Multiple Dispatch and Roles in OO Languages: $\mathcal{F}ickle_{MR}$

Submitted By:

Asim Anand Sinha
aas04@doc.ic.ac.uk

Supervised By:

Sophia Drossopoulou
scd@doc.ic.ac.uk

Department of Computing
IMPERIAL COLLEGE LONDON

MSc Project Report 2005

September 13, 2005

Abstract

Object-Oriented Programming methodology has been widely accepted because it allows easy modelling of real world concepts. The aim of research in object-oriented languages has been to extend its features to bring it as close as possible to the real world. In this project, we aim to add the concepts of *multiple dispatch* and first-class *relationships* to a statically typed, class based langauges to make them more expressive. We use $\mathcal{F}ickle_{||}$as our base language and extend its features in $\mathcal{F}ickle_{MR}$. $\mathcal{F}ickle_{||}$is a statically typed language with support for object *reclassification*, it allows objects to change their class dynamically. We study the impact of introducing multiple dispatch and roles in $\mathcal{F}ickle_{MR}$. Novel idea of *relationship reclassification* and more *flexible* multiple dispatch algorithm are most interesting. We take a formal approach and give a static type system and operational semantics for language constructs.

Acknowledgements

I would like to take this opportunity to express my heartful gratitude to DR. SOPHIA DROSSOPOULOU for her guidance and motivation throughout this project.

I would also like to thank ALEX BUCKLEY for his time, patience and brilliant guidance in the abscence of my supervisor.

Contents

Notations and Symbols

$\sqsubseteq$	$the subclass relation$
$\sqsubseteq_R$	$the\ subrelationship\ relation$
$\leq$	$the\ subtype\ relation$
χ	$heap$
σ	$stack$
$\vdash$	$judgement$
$\sqcup_p$	$least-upper-bound\ with\ relation\ to \leq$ in program P
$\leq_{spec}$	$compile-time\ more\ specific$
$\leq_{sped-d}$	$run-time\ more\ specific$
$\tau_1 \times \ldots \times \tau_n$	$argument-type$

Chapter 1

Introduction

> *I rejected multi-methods with regret because I liked the idea, but couldn't find an acceptable form under which to accept it.* **Bjarne Stroustrup**

In a typed object-oriented language, method selection is usually carried out at run-time due to polymorphism. Languages like Smalltalk, Java and C++ support *single dispatching*, in which the method to be executed is selected by the dynamic type of the receiver object and the static types of the method parameters, completely ignoring the dynamic types of the parameters. For example, consider the Java program in Figure 1.1, we have a 2-D point and a 3-D point, both of which have a `compare` method (`Point3D` class overrides `compare` method in Point2D). The `compare` method in class Point2D is more appropriate for comparing two 2-D points and the `compare` method in class Point3D is more appropriate for comparing two 3-D points. In lines 12 and 13, we create one instance of `Point3D` class `p1` and one instance of `Point2D` class `p2`. In line 14, invoking the `compare` method from `p1` unsurprisingly calls the `compare` method in Point2D class by matching the static types of the actual and formal arguments. In line 15, we assign an instance of `Point3D` class to `p2`. Now, `p2` has a static type `Point2D` and a dynamic type `Point3D`. In line 16, `p1.compare(p2)` still calls the `compare` method from `Point3D` class even though we are comparing two `Point3D` class instances. As mentioned above, in single dispatch the appropriate method is selected by the dynamic type of the receiver object and the static types of the arguments, therefore the dynamic type of `p2` plays no role in the method selection. We require a mechanism which takes the dynamic types of the parameters in consideration too, while selecting the appropriate method.

This gives rise to the concept of *multiple dispatch*, or *multi-methods*. Multi-methods are set of methods that are selected on the basis of the dynamic types of all the parameters, not just the receiver and multiple-dispatch is the mechanism of selecting the appropriate methods at run time. Multimethods offer higher expressability to a statically typed object-oriented programming during run-time. A language with support for multi-methods, for example, MultiJava or CLOS would call the appropriate method; in the case of figure 1.1, `compare` method from `Point3D` class. Stroustrup, the designer of C++, regrets that he was unable to consider providing multi-methods in his language, essentially because, as he admits, he was not able to find out how to do it (see Section 13.8 of [18]).

```
1. class Point2D{
2.      int x,y;
3.      int compare(Point2D p){
4.              if (p.x < this.x && p.y < this.y) return 1;
5.              else return 2;
        }
   }
6. class Point3D extends Point2D{
7.      int z; //inherits x,y
8.      int compare(Point3D p){
9.              if (p.x < this.x && p.y < this.y && p.z < this.z) return 1;
10.             else return 2;
        }
   }
11. class Test{
12.     Point3D p1 = new Point3D(...);
13.     Point2D p2 = new Point2D(...);
14.     p1.compare(p2); //calls compare from Point2D class
15.     p2 = new Point3D(...);
16.     p1.compare(p2);//calls compare from Point2D class
    }
```

Figure 1.1: Points Example in Java

Multiple dispatch has been implemented in some object-oriented languages such as MultiJava[6], Common Lisp Object System (CLOS)[9], Dylan and Cecil[4]. Unfortunately these languages are either not statically typed or have to compromise on several important features in a language, for example, modularity, separate-compilation properties(like in Java and C++) and symmetric treatment of parameters.

The concept of roles(relationships) have been around for a long time. It comes from database modelling languages like E-R diagrams and system modelling languages like UML. In roles based languages, objects can be assigned roles (relationships) which makes it easy for system designers to model the real world. For example, in Figure 1.2, a Teacher 'teaches' a Student, where an instance of `Teacher` is related to an instance of `Student` by an instance of relationship `Teaches`. Currently, languages do not support the concept of roles, hence the programmer is over-burdened with petty ways of implementing this feature, for example, in Figure 1.3, by creating a table of tuple $(Object \times Relationship \times Object)$ and searching through this table each time to retrieve the tuples. The relationships can be one-to-one, many-to-one, many-to-many and one-to-many. In this example many teachers can teach many students, hence it is many-to-many.

An object's roles may change with time, for example a `RetiredTeacher` may no more be related to a `Student` by role `teaches`. When adding roles in $\mathcal{F}ickle_{||}$, we have to consider what happens to an object's roles once it is reclassified: we could detele, add or simply *reclassify* roles. We may also have to

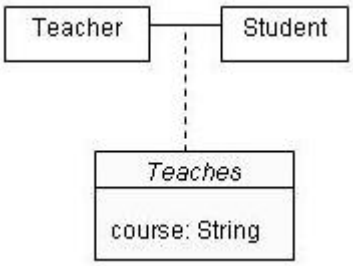

Figure 1.2: Teacher-Student Relationship

Object	*Relationship*	*Object*
William: Teacher	Teaches{course: programming}	Alex: Student
William: Teacher	Teaches{course: programming}	John: Student
William: Teacher	Teaches{course: type systems}	Alex: Student
Smith: Teacher	Teaches{course: programming}	Alex: Student
Smith: Teacher	Teaches{course: programming}	John: Student
...	...	...

Figure 1.3: Teacher-Student Table

update roles of other objects which were related to the object being reclassified. When deleting a role, we have to make sure there are no aliases still pointing to the old role. In databases, this is achieved by updating tuples with the help of foreign keys. It would be extremely useful if a database language supported reclassification and roles, where both: the object and its relationships could be reclassified dynamically. We study the reclassification semantics in $\mathcal{F}ickle_{||}$ and apply it to the novel concept of relationship reclassification. Another issue that we must think about is how to design *m-to-n relationships*.

In this project we implement multiple dispatch and roles in $\mathcal{F}ickle_{MR}$, an extension of $\mathcal{F}ickle_{||}$. Most interesting parts of the proejct are the more flexible multiple dispatch algorithm in $\mathcal{F}ickle_{MR}$ than its counterparts and the novel idea of relationship reclassification and reclassification directives.

Chapter 2 gives a basic theory of objects, object-oriented jargon, and existing research in multiple-dispatch and relationships. Chapter 4 gives an introduction to various functions for type-system and operational semantics which are used in the next chapters. Chapter 5 talks about our approach to multiple dispatch including the type-rules, operational semantics and a novel multiple-dispatch algorithm. Chapter 6 talks about relationships including extended RelJ and operational semantics for relationship reclassification.

Chapter 2

Background

This chapter gives the background research in *multiple dispatch* and *relationships* over several years. Multiple dispatch and relationships are programming language features and are studied in Programming Languages research. This chapter begins by introducing some theory and jargon related with object-oriented calculus. It then goes on to describe related work in multiple dispatch by colleagues with similar research interests. We then talk about the related work in relationships by Wren et al and Ghelli et al. For each topic, we also discuss the main challenges in their implementation at theoretical and prototypical level.

2.1 Theory of Objects

The popularity of Object-Oriented Programming languages shows its effectiveness and ease of use. Languages like C++ and Java have rapidly replaced procedural and functional languages like C. Concepts in object-oriented languages get their motivation from the real world, thus making it easier and more fun to model software systems. `Objects` can be thought of as entities of the real world we wish to model. For example, while modelling a solar system, we could treat `planets` and `stars` as objects. Objects have data (`fields`) like `planetName` and behaviour(`methods`) like `rotate` and `revolve`. The interaction and relationship between the various objects is modelled using object-oriented features like method calls, inheritance, association etc.

Simula was the first object-oriented language to be developed, which was developed for simulations[7]. Its concepts were later refined by the developers of Smalltalk, C++ and Java. Since then, the object-oriented methodology has been exploited in a wide range of applications, including development of user-interfaces, databases, operating systems and many more. A large part of the success of obect-oriented languages is because of its support for *software reuse*, i.e it allows reuse of software components.

The most important property of object-oriented languages is *polymorphism*, i.e. the property to take many forms. As described in [1], `Ad hoc polymorphism` simply refers to function overloading while `subtype polymorphism` allows a `subtype` to appear wherever a `base-type` is expected, also known as the

principle of substitutability. If $a : A$(a is an instance of class A) and $A <: B$(B inherits from(extends) A), then $a : B$(a is an instance of class B) ; we call this *subsumption*. When subsumtion occurs i.e. a subtype is converted to a base-type, the extra fields defined in base-type is only temporarily hidden (from field access) but not removed, so if we were to type-cast it back to the subtype, we would retrieve the extra fields with the original values. This is semantically necessary as we shall see in the following paragraphs.

```
/* ReCell <: Cell, Cell has a method 'set' which sets a field value to the
argument value. ReCell overrides 'set' and allows 'undo'*/
public class Cell{
        int value;

        void set(int n){
                value = n;
        }
}
public class ReCell extends Cell{
        int backup;

        void set(int n){
                backup = n;
        }
}
public class Test{
        public void g(Cell x){
                x.set(3);
        }
        //....
        //executing this code in main method
        ReCell myReCell=new ReCell(5);
        g(myReCell);
}
```

Figure 2.1: Dynamic Dispatch Example

With the introduction of subsumption, we have to re-examine the meaning of method invocation. For example, consider Figure 2.1 taken from [1]. We have a class Cell, which has a `set` method to set the value of `value` field. ReCell inherits from Cell and overrides `set` to allow `undo`; has a `backup` field and copies the `value` to `backup` before setting the new value. The method call `g(myReCell)` will behave differently in languages with static and dynamic dispatch. In *static dispatch*, the method resolution is based on the compile-time type information available for `x`. In programming languages which allow only static dispatch, the `set` method from class `Cell` would be executed. To provide flexibility to objects at run-time, all object-oriented languages implement *dynamic dispatch*. In dynamic dispatch, the method resolution is carried out at run-time, hence is based on the dynamic type of the objects. For example, in Java, the `set` method from class `ReCell` would be executed. Dynamic dispatch is very important to an object-oriented

language as this allows objects to behave autonomously.

Another important feature of multiple dispatch is that it can make use of the fields that become hidden (from field access) as a result of subsumption. For example, if an application of subsumption from `ReCell` to `Cell` were to force a `reCell` to a `cell` by cutting off its additional attributes (`backup`), then a dynamically dispatched invocation of `set` would fail. Although the field `backup` cannot be accessed directly, it is used by the method `set`.
The collection of `set` methods in Figure 2.1 is called a generic function. A *generic function* is a set of methods with the same name and number of arguments which differ in the types of the receiver object and the parameters; they define type-specific operations for those types. To take advantage of generic functions, the method resolution in dynamic dispatch is carried out at run-time unlike static dispatch where the method resolution is carried out at compile-time and is guaranteed not to cause `method-not-found` errors.

We briefly mention the concepts of *covariance* and *contravariance* as taken from [1]. The product type of an operation $A \times B$ is the type of pairs with left component of type A and right component of type B. We say that $\times$ is a *covariant* operator:
$A \times B <: A' \times B'$ provided that $A <: A'$ and $B <: B$
The type A $\rightarrow$ B is the type of functions with argument type A and the result type B. We call $\rightarrow$ the *contravariant* operator in its left argument because A $\rightarrow$ B varies in the opposite sense as A; the right argument is instead covariant:
$A \rightarrow B <: A' \rightarrow B'$ provided that $A' <: A$ and $B <: B'$
Finally, consider pairs whose components can be updated; we indicate their type by $A \sharp B$. The operator $\sharp$ does not enjoy any covariance or contravariance properties:
$A \sharp B <: A' \sharp B'$ provided that $A = A'$ and $B = B'$

2.2 Single and Double Dispatch

In most object-oriented languages, a message is sent to a distinguished receiver object. The run-time or dynamic type of the receiver determines the method that is invoked by the message. Other arguments of the message are passed on to the invoked method but `do not` participate in method lookup. Languages where methods are invoked in this manner are called *uni-dispatch* or *single-dispatch language*. Popular examples of single-dispatching languages are C++ and Java.

Single-dispatching works well for messages in which only the dynamic type of the receiver object defines the behaviour of the method invoked. Programmers of these languages do not realise the limitation of single-dispatch until they write code for inter-dependent behaviour between two or more objects which may have different static and dynamic types and which have different behaviour for each type. For example, consider *binary methods*[]. Binary methods can implemented by double-dispatching mechanism.

```
package java.awt;
class Component {
        //double dispatch event to subComponent

        void processEvent(AWTEvent e){
                if (e instanceof FocusEvent){
                        processEvent((FocusEvent)e);
                }else if (e instanceof MouseEvent){
                        switch(e.getID()){
                        case MouseEvent.MOUSE_RELEASED:
                        ...
                        case MouseEvent.MOUSE_EXITED:
                                processEvent((MouseEvent)e);
                                break;
                        case MouseEvent.MOUSE_MOVED:
                        case MouseEvent.MOUSE_DRAGGED:
                                processMouseMotionEvent((MouseEvent)e);
                                break;
                        }
                }else if (e instanceof KeyEvent){
                        processKeyEvent((KeyEvent)e);
                }else if (e instanceof ComponentEvent){
                        processComponentEvent((ComponentEvent)e);
                }else if (e instanceof InputMethodEvent){
                        processInputMethodEvent((InputMethodEvent)e);
                }
                //other events ignored by Component
        }

        void processFocusEvent(FocusEvent e){...}
        void processMouseEvent(MouseEvent e){...}
        void processMouseMotionEvent(MouseEvent e){...}
        void processKeyEvent(KeyEvent e){...}
        void processComponentEvent(ComponentEvent e){...}
        void processInputMethodEvent(InputMethodEvent e){...}
}
```

Figure 2.2: Double Dispatch in Java

Double dispatch is mainly a user-defined mechanism which dispatches the appropriate method by checking the dynamic type of the arguments and then type-casing to dispatch the appropriate method. There are many examples of binary methods which use double-dispatch[14, 16]. The Java AWT example from [**?**] is a very convincing example to show the complexity of implementing binary methods in single-dispatching languages.

Figure **??** gives the Java code for capturing various events relating to the component. An event in Java controls the behaviour of its component. `AWTEvent` is the base class for all events in Java AWT. Specific events such as `MouseEvent`, `FocusEvent`, `KeyEvent` etc. are direct or indirect subclasses of `AWTEvent`. A component performs specific action depending on the type of event received, by passing the event as argument to the `processEvent` method. Due to the limitation of single dispatch languages, `processEvent` must accept the event as an `AWTEvent`. It then performs type-casing on the dynamic type of the event and calls the appropriate method for that particular event. In Java, the dynamic type of an object is checked by the `instanceof` keyword.

Double-dispatch suffers from a number of disadvantages as we can see in Figure 2.2. First, type-casing is tedious and error-prone. Second, it has the overhead of invoking a second method. Third, to extend the program by adding a new event, it would require adding another `else if` statement.

2.3 Multiple Dispatch

Looking at the limitations of single-dispatch languages in Figure **??**, we stress the need for multiple dispatch in object-oriented languages. Figure 2.3 gives the code equivalent to the one in Figure 2.2, but in this one we assume that JVM supports multiple dispatch. In this example, there is no need to check the dynamic type of the argument explicitly because multiple dispatch automatically invokes the most specific method for the dynamic types of the arguments. The code is much simpler and shorter. Extending this code to add a new event can easily be done by adding a new `processEvent` method with the new event type.

Multiple dispatch is common in dynamically typed languages because the type-checking is performed at run-time alone so method selection is based on the dynamic type of the arguments alone. On the other hand, multiple dispatch in statically-typed languages is possible but poses some crucial restrictions and hence is not popular in such languages.

