cn}_d, \mathtt{mn}) = \overline{\mathtt{Cn}_i} \qquad \mathtt{stmt}_t = \big\{\overline{\mathtt{x\ \textbf{to}\ Cn}_i\mathtt{;}}\mathtt{vn\ =\ x.mn(\overline{a});}\big\} \end{array}}{(\langle \mathtt{vn\ =\ x.mn(\overline{a});}\ \overline{\mathtt{stmt}}, L\rangle : S, H, P) \longrightarrow (\langle \mathtt{stmt}_t\ \overline{\mathtt{stmt}}, L\rangle : S, H, P)}\ \text{R-MI-I}\textsc{ndirect}$$

$$\frac{\begin{array}{c} L(x) = \mathit{objId} \qquad H(\mathit{objId}) = (\mathtt{Cn}_s, \mathtt{Cn}_d, F) \\ \textsf{find-class}(P, \mathtt{Cn}_d) = \mathtt{cd}_d \qquad \neg\textsf{find-method}(\mathtt{cd}_d, \mathtt{mn}) \qquad \neg\textsf{transition-path}(P, \mathtt{Cn}_s, \mathtt{Cn}_d, \mathtt{mn}) \end{array}}{(\langle \mathtt{vn\ =\ x.mn(\overline{a});}\overline{\mathtt{stmt}}, L\rangle : S, H, P) \longrightarrow (\langle \mathtt{Unsupported\text{-}Swap\text{-}Exception}, L\rangle : S, H, P)}\ \text{R-MI-USE}$$

Figure 4: Reduction rules for method invocation.

having a class that supports the combination of behaviour cannot be realised using the technique of multiple specialisations. This limitation was the driving motivation when designing JitDS.

3.2 Small-step Operational Semantics: Finding a Method

Just-in-time inheritance is not just a mind experiment to create a dynamic and implicit multiple inheritance mechanism. It has been given an executable implementation and a operational semantics (see [5]). Here, we focus on a specific part of the latter: the reduction rules for method invocation. We focus on method invocation because it is the most important feature when studying inheritance of behaviour. Figure 4 describes the method lookup as we define it for just-in-time inheritance. For completeness, however, we first introduce the auxiliary functions and constructs needed to understand the reduction rules.

3.2.1 Auxiliary Functions and Constructs

Foremost, we have a program P which is nothing more than a collection of class and just-in-time class definitions. The semantics of such a program is defined in terms of *reduction rules*. Such a rule defines the reduction from one configuration C to another configuration C'. A configuration C is a triple containing, from right to left, the program P, a heap H, and a stack $s_0 : S$, with s_0 the top stack frame and S again a stack. A stack frame is again a pair containing a sequence of statements and a local store L.[4]

Further, we make use of some auxiliary constructs and functions. Local stores (L) map from variable names to object identifiers (objId), and heaps (H) map from object identifiers to objects. An object itself is a triple $(\mathtt{Cn}_s, \mathtt{Cn}_d, F)$, with $\mathtt{Cn}_s$ the just-in-time class of the object, $\mathtt{Cn}_d$ the currently favoured parent, and F a mapping from field names to object identifiers (objId). The function find-class looks for a class definition $\mathtt{cd}$ in program P for a given name $\mathtt{Cn}$. The function find-method looks for a method definition $\mathtt{md}$ in a class definition $\mathtt{cd}$ for a given name $\mathtt{mn}$. Finally, transition-path tries to find a path of representations ($\overline{\mathtt{Cn}_i}$) from the currently favoured parent to another parent that has an implementation for the method $\mathtt{mn}$. Concretely, $\textsf{transition-path}(P, \mathtt{Cn}_s, \mathtt{Cn}_d, \mathtt{mn})$ looks in the program P for a given just-in-time class $\mathtt{Cn}_s$, in $\mathtt{Cn}_s$ it looks for a path $\overline{\mathtt{Cn}_i}$, originating in $\mathtt{Cn}_d$ and ending in a representation class that has an implementations for the method $\mathtt{mn}$.

3.2.2 Reduction of a Method Invocation

Here, we focus on the reduction of a method invocation, which has the form `vn = x.mn(a̅);`. R-MI-NPE and R-MI-DIRECT describe the simplest cases: when the receiver is null and when the method is simply found in the currently favoured parent, respectively. Concretely, R-MI-NPE reduces to a null pointer exception configuration, the equivalent of a runtime exception. R-MI-DIRECT looks up and finds the requested method in the current favoured parent and executes its body in the context of new stack frame.