2.3.1 Challenges in Implementation

From the Java AWT example in Figure 2.3 it is quite apparent that multiple-dispatch is a desirable feature. As mentioned before, it is already quite common in dynamically-typed languages like CLOS, CECIL and PERL because the all the type-checking is performed at run-time alone. As a result, the language run-time system can optimally choose the right method to dispatch for a particular method call. This is the easiest way to implement multiple-dispatch. However, we believe in static type-system which ensures that a type-

```
package java.awt;
class Component {
        //double dispatch event to subComponent
        void processEvent(AWTEvent e){...}

        void processEvent(MouseEvent e){
                switch(e.getID()){
                case MouseEvent.MOUSE_RELEASED:
                ...
                case MouseEvent.MOUSE_EXITED:
                        processEvent((MouseEvent)e);
                        break;
                case MouseEvent.MOUSE_MOVED:
                case MouseEvent.MOUSE_DRAGGED:
                        processMouseMotionEvent((MouseEvent)e);
                        break;
                }
        }

        void processEvent(FocusEvent e){...}
        void processMouseEvent(MouseEvent e){...}
        void processMouseMotionEvent(MouseEvent e){...}
        void processEvent(KeyEvent e){...}
        void processEvent(ComponentEvent e){...}
        void processEvent(InputMethodEvent e){...}
}
```

Figure 2.3: Equivalent Code for Figure 2.2 in Multi-dispatch Java

correct code will not produce unanticipated errors at run-time. This is essential for the integrity of a software system. There are several issues that arise in type-checking multimethods.

Separate Compilation Problem

Languages like Java and C++ allow programs to be decomposed into several *compilation units* e.g. classes. The property of *separate compilation* is that if the individual compilation units type-check correctly in isolation, then the whole program is type-correct when all the separate compilation units are linked at run-time. This is an important property as it allows incremental development of programs i.e. a new class can be added to a program by compiling the new class alone, without having to re-compile the whole program.

With the introduction of multimethods (internal and external) type-checking must be performed on the whole multimethod to detect errors described next. Since there are no restriction to place the whole multimethod in the same compilation unit, the compiler has to check all the compilation units to type-check multimethods. This would not allow separate compilation of classes. Chambers et al suggest performing the type-check for ambiguities in multimethods at link time when all compilation units are available.

Message-Ambiguous and Message-Not-Understood Errors

Two methods `equals(A x,B y)` and `equals(B x, A y)` are ambiguous for a method call `equals(b,b)` where `B` is subclass of `A` and `b` is an instance of `B`. It is ambiguous because both the methods are equally specialized for the method call. This topic is discussed in detail in Section 5.6. This error can occur at compile time in single-dispatch languages like Java. In the case of multimethods, it is a bigger issue because in a multimethod, there are many methods which may cause ambiguities between each other and we can only type-check ambiguities if we make our system very restrictive. Chambers et al have suggested two type-checking solutions to make sure that a type-checked code does not cause message-ambiguous-error at run-time.

The first solution is to make sure that the multi-method is *fully* and *unambiguously* implemented. This is a general strategy that can be applied to internal and external methods both. By *fully* we mean that there must be a method for all possible combinations of arguments. By doing this, there will always be a clear match for a method call with any type of arguments. This is however very restrictive, as the programmers may not want to define methods for all possible combination of argument types. In programs like Java AWT, which contains long class hierarchies and methods with many arguments, it would become extremely tedious and wasteful to define a multimethod *fully*.

The second solution is specific to internal multi-methods. In this approach, a method is allowed to override another method only if the arguments of the overridden method is a super-type of the arguments of the overriding method. This would forbid two internal methods of the type `equals(A x,B y)` and `equals(B`

`x, A y)` because the arguments are not subtype of the arguments of the other method. This restriction is less restrictive than the first solution.

2.4 Symmetric vs Asymmetric Multiple Dispatch

Multiple dispatch is *symmetric* if the rules for method lookup treat all dispatched arguments identically. *Asymmetric* multiple dispatch typically uses lexicographic ordering, where earlier arguments are more important; a variant of this approach selects methods partly on the textual ordering of their declarations. Symmetric multiple dispatch is used in MultiJava, Cecil, Dylan, Kea, the $\lambda\&$-calculus, $\mathrm{ML}_{\leq}$ and Tuple. A major obstacle to adding symmetric multimethods to an existing statically-typed programming language has been their modularity problem: independently-developed modules, which typecheck in isolation, may cause type errors when combined. Previous work on adding multimethods to an existing statically-typed object-oriented language has either been done by forcing global typechecking[16] or by employing asymmetric multiple dispatch in order to ensure safety of modular typechecking[2, 3]. Property of separate compilation is necessary to provide flexibility, as in Java and symmetric multi-methods are important so that we dont have to specify or order our arguments, so in this project, we will try to come up with an approach that allows separate compilation and supports symmetric multi-methods.

Implementing multimethoding along with static typechecking poses a challenge. Static typechecking in the absence of polymorphism requires that we ensure at compile time that every operation receives the proper number of arguments and that these arguments are instances of the proper data types. To allow for subtype polymorphism, this constraint is relaxed in the following way: We assume that it is admissible for a variable of a formal argument of a function to be bound at run-time to a value that is an instance of its static type or an instance of any subtype of its static type. However, because instances of subtypes may be substituted for instances of super-types at run time, the method that is selected for execution at run time for a given generic function invocation may be different from the method that would have been selected, were the arguments of the same types as those of the static call. Therefore for each generic function invocation, we need to be able to determine that there exists some method whose formal argument types are supertypes of the types of the static argument types, that the run time type of each actual argument will be a subtype of the type of the corresponding static formal argument, and that the run-time type of the result will be consistent with the context in which it occurs.

2.4.1 Static Type-checking of Asymmetric Multimethods

In [2], the authors present a step-wise execution of a method invocation and provide some conditions for the result to be consistent. Due to sub-type polymorphism, a generic method call may match one or more methods. Multi-methods can be statically type checked as follows:

1. *Insure that all method definitions are mutually consistent.*
   Two methods $m_i(T_i^1, T_i^2, \ldots, T_i^n) \rightarrow R_i$ and $m_j(T_j^1, T_j^2, \ldots, T_j^n) \rightarrow R_j$ of a generic function M are *mutually consistent* if whenever they are both applicable for arguments of types $T^1, \ldots, T^n$ and

m_i is more specific than m_j, then $R_i \preceq R_j$. A generic function is consistent if all of its methods are mutually consistent.

2. *For each static generic function invocation* $m(T^1, T^2, \ldots, T^n)$*, apply the following procedure:*
   (a) Determine that there is atleast one method that is applicable for the invocation.
   (b) Find the definition of the most specific applicable method m_k for the argument types $T^1, \ldots, T^n$.
   (c) Verify for the method $m_k(T_1^k, T_2^k, \ldots, T_n^k) \rightarrow R_k$ that if the result of the function incovation $m(T^1, T^2, \ldots, T^n)$ is assigned to a variable $r : R \leftarrow m(T^1, T^2, \ldots, T^n)$, then $R_k \preceq R$.

Step (1) is performed once at method definition time. Step (2) is performed at compile time for each generic function invocation. Agrawal et. al give a graph based algorithm for computing and invoking the most specific method whenever a generic function call is made. A *method graph* for a confusable set is an undirected graph such that

1. For every method m_i in the confusable set, there is a node labelled by that method.
2. There is an edge between the nodes labelled m_i and m_j if and only if m_i and m_j are confusable.

A *method precedence graph* for a confusable set is a directed graph such that

1. For every method m_i in the confusable set, there is a node by that method.
2. There is an arc from node m_i to m_j if and only if m_i and m_j are confusable and m_i is more specific than m_j.

Consider the case when A and B are types such that $B \preceq A$ and confusable methods $m_1(A, B)$ and $m_2(B, A)$. In this case an invocation $m(b, b)$ cannot be resolved to either $m_1(A, B)$ or $m_2(B, A)$ using only argument subtype precedence. In this case an argument order is required (for example, left to right). A method precedence graph induces a graph over the result types of the corresponding methods. A *result graph* for a confusable set is a directed graph such that

1. For every result type R_i that is a result type of some method m_i in the confusable set, there is a node labelled by that result type.
2. There is an arc from node R_i to R_j if and only if there is an arc from node m_i to m_j in the method precedence graph.

Given a set of confusable methods, we require an ordering on their specificity to build the *method precedence graph*. There are several design issues when ordering the applicable functions according to their specificity. These design issues are very important to the run-time performance and flexibility of multimethods. These can be classified as:

1. *Arbitrary Global Precedence*: The user orders confusable methods in any way the user pleases. This however imposes extra burden on the user and is clearly not a good solution.

2. *Argument Subtype Precedence*: In a generic function, the subtype relationships between the corresponding formal argument types of two confusable methods $m_i(T_i^1, T_i^2, \ldots, T_i^n)$ and $m_j(T_j^1, T_j^2, \ldots, T_j^n)$ are used to establish the precedence between those methods. If $\forall k, 1 \leq k \leq n, T_i^k \preceq T_j^k$, and $\exists l, 1 \leq l \leq n$, such that $T_i^l \preceq T_j^l$, we say that m_i *precedes* m_j. This method is not sufficient for multimethods and multiple inheritance.

3. *Argument Order Precedence*: An `argument order` is a total order over the argument positions of the methods of a generic function. It may correspond to the left-to-right ordering of arguments or it may be any permutation of that order. This approach is known as *asymmetric* multiple dispatch. This is the easiest to implement but requires the user to order the actual/formal arguments in the appropriate way.

Further to the graphs, Agrawal et. al give an algorithm for method resolution, and calculate the complexity of the algorithm to be $O(m(m+1)/2)$ where m is the number of methods of the generic function. The approach here is that of **asymmetric multiple dispatch** since the arguments are ordered in some ways.

Another approach to asymmetric multi-methods can be seen in [3]. The solution Castagna et. al present here is an extension to Java respecting the modularity and separate compilation properties of Java, but it treats the arguments asymmetrically i.e. in some order. They use *parasitic methods* which is a Java method which additionally extend the functionalities of other methods(the *host* methods) for certain argument cases. If a host method is called with arguments that fit the parasitic method's parameter types, the parasitic method body is called instead of the host method body.

2.4.2 Global Type-checking Of Multimethods

Another approach to multimethods has been in form of forcing the entire program to be type-checked as a whole. This however compromises on the flexibility that Java-like languages offer of *separate typechecking* of classes. *Tuple* is a class-based language that supports global type-checking. In [16], the authors propose a new way to add multimethods to a single dispatch languages like Java and C++. The idea here is to add tuples as primitive expressions and to allow messages to be sent to tuples. Selecting a method based on the dynamic classes of the elements of the tuple gives multiple dispatch. Tuple is a simple language, not very elegant but demonstrates that is can be added to existing single dispatch languages without affecting the semantics or typing of existing programs written in these languages. The authors implement SDCore, a class-based single dispatching language which is extended to allow multiple dispatch.

SDCore is similar to a conventional OO language like Java and C++. To keep things simple, it implements single inheritance. Dynamic dispatching is used to determine which method to invoke. In particular, the receiver's class is first checked for a method implementation of the right name and number of arguments. If no such method exists, then that class's immediate superclass is checked and so on up the inheritance hierarchy until a valid implementation is found otherwise a *message-not-understood* error occurs. To statically check a message send expression of the form $E_0.I(E_1, \ldots, E_n)$, we check that the static type of E_0 is a

subtype of a class which contains method I of type $(T_1, \ldots, T_n) \rightarrow T_{n+1}$, where the $E_1, \ldots, E_n$ are subtypes of $T_1, \ldots, T_n$ respectively. In Figure 1.1 if `p1:Point3D` and `p2:Point2D` to start with, and later `p2` is assigned a dynamic Point3D type: `p1.compare(p2)` will fail to call the `compare` method from Point3D class. Multimethoding allows method selection to be based on the dynamic types of the method arguments too, hence providing more flexibility and expressability.

```
tuple class (p1:Point, p2:Point){
        method equal(): bool
        { p1.x()==p2.x() and p1.y()==p2.y()}
}
tuple class (cp1:ColorPoint, cp2:ColorPoint){
        method equal(): bool
        {cp1.x()==cp2.x() and cp1.y()==cp2.y() and cp1.color()==cp2.color()}
}
```

Figure 2.4: Point Class Structure in Tuple

In Tuple, the expression $(E_1, \ldots, E_n)$ creates a tuple of size n with components $v_1, \ldots, v_n$ where each v_i is the value corresponding to E_i. In Figure 2.4, instead of defining the `equals` method within the `Point` and `ColorPoint` classes, two new *tuple* classes are declared with a tuple of two receivers each `p1`,`p2`,`cp1` and `cp2`. These receivers can be used by all methods inside the tuple class. However, note that we used `p1.x()` and not `p1.x` to access the field since tuple classes are client code and have no priviliged access to the fields of such components (Point and ColorPoint). This is important if we wish to resepect the encapsulation properties of languages like C++ and Java. There can be more than one tuple class for a given tuple of classes. Since no changes are made to `Point` and `ColorPoint` classes, their subtype relation is unchanged.

The syntax for sending messages to a tuple is the same as that of an instance. For example, `(p1,p2).equal()` sends the message `equal()` to the tuple `(p1,p2)`, which will invoke one of the two `equal` method. Just as method lookup in single dispatching relies on the dynamic type of the receiver, here it relies on the dynamic types of the tuple components. The semantics of message sends to an instances remain the same. The use of dynamic classes distinguishes multiple dispatch from static overloading(as found in primitive languages).

In order to type-check tuples, the authors add *product types* of the form $(T_1, \ldots, T_n)$: these are types of tuples containing elements of types $T_1, \ldots, T_n$. A product type $(T'_1, \ldots, T'_n)$ is a subtype of $(T_1, \ldots, T_n)$ if each T'_i is a subtype of T_i. Consider an invocation of tuple `(Point, ColorPoint)`, since no tuple classes define such tuple of receivers, a *method-ambiguous* error will occur. This issue is solved by ensuring two properties : monotonicity and ensure that two methods are not ambiguous. In order to ensure the above mentioned properties, the entire program has to be type-checked as a whole.

2.4.3 Best Solution: Ensuring Symmetry and Separate Type-checking

Java's property of separate compilation provides us with flexibility and is too important to be compromised. Symmetry, on the other hand is important too as we do not need to order out formal and actual arguments. The best solution to multi-methods would be one that supports both: symmetry and separate type-checking. These two properties have not been achieved in a class-based statically typed OO language. In [17], the authors suggest their ideas in the form of language Dubious. Dubious includes:

- a classless object model with explicitly declared objects and inheritance (but omits dynamically created objects and state).
- first-class generic functions (but omits kexically nested closures).
- explicitly declared function types (but omits explicitly declared object types and subtyping independent of objects and inheritance), and
- modules to support separate typechecking (but omits nested modules, parameterized modules, and encapsulation mechanisms).

Since it is a classless language, the **object** declaration creates a fresh object with a unique statically known identity. The declaration also names the objects from which the new object directly inherits (it's parent). The **isa** relation plays the role of inheritance and instantiation. We will skip discussions on Dubious' syntax and dive straight into it's type checking mechanism.

Dubious' static type system ensures that legal programs do not have `message-not-understood` error or `message-ambiguous` errors. Ruling out `message-ambiguous` errors involves two kinds of checks: *client-side* and *implementation-side*. Client-side checks are local checks on the declarations and expressions. The most important client-side check is that for each message send expression $E(E_1, \ldots, E_n$ in the program, E conforms to an arrow type $(T_1, \ldots, T_n) \rightarrow T$ and each E_i conforms to T_i. The message send is then given the type T. Implementation-side checks ensure that each generic function o correctly implements the arrow type $(T_1, \ldots, T_n) \rightarrow T$ to which it conforms. For every object tuple $(o_1, \ldots, o_n)$ such that each o_i conforms to T_i and o_i is not abstract, there must exist a most specific applicable method in o. Client-side checks on the declarations in a module require only type information from imported modules, and they can therefore be performed on a module-by-module basis. Implementation-side checks assume global knowledge of program's objects and methods. Therefor implementation-side typechecking is deferred until link-time, when all modules are present. This global typechecking algorithm is called *System G*. For semantics of System G please refer to Appendic B.2 in [17].

2.5 Relationships

The idea of *relationships* is not new. Relationships are widely used in modelling database systems. The idea of relationships in class based languages is however pretty new. Wren et al presented a paper on first

class relationships in FOOL/ECOOP 2005 [15]. In that paper they present the RelJ calculus which is a subset of Java with first class relationships. The instances of relationships are called *relations* just like the instances of classes are called objects. Since relations are first class they can be stored in variables and fields.

Having support for relationships in object-oriented programming languages would be a distinct advantage to programmers. The programmers are forced to implement relationships between objects by resorting to complicated ways. For example by creating a table of the type $(Object \times Object \rightarrow Relation)$. With the help of relationships, they can concentrate on their system design in a high level way and not have to worry about implementing relationships.

2.5.1 RelJ Calculus

We give a basic introduction to the RelJ calculus here. Relationships are declared in RelJ by a statement of the form

relationship r extends $r^{'}$ from n_1 to n_2 $\{FieldDecl^*\}$

where r is the name of the relationship, $r^{'}$ is the super-relationship of r, n_1 and n_2 are classes or relationships which are related through r. By allowing n_1 and n_2 to be class and relationship both, RelJ allows relationships to participate in further relationships which is called *aggregation* in E-R modelling. In RelJ relationships have fields but they do not have methods.

The authors argue that inheritance in relationships is better explained by delegation. Delegation is another way to implement inheritance. In delegation based inheritance, each relationship in the inheritance hierarchy has an instance of itself in the store which delegates to its super-relations. In [15] the authors give a convincing example why delegation is more suited for relationship inheritance. They impose two invariants. `Invariant 1` says that if relationship r extends $r^{'}$, for every instance of r between n_1 and n_2 there is an instance of $r^{'}$ also between n_1 and n_2. This means that for every sub-relations between n_1 and n_2 there is an instance of its super-relations between n_1 and n_2. Invariant 2 says that for every relationship r and pair of objects o_1 and o_2, there is at most one instance of r between o_1 and o_2. This invariant makes sure that if two sub-relationships extend the same super-relationship, then if we create instances of the two sub-relationships between o_1 and o_2 then both of them will delegate to the same instance of super-relationship.

Relations cannot be created explicitly in RelJ like `new` creates a new object in Java. Relations are created implicitly when a new object or relation is placed into the relationship. For example by an expression of the form $e.r+ = e^{'}$ which creates a new r relation between objects/relations resulting from expressions e and $e^{'}$, and returns this relation. A relation between two objects/relations is removed by $e.r- = e^{'}$ which returns the deleted relation. Removal of relations causes dangling pointers.

2.5.2 Challenges in Extending $\mathcal{F}ickle_{MR}$ with Relationships

Having first class relationships means they are run-time objects which can be stored in heap along with classes. This means that everytime, a memory location on the heap is accessed, the system must check to see if it is an object or relation. Because of this the operational semantics become complicated. While typechecking method calls, the agruments to a method call may be objects or relations, so the system has to check each argument to see if it is an object or relation.

Another challenging and most interesting part of this project is the idea of *relationship reclassification*. $\mathcal{F}ickle_{||}$allows objects to reclassify at run-time. What happens to the object's relationships when it is reclassified? It would be extremely wasteful of resources if we discard the object's relations. We suggest the novel idea of relationship reclassification. So if an object reclassifies, its relations will reclassify too (if the relationship hierarchy permits this). It is possible to perform automatic reclassification of an object's relations, however it is quite restrictive as discussed in section 6.4. In SQL, changes between objects and its relations are done by explicitly defining assoications between classes and their relationships. If the objects of those classes are updated, a message is sent to the associated relations to update in the defined manner. We introduce the concept of *reclassification directives*. The reclassification directives are user-defined piece of code which defines set of actions to be performed when an object/relation reclassifies to another object/relation. This is discussed in more detail in section 6.4.

Relationship are just like classes with an additional `from` and `to` attributes. It would make it easy for us if there was a way to unify relationships and classes. At the moment, we do not think this is possible because of two reasons. The first and less important reason being that relationships have extra `from` and `to` attributes which classes do not have. The second and more important reason is that relationship inheritance is done by delegation whereas class inheritance is not. As a result, the creation of new objects is very different from the way new relations are created. This effects the way relationships are reclassified too. We discuss this in greater detail in Relationships chapter.

Chapter 3

$\mathcal{F}$ickle$_{||}$

In class-based object-oriented languages, an object's behaviour is determined by its class. Although an object of a subclass is allowed to have a dynamic type which is a subtype of its static type, this often is not enough to represent the change in a system as we shall see. Usually statically typed, class-based programming languages do not provide support for *reclassification*. Reclassification changes the class membership of an object at run-time while retaining its identity. For example, `students` paying reduced and `employees` paying full conference fees are best described through distinct classes `Stdt` and `Empl` with different `fee()` methods. In a language without reclassification, how would we model a case where `mary`, who was a `student`, became an `employee`. In database systems, this would typically be done by creating a new Empl instance for mary and deleting the old student instance. This approach is crude and computationally extensive. $\mathcal{F}ickle_{||}$[11] is an imperative, statically typed, class-based object-oriented language. It provides support for reclassification. This section provides a brief overview of $\mathcal{F}ickle_{MR}$.

Since $\mathcal{F}ickle_{MR}$ is a class-based object-oriented language, classes are types and subclasses are subtypes. It achieves dynamic reclassification of objects by explicitly changing the class membership of objects. A `re-classification` operation changes the class membership of an object while preserving its identity; it maintains all fields common to the the class of the object being reclassified and the class to which it is reclassifying. `State` classes are those classes which are allowed to reclassify. A Root class is the superclass of state classes. Each state class has exactly one root class. Classes with the same same root class are allowed to reclassify to each other. Since state classes can reclassify an alias to an object before reclassification will become type incorrect after reclassification. For this reason, $\mathcal{F}ickle_{||}$does not allow instance variables of state class type but it allows parameters of state class types in methods because the lifetime of parameters are restricted to the methods they are declared in. Fickle[10] did not allow both: parameters and instance variables of state class type, which was later changed in $\mathcal{F}ickle_{||}$. Drossopoulou et al prove that $\mathcal{F}ickle_{||}$type system is sound, so execution of a well-typed $\mathcal{F}ickle_{||}$program produces a value of the expected type, or a null-pointer exception, but does not get stuck.

Smalltalk and CLOS support changing of class membership at run-time; however, these languages are not

statically typed. $\mathcal{F}ickle_{||}$[11] is an extension of $\mathcal{F}$ickle. $\mathcal{F}$ickle only allows the receiver to be of a state class and to be reclassified, whereas $\mathcal{F}ickle_{||}$ also allows parameters to be of a `state` class and to be reclassified.

3.1 $\mathcal{F}ickle_{||}$ Syntax

The syntax of $\mathcal{F}ickle_{||}$ is given in Figure 3.1. A $\mathcal{F}ickle$ program is a sequence of class definitions, consisting of `field` and `method` definitions. Class definitions may be preceded by the keywords `state` or `root` indicating whether the class is a state class or root class. Reclassification can occur only with `state` classes. Objects of a state class `c` may be re-classified to class c′, where c′ must be a subclass of the uniquely defined root superclass of `c`. The type-system ensures that re-classification does not cause access to fields and methods that do not belong to it. Any subclass of a state or a root class must be a state class. Objects of a `non-state`, `non-root` class `c` behave like regular `Java` objects, i.e. are never re-classified.

$$
\begin{array}{lcl}
prog & ::= & class^* \\
class & ::= & [\mathbf{root} \mid \mathbf{state}]\ class\ c\ \mathbf{extends}\ c\{field^*\ meth^*\} \\
field & ::= & type\ f \\
meth & ::= & type\ m\ (par^*)\ \mathbf{eff}\{e\} \\
type & ::= & \mathbf{bool} \mid c \\
par & ::= & type\ x \\
eff & ::= & \{(c \Downarrow c^{'})^*\} \\
e & ::= & \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid var := e \mid e; e \mid sVal \mid \\
 & & this \mid var \mid new\ c \mid e.m(e^*) \mid id \Downarrow c \\
var & ::= & x \mid e.f \\
sVal & ::= & \mathbf{true} \mid \mathbf{false} \mid \mathbf{null} \\
id & ::= & \mathbf{this} \mid x
\end{array}
$$

Figure 3.1: Syntax of $\mathcal{F}ickle_{||}$

The methods signatures have a set of effects which define the kind of reclassifications taking place in the method. The syntax for effects has been taken from $\mathcal{F}ickle_{|||}$ which changes the syntax of effects from $\mathcal{F}ickle_{||}$. In $\mathcal{F}ickle_{||}$the effects were a set of root classes whose state classes were being reclassified. In $\mathcal{F}ickle_{|||}$ the effects are a set of tuples of the type $c \Downarrow c^{'}$ which tell the compiler exactly what reclassifications are occuring in the method.
Method declaration have the following shape:

$$tm(t_1 x)\{c_1, ..., c_n\}e$$

where *t* is the resulting type, t_1 is the type of the formal parameter `x`, and `e` the body. We require root

classes to extend only non-root and non-state classes, and state classes to extend either root classes or state classes. The judgement $\vdash$ P$\Diamond_h$ (the inheritance hierarchy is well formed) asserts that program P satisfies these conditions, as well as the requirements for acyclic inheritance and unique definitions.
For a more comprehensive introduction to $\mathcal{F}ickle_{||}$please see [10, 11, **?**].

3.2 An Example

```
abstract root class Player extends Object{
  bool brave;

  abstract bool wake(){ };
  abstract Weapon kissed(){Player};
}

state class Frog extends Player{
  Vocal pouch;

  bool wake(){ } {pouch.blow(); brave}
  Weapon kissed(){Player}{this⇓Prince; sword := new Weapon}
}

state class Prince extends Player{
  Weapon sword;

  bool wake(){ }{sword.swing(); brave}
  Weapon kissed(){Player}{sword}
  Frog cursed(){Player}{this⇓Frog; pouch := new Vocal; this}
}

class Princess extends Object{
  bool walk1(Frog mate){Player}{mate.wake(); mate.kissed(); mate.wake()}
  Weapon walk2(Frog mate){Player}{mate⇓Prince; mate.sword :=new Weapon}
}
```

Figure 3.2: A $\mathcal{F}ickle_{||}$ program

The example in Figure 3.2 has been taken from [11]. `Frog` and `Prince` extend `player`. Player is declared as root class while Frog and Prince are state classes and are allowed to mutate. However, state classes cannot be used as types for fields. When woken up, a frog inflates its pouch while a prince swings his sword. When kissed, a frog turns into a prince; when cursed, a prince turns into a frog. The root classes define the fields and methods common to their state subclasses. The subclasses of root classes must be state classes. A state class c must have a (possibly indirect) root superclass c'; objects of class c may be reclassified to any subclass of c'. Annotations like and Player before method bodies are *effects*; they list the root classes of all objects that may be reclassified by invocation of that method.

In the `kissed` method of Frog class, `this`⇓`Frog` statement changes the object's class membership to that of Prince which has the field `sword` and can no longer access the `pouch` field. The field `brave` retains it's value. This mechanism can be used to transmit some information about the object before the reclassification to the object after the reclassification. Consider the following code:

```
1. Frog mate := new Frog;          //creates an instance of Frog
2. mate.wake();                    //Frog inflates pouch
3. mate.kissed();                  //Frog gets reclassified into Prince
4. mate.wake();                    //Prince swings sword
```

If `mate` is bound to a `Frog` object with field `brave` containing `true`, then after performing reclassification (`mate.kissed()`) mate's `brave` field will still contain `true`. Reclassification removes from the object all fields that are not defined in its root superclass and adds the remaining fields of the target class. Reclassification is transparent to aliasing, For instance consider the following code:

```
1. Player p1,p2;
2. p1 := new Prince;
3. p2 := p1;
4. p1.cursed();
5. p2.wake();                      //inflates pouch
```

In the presence of aliasing we have to be sure that no reclassification on an object occurs while a method is being executed from one of its aliases. Because the class membership of objects of state class is transient, access to their members(e.g. to `sword` from `Prince`) is only legal in contexts where it is certain that the object belongs to the particular class. This can be done to "local" entities, i.e. for parameters, the receiver `this` and for local variables. But it cannot be done for fields, as their lifetime exceeds a method activation. Therefore, state classes are not allowed to appear as types of fields. For example, the following is illegal:

```
class Witch{
        Prince friend;                  //illegal
        Weapon search(){friend.sword()}
}
```

If on the other hand, had it been legal to have state classes as types of fields, then in the following code, line 2 would cause a `fieldNotFoundError`:

Thus, state classes mustn't be allowed as types of fields, however they can appear as types of `this`, parameters and return types of methods. In `walk1` method of `Princess` class in Fig 3.2, state class `Frog` is the parameter type of `mate` and the return type of method `cursed` in `Prince` class. For detailed operational semantics and typing rules please refer to [11].

```
// w:Witch, p1:Prince, w.friend := p1;
1. p1.curse();                    // p1 is now a Frog
2. w.search();                    // error
```

3.3 Operational Semantics

This section describes the operational semantics for reclassification operator in $\mathcal{F}ickle_{||}$. The operational semantics rewrites pairs of expressions and stores into pairs of values, exceptions, or errors and stores in the context of a program P. The signature of the rewriting relation $\rightsquigarrow$:

$$\rightsquigarrow : prog \longrightarrow e \times store \longrightarrow (val \cup dev) \times store$$

where *store* maps `this` to an `address`, variable `x` to a value *val* and `address` to `object`. Values in $\mathcal{F}ickle_{||}$are `true`, `false`, `null` and `addresses`. Deviations may be `nullPtrException` and `stuckError` but the type-system ensures that a well-typed $\mathcal{F}ickle_{||}$program does not cause stuckError at run-time. Objects have their class identifier and contains field-value pairs.

$$\begin{array}{lcl} store & ::= & (\{this\} \longrightarrow addr) \cup (x \longrightarrow val) \cup (addr \longrightarrow object) \\ val & ::= & sVal \cup addr \\ dev & ::= & \{nullPtrExc, stuckErr\} \\ object & ::= & \{\ [[\ c\ f_1 : v_1, \ldots, f_r : v_r\ ]]\ \mid\ f_1, \ldots, f_r\ area; field\ identifiers, \\ & & \quad v_1, \ldots, v_r\ \in\ val, c\ is\ class\ identifier\} \end{array}$$

3.4 Type System

Chapter 4

Syntax and Basic Judgements

In this chapter we define the basic semantics for $\mathcal{F}ickle_{MR}$, that will help us define the *operational semantics* and *type-system* for multiple-dispatch and relationships in the next chapters.

4.1 Syntax of $\mathcal{F}ickle_{MR}$

$prog$	::=	$class^* \mid meth^* \mid relationship^*$
$class$	::=	$[\mathbf{root} \mid \mathbf{state}]\ class\ c\ \mathbf{extends}\ c\{field^*\ meth^*\ relDir^*\}$
$relationship$	::=	$[\mathbf{root} \mid \mathbf{state}]\ \mathbf{relationship}\ rel\ \mathbf{extends}\ rel\ \mathbf{from}\ [class \mid rel]\ to\ [class \mid rel]$ $\{field^*\ meth^*\ relDir^*\}$
$field$	::=	$type\ f$
$meth$	::=	$type\ m\ (par^*)\ \mathbf{eff}\{e\}$
$relDir$	::=	$\mathbf{upon\ reclassification\ to}\ [c \mid r]\{e\}$
$type$	::=	$\mathbf{bool} \mid c \mid rel \mid set < n >$
par	::=	$type\ x \mid type@type\ x$
eff	::=	$\{c^*\}$
e	::=	$\mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid var := e \mid e; e \mid sVal \mid$ $this \mid var \mid new\ c \mid e.m(e^*) \mid m(e^*) \mid id \Downarrow c$
var	::=	$x \mid e.f$
$sVal$	::=	$\mathbf{true} \mid \mathbf{false} \mid \mathbf{null} \mid empty$
id	::=	$\mathbf{this} \mid x$

Figure 4.1: Syntax of $\mathcal{F}ickle_{MR}$

Figure 4.1 defines the syntax for $\mathcal{F}ickle_{MR}$. The syntax presented here is an extension of $\mathcal{F}ickle_{||}$[11]. To allow multiple dispatching, we allow formal parameters to have types of the form $type@type$. To have *external methods*, we allow methods to be declared outside classes. Calls to *external methods* are of the form

$m(e^*)$, without a receiver object. To differentiate between internal and external method calls, we require internal method calls to have a distinguished receiver object. The next extension is to allow relationships. Relationships have a `from` and `to` attributes which range over classes and relationships. We also allow *reclassification directives* which defines the user-defined action when reclassification occurs in both: relations and objects. We use the set type from RelJ. The set type of the form set¡*n*¿ is a set containing number of *n* types. This will be introduced in chapter 6.