However, when the current favoured parent does not provide an implementation (*cf.* $\neg\textsf{find-method}(\mathtt{cd}_d, \mathtt{mn})$ in R-MI-INDIRECT), then a transition path is computed and translated into a sequence of swap-statements. Implicitly, the swap-statements change the favoured parent of the just-in-time object, and the method invocation can now be reduced by R-MI-DIRECT. Finally, R-MI-USE describes what happens if no method implementation ($\neg\textsf{find-method}(\mathtt{cd}_d, \mathtt{mn})$) and no transition path ($\neg\textsf{transition-path}(P, \mathtt{Cn}_s, \mathtt{Cn}_d, \mathtt{mn})$) is found: the original configuration is reduced to an exception-

[4] In [5], a stack frame is actually a triple. Here, for the sake of simplicity, we ignore the *invocation context*, which is of no particular interest in the discussion on method invocation.

Listing 6: ClosedFile.

```
class ClosedFile {

  var path;

  ClosedFile(path) {
    this.path = path;
  }

  getPath() {
    return this.path;
  }

  move(newPath) {
    System.movef(this.path,
                 newPath);
    this.path = newPath;
  }
}
```

Listing 7: OpenFile.

```
class OpenFile {

  var path, filePtr;

  getPath() {
    return this.path;
  }

  read() {
    return this.filePtr.readln();
  }

  write(txt) {
    this.filePtr.write(txt+"\n");
  }
}
```

Listing 8: LockedFile.

```
class LockedFile {

  var path;

  getPath() {
    return this.path;
  }

  lock() {
    /* do nothing */
  }

}
```

configuration, which is the equivalent of a runtime exception.

From this fragment of the formal semantics of JitDS it is clear why the just-in-time inheritance mechanism is dynamic and implicit. The inheritance mechanism is *dynamic* because the method lookup is dependent on the favoured parent of an object (*cf.* $(\mathrm{Cn}_s, \mathrm{Cn}_d, F)$) and because the favoured parent Cn_d can change at runtime. At the same time, the inheritance mechanism is *implicit* because the method lookup strategy (*cf.* the combination of R-MI-NPE, R-MI-DIRECT, R-MI-INDIRECT, and R-MI-USE) is uniform for all objects and automatically applies the dynamic rule.

4. Applicability of Just-in-Time Inheritance

The core contribution of this paper is to *identify* just-in-time inheritance as the first multiple inheritance system that is both *dynamic* and *implicit* at the same time. Just-in-time inheritance is by no means the silver bullet to solve all problems heir to multiple inheritance: it only models the "is-a *OR* is-a" relation instead of the "is-a *AND* is-a" relation that multiple inheritance is traditionally used for. Hence, just-in-time inheritance is targeted specifically towards applications where changing the representation of an object at runtime is a key functionality.

De Wael et al. [6] identify the need for two kinds of representation changes: functional and non-functional representation changes. Functional representation changes are needed to realise the intended semantics of a program. In the upcoming *File Example*, for instance, we model a file with different *modes* such as open and closed. Non-functional representation changes do not change the semantics of a program but are introduced to realise non-functional requirements. In the upcoming *Matrix Example*, for instance, we model a matrix with different representations where changing between the representations improves the performance of the program.

The scenarios of the two example programs are in contrast to scenarios where multiple inheritance is generally used for. Multiple inheritance is traditionally used to create structural unions of state and behaviour *e. g.*, in the "PhdStudent extends Employee, Student"-example. In contracts, just-in-time inheritance is best used to model scenarios where the inheritance relation combines parents that are relatively *equivalent*.[5] This implies that just-in-time inheritance effectively models an "is-a *OR* is-a"-relation. In scenarios where multiple inheritance is used to compose *augmenting* parents,[5] the more traditional "is-a *AND* is-a"-relation is modelled. In these scenarios, just-in-time inheritance is a less interesting candidate.

Because JitDS relies heavily on representation changes, it benefits from using the dynamic just-in-time inheritance mechanism. For a more thorough motivation behind JitDS, an evaluation in terms of performance, and a discussion of the language implementation, we refer to our earlier work (see [5, 6]).

File Example We implement the example from Plaid [23], here using JitDS. The *File Example* is an example of a program with *functional representation changes*, *i. e.*, representation changes that are needed to realise the intended semantics of a program. In this example we present the class File, which inherits functionality from three other classes ClosedFile (listing 6), OpenFile (listing 7), and LockedFile (listing 8). All three classes implement the method getPath. Further, ClosedFile is the only class that provides functionality for moving a file (*cf.* move), OpenFile is the only class that provides functionality for reading and writing (*cf.* read and write), and LockedFile provides the method lock which does nothing but extending the interface of LockedFile.