4.2 Judgements

We begin our formalization by imposing a *unique definitions* judgement $\vdash\ P\ \diamondsuit_u$, which guarantees that all class, field, method and relationship names are unique in a program P. Please note that through the uniqueness of method identifiers we eliminate method overloading which means a class can have only one implementation of method with the same name. This helps us to keep our semantics simple.

$$
\begin{array}{lcl}
\forall c & : & (\mathtt{P{=}P}_1\ class\ c\ extends\ c'\{\ldots\}\ P_2,\ P = P_3\ class\ c\ extends\ c''\{\ldots\}\ P_4) \\
 & & \Rightarrow P_1 = P_3, P_2 = P_4 \\
\forall r & : & \mathtt{P{=}P}_1\ \ relationship\ r\ extends\ r^{'}\ from\ n_1\ to\ n_2\{\ldots\}\ \ P_2 \\
 & & \mathtt{P{=}P}_3\ \ relationship\ r\ extends\ r^{''}\ from\ n_1^{'}\ to\ n_2^{'}\{\ldots\}\ \ P_4 \\
 & & \mathtt{P}_1 = P_3, P_2 = P_4 \\
\forall f & : & \mathtt{P{=}P}_1\ class\ c\ extends\ c'\{defs_1\ t\ f\ defs_2\}\ P_2, \\
 & & \mathtt{P{=}P}_1\ class\ c\ extends\ c''\{defs_3\ t^{'}\ f\ defs_4\}\ P_2 \\
 & & \Rightarrow defs_1 = defs_3, defs_2 = defs_4; \\
\forall m & : & \mathtt{P{=}P}_1\ class\ c\ extends\ c'\{defs_1\ t\ m(t_1\ x)\{e\}\ defs_2\}\ P_2, \\
 & & \mathtt{P{=}P}_1\ class\ c\ extends\ c''\{defs_3\ t^{'}\ m(t_1^{'}\ x)\{e\}\ defs_4\}\ P_2 \\
 & & \Rightarrow defs_1 = defs_3, defs_2 = defs_4; \\
\forall m & : & \mathtt{P{=}P}_1\ rel\ r\ \ldots\{defs_1\ t\ m(t_1\ x)\{e\}\ defs_2\}\ P_2, \\
 & & \mathtt{P{=}P}_1\ rel\ r\ \ldots\{defs_3\ t^{'}\ m(t_1^{'}\ x)\{e\}\ defs_4\}\ P_2 \\
 & & \Rightarrow defs_1 = defs_3, defs_2 = defs_4; \\
\hline
 & & \vdash\ P\ \diamondsuit_u
\end{array}
$$

Figure 4.2: Unique Judgement

We define the class-lookup and relationship-lookup function for a program P with $\vdash\ P\ \diamondsuit_u$ by :

$$
\mathcal{C}(P,c) = \begin{cases} class\ c\ extends\ c'\{cBody\} & if P = P'\ class\ c\ extends\ c'\{cBody\}\ P'' \\ \mathcal{U}df & otherwise \end{cases}
$$

$$
\mathcal{R}(P,r) = \begin{cases} relationship\ r\ extends\ r'\ from\ n_1\ to\ n_2\{rBody\} & if P = P'\ relationship\ r\ extends\ r'\ from\ n_1\ to\ n_2\{rBody\}P'' \\ \mathcal{U}df & otherwise \end{cases}
$$

where c, r are class and relationship identifiers respectively.

$$\frac{\vdash P\diamondsuit_u}{\text{P}\vdash Object \sqsubseteq Object} \qquad \frac{\vdash P\diamondsuit_u \quad \text{P=}\ldots\text{class c extends c}'\{\ldots\}\ldots}{\begin{array}{l}\text{P}\vdash c \sqsubseteq c\\ \text{P}\vdash c \sqsubseteq c'\end{array}} \qquad \frac{\text{P}\vdash c \sqsubseteq c' \quad \text{P}\vdash c' \sqsubseteq c''}{\text{P}\vdash c \sqsubseteq c''}$$

$$\frac{\vdash P\diamondsuit_u}{\text{P}\vdash Relation \sqsubseteq_R Relation} \qquad \frac{\vdash P\diamondsuit_u \quad \text{P=}\ldots\text{relationship r extends r}'\ldots\{\ldots\}\ldots}{\begin{array}{l}\text{P}\vdash r \sqsubseteq_R r\\ \text{P}\vdash r \sqsubseteq_R r'\end{array}} \qquad \frac{\text{P}\vdash r \sqsubseteq_R r' \quad \text{P}\vdash r' \sqsubseteq_R r''}{\text{P}\vdash r \sqsubseteq_R r''}$$

Figure 4.3: Subclasses and Subrelations

Figure 4.3, gives the rules for well-formedness of objects and relationships hierarchies. The symbol $\sqsubseteq$ has the usual meaning of subclass, i.e. $c \sqsubseteq c'$ means c is a subclass of c'. The symbol $\sqsubseteq_R$ has the meaning of sub-relation, i.e. $r \sqsubseteq_R r'$ means r is a subrelation of r'. The above rules assert that each class is a subclass of itself and any of its superclasses that it extends directly or indirectly. The same applies for relationships as stated in the rules. $\sqsubseteq$ and $\sqsubseteq_R$ are transitive.

$$\frac{\begin{array}{ll}\forall c, c': & \text{P}\vdash c \sqsubseteq c' \text{ and } P \vdash c' \sqsubseteq c \Longrightarrow c = c'\\ & \text{C(P,c) = class c extends c}'\{\ldots\} \Longrightarrow \mathcal{C}(P, c') = class\ c' \ldots\\ & \text{C(P,c) = root class c extends c}'\{\ldots\} \Longrightarrow \mathcal{C}(P, c') = class\ c' \ldots\\ & \text{C(P,c) = state class c extends c}'\{\ldots\} \Longrightarrow ((\mathcal{C}(P, c') = root\ class\ c' \ldots) \text{ or}\\ & \quad (\text{C(P,c}') = state\ class\ c' \ldots))\\ \forall r, r': & \text{P}\vdash r \sqsubseteq_R r' \text{ and } P \vdash r' \sqsubseteq_R r \Longrightarrow r = r'\\ & \text{R(P,r) = relationship r extends r}' \ldots \{\ldots\} \Longrightarrow \mathcal{R}(P, r') = relationship\ r' \ldots\\ & \text{R(P,r) = root relationship r extends r}' \ldots \{\ldots\} \Longrightarrow \mathcal{R}(P, r') = relationship\ r' \ldots\\ & \text{R(P,r) = state relationship r extends r}' \ldots \{\ldots\} \Longrightarrow ((\mathcal{R}(P, r') = root\ relationship\ r' \ldots) \text{ or}\\ & \quad (\text{R(P,r}') = state\ relationship\ r' \ldots))\end{array}}{\vdash P\diamondsuit_h}$$

Figure 4.4: Well-formed inheritance hierarchy

Figure 4.4 asserts that the class hierarchy in a program P is well-formed if it is acyclic, root classes extend only non-root and non-state classes, and state classes extend either root or state classes. If the same applies to relationships too, then we say that a *program is well formed.* We say that an *environment is well formed,* $P \vdash \Gamma\diamondsuit$, if it maps `this` to a class or relationship, and `var` to any type. We also need the judgements

$P \vdash c\diamondsuit_c$ asserting that c is a class, $P \vdash r\diamondsuit_r$ asserting that r is a relationship and $P \vdash t\diamondsuit_t$ asserting that t is a type. The well-formedness of environment is shown by the rules in Figure 4.5.

$$\frac{\vdash P\diamondsuit_u \quad \text{C(P,c) = class c ...}}{\text{P}\vdash c\diamondsuit_c \quad \text{P}\vdash c\diamondsuit_t} \qquad \frac{\vdash P\diamondsuit_u \quad \text{R(P,r) = relationship r ...}}{\text{P}\vdash r\diamondsuit_r \quad \text{P}\vdash r\diamondsuit_t} \qquad \frac{\vdash P\diamondsuit_u}{\text{P}\vdash bool\diamondsuit_t} \qquad \frac{\text{P}\vdash \Gamma(x)\diamondsuit_t \quad \text{P}\vdash \Gamma(this)\diamondsuit_c \quad \text{P}\vdash \Gamma(this)\diamondsuit_r}{\Gamma P\Gamma\diamondsuit}$$

Figure 4.5: Well-formed Environment

4.3 Lookup Functions

In this section we define various lookup functions for classes and relationships. In addition to the *field and method lookup* functions, we define the reclassification directive lookup function which will be discussed later. For a program P which satisfies the judgements $\vdash P\diamondsuit_u$ and $\vdash P\diamondsuit_h$ for *unique definitions* and *well-formed inheritance hierarchy*, and a class identifier *c* where $\mathcal{C}(P,c) = [root|class]\ class\ c\ extends\ c'\{cBody\}$, field identifier *f* and method identifier *m*, we define the following field and method lookup functions in Figure 4.6:

$$\begin{aligned}
\mathcal{FD}(P,c,f) &= \begin{cases} t & if\ cBody = \ldots t\ f \ldots \\ \mathcal{U}df & otherwise \end{cases} \\
\mathcal{F}(P,c,f) &= \begin{cases} \mathcal{FD}(P,c,f) & if\ \mathcal{FD}(P,c,f) \neq \mathcal{U}df \\ \mathcal{F}(P,c,f) & otherwise \end{cases} \\
\mathcal{F}(P,Object,f) &= \mathcal{U}df \\
\mathcal{F}s(P,c) &= \{f|\mathcal{F}(P,c,f) \neq \mathcal{U}df\} \\
\mathcal{MD}(P,c,m) &= \begin{cases} t\ m(t_1\ x_1,\ldots,t_n\ x_n)\ \textit{eff}\ \{e\} & if\ cBody = \ldots t\ m(t_1\ x_1,\ldots,t_n\ x_n)\ \textit{eff}\ \{e\}\ldots \\ \mathcal{U}df \end{cases} \\
\mathcal{M}(P,c,m) &= \begin{cases} \mathcal{MD}(P,c,m) & if\ \mathcal{MD}(P,c,m) \neq \mathcal{U}df \\ \mathcal{M}(P,c,m) & otherwise \end{cases} \\
\mathcal{M}(P,Object,m) &= \mathcal{U}df \\
\mathcal{M}s(P,c) &= \{m|\mathcal{M}(P,c,m) \neq \mathcal{U}df
\end{aligned}$$

Figure 4.6: Field and Method Lookup Functions for Classes

The $\mathcal{FD}(P,c,f)$ and $\mathcal{MD}(P,c,m)$ functions are used to fetch the fields and methods in class *c*, while the

$\mathcal{F}(P, c, f)$ and $\mathcal{M}(P, c, m)$ functions are used to fetch the fields and methods from *c* or its superclasses. The recursive lookup stops at `Object` in the case of classes and `Relation` in the case of relationships. These functions are the same as presented in **??** because the structure of classes have not changed.

The field and method lookup functions for relationships in Figure 4.7 are similar to the ones for the classes. For a program P which satisfies the judgements $\vdash\ P \diamondsuit_u$ and $\vdash\ P \diamondsuit_h$, and a relationship identifier *r* where $\mathcal{R}(P, r) = relationship\ r\ extends\ r^{'}\ from\ n_1\ to\ n_2\{rBody\}$, field identifier *f* and method identifier *m*, we define the following field and method lookup functions in Figure 4.7:

$$
\begin{aligned}
\mathcal{FD}_R(P, r, f) &= \begin{cases} t & if\ rBody = \ldots t\ f \ldots \\ \mathcal{U}df & otherwise \end{cases} \\
\mathcal{F}_R(P, r, f) &= \begin{cases} \mathcal{FD}_R(P, r, f) & if\ \mathcal{FD}_R(P, r, f) \neq \mathcal{U}df \\ \mathcal{F}_R(P, r^{'}, f) & otherwise \end{cases} \\
\mathcal{F}_R(P, Relation, f) &= \mathcal{U}df \\
\mathcal{F}s_R(P, r) &= \{f | \mathcal{F}_R(P, r, f) \neq \mathcal{U}df\} \\
\mathcal{MD}_R(P, r, m) &= \begin{cases} t\ m(t_1\ x_1, \ldots, t_n\ x_n)\ \emptyset\ \{e\} & if\ rBody = \ldots t\ m(t_1\ x_1, \ldots, t_n\ x_n)\ \emptyset\ \{e\} \ldots \\ \mathcal{U}df \end{cases} \\
\mathcal{M}_R(P, r, m) &= \begin{cases} \mathcal{MD}_R(P, r, m) & if\ \mathcal{MD}_R(P, r, m) \neq \mathcal{U}df \\ \mathcal{M}(P, r^{'}, m) & otherwise \end{cases} \\
\mathcal{M}_R(P, Relation, m) &= \mathcal{U}df
\end{aligned}
$$

Figure 4.7: Field and Method Lookup Functions for Relationships

The rules in Figure 4.7 allow us to lookup fields and methods in a relationship. The structure of lookup functions are same as the ones for classes but we require different functions because relationships have different structure from classes.

In $\mathcal{F}ickle_{MR}$, we allow external methods i.e methods to be declared outside classes as shown in Figure 4.1. The external method lookup assumes global knowledge of a program `P` and looks for the method `m` declared outside classes. There may be more than one method `m` for example in the case of a multimethod as expressed in Figure 4.3 therefore $\mathcal{M}_{ext}(P, m)$ returns a set of external methods.
We require another lookup function for reclassification directives. In Figure 4.9, we define the function $\mathcal{DR}(P, c, c^{''})$ for a program P, class identifier c and $c^{''}$ where $\mathcal{C}(P, c) = class\ c\ extends\ c^{'}\{cBody\}$. The function $\mathcal{DR}_R(P, r, r^{''})$ is the equivalent function for relationships for a program P, relationship indentifier r and $r^{''}$ where $\mathcal{R}(P, r) = relationship\ r\ extends\ r^{'}\{rBody\}$. The function $\mathcal{DR}(P, c, c^{''})$ looks for the

$$\mathcal{M}_{ext}(P,m) \;=\; \left\{ \; \tau\ m(\tau_0\ x_0,\ldots,\tau_n\ x_n)\mathit{eff}\{e\} \quad | \quad if\ P = P_1\ \tau\ m(\tau_0\ x_0,\ldots,\tau_n\ x_n)\{e\}\ P_2 \right.$$

Figure 4.8: Method Lookup Function for External Methods

reclassification directive for reclassifying objects of class c to $c^{''}$, in class c or its superclasses. The same concept applies to $\mathcal{DR}_R(P,r,r^{''})$, it looks for reclassification directive for reclassifying relation r to $r^{''}$ in r or its super-relations.

$$\begin{array}{lcl}
\mathcal{DRD}(P,c,c^{''}) & = & \left\{ \begin{array}{ll} e & if\ cBody = \ldots upon\ reclassification\ to\ c^{''}\{e\}\ldots \\ \emptyset & otherwise \end{array} \right. \\
\mathcal{DR}(P,c,c^{''}) & = & \left\{ \begin{array}{ll} \mathcal{DRD}(P,c,c^{''}) & if\ \mathcal{DRD}(P,c,c^{''}) \neq \mathcal{U}df \\ \mathcal{DR}(P,c^{'},c^{''}) & otherwise \end{array} \right. \\
\mathcal{DR}(P,Object,c^{''}) & = & \emptyset \\
 & & \\
\mathcal{DRD}_R(P,r,r^{''}) & = & \left\{ \begin{array}{ll} e & if\ rBody = \ldots upon\ reclassification\ to\ r^{''}\{e\}\ldots \\ \emptyset & otherwise \end{array} \right. \\
\mathcal{DR}_R(P,r,r^{''}) & = & \left\{ \begin{array}{ll} \mathcal{DRD}_R(P,r,r^{''}) & if\ \mathcal{DRD}(P,r,r^{''}) \neq \mathcal{U}df \\ \mathcal{DR}_R(P,r^{'},r^{''}) & otherwise \end{array} \right. \\
\mathcal{DR}_R(P,Relation,r^{''}) & = & \emptyset
\end{array}$$

Figure 4.9: Directive-Lookup Function

The final lookup function is for finding the `root` class or root relation. This function is used to ensure that while reclassification, the class/relation being reclassified and the class/relation that they are being reclassified to have the same root class. This in turn ensures that the two entities are allowed to reclassify to each other. For a program P and class identifier c such that $\mathcal{C}(P,c) = [state \mid root]\ class\ c\ extends\ c^{'}\ldots$, and for relationship identifier r such that $\mathcal{C}(P,c) = [state \mid root]\ rel\ r\ extends\ r^{'}\ldots$, we define $\mathcal{R}oot(P,c)$ and $\mathcal{R}oot_R(P,r)$ functions in Figure 4.10.

4.4 Typing Environment

In this section we describe the basic type-system for $\mathcal{F}ickle_{MR}$. In later chapters, we extend the type-system as we introduce more features to the language. The typing environment Γ maps `this` to a class or relationship and a variable `id` to a type t. Environment lookup has the form $\Gamma(id)$ and update has the form $\Gamma[id \mapsto t]$. Environment update is performed in $\mathcal{F}ickle_{||}$when reclassification occurs.

$$
\begin{array}{lcl}
\mathcal{DR}oot(P,c) & = & \begin{cases} c & if\ \mathcal{C}(P,c) = root\ class\ c\ extends\ c^{'}\ldots \\ \mathcal{U}df & otherwise \end{cases} \\
\mathcal{R}oot(P,c) & = & \begin{cases} \mathcal{DR}oot(P,c) & if\ \mathcal{DR}oot(P,c) \neq \mathcal{U}df \\ \mathcal{R}oot(P,c^{'}) & otherwise \end{cases} \\
\mathcal{R}oot(P,Object) & = & \mathcal{U}df \\
 & & \\
\mathcal{DR}oot_R(P,r) & = & \begin{cases} r & if\ \mathcal{R}(P,c) = root\ rel\ r\ extends\ r^{'}\ldots \\ \mathcal{U}df & otherwise \end{cases} \\
\mathcal{R}oot_R(P,r) & = & \begin{cases} \mathcal{DR}oot_R(P,r) & if\ \mathcal{DR}oot_R(P,r) \neq \mathcal{U}df \\ \mathcal{R}oot_R(P,r^{'}) & otherwise \end{cases} \\
\mathcal{R}oot_R(P,Relation) & = & \mathcal{U}df
\end{array}
$$

Figure 4.10: Root Class and Relationship Lookup

$$
\Gamma(id) = \begin{cases} t & if\ id = x \\ c & if\ id = this \\ \mathcal{U}df & otherwise \end{cases}
\qquad
\Gamma[id \mapsto t](id^{'}) = \begin{cases} t & if\ id = id^{'} \\ \Gamma(id^{'}) & otherwise \end{cases}
$$

Figure 4.11: Evironment Lookup and Update

Like in $\mathcal{F}ickle_{||}$, typing an expression e in the context of program P and environment Γ has the form:

$$P, \Gamma \vdash e : t[]\Gamma^{'}[]\phi$$

t is the type returned after type-checking the expression $e, \Gamma^{'}$ is the updated environment resulting from evaluating e and ϕ estimates the reclassification effects from evaluating e.

With $t \sqcup_p t^{'}$ in Figure 4.12 we denote the *least upper bound* of t and $t^{'}$ with respect to $\leq$ in P. The least upper bound of two types is their most immediate common super-type. The least upper bound of two environments Γ and $\Gamma^{'}$ is an environment where the variables common to Γ and $\Gamma^{'}$ are mapped to their least upper bounds. The definition for least upper bound operation has been taken from $\mathcal{F}ickle_{|||}$ [8].

The well-formedness of the environment is given in Figure 4.5. All classes are direct or indirect sub-classes of Object and all relationships are direct or indirect sub-relationships of Relation. A statement of the form $\Gamma \vdash X$ is read as the environment Γ justifies X.

$$t_1 \sqcup_p t_2 \quad = \quad \begin{cases} t & if\ t_1 \leq t\ \ t_2 \leq t\ \ \forall t'(t_1 \leq t'\ and\ t_2 \leq t') \Rightarrow t \leq t' \\ \mathcal{U}df & otherwise \end{cases}$$

$$\Gamma \sqcup_p \Gamma' \quad = \quad \left\{\ id : (t \sqcup_p t')\ \middle|\ \Gamma(id) = t\ and\ \Gamma(id') = t'\ \right\}$$

$$\phi \sqsubseteq \phi' \quad iff \quad \forall c \Downarrow c' \in \phi : \exists d \Downarrow d' \in \phi' \quad c \sqsubseteq d\ and\ c' \sqsubseteq d'$$

$$\begin{aligned} \phi \sqcup_p \phi' \quad = \quad & \mathbf{let}\ \phi_0 = \{c \Downarrow (c' \sqcup_p (\sqcup_p \{d' \mid d \Downarrow d' \in \phi'\ and\ d \sqsubseteq c\})) \mid c \Downarrow c' \in \phi\} \\ & \quad \cup \{d \Downarrow d' (d' \sqcup_p (\sqcup_p \{c' \mid c \Downarrow c' \in \phi\ and\ c \sqsubseteq d\})) \mid d \Downarrow d' \in \phi'\} \\ & \mathbf{in}\ \{c_0 \Downarrow (c_0' \sqcup_p (\sqcup_p \{d_0' \mid d_0 \Downarrow d_0' \in \phi_0\ and\ (c_0' \sqsubseteq d_0\ or\ d_0 \sqsubseteq c_0')\})) \mid c_0 \Downarrow c_0' \in \phi_0\} \end{aligned}$$

Figure 4.12: Least Upper Bound for Effects

Variables in $\mathcal{F}ickle_{||}$can be primitive, class and relationship. `bool` is the only primitive type to keep the semantics simple. In addition to these, there are two new types: *method-type* and *argument-type*. A method-type *MT* for a method declaration $T\ m(T_1\ x_1, \ldots, T_n\ x_n)$ is denoted by $T_1\ \times\ \ldots\ \times\ T_n\ \rightarrow\ T$. The argument-type AT for the same method is $T_1\ \times\ \ldots\ \times\ T_n$. The judgement $\diamondsuit_{MT}$ and $\diamondsuit_{AT}$ is given in Figure 4.13. The function $Args(MT)$ gives the argument-tuple (AT) and function $Res(MT)$ gives the result type of MT.

$$\frac{\begin{array}{l}\Gamma\ \vdash\ T \diamondsuit_t\ T\ =\ void \\ \Gamma\ \vdash\ T_i \diamondsuit_t\ i\ \in\ \{1, \ldots, n\}, n \geq 0\end{array}}{\begin{array}{l}\Gamma\ \vdash\ T_1\ \times\ \ldots\ \times\ T_n \diamondsuit_{AT} \\ \Gamma\ \vdash\ T_1\ \times\ \ldots\ \times\ T_n\ \rightarrow\ T \diamondsuit_{MT}\end{array}} \qquad \begin{array}{l} Args(MT)\ =\ T_1\ \times \ldots \times\ T_n \\ Res(MT)\ =\ T \\ \quad whereMT = T_1\ \times\ \ldots\ \times\ T_n\ \rightarrow\ T \end{array}$$

Figure 4.13: Method and Argument Types

Figure 4.14 gives the subtyping rules. The first rule says that every type is a subtype of itself. The second and third rules say that `null` is a subtype of `T` iff `T` is a subtype of `Object` or `Relation`. The next two rules say that relations are subtypes of `Relation` and objects of `Object`. The next two rules say that a type T is a subtype of T' if T is a subclass or sub-relation of T'. The last rule says that an an argument-type $T_1 \times \ldots \times T_n$ is a subtype of another argument type $T_1' \times \ldots \times T_n'$ if all T_i is a subtype of T_i'.

The $\mathcal{ST}(T)$ function in Figure 4.15 gives the static type of T. If $T = \tau_1 @ \tau_2$ then it returns τ_1 , which is

$$\frac{P \vdash T \diamondsuit_t}{P \vdash T \leq T} \qquad \frac{P \vdash T \diamondsuit_c}{P \vdash T \leq Object} \qquad \frac{P \vdash T \diamondsuit_r}{P \vdash T \leq Relation}$$

$$\frac{P \vdash T \sqsubseteq T'}{P \vdash T \leq T'} \qquad \frac{P \vdash T \sqsubseteq_R T'}{P \vdash T \leq T'} \qquad \frac{P \vdash T_i \leq T_i' : \forall i \in \{1, \ldots, n\}}{P \vdash T_1 \times \ldots \times T_n \leq T_1' \times \ldots \times T_n'}$$

Figure 4.14: The Subtype Relation

$$\mathcal{ST}(T) = \begin{cases} \tau_1 & if\ T = \tau_1 @ \tau_2 \\ T & otherwise \end{cases} \qquad \mathcal{ST}(AT) = \begin{array}{l} \tau_1 \times \ldots \times \tau_n \\ where\ AT = T_1 \times \ldots \times T_n \\ and\ \tau_i = \mathcal{ST}(T_i)\ :\ \forall i \in \{1, \ldots, n\} \end{array}$$

$$\mathcal{DT}(T) = \begin{cases} \tau_2 & if\ T = \tau_1 @ \tau_2 \\ T & otherwise \end{cases} \qquad \mathcal{DT}(AT) = \begin{array}{l} \tau_1 \times \ldots \times \tau_n \\ where\ AT = T_1 \times \ldots \times T_n \\ and\ \tau_i = \mathcal{DT}(T_i)\ :\ \forall i \in \{1, \ldots, n\} \end{array}$$

Figure 4.15: The Static type and Dynamic type Functions

the static part of an @-type, otherwise it returns T itself. If $T = \tau_1 @ \tau_2$ then $\mathcal{DT}(T)$ returns τ_2 the dynamic type of T, otherwise it returns T itself. The last two rules perform the same on argument-types instead of single argument types.

4.4.1 Typing Rules

In this section, we give the typing rules from $\mathcal{F}ickle_{||}$. Figure 4.16 gives the typing rules for conditional statements (COND), field assignment to objects (CLASSFldAss) and relations (RELFldAss), local variable assignment (VarAss), expression sequence (ExpSeq), class and relationship field access (CLASSFldAccess, RELFldAccess), local variable access (ID), new operator (NEW) and booleans (BOOL).

4.5 The Operational Semantics of $\mathcal{F}ickle_{MR}$

Operational semantics describe the execution of expressions. We extend $\mathcal{F}ickle_{||}$by introducing a heap $\mathcal{X}$, for storing dynamically allocated objects. For $\mathcal{F}ickle_{MR}$, execution takes place in the context of a given program(P), and rewrites the tuples of expressions (e), stacks (σ), and heaps ($\mathcal{X}$) into pairs of results (res) and heaps. Therefore the signature of the rewriting relation $\rightsquigarrow$ is:

$$\rightsquigarrow: P \longrightarrow e \times \sigma \times \mathcal{X} \longrightarrow res \times \mathcal{X}$$

The stack(σ) is a tuple which maps *this* to an address and variable *x* to a value *val*. The heap($\mathcal{X}$) maps

(COND)
$$\frac{P,\Gamma \vdash e : bool[]\Gamma^{'}[]\phi \quad P,\Gamma^{'} \vdash e_1 : \tau_1[]\Gamma^{''}[]\phi^{'} \quad P,\Gamma^{''} \vdash e_2 : \tau_2[]\Gamma^{'''}[]\phi^{''}}{P,\Gamma \vdash \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 : \tau_1 \sqcup_p \tau_2[]\Gamma^{''} \sqcup_p \Gamma^{'''}[]\phi \sqcup_p \phi^{'} \sqcup_p \phi^{''}}$$

(CLASSFldAss)
$$\frac{P,\Gamma \vdash e : c[]\Gamma^{'}[]\phi \quad P,\Gamma^{'} \vdash e^{'} : \tau[]\Gamma^{''}[]\phi^{'} \quad \mathcal{F}(P,\phi^{'}@_p c,f) = \tau^{'} \quad P \vdash \tau \leq \tau^{'}}{P,\Gamma \vdash e.f := e^{'} : \tau[]\Gamma^{''}[]\phi \sqcup_p \phi^{'}}$$

(RELFldAss)
$$\frac{P,\Gamma \vdash e : r[]\Gamma^{'}[]\phi \quad P,\Gamma^{'} \vdash e^{'} : \tau[]\Gamma^{''}[]\phi^{'} \quad \mathcal{F}_R(P,\phi^{'}@_p r,f) = \tau^{'} \quad P \vdash \tau \leq \tau^{'}}{P,\Gamma \vdash e.f := e^{'} : \tau[]\Gamma^{''}[]\phi \sqcup_p \phi^{'}}$$

(VARAss)
$$\frac{P,\Gamma \vdash e : \tau[]\Gamma^{'}[]\phi \quad \Gamma^{'}(x) = \tau^{'} \quad P \vdash \tau \leq \tau^{'}}{P,\Gamma \vdash x := e : \tau[]\Gamma^{'}[]\phi}$$

(EXPSeq)
$$\frac{P,\Gamma \vdash e_1 : \tau_1[]\Gamma^{'}[]\phi \quad P,\Gamma^{'} \vdash e_2 : \tau_2[]\Gamma^{''}[]\phi^{'}}{P,\Gamma \vdash e_1; e_2[]\Gamma^{''}[]\phi \cup \phi^{'}}$$

(CLASSFldAccess)
$$\frac{P,\Gamma \vdash e : c[]\Gamma^{'}[]\phi \quad \mathcal{F}(P,c,f) = \tau}{P,\Gamma \vdash e.f : \tau[]\Gamma^{'}[]\phi}$$

(RELFldAccess)
$$\frac{P,\Gamma \vdash e : r[]\Gamma^{'}[]\phi \quad \mathcal{F}_R(P,r,f) = \tau}{P,\Gamma \vdash e.f : \tau[]\Gamma^{''}[]\phi}$$

(ID)
$$\frac{P \vdash \Gamma\diamondsuit}{P,\Gamma \vdash id : \Gamma(id)[]\Gamma[]\{\ \}}$$

(NEW)
$$\frac{P,\Gamma \vdash c\diamondsuit_c}{P,\Gamma \vdash \texttt{new } c : c[]\Gamma[]\{\ \}}$$

(BOOL)
$$\frac{}{P,\Gamma \vdash true : bool[]\Gamma[]\{\ \} \quad P,\Gamma \vdash false : bool[]\Gamma[]\{\ \}}$$

(NEW)
$$\frac{P,\Gamma \vdash c\diamondsuit_c}{P,\Gamma \vdash \texttt{new } c : c[]\Gamma[]\{\ \}}$$

Figure 4.16: Typing Rules For $\mathcal{F}ickle_{MR}$

$$\frac{\begin{array}{l} cBody = class\ c\ extends\ c^{'}\{mBody_1, \ldots, mBody_r, reclDir_1, \ldots, reclDir_v\} \\ mBody_i = \tau_i m_i(\tau_{0_i}\ x_{0_i}, \ldots, \tau_{n_i}\ x_{n_i})\{stmts_i\} \quad \forall i \in \{1, \ldots, r\} \\ P, \Gamma, this : c \vdash mBody_i : \tau_{0_i} \times \ldots \tau_{n_i} \rightarrow \tau_i \\ \forall j \in \{0, \ldots, n\} : \ (\tau_{j_i} = \tau^{'}@\tau^{''}) \Rightarrow \exists c^{''} : c \sqsubseteq c^{''} \\ \quad and\ \mathcal{M}(P, c^{''}, m_i) = \tau^{''}\ m_i(\tau_0^{''}\ x_0^{''}, \ldots, \tau_n^{''}\ x_0^{''})\{stmts^{''}\} \\ \quad and\ \mathcal{ST}(\tau^{'}) \leq \tau_j^{''} \\ reclDir_k = upon\ reclassifcation\ to\ c_k\{e_k\} \quad \forall k \in \{1, \ldots, v\} \\ \mathcal{R}oot(P, c) = \mathcal{R}oot(P, c_k) \end{array}}{P, \Gamma \vdash cBody : c}$$

$$\frac{\begin{array}{l} P = cBody_1 \ldots cBody_q\ rBody_1 \ldots rBody_r\ methBody_1 \ldots methBody_s \\ cBody_i = class\ c_i \ldots \quad \forall i \in \{1, \ldots, q\} \\ P, \Gamma \vdash cBody_i : c_i \quad foralli \in \{1, \ldots, q\} \\ rBody_j = relationshipr_j \ldots \quad \forall j \in \{1, \ldots, r\} \\ P, \Gamma \vdash rBody_j : r_j \quad \forall j \in \{1, \ldots, r\} \\ methBody_k = \tau_k\ m_k(\tau_{0_k}\ x_{0_k}, \ldots, \tau_{n_k}\ x_{n_k})\{stmts_k\} \quad \forall k \in \{0, \ldots, n\} \\ P, \Gamma \vdash methBody_k : \tau_{0_k} \times \ldots \times \tau_{n_k} \rightarrow \tau_k \\ \forall h \in \{0, \ldots, n\} : \ (\tau_{h_k} = \tau^{'}@\tau^{''}) \Rightarrow \exists methBody_u \quad u \in \{1, \ldots, s\} \\ \quad where\ mBody_u = \tau^{''}\ m_i(\tau_0^{''}\ x_0^{''}, \ldots, \tau_n^{''}\ x_0^{''})\{stmts^{''}\} \\ \quad and\ \mathcal{ST}(\tau^{'}) \leq \tau_j^{''} \end{array}}{\Gamma \vdash P_{\diamondsuit}}$$

Figure 4.17: Typechecking Classes, Relationships and Well-formedness of Program in Γ

addresses to objects allocated via the *new* operator. The result *res* is a value (*true*, *false* or *null*), an address or an exception *dev*.

$$
\begin{array}{lcl}
stack & ::= & (\{this\} \longrightarrow addr) \ \cup \ (var \longrightarrow val) \\
heap & ::= & addr \ \longrightarrow \ \{object \cup relation\} \\
val & ::= & \{true, false, null\} \ \cup \ addr \\
object & ::= & \{[[\ c \mid f_1 \ : \ v_1, \ldots, f_r \ : \ v_r \ ]] \\
 & & \quad f_1, \ldots, f_r, c \ identifiers, \ \ v_1, \ldots, v_r \ \in \ val\} \\
relation & ::= & [[\ r \mid from \ : \ addr_1, to \ : \ addr_2, f_1 \ : \ v_1, \ldots, f_r \ : \ v_r \ ]] \\
 & ::= & \quad f_1, \ldots, f_r, c \ identifiers, \ \ v_1, \ldots, v_r \ \in \ val\} \\
addr & ::= & \{\ \iota_i \mid i \ is \ a \ natural \ number\} \\
dev & ::= & \{nullPntrExc, \ stuckErr\} \\
res & ::= & dev \ \cup \ val
\end{array}
$$

Figure 4.18: Stack and Heap Structure of $\mathcal{F}ickle_{MR}$

We do not allow pointer to pointers (pointers are implicit), but this restriction can be relaxed by having heap map addresses to addresses. The deviations (dev) are cause by abnormal termination of program which may be the result of dereferencing a null-pointer or because of the evaluation of an expression being stuck. We will later prove the soundness of our type system which ensures that well-typed expressions are never stuck, although they may throw a *null-pointer* exception.