The just-in-time class File combines these three classes (listing 9) and provides three transition functions. The first two transition functions implement the logic needed to open and close an actual file (on disk). Opening an actual file relies on System.fopen and closing a file requires the write buffer to be flushed and the actual file to be closed (flush

[5] Terminology by Chambers et al. [4].

Listing 9: The just-in-time class `File`.

```
class File
combines ClosedFile, OpenFile, LockedFile {

  ClosedFile to OpenFile {
    target.path = source.path;
    target.filePtr = System.fopen(source.path);
  }

  OpenFile to ClosedFile {
    target.path = source.path;
    source.filePtr.flush();
    source.filePtr.close();
  }

  ClosedFile to LockedFile {
    target.path = source.path;
  }
}
```

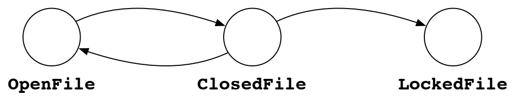


Figure 5: The transition graph for the just-in-time class `File` from listing 9.

Listing 10: A small example program using `File`.

```
1 f = new File("temp.txt");
2 f.write("test123\n");
3 f.lock();
4 f.write("test123\n");
```

and `close`). The third transition function is straightforward. The resulting transition graph is shown in figure 5. Note that there is no out-edge in `LockedFile`. This implies that a just-in-time object of the class `File` that transitions to the `LockedFile` representation, will stay in that representation forever.

Consider the transcript in listing 10. On line 1, a new file-object is created. Here, the initial representation does not need to be specified: it is unambiguously the constructor of `ClosedFile` that will be invoked, since the other two file classes do not provide a constructor. On line 2, a sequence of characters is written to the file, which succeeds because the file changes its favoured parent to `OpenFile`, using the first transition function. On line 4, the method `lock` is invoked. This also succeeds. However, because `lock` is only supported for files in the `LockedFile` representation, the second (open to closed) and third (closed to locked) transition function are executed subsequently. On line 4, again a sequence of characters is written to the file. In order to support I/O operations, the file needs to adhere to the `OpenFile` representation, which requires a *specialised swap*. Issuing a specialised swap, however, will raise a runtime exception, since `OpenFile` is not reachable from `LockedFile` in the transition graph (*cf.* figure 5). Finally, no explicit swap

Listing 11: The just-in-time class `Matrix`.

```
class Matrix
combines RowMajorMatrix, ColMajorMatrix {
  RowMajorMatrix to ColMajorMatrix { ... }
  ColMajorMatrix to RowMajorMatrix { ... }
}
```

Listing 12: A small example program using `Matrix`.

```
1 mA = loadMatrixFromFile("mat001.txt");
2 mB = loadMatrixFromFile("mat002.txt");
3 mC = multiply(mA, mB);
```

statements are *needed* in listing 10 because calling a specialised method invokes the intended representation changes implicitly.

Matrix Example The *Matrix Example* is an example of a program with *non-functional representation changes*, *i. e.*, representation changes that are not intended to change the semantics of a program, but that are introduced to realise a non-functional property. In this program, the representation changes are introduced to improve the performance of the application. The Matrix Example uses the class `Matrix`, which inherits functionality from the two previously introduced classes `RowMajorMatrix` and `ColMajorMatrix` (see listing 11), including the two first transition functions from listing 5, *i. e.*, from “row to col” and from “col to row”.

Consider the transcript in listing 12. In this simple program two matrixes are read from disk (lines 1–2) and are then multiplied (line 3). The implementation of `multiply` is shown in listing 13. The classic matrix-matrix multiplication algorithm is implemented on lines 12–24. The code on lines 2–10, however, is only added to improve the performance of the application. When the matrixes are too large to fit in the cache, changing the representations of `mA` and `mB` to `RowMajorMatrix` and `ColMajorMatrix`, respectively, greatly reduces the execution time due to a significant reduction in cache misses [5, 6]. This example formed the main motivation for designing JitDS, where additional constructs (swap rules) allow the developer to disentangle the application logic (lines 13–27) from the representation changing logic (lines 3–11). In contrast to the file example, the representation changes in this example are explicit, namely on line 7 and line 10. However, for the method invocations of `get`, `set`, `getRows`, and `getCols`, it is decided implicitly which representation to use.

4.1 Modularity and Design Patterns

Scenarios such as the file and the matrix example would traditionally be implemented as a single *class with modes* or

Listing 13: Matrix multiplication.

```
multiply(mA, mB) {
  /*  representation changing logic  */
  sizeA = mA.getRows() * mA.getCols();
  sizeB = mB.getRows() * mB.getCols();

  if ( sizeA > (CACHE_SIZE/2) )
    mA to ColMajorOrder;

  if ( sizeB > (CACHE_SIZE/2) )
    mB to RowMajorOrder;

  /* matrix-matrix multiply algorithm */
  rows = mA.getRows();
  cols = mB.getCols()
  mC = new Matrix.RowMajorMatrix(rows, cols);
  for (row in rows) {
    for (col in cols) {
      temp = 0.0
      for (k in mA.getCols() ) {
        temp += mA.get(row, k) * mB.get(k, col);
      }
      mC.set(row, col, temp);
    }
  }

  return mC;
}
```

Listing 14: The method `read` in a class with modes.

```
read() {
  switch ( this.mode ) {
    case CLOSED:
      this.filePtr = System.fopen( this.path );
      this.mode = OPEN;
      break;
    case LOCKED:
      throw new Exception( "Cannot␣open␣locked␣file." );
  }

  return this.filePtr.readln();
}
```

using a variation of the *strategy pattern*[6] as they are described by Gamma et al. [10]. We now argue that either technique hampers modularity and requires significant boilerplate code.

Implementing a class with modes hampers modularity because the functionality that used to be expressed in the different file classes (*i. e.*, `ClosedFile`, `OpenFile`, and `LockedFile`), now has to be combined into one class with three *modes*. This is, by definition, the opposite of modularity. Furthermore, a lot of boilerplate code is needed to check the current mode and to react accordingly. The implementation of `read`, for instance, needs guards expressing that: when the file is closed it needs to be opened first; and reading from a locked file is not possible. A possible implementation of the `read` method with these guards is given in listing 14. Similar guards have to be introduced in all methods, *i. e.*, all methods as they are found in `ClosedFile`, `OpenFile`, **and** `LockedFile`.

[6] In this context, strategy pattern, state pattern, or bridge pattern can considered to be similar.

Listing 15: The class `ClosedFile` with an extended interface compared to listing 6.

```
class ClosedFile {

  var path;

  ClosedFile(path) { this.path = path;}

  getPath() { return this.path;}

  move(newPath) {
    System.movef(this.path, newPath);
    this.path = newPath;
  }

  read()     { throw new Exception(); }
  write(txt) { throw new Exception(); }
  lock()     { throw new Exception(); }
}
```

Checking for the *mode* of an object and changing the behaviour accordingly (*e. g.*, the switch statement in listing 14) is considered to be a bad smell. To resolve this particular bad smell, Fowler [9] suggests the *Replace Type Code with State/Strategy* refactoring.

Realising a just-in-time class with the strategy pattern, however, also requires a lot of boilerplate code. In the classic strategy pattern, the changeable strategy classes adhere to the exact same interface. The representations of a just-in-time class, however, can have partially diverging interfaces (*e. g.*, `nonZeroElementIterator` in `SparseMatrix`). To cope with this issue, one could extend the original representation classes to match the unified interface (*i. e.*, the union of the interfaces of all representation classes). The methods that where previously unsupported could, for instance, be implemented by stubs that throw an exception. Listing 15 shows such an adaptation for the `ClosedFile` class. This, however, reduces modularity again.

One of the current implementations of JitDS [5, Chapter 7.3] is a JitDS-to-Java transpiler, which translates just-in-time classes into simple classes with strategies/state. Of course the transpiler takes care of generating the boilerplate code, which then no longer has to be written by the developer. This also improves modularity because the representation classes do not need to be aware of each other.

5. Related Work

In JitDS, the feature lookup in the hierarchy chain is dynamic because it effectively supports a restricted form of dynamic object reclassification, called homomorphic reclassification [6]. In this context, JitDS is related to other languages with support for dynamic object reclassification. These include, but are not limited to: Smalltalk, where any object can become (`a become: b`) any other object [11]; Self, where it is possible to dynamically update the parent pointer(s) and to re-order their priority [4]; Fickle$_{II}$, which is a language specifically designed to support dynamic object reclassification [7]; and Plaid, which is a typestate-oriented programming language [23].

Smalltalk does not support multiple inheritance. Self does support multiple inheritance and introduces *prioritised multiple inheritance*, which puts the responsibility for resolving ambiguity explicitly with the developer.

Fickle$_{II}$ and Plaid are, w.r.t. motivation, the two programming languages with the most resemblance to JitDS. Both Fickle$_{II}$ and Plaid support (functional) representation changes. However, in these programming languages, the relation between the different representations of an ADT is modelled as a sub-type relation and thus no multiple inheritance is needed. In JitDS, this relation is modelled as a super-type relation, hence the discussion on multiple inheritance. The difference in approach stems from the fact that Plaid and Fickle aim to support functional representation changes, whereas the focus of JitDS lays with non-functional representation changes.