4.5.1 Updates

For object $o = [[c \ || \ f_1 : v_1, \ldots, f_l : v_l, \ldots, f_r : v_r]]$, heap $\mathcal{X}$, value v, address ι, field identifier f, we define:

$$
\begin{array}{lcl}
o(f) & = & \begin{cases} v_l & if \ f = f_l \ for \ some \ l \ \in \ 1, \ldots, r \\ \mathcal{U}df & otherwise \end{cases} \\
o[f \mapsto v] & = & [[c \ || \ f_1 : v_1, \ldots, f_l : v, \ldots, f_r : v_r]] \ if f_l = f \ for \ some \ l \ \in \ 1, \ldots, r \\
\mathcal{X}[\iota \mapsto o] & & gives \ a \ new \ heap, \ say \ \mathcal{X}^{'} \equiv \mathcal{X}[\iota \mapsto o], so \ that \ \mathcal{X}^{'}(\iota) = o, and \ \mathcal{X}^{'}(\iota^{'}) = \mathcal{X}(\iota^{'}) \ if \ \iota^{'} \neq \iota.
\end{array}
$$

(IFTrue)
$$\frac{e, \sigma, \chi \rightsquigarrow_p true, \chi' \quad e_1, \sigma, \chi' \rightsquigarrow_p val, \chi''}{if\ e\ then\ e_1\ else\ e_2, \sigma, \chi \rightsquigarrow_p val, \chi''}$$

(IFFalse)
$$\frac{e, \sigma, \chi \rightsquigarrow_p false, \chi' \quad e_2, \sigma, \chi' \rightsquigarrow_p val, \chi''}{if\ e\ then\ e_1\ else\ e_2, \sigma, \chi \rightsquigarrow_p val, \chi''}$$

(VARAss)
$$\frac{e, \sigma, \chi \rightsquigarrow_p val, \chi' \quad \sigma[x \mapsto val]}{x := e, \sigma, \chi \rightsquigarrow_p val, \chi'}$$

(CLASSFldAss)
$$\frac{e, \sigma, \chi \rightsquigarrow_p \iota, \chi' \quad e', \sigma, \chi' \rightsquigarrow_p val, \chi'' \quad \chi''(\iota)(f) \neq \mathcal{U}df \quad \chi''' = \chi''[\iota \mapsto \chi''(\iota)[f \mapsto val]]}{e.f := e', \sigma, \chi \rightsquigarrow_p val, \chi'''}$$

(CLASSFldAccess)
$$\frac{e, \sigma, \chi \rightsquigarrow_p \iota, \chi' \quad \chi'(\iota)(f) = val}{e.f, \sigma, \chi \rightsquigarrow_p val, \chi'}$$

(VAR)
$$\frac{\sigma(id) = val}{id, \sigma, \chi \rightsquigarrow_p val, \chi}$$

(EXPSeq)
$$\frac{e_1, \sigma, \chi \rightsquigarrow_p val', \chi' \quad e_2, \sigma, \chi' \rightsquigarrow_p val'', \chi''}{e_1; e_2, \sigma, \chi \rightsquigarrow_p val'', \chi''}$$

(VAL)
$$\frac{}{val, \sigma, \chi \rightsquigarrow_p val, \chi}$$

(THIS)
$$\frac{\sigma = (\iota, v_1, \ldots, v_n)}{this, \sigma, \chi \rightsquigarrow_p \iota, \chi}$$

(NEW)
$$\frac{\iota\ is\ new\ in\ \chi \quad \mathcal{F}s(P, c) = \{f_1, \ldots, f_n\} \quad v_i\ initial\ value\ for\ \mathcal{F}(P, c, f_i)\ (\forall i \in \{1, \ldots, n\} \quad \chi' = \chi[\iota \mapsto [[\ c\ \mid\ f_1 : v_1, \ldots, f_n : v_n\ ]]\ ]}{new\ c, \sigma, \chi \rightsquigarrow_p val, \chi'}$$

Figure 4.19: Operational Semantics Rules

Chapter 5

Multiple Dispatch: Concept & Design

In a typed object-oriented language, method selection is usually carried out at run-time, due to polymorphism. Languages like Smalltalk, Java and C++ support *single dispatching*, in which the method to be executed is selected by the dynamic type of the *receiver* object and the static types of the parameters; completely ignoring the dynamic types of the method arguments. This restricts us from expressing 'change' in the real world. For example, in Figure 1.1, if we coerce a 2-D point `p2` to a 3-D point `p1` and wish to compare it to another 3-D point, languages mentioned above still choose the 2-D `compare` method: it still treats the dynamically typed 3-D point `p2` on the basis of its static type. Multiple dispatch allows us to be more flexible when modelling these types of changes in the real world. There are cumbersome ways to implement multiple dispatch in a single dispatch language: but the length of code is considerably longer and it imposes unnecessary burden on the programmer to carefully type-case all the argument types.

5.1 Points Example for Multimethods

The program in Figure 5.1, is an extension of the previous Points example (in Java) by adding a `Point4D` class and adapting the code to perform the desired operations. The hierarcy of classes has `Point2D`, `Point3D` and `Point4D` (with *time* as fourth dimension). We assume that two points are equal if the coordinates common to them have the same value. We deliberately extended the previous Points example with `Point4D` class to demonstrate that each time we add a sub-class to the hierarchy, in order for that sub-class to override the `equals` method, it will require one more implementation of `equals` method with argument of the immediate super-type. As we mentioned earlier, in single dispatch, the method selection is performed on the dynamic type of the receiver object and the static type of the arguments. For class `Point4D` there are two classes whose instances may have a different static type but a `Point4D` dynamic type i.e. `Point2D` and `Point3D`(its super-types). Class `Point4D` therefore requires two implementations of `equals` methods with `Point2D` and `Point3D` types as arguments, and one implementation for the efficient `equals` method on `Point4D` type. In general a `PointND` class would require $N-1$ implementations of `equals` methods.

```
public class Point2D{
        public int x,y;

        public Point2D(int x, int y){this.x=x;this.y=y;}
        public boolean equals(Point2D p1){return this.x==p1.x && this.y==p1.y;}
}

public class Point3D extends Point2D{
        public int z;

        public Point3D(int x,int y,int z){super(x,y);this.z=z;}
        public boolean equals(Point2D p1){
                if (p1 instanceof Point3D) {return equals((Point3D)p1);}
                else return super.equals(p1);
        }
        public boolean equals(Point3D p1) {
                return this.x==p1.x && this.y==p1.y && this.z==p1.z;
        }
}

public class Point4D extends Point3D{
        public int time;

        public Point4D(int x,int y, int z,int time){super(x,y,z);this.time=time;}
        public boolean equals(Point2D p1){
                if(p1 instanceof Point4D){return equals((Point4D)p1);}
                else return super.equals(p1);
        }
        public boolean equals(Point3D p1){
                        if(p1 instanceof Point4D){return equals((Point4D)p1);}
                        else return super.equals(p1);
        }
        public boolean equals(Point4D p1){
                return this.x==p1.x && this.y==p1.y && this.z==p1.z && this.time==p1.time;
        }
}

public class Test{
        public static void main(String param[]){
                Point4D p4d=new Point4D(1,2,3,4);
                Point3D p3d=new Point3D(1,2,3);
                Point2D p2d=new Point2D(1,2);

                System.out.println(p4d.equals(p3d));//should print true
                System.out.println(p4d.equals(p2d));//should print true

                p2d=new Point4D(1,2,3,5);
                System.out.println(p4d.equals(p2d));//should print false

                p3d=new Point4D(1,2,3,4);
                System.out.println(p4d.equals(p3d));//should print true
        }
}
```

Figure 5.1: A Java Code For Points Example: Simulating Multiple Dispatch using Single Dispatch

```
public class Point2D{
        public int x,y;

        public Point2D(int x, int y) {this.x=x;this.y=y;}
        public boolean equals(Point2D p1) {return this.x==p1.x && this.y==p1.y;}
}

public class Point3D extends Point2D{
        public int z;

        public Point3D(int x,int y,int z) {super(x,y);this.z=z;}

        public boolean equals(Point2D@Point3D p1){
                return this.x==p1.x && this.y==p1.y && this.z==p1.z;
        }
}

public class Point4D extends Point3D{
        public int time;

        public Point4D(int x,int y, int z,int time) {super(x,y,z);this.time=time;}

        public boolean equals(Point2D@Point4D p1){
                return this.x==p1.x && this.y==p1.y && this.z==p1.z && this.time==p1.time;
        }
}
```

Figure 5.2: A MultiJava Code Equivalent to standard Java code in 5.1

This would become extremely tedious and error-prone when modelling large-scale systems, especially when we wish to perform multiple dispatch on more than one method argument. In some sense, the subclasses filter the method each time a generic function invocation occurs. This could have serious effects on the run-time efficiency of the program. To sum it all, single-dispatching limits expressiveness, it forces us to write longer code, imposes the burden of carefully considering each argument type-specific behaviour on the programmer and increases the run-time complexity of the program. We require a mechanism that offers more expressiveness to model the 'change' in the real world. For example Mark is a student, so he does not pay tax, where tax is calculated by the Tax Office. After graduating Mark becomes an Employee, but the Tax Office still calculates his tax according to his student status (static-type). Multiple dispatch offers a natural way to define operations in the real world where entities may change.

In Figure 5.2, we have a MultiJava code which is equivalent to the standard Java code in Figure 5.1. MultiJava is an extension of Java, which supports multiple dispatch and *open classes*[6]. In this program the only deviation from standard Java is the '@' keyword. An expression of the form `Point2D@Point3D p1` in the method arguments performs multiple dispatch on the argument `p1`. In this approach each Point

need only define their specialized `equals` method. The dynamic dispatch is handled by the environment at run-time. This example demonstrate how concise and intuitive the relevant code becomes. The programmer need not worry about type-casing all possible combinations of arguments; every time a new Point is added to the class hierarchy, it only needs to define its specialzed `equals` method i.e. comparing its own types. Multimethods offer ease of extending existing source code, thus also supporting the concept of open classes.

5.2 Type-checking Method Calls in Single-Dispatch Languages

Before proceeding onto the formal semantics for muliple dispatch in $\mathcal{F}ickle_{MR}$, it is worth describing how method calls are typed in single dispatching class-based languages. For the sake of simplicity we ignore *overloading* in classes.

$$\frac{\begin{array}{c} P,\Gamma \vdash e_0 : c \\ M(P,c,m) = \tau\ m(\tau_1\ x_1,\ldots,\tau_n\ x_n)\mathit{eff}\{e\} \\ P,\Gamma \vdash e_i : \tau_i^{'} \\ P,\Gamma \vdash \tau_i^{'} \leq \tau_i\ \forall i \in \{1,\ldots,n\} \end{array}}{P,\Gamma \vdash e_0.m(e_1,\ldots,e_n) : \tau}\ \text{meth-calls} \tag{5.1}$$

The compile-time type checking ensures that there is one method applicable for the method call $e_0.m(e_1,\ldots,e_n)$. To do so, the static type of the expression e_0 is consulted with the environment Γ which would be a class c. The method m is then searched for in class c or its super-classes with the constraint that the actual parameters being a pointwise super-type of its formal parameters, as specified in the rule 5.1

Dynamic-dispatch refers to specialization of a method's behaviour in the subclasses. Overriding a method requires the static types of the method arguments to be the same. During run-time, the method lookup function then starts upward lookup from the *dynamic* class of the receiver. There may be more than one *applicable method* for a method call but the most specific method is selected. Drossopoulou et. al give a type-checking rule for dynamic dispatch in Java [12]. To enable dynamic dispatch, the *argument tuple* of the most-specific method from compile time is annotated with its corresponding method call, as described by the translation from $Java_s$ to $Java_{se}$. This information is later used by the operational semantics to perform dynamic dispatch.

5.3 The @-Type in Multimethods

The @-operator for multiple dispatch was introduced by Chambers et al in [5]. As expressed in Figure 4.1 and as used in Figure 5.2, formal parameters can have types of the form $\tau_1@\tau_2$. The type preceding the @ symbol is the static type of the argument while the type following the @ symbol is the dynamic type of the argument, also known as *explicit specializer* of the formal. Giving separate types to a method argument is the key to multiple dispatch.

In a @-type of the form $\tau_1@\tau_2$, τ_1 is the type that the compiler sees i.e the static type. Multimethods are declared by having the same static type in the method arguments. For example, in Figure 5.2, all the `equals` method in the multimethod has the same static type i.e. Point2D. The static type of an @-type can be retrieved by $\mathcal{ST}$ function described in Figure 4.15.

τ_2 is the dynamic type and thus does not play any role in method selection until run-time. During run-time, τ_2 becomes the type of the method argument. The dynamic type of an @-type can be retrieved by $\mathcal{ST}$ function described in Figure 4.15.
We would like multiple-dispatch in $\mathcal{F}ickle_{MR}$ to unify all types of method lookup including dynamic dispatch and multiple dispatch. As discussed briefly in the previous section, dynamic dispatch works if the static types of the overriding methods are the same or pointwise supertypes of the overridden methods. We approach a solution to multiple dispatch along the lines of dynamic dispatch so that we do not effect the semantics of dynamic dispatch. The @-operator lets us do exactly that.

5.4 Modular Multimethods in $\mathcal{F}ickle_{MR}$

Object-Oriented programming allows development of programs in increments by code reuse facilities such as inheritance. Multimethods would be very limited if we had to modify the source code of existing classes in order to add new inter-class behaviours. Additionally, a method describing some behaviour between two or more classes may not necessarily belong to either class. For example, consider a multimethod *draw*, with the methods {*draw(Rectangle,Canvas),draw(Circle,ColoredCanvas), ...*} to draw an arbitrary shape on a medium such as canvas, panel etc. Introducing a new type of canvas (GridCanvas) would require changing the source code of classes such as *Rectangle* and *Circle* to add new method *draw(Rectangle,GridCanvas) and draw(Circle,GridCanvas)* respectively. This reinforces the idea of being able to add multi-methods in a modular way, i.e allowing multi/methods to be declared outside classes, giving way to the concept of *open classes*[6]. We differentiate between the external and internal methods calls by using the notation $e.m(e^*)$ to refer to an internal method and $m(e^*)$ for an external method call (see Figure 4.1).

5.4.1 Defining External Multimethods

In $\mathcal{F}ickle_{MR}$, we allow multimethods to be declared outside classes. The complete syntax is given in Figure 4.1. We refer to any methods declared outside classes as *external* and the ones declared inside classes as *internal*. As a principle, any set of methods which define some inter-class behaviours should always be declared as external multimethods. This gives the freedom of adding new classes to the existing class hierarchy and declaring the inter-class behaviours in a modular way. The `equals` multimethod in Figure 5.2 can also be added to the program in the way given in Figure 5.3.

The set of *equals* methods above form the equals multimethod. The method whose arguments is a pointwise supertype of all other methods is called the *top-level* method. To evaluate a *message send* of the form

```
boolean equals(Point2D p1,Point2D p2){ return p1.x==p2.x && p1.y==p2.y; }

boolean equals(Point2D@Point3D p1,Point2D@Point3D){
        return p1.x==p2.x && p1.y==p2.y && p1.z==p2.z;
}
boolean equals(Point2D@Point4D p1, Point2D@Point4D p2){
        return p1.x==p2.x && p1.y==p2.y && p1.z==p2.z && p1.time==p2.time;
}
```

Figure 5.3: External multimethod declaration

$m(e_0, \ldots, e_n)$, we evaluate each expression e_i to a type t_i and then invoke the *most-specific applicable method*. We say a method is *applicable* for a message send, if it has the same name, number of arguments and the formal parameters are pointwise super-types of the actual parameters in the message send. There may be many applicable methods for a particular message send so we pick the unique applicable method whose specializers pointwise inherit from the specializers of every applicable method.

We define a function $\mathcal{M}_{ext}(P, m)$ for external method lookup in Figure 4.3. The difference between internal and external methods is that since external methods do not belong to any class in particular they do not have a *this* pointer or a reciever object. External methods also effect the property of *seperate compilation* which is apparent from the external method lookup function. While looking for an external method, the whole program has to be checked. This violates the property of *seperate compilation* which says that if individual compilation-units (classes in Java,C++) are well-typed then at link-time, the whole program will be well-typed. This allows individual compilation units to be compiled seperately. As a result a programmer can add new compilation units to an existing program without having to recompile the whole program.

5.5 Type-checking Multimethods

It is quite obvious that multimethods is an extremely important feature in a statically-typed object-oriented language. It however comes with a price: static type-checking of multimethods is a challenge. Existing solutions of symmetric multiple dispatch by Chambers et al forces global typechecking of multimethods and perform *most-specific method* lookup at run-time [17, 5, 16]. As a result a *message-ambiguous-error* can occur at run-time. They also impose a greater restriction on method overriding. Various approaches to type-checking multimethods have been considered over the years but all of them compromise on some important feature of language design. The lack of an optimal solution may be the reason why C++ and Java do not offer inbuilt support for multimethods. Our aim is to have type-checking rules that statically ensure that no `message-not-understood` and `message-ambiguous` errors occur at run-time. Ruling out these errors at compile time, ensures that our code is safe to be executed without errors.

We develop upon the two phase type-checking strategy by Chambers, Leavens and Millstein**??**. The type-checking is divided into two phases: *Client-side* and *Implementation-side*. Although we take this approach, we deviate from their type-checking rules to allow for a more flexible multimethod declaration.

5.5.1 Client-Side Typechecking

Client-side checks are local checks on message send expressions. This is a typical check that is performed in conventional languages like Java and C++. Client-side checks ensures that the static types of the arguments and the receiver (if any) in a message-send expression corresponds to a method declaration. There may be more than one applicable method for each message send, for example in the case of a multimethod. In that case, the *most specific method* is selected. In $\mathcal{F}ickle_{MR}$ we do not worry about importing modules or compilation units as we consider them to be globally available (this helps us keep the semantics simple). The syntax for internal message send is different from the syntax for external message send. For this reason we have separate type-checking rules for these two kinds of message-send expressions.

Typechecking for Internal Methods

Internal methods i.e. methods declared inside classes, are invoked by message-send expressions of the form $e.m(e_0, \ldots, e_n)$ where e corresponds to the *receiver* type. We impose the restriction that all method calls to internal methods must have an explicit receiver object. This restriction helps us distinguish between internal and external method calls. We develop our type-system from that given by Drossopoulou et. al[12, 13] which gives the semantics in presence of method overloading. They annotate the method calls with the most-specific-method from compile-time. The run-time system simply peforms dynamic dispatch on the most-specific-method from compile-time. In our approach we do not allow method overloading through the *unique judgement rule* in Figure 4.2. Figure 5.4 gives the client-side type-rule which selects the most-specific method for the message-send $e.m(e_0, \ldots, e_n)$.

$$\frac{\begin{array}{l} P, \Gamma \vdash e : c[]\Gamma_0[]\phi_0 \\ P, \Gamma_i \vdash e_i : \tau_i[]\Gamma_{i+1}[]\phi_{i+1} \quad \forall i \in \{0, \ldots, n\} \\ \mathit{MostSpec}(\Gamma_{n+1}, c, m, \tau_0 \times \ldots \times \tau_n) = \{(c', MT)\} \end{array}}{P, \Gamma \vdash e.m(e_0, \ldots, e_n) : Res(MT)[]\Gamma_{n+1}[]\{\phi_0 \cup \ldots \cup \phi_{n+1}\}}$$

Figure 5.4: Client-side Type-check for Internal Method Calls

The *ApplMeths* function in Figure 5.5 takes inputs environment Γ, class of receiver object c, name of method m, argument-type of the method call AT and returns a set of tuples, of class of the method and method signatures which are `applicable` for the method call. A method is applicable for a method call if it has the same name, number of arguments and if the argument-types of the message-call are subtypes of the argument-types of the method itself. Please note that we use the function $\mathcal{ST}$ to get the static type of the

formal paramets of methods. As mentioned in Section 5.3, for an @-type, the compiler only sees its static type. There may be more than one *applicable* methods, for example, in a call to a multimethod. Possible errors are discussed in next section. The binary relation $\leq_{spec}$ is used to give `total` order to the set of *applicable* methods. A tuple of class and method-type $(c^{'}, MT^{'})$ is less specific than (c, MT) if c is a subtype of $c^{'}$ and MT is a subtype of $MT^{'}$. The *MostSpec* function uses the ordering $\leq_{spec}$ to find the *most specific* (c, MT) tuple if any for the corresponding method call. Since $\leq_{spec}$ is reflexive, a tuple (c, MT) is always more-specific than itself. The auxillary functions for *MostSpec* function is given in Figure 5.6.

$$
\begin{array}{lcl}
\mathit{ApplMeths}(\Gamma, c, m, AT) & = & \{(c^{'}, MT^{'}) \mid (c^{'}, MT^{'}) \in \mathcal{M}s(P, c, m) \ \mathit{and}\ AT \leq \mathcal{ST}(\mathit{Args}(MT^{'}))\} \\
\\
(c^{'}, MT^{'}) \leq_{spec} (c, MT) & \mathit{if} & c \leq c^{'} \ \mathit{and}\ \mathcal{ST}(\mathit{Args}(MT)) \leq \mathcal{ST}(\mathit{Args}(MT^{'})) \\
\\
\mathit{MostSpec}(\Gamma, c, m, AT) & = & \{(c^{'}, MT^{'}) \mid (c^{'}, MT^{'}) \in \mathit{ApplMeths}(\Gamma, c, m, AT)\ \mathit{and} \\
 & & (c^{''}, MT^{''}) \leq_{spec} (c^{'}, MT^{'})\ \forall (c^{''}, MT^{''}) \in \mathit{ApplMeths}(\Gamma, c, m, AT)\}
\end{array}
$$

Figure 5.5: *MostSpec* Function for Internal Method-Calls

$$
\begin{array}{lcl}
\mathcal{M}s(P, c, m) & = & \left\{ \begin{array}{ll} (c, \mathit{MethType}(\mathcal{MD}(P, c, m))) & \mathit{if}\ \mathcal{MD}(P, c, m) \neq \phi \\ \phi & \mathit{otherwise} \end{array} \right\} \\
 & & \cup\ \mathcal{M}s(P, c^{'}, m)\ \mathit{where}\ P = P_1\ \mathit{class}\ c\ \mathit{extends}\ c^{'}\{\ldots\}\ P_2 \\
\\
\mathcal{M}s(P, \mathit{Object}, m) & = & \phi \\
\\
\mathit{MethType}(\tau\ m(\tau_0\ x_0, \ldots, \tau_n\ x_n)) & = & \tau_0 \times \ldots \times \tau_n \rightarrow \tau
\end{array}
$$

Figure 5.6: Auxillary functions for *MostSpec* function

Typechecking for External Methods

External methods i.e methods declared outside classes, are invoked by message-send expressions of the form $m(e_0, \ldots, e_n)$. An external method does not have a receiver object, hence does not have a `this` pointer. The client-side type-checking rule for external method is given in Figure 5.7. This rule is the same as the corresponding rule for internal methods, except that it does not have a receiver object.

$$\frac{P, \Gamma_i \vdash e_i : \tau_i[]\Gamma_{i+1}[]\phi_i \;\; \forall i \in \{0, \ldots, n\} \quad MostSpec_{ext}(\Gamma_{n+1}, m, \tau_0 \times \ldots \times \tau_n) = \{MT\}}{P, \Gamma_0 \vdash m(e_0, \ldots, e_n) : Res(MT)[]\Gamma_{n+1}[]\{\phi_0 \cup \ldots \cup \phi_n\}}$$

Figure 5.7: Client-side Type-check for External Method Calls

In Figure 5.8, the *MostSpec* and *ApplMeths* have been modified a little. Since external method don't belong to a class, the tuple (c, MT) needed modification. $ApplMeths_{ext}$ returns the set of `method-types` from those external methods which are eligible for the method call. In this function can use the normal subtype relation $\leq$ to give ordering to the set of method-types returned by the $ApplMeths_{ext}$, since the comparison is between argument-types, unlike class and method-type typle in the equivalent function for internal methods. The $MostSpec_{ext}$ then finds the most-specific method from the ones returned by the $ApplMeths_{ext}$ function. We say $AT \leq AT^{'}$ if all the n types in AT are pointwise sub-types of all the n types in $AT^{'}$ as shown in Figure 4.14. The function $\mathcal{M}s_{ext}(P, m)$ in Figure 5.9 is a slight modification of the external method lookup function $\mathcal{M}_{ext}(P, m)$. $\mathcal{M}s_{ext}(P, m)$ returns the set of external methods as method-types which allows us to write shorter semantics.

$$ApplMeths_{ext}(\Gamma, m, AT) = \{MT^{'} \mid MT^{'} \in \mathcal{M}s_{ext}(P, m) \; and AT \leq \mathcal{ST}(Args(MT^{'}))\}$$

$$MostSpec_{ext}(\Gamma, m, AT) = \{MT^{'} \mid MT^{'} \in ApplMeths_{ext}(\Gamma, m, AT) \; and \; \forall MT^{''} \in ApplMeths_{ext}(\Gamma, m, AT) : \mathcal{ST}(Args(MT^{'})) \leq \mathcal{ST}(Args(MT^{''}))\}$$

Figure 5.8: *MostSpec* Function for External Method-Calls

$$\mathcal{M}s_{ext}(P, m) = \left\{ MethType(\tau \; m(\tau_0 \; x_0, \ldots, \tau_n \; x_n)) \;\middle|\; P = P_1 \; \tau \; m(\tau_0 \; x_0, \ldots, \tau_n \; x_n) \mathit{eff} \{e\} \; P_2 \right\}$$

Figure 5.9: Generic function lookup for External Methods

For external method lookup function to work, we must have a judgement $\vdash P \diamondsuit_{extMeths}$ which says that no two external methods can have the same signature. This is because otherwise it would become undecidable which of those two method to dispatch. The judgement is given in Figure 5.10. Unlike the uniqueness judgement $\Gamma \diamondsuit_u$ for methods where each method name is unique in a class, the judgement $\vdash P \diamondsuit_{extMeths}$

looks at the uniqueness of the whole signature of the method.

$$\frac{\begin{array}{ll}\forall m & : \; (P = P^{'}\ t\ m(t_1\ x_1, \ldots, t_n\ x_n)\{mBody\}\ P^{''}) \Rightarrow \\ & \quad P^{'} \neq P_1^{'}\ t\ m(t_1\ x_1^{'}, \ldots, t_n\ x_n^{'})\{mBody\}\ P_2^{'} \wedge \\ & \quad P^{''} \neq P_1^{''}\ t\ m(t_1\ x_1^{''}, \ldots, t_n\ x_n^{''})\{mBody\}\ P_2^{''}\end{array}}{\vdash\ P \diamondsuit_{extMeths}}$$

Figure 5.10: Unique Judgement for External Methods

5.5.2 Implementation-Side Typechecking

The implementation-side checks deal with method declarations i.e. method signatures and method bodies. They guarantee that each multimethod in the program is *fully* and *unambiguously* implemented. These checks ensure that no *message-ambiguous-error* occurs at run-time, but they make multimethod declaration quite rigid.

For our implementation-side check, we check method declarations to ensure that any arguments of `@`-type is well typed. We also check the method bodies and their return type. The first check is performed on each method declaration M in isolation. For each explicit specializer S of a method M, its associated static type must be a proper supertype of S (expSplz rule).

$$\frac{P\ \vdash\ \tau_2\ \leq\ \tau_1}{P\ \vdash\ \tau_1@\tau_2\ var\ :\ \tau_1@\tau_2}$$

Figure 5.11: Implementation-side Explicit Specializer Check

In Figure 5.12, the second implementation-side check type-checks the method bodies. The result of type-checking $stmts$ is the type of return expression, which we assume to be the last statement in a method body. The type of the return expression must be a sub-type of the return type in the method signature.

$$\frac{\begin{array}{l}mBody = \tau\ m(\tau_0\ x_0, \ldots, \tau_n\ x_n)\{stmts\} \\ P, \tau_0\ x_0, \ldots, \tau_n\ x_n \vdash stmts : \tau^{'} \\ P \vdash \tau^{'} \leq \tau\end{array}}{P \vdash mBody : \tau_0 \times \ldots \times \tau_n \rightarrow \tau}$$

Figure 5.12: Implementation-side Method Body Type-check

The next rule is required to make sure that there is a `top` method for each multimethod. A top method is one whose arguments are the supertypes of the arguments in all other methods belonging to a multimethod.

Although this implementation-side check is not mentioned in any of the papers, we believe this is essential inorder to eliminate possibility of *message-not-understood* error. This error arises if a method call does not correspond to any method declaration. For example, consider the program 5.13, At compile time, the compiler performs method lookup for message-send expression `b.compare(a)`. The compiler finds the `compare(A@B arg){...}` method in class `B` which is applicable for this method call because the static type `arg` is `A`, so no type-error occurs. At run-time the dynamic type of `arg` becomes `B` so this method is no longer applicable for the same message-send. This causes a *message-not-understood* error at run-time.

```
class A {  }

class B extends A{ ... boolean compare(A@B arg){ ... } ... }

class Test{
        A a = new A();
        B b = new B();
        b.compare(a); //message-notunderstood-error at run-time
}
```

Figure 5.13: Message-Not-Understood-Error in Multimethods

This error can be prevented by ensuring that class `A` has a method declaration `boolean compare(A arg){ ... }` which becomes the `top` method of the `compare` multimethod. This is expressed formally by the rule in Figure 5.14. This check is used while typechecking class definitions given in Figure 4.17.

$$\frac{\begin{array}{ll}\exists c' & P = \ldots\ class\ c'\{\ldots \tau'\ m(\tau_0'\, x_0, \ldots, \tau_n'\, x_n)\{\ldots\}\ldots\} \\ & and\ c \sqsubseteq c' \\ & and\ \tau_0' = \mathcal{ST}(\tau_0), \ldots, \tau_n' = \mathcal{ST}(\tau_n)\end{array}}{P, \Gamma \vdash class\ c\ \{\ldots \tau\ m(\tau_0, \ldots, \tau_n)\ldots\} : c}$$

Figure 5.14: Implementation-side Top Method Check

The final implementation-side check is on all the methods declarations in a multimethod. It ensures that a multimethod is fully implemented. This check however forces global type-checking i.e. typechecking of all compilation units. As discussed in Section 2.3.1, this compromises the much desired property of *separate compilation*. In $\mathcal{F}ickle_{||}$, we assume global knowledge of programs and thus do not have an import statement. For this reason we do not worry about separate compilation but talk about in Chapter 8 as further work.

There are two ways to ensure that multimethods declarations are unambiguous. The *first* solution is to

ensure that a multimethod is *fully* implemented. By *fully*, we mean that there is a method for each combination of the dynamic types of arguments. For example, consider the multijava code in Figure 5.2. Let us assume that the `equals` was declared as external multimethod. The method `equals` takes two `Point2D` arguments. Since `Point2D` has two subclasses `Point3D` and `Point4D`, either of these three types can occur at both of the two argument positions. As a result there must be 2^3 `equals` methods with all the combinations of argument types: `equals(Point,Point)`,...,`equals(Point3D,Point3D)`.

The *second* solution is to force any overriding method to have argument types which are pointwise subtypes of the argument types of the overridden methods. This check is enough on its own to ensure the abscence of *method-ambiguous-error* at run-time.

5.6 Message-Ambiguous-Error

A *message-ambiguous-error* is reported if two or more methods are equally specialized for a particular method call. In order to make programming deterministic, instead of choosing one randomly, the compiler/run-time environment throws an error. For example, in Figure 5.15, we methods *compare(A,B)* and *compare(B,A)*, where B is a subclass of A and a method invokation of the form *compare(b1,b2)* with b1, b2 instances of class B.

```
class A { ... boolean compare(B b){ ... } ... }
//equivalent external method: compare(A a,B b)

class B extends A{ ... boolean compare(A a){ ... } ... }
//equivalent external method: compare(B b,A a)

class Test{
        B b1, b2;
        b1.compare(b2); //compare(b1,b2) -> message-ambiguous-error
}
```

Figure 5.15: Message-Ambiguous-Error in Multimethods

In Figure 5.15, the compare message-send in the Test class cannot be resolved to either of the two methods in the compare multimethod because both of them are equally specialized for the argument types of the message-send. In such cases the compiler throws a *message-ambiguous-error*. If no *applicable* method is found, then a *message-not-understood-error* is thrown.

Formally, we can say that a *message-not-understood-error* occurs if the function *ApplMeths* returns an empty set and *message-ambiguous-error* occurs if *ApplMeths* function returns a non-empty set but *MostSpec* returns a an empty set as shown in figure 5.16.

$$\frac{P,\Gamma \vdash e : c[]\Gamma_0[]\phi_0 \quad P,\Gamma_i \vdash e_i : \tau_i[]\Gamma_{i+1}[]\phi_{i+1} \;\; \forall i \in \{0,\ldots,n\} \quad \mathit{ApplMeths}(\Gamma_{n+1}, c, m, \tau_0 \times \ldots \times \tau_n) = \emptyset}{\Gamma \vdash e.m(e_0,\ldots,e_n) : \mathit{message-not-understood-error}}$$

$$\frac{P,\Gamma_i \vdash e_i : \tau_i[]\Gamma_{i+1}[]\phi_i \;\; \forall i \in \{0,\ldots,n\} \quad \mathit{ApplMeths}_{ext}(\Gamma_{n+1}, m, \tau_0 \times \ldots \times \tau_n) = \emptyset}{\Gamma_0 \vdash m(e_0,\ldots,e_n) : \mathit{message-not-understood-error}}$$

$$\frac{P,\Gamma \vdash e : c[]\Gamma_0[]\phi_0 \quad P,\Gamma_i \vdash e_i : \tau_i[]\Gamma_{i+1}[]\phi_{i+1} \;\; \forall i \in \{0,\ldots,n\} \quad \mathit{ApplMeths}(\Gamma_{n+1}, c, m, \tau_0 \times \ldots \times \tau_n) \neq \emptyset \quad \mathit{MostSpec}(\Gamma_{n+1}, c, m, \tau_0 \times \ldots \times \tau_n) = \emptyset}{\Gamma \vdash e.m(e_0,\ldots,e_n) : \mathit{message-ambiguous-error}}$$

$$\frac{P,\Gamma_i \vdash e_i : \tau_i[]\Gamma_{i+1}[]\phi_i \;\; \forall i \in \{0,\ldots,n\} \quad \mathit{ApplMeths}_{ext}(\Gamma_{n+1}, m, \tau_0 \times \ldots \times \tau_n) \neq \emptyset \quad \mathit{MostSpec}_{ext}(\Gamma_{n+1}, m, \tau_0 \times \ldots \times \tau_n) = \emptyset}{\Gamma_0 \vdash m(e_0,\ldots,e_n) : \mathit{message-ambiguous-error}}$$

Figure 5.16: Message Ambiguous and Message-Not-Understood Error

One approach of dealing with *message-ambiguous-error* is to have *asymmetric* multiple dispatch like [2]. Because the arguments are ordered in some way, the method call is always resolved to some method (under the typing constraints). For example, in Figure 5.15, if the method dispatch worked with lexicographic ordering, then it would dispatch the `compare` method from class `B`. There would never be ambiguity. However, we strongly believe in symmetric multiple dispatch and hence we choose to throw a compile-time *message-ambiguous-error*, guaranteeing that this error does not occur during the execution of a $\mathcal{F}ickle_{MR}$ program.