The idea of non-functional representation changes itself is also not new. Bolz et al. [2], for instance, implemented a PyPy interpreter where the implementation of a collection can implicitly change at runtime to improve performance (storage strategies). Efforts with a similar motivation as storage strategies include [16, 21, 24, 26]. The examples described in these papers are typical scenarios where the "is-a *OR* is-a"-relation could be a good fit.

In the context of multiple inheritance all the languages used as examples in section 2 are related work [1, 8, 14, 17, 25]. Other approaches exist where method resolution is dynamically decided. Context-oriented programming (COP), for instance, identifies that contemporary method lookup mechanisms can be as complex as four dimensional and effectively allow *any computable property* (the "current context", *cf.* the fourth dimension) to have an influence on method resolution [13].

6. Conclusion

Many techniques exist to mitigate the ambiguity that arises when inheriting from multiple classes at the same time. Most of these techniques define the lookup behaviour statically and only a few languages allow the developer to change the lookup behaviour at runtime. None of the proposed solutions are both *dynamic* and *implicit*. In this paper, we introduce just-in-time inheritance, which is the first inherently dynamic ambiguity resolution mechanism that does not rely on explicit programmer intervention, but that handles all invocations automatically and uniformly. The key idea behind just-in-time inheritance is the "favoured parent", which reduces the possibility of ambiguity by changing the lookup hierarchy at runtime. In terms of applicability, we conclude that just-in-time inheritance is not a general solution to the ambiguity introduced by multiple inheritance, but excels at modelling the "is-a *OR* is-a"-relation.

Acknowledgments

The work in this text is supported by a research grant of IWT (Innovation through Science and Technology, Flanders).

References

[1] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A Monotonic Superclass Linearization for Dylan. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 69–82, 1996.

[2] C. F. Bolz, L. Diekmann, and L. Tratt. Storage Strategies for Collections in Dynamically Typed Languages. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '13, pages 167–182, 2013.

[3] T. A. Cargill. The Evolution of C++. chapter The Case Against Multiple Inheritance in C++, pages 101–109. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-73107-x.

[4] C. Chambers, D. Ungar, B.-W. Chang, and U. Hlzle. Parents are shared parts of objects: Inheritance and encapsulation in self. In *LISP and Symbolic Computation*, pages 207–222, 1991.

[5] M. De Wael. *Just-in-Time Data Structures*. PhD thesis, Vrije Universiteit Brussel, Belgium, May 2016.

[6] M. De Wael, S. Marr, J. De Koster, J. B. Sartor, and W. De Meuter. Just-in-time Data Structures. In *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '15, pages 61–75, 2015.

[7] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: Fickle(ii). *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24:153–191, 2002.

[8] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-51459-1.

[9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

[11] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

[12] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996. ISBN 0201634511.

[13] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.

[14] R. Howard. The Eiffel Programming Language. *Dr. Dobb's Journal*, 18(11):68–73, October 1993. ISSN 1044-789X.

[15] T. Jones, M. Homer, J. Noble, and K. Bruce. Object Inheritance Without Classes. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 56 of *ECOOP '16*, pages 13:1–13:26. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. ISBN 978-3-95977-014-9.

[16] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '11, pages 86–97, 2011.

[17] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0262111586.

[18] H. Lieberman, L. Stein, and D. Ungar. Treaty of Orlando. In *Addendum to the Proceedings of the Conference on Object-Oriented programming, systems, languages, and applications*, OOPSLA '87, pages 43–44, 1987.

[19] Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real World OCaml*. O'Reilly Media, 2013. ISBN 978-1449323912.

[20] M. Odersky and Others. The Scala Language Specification. 2004. URL `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.

[21] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive Selection of Collections. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI) '09*, pages 408–418, 2009.

[22] G. L. Steele, Jr. *Common LISP: The Language (2Nd Ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

[23] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter. First-class State Change in Plaid. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '11, pages 713–732, 2011.

[24] V. Ureche, A. Biboudis, Y. Smaragdakis, and M. Odersky. Automating ad hoc data representation transformations. Technical report, EPFL, 2015. URL `http://infoscience.epfl.ch/record/207050`.

[25] G. van Rossum. The Python Language Reference. Technical report, 1990-2015.

[26] G. H. Xu. Coco: Sound and adaptive replacement of java collections. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '13, pages 1–26, 2013.