5.6.1 Error Introduced by ⇓ Operator

The main feature of $\mathcal{F}ickle$ is that it allows objects of a particular class to reclassify to another class with certain constraints. We have observed that the reclassification operator ⇓, can cause *message-ambiguous-error*. For example, consider the $\mathcal{F}ickle_{MR}$program in Figure 5.17. We declare `Player` to be the root class for the class hierarchy. This implies that any two subclasses of `Player` can reclassify to each other. The method `strengthen` makes the players stronger by giving them weapons (sword and gun), i.e. reclassifies a `WeakPrince` to `StrongPrince` by giving it a sword (`SwordedPrince`) and `SwordedPrince` to a `GunnerPrince` by giving it a gun. The method `fight` is a binary multimethod which takes two `Player` objects and returns a `boolean`.

The method `test1` has no reclassification operation. At compile-time the list of applicable methods for `p1.fight(p2)` is {Player.fight(Player),WeakPlayer.fight(Player@WeakPlayer)}. The most-specific method is WeakPlayer.fight(Player@WeakPlayer) at compile-time and run-time. On the other hand, in the method `test2`, p1 is reclassified from WeakPrince to SwordedPrince. The applicable methods now are {Player.fight(Player), StrongPrince.fight(Player@SwordedPrince), SwordedPrince.fight(Player@StrongPrince)}. The most specific method at compile-time is SwordedPrince.fight(Player@StrongPrince) but at run-time there will be a method-ambiguous-error because StrongPrince.fight(Player@SwordedPrince) and SwordedPrince.fight(Player@StrongPrince) are equally specialized for the method call (looking at the dynamic type of the method arguments).

The implementation-side typechecking strategies discussed in Section 5.5.2 are capable of eliminating ambiguities at run-time, hence any well-typed $\mathcal{F}ickle_{MR}$ program is free of this error.

5.7 Operational Semantics for Multiple Dispatch

Operational semantics define the run-time evaluation of expressions. In this section we define the operational semantics and the algorithm for multiple-dispatch in $\mathcal{F}ickle_{MR}$. To perform multiple-dispatch, we need to select the most specialized method based on the dynamic type of the actual parameters. This requires atleast some type-checking at run-time. Although this incurs some run-time complexity, multiple-dispatch gives substantial flexibility to a language and is certainly worth the extra overhead. Through our type-system, we aim to minimize errors and the run-time overheads.

```
abstract root class Player extends Object{
  bool brave;

  bool fight(Player p){false };
  void strengthen(){};
}

state class WeakPrince extends Player{
  ...

  bool fight(Player@WeakPrince p){true}
  void strengthen(){(WeakPlayer⇓SwordedPrince)}{this⇓SwordedPrince}
}

state class StrongPrince extends Player{
  ...

  bool fight(Player@SwordedPrince){}{false} //fight(StrongPrince,SwordedPrince)
  void strengthen(){}{}
}

state class SwordedPrince extends StrongPrince{
  ...

  bool fight(Player@StrongPince){ }{true} //fight(SwordedPrince,StrongPrince)
  void strengthen(){(SwordedPrince⇓GunnerPrince)}{this⇓GunnerPrince}
}

state class GunnerPrince extends StrongPrince{
  ...

  bool fight(Player@GunnerPince){ }{false}
  void strengthen(){}{}
}

class Game extends Object{
        void test1(WeakPrince){
                Player p1=new WeakPrince();
                Player p2=new SwordedPrince();
                print(p1.fight(p2)); // false: from Player.fight(Player)
        }

        void test2(){
                Player p1=new WeakPrince();
                Player p2=new SwordedPrince();
                p1⇓SwordedPrince;
                print(p1.fight(p2)); // method ambiguous error
        }
}
```

Figure 5.17: Method-Ambiguous Error in $\mathcal{F}ickle_{MR}$ program, resulting from ⇓ operator

$$\mathcal{MM}_{app}(P, c, m, DTup) = \left\{ \begin{array}{ll} (c, mBody_i) | & mBody_i = \tau\ m(\tau_0\ x_0, \ldots, \tau_n\ x_n)\{e\} \\ & DTup\ =\ \langle c_0, \ldots, c_n \rangle \\ & c_j\ \leq\ \mathcal{DT}(\tau_j)\ \ (\forall j \in \{0 \ldots n\}) \end{array} \right\} \cup \mathcal{MM}_{app}(P, c^{'}, m, \langle c_0, \ldots, c_n \rangle)$$

$$\mathcal{MM}_{app}(P, Object, m, DTup) = \emptyset$$

$$\mathcal{MM}_{ext-app}(P, m, DTup) = \left\{ \begin{array}{ll} mBody | & mBody \in \mathcal{M}_{ext}(P, m) \\ & where\ mBody = \tau\ m(\tau_0\ x_0, \ldots, \tau_n\ x_n)\{e\} \\ & DTup\ =\ \langle c_0, \ldots, c_n \rangle \\ & c_j\ \leq\ \mathcal{DT}(\tau_j)\ \ (\forall j \in \{0 \ldots n\}) \end{array} \right\}$$

Figure 5.18: Applicable Methods Functions for Multiple Dispatch

The funtion $\mathcal{MM}_{app}(P, c, m, DTup)$ in Figure 5.18 returns a set of class $\times$ method-body tuples ***(c,mBody)*** which are applicable for the dynamic types of the arguments of a method call denoted by dynamic-type tuple ***DTup*** and receiver object of class c. A method is applicable for a method call if the dynamic types of its formal parameters are pointwise supertypes of the dynamic types of the arguments of method call. We define $\mathcal{MM}_{app}(P, c, m, DTup)$ for $\mathcal{C}(P, c) = class\ c\ extends\ c^{'}\{mBody_1 \ldots mBody_r\}$. This function is for internal method calls and thus has to check the super-classes of the receiver class c too for method declaration m. $\mathcal{MM}_{ext-app}(P, m, DTup)$ function is for external method calls. It returns those external methods which are applicable for the method call. Since external methods have no classes or class hierarchy, there is no recursion.

The operational semantics rules for method-calls are given in Figure 5.19. The first rule is for internal method calls. It first evaluates the expression e_0 to determine the address of the receiver object. The expressions may update the heap, therefore for each argument expression $e_1, \ldots, e_n$ it evaluates these expressions in the heap updated by their previous expression. To simplify the rule, all expressions evaluate to addresses (i.e. does not consider bool type) and through those addresses the run-time system evaluates the dynamic class of each argument expression. The program, class of the receiver object, method name and a tuple of dynamic classes of the argument expressions is passed to function *MethBody*. The function *MethBody* in Figure 5.20 returns the *most-specific-method* to dispatch at run-time. The stack is updated to contain new variables for the parameters of the selected method whose values are the addresses from evaluation of $e_1, \ldots, e_n$. The variables in the body of the method are replaced with the new variables in stack. The first variable in stack contains the address to the receiver object commonly known as $this$ pointer. The method body is executed in the new stack and heap which results in a value val.

$$\frac{\begin{array}{l} e_i, \sigma, \chi_i \rightsquigarrow_p \iota_i, \chi_{i+1} \ \ (\forall i \in \{0, \ldots, n\} \\ \chi_{n+1}(\iota_i) = [[\ c_i || \ldots]] \ \ (\forall i \in \{0, \ldots, n\} \\ MethBody(P, c_0, AT, \langle c_1, \ldots, c_n \rangle) = m(\tau_1\ x_1, \ldots, \tau_n\ x_n)\{e\}) \\ z_i\ new\ in\ \sigma \\ \sigma^{'} = \sigma[z_i \mapsto \iota_i] \ \ (\forall i \in \{0, \ldots, n\}) \\ e^{'} = e[z_0/this, z_1/x_1, \ldots, z_n/x_n] \\ e^{'}, \sigma^{'}, \chi_{n+1} = val, \chi^{'} \end{array}}{e_0.m(e_1, \ldots, e_n), \sigma, \chi_0 \rightsquigarrow_p val, \chi^{'}}$$

$$\frac{\begin{array}{l} e_i, \sigma, \chi_i \rightsquigarrow_p \iota_i \ \ (\forall i \in \{0, \ldots, n\}) \\ \chi_{n+1}(\iota_i) = [[\ c_i \ || \ldots]] \ \ (\forall i \in \{0, \ldots, n\}) \\ MethBody_{ext}(P, m, \langle c_0, \ldots, c_n \rangle) = m(\tau_0\ x_0, \ldots, \tau_n\ x_n)\{e\}) \\ z_i\ new\ in\ \sigma \\ \sigma^{'} = \sigma[z_i \mapsto \iota_i] \ \ (\forall i \in \{0, \ldots, n\}) \\ e^{'} = e[z_i/x_i] \ \ (\forall i \in \{0, \ldots, n\}) \\ e^{'}, \sigma^{'}, \chi_{n+1} = val, \chi^{'} \end{array}}{m(e_0, \ldots, e_n), \sigma, \chi \rightsquigarrow_p val, \chi^{'}}$$

Figure 5.19: Operational semantic for Method Calls in $\mathcal{F}ickle_{MR}$

The second rule in Figure 5.19 is for external method calls. Since external methods do not have a receiver object, they do not have a $this$ pointer and since the method lookup for external methods is different it uses a variant $MethBody_{ext}$ function to get the most-specific method. The rest is same as the rule for internal methods described above.

5.7.1 Multiple-dispatch Algorithm in $\mathcal{F}ickle_{MR}$

There are two types of algorithms that can be implemented for multiple dispatch: *asymmetric* and *symmetric*. In asymmetric multiple dispatch the right method to be invoked is determined by some type of ordering on the arguments, as discussed in **??**. Although this approach is quite simple to implement, it imposes the burden of ordering the arguments of a method call, on the programmer. It is certainly unacceptable to programmers and would become a drawback to the language's popularity. We believe that the users of $\mathcal{F}ickle_{MR}$ should not have to worry about $\mathcal{F}ickle_{MR}$'s multiple-dispatch algorithm i.e. to order the arguments of their method call in the way that our multiple-dispatch algorithm works. For example, consider a method call of the type $equals(p2D, p3D, p3D)$. Since the asymmetric algorithm treats some arguments more importantly than others in invoking the right method, the programmer is required to have a deep understanding of the multiple dispatch algorithm.

On the other hand, *symmetric multiple dispatch* algorithm treats all arguments in the same way i.e. all arguments are equally important in choosing the right method to dispatch. For this reason we develop a symmetric multiple dispatch algorith for $\mathcal{F}ickle_{MR}$. An interesting observation is that, in `dynamic dispatch`, the `receiver` object is more important than the other arguments. This is asymmetric in some sense, but we will refer to a multiple dispatch algorithm as symmetric when it considers ordering on all but the receiver object. Dynamic dispatch is quite intuitive and appealing, hence we do not consider this as a hindrance to $\mathcal{F}ickle_{MR}$'s usability.

$$
\begin{aligned}
MethBody_{ext}(P, m, DTup) \quad = \quad & \left\{ \begin{array}{l} m(\tau_0\, x_0, \ldots, \tau_n\, x_n)\{e\} \mid \\ \quad m(\tau_0\, x_0, \ldots, \tau_n\, x_n)\{e\} \in \mathcal{MM}_{ext-app}(P, m, DTup)\ and \\ \quad \forall(m(\tau_0^{'}\, x_0^{'}, \ldots, \tau_n^{'}\, x_n^{'})\{e^{'}\}) \in \\ \qquad \{\mathcal{MM}_{ext-app}(P, m, DTup) \backslash m(\tau_0\, x_0, \ldots, \tau_n\, x_n)\{e\}\} \ : \\ \qquad\quad \tau_0 \times \ldots \times \tau_n \leq_{spec-d} \tau_0^{'} \times \ldots \times \tau_n^{'} \end{array} \right. \\
MethBody(P, c, m, DTup) \quad = \quad & \left\{ \begin{array}{l} m(\tau_0\, x_0, \ldots, \tau_n\, x_n)\{e\} \mid \\ \quad (c^{'}, m(\tau_0\, x_0, \ldots, \tau_n\, x_n)\{e\}) \in \mathcal{MM}_{app}(P, c, m, DTup)\ and \\ \quad \forall(c^{''}, m(\tau_0^{'}\, x_0^{'}, \ldots, \tau_n^{'}\, x_n^{'})\{e^{'}\}) \in \\ \qquad \{\mathcal{MM}_{app}(P, c, m, DTup) \backslash (c^{'} m(\tau_0\, x_0, \ldots, \tau_n\, x_n)\{e\}\} \ : \\ \qquad\quad c^{'} \times \tau_0 \times \ldots \times \tau_n \leq_{spec-d} c^{''} \times \tau_0^{'} \times \ldots \times \tau_n^{'} \end{array} \right. \\
\tau_1 \times \ldots \times \tau_n \leq_{spec-d} \tau_1^{'} \times \ldots \times \tau_n^{'} \quad = \quad & iff\ \{1, \ldots, n\} = I \uplus J \\
& \forall i \in I \ \ \tau_i \leq \tau_i^{'} \ \wedge \ \tau_i \neq \tau_i^{'} \\
& |I| > J
\end{aligned}
$$

Figure 5.20: Run-Time Multiple-dispatch Algorithm

We develop a more flexible multiple-dispatch algorithm than the ones proposed by Castagna, Chambers and Millstein [5, 17, 16, 6] and Agrawal et al [2]. Agrawal et al give a graph based method resolution algorithm (see Section 2.4.1) to find the *most-specific* method. This algorithm was the key inspiration but was asymmetric in nature. On the other hand, the algorithm by Castagna, Chambers and Millstein defined a method $m(\tau_0, \ldots, \tau_n)$ to be more specific than $m(\tau_0^{'}, \ldots, \tau_n^{'})$ if $\forall i(i \in \{0, \ldots, n\}) \ : \tau_i \leq \tau_i^{'}$. In our algorithm, we relax this restriction by specifying a method's specificity by the number of argument places it is most specific in. The comparison is given in Section 5.8.

The multiple dispatch algorithm is given in Figure 5.20. The definition of *more-specific-than* ($\leq_{spec-d}$) has changed from the one at compile time ($\leq_{spec}$). The purpose of having static type-checking is to en-

sure that the program is safe to be executed. At compile-time, we wanted to detect and report $method - ambiguous - error$ thats why we described *compile-time-more-specific-than* as being more-specific($\leq$) in `all` argument positions. At run-time we want to make multiple-dispatch as flexible as possible, so we describe the *run-time-more-specific-than* as being more specific in the `number` of argument places. The binary relation $\leq_{spec-d}$ compares two argument-types and returns `true` if the argument-type can be split into two disjoint sets I and J such that the arguments of the first set I are more-specific ($\leq$) than the arguments of set J. The cardinality of I must be strictly greater than that of J. For example, $B \times A \times B$ is more specific than $A \times B \times A$ because it is more-specific in 2 argument places whereas $A \times B \times A$ is less specific than $B \times A \times B$ in only 1 argument place, where $B \sqsubseteq A$.

The function $MethBody_{ext}(P, m, DTup)$ for external methods takes a program, method-name and a tuple of dynamic types of the arguments of the method call. It returns the most-specific method from the set $\mathcal{MM}_{ext-app}(P, m, DTup)$ by comparing the method against rest of the methods in $\mathcal{MM}_{ext-app}(P, m, DTup)$. The comparison is made using the run-time-more-specific-than binary relation $\leq_{spec-d}$ as described above. The function $MethBody(P, c, m, DTup)$ for internal methods takes an extra argument c which is the dynamic class of the receiver object in the method call. Its functionality is exactly the same, with the difference that it compares the class of the methods along with the argument types. This unifies dynamic dispatch with multiple dispatch. More on this topic in the next section.

5.8 Evaluation of Multiple Dispatch in $\mathcal{F}ickle_{MR}$

In this section we evaluate multiple-dispatch in $\mathcal{F}ickle_{MR}$with the symmetric multiple-dispatch in other languages like Tuple[16], Multi-Java **??** and Dubious **??**. These languages have been developed by Todd Millstein, Craig Chambers and Gary T. Leavens and thus have similar type-system and algorithm for multiple-dispatch.

Static Typechecking

The typechecking strategy in $\mathcal{F}ickle_{MR}$ is almost the same. We developed our type-system from the two phase type-checking strategy by Chambers et al, client-side and implementation-side. The only difference is that we explicitly check for the top-method for a multimethod (see Figure 5.14). For example, in Figure 5.22, if method1 was not present the type-system would report a *message-not-understood-error*. To see why the top method is important, assume that `method1` and the above mentioned type-checking rule were absent. Create three Point2D instances with same static-type Point2D as shown in Figure 5.21 and call $compare(p1, p2, p3)$. This will type-check correctly at compile-time because static type of arguments of `method2` and `method3` are Point2D. At run-time however the dynamic-type of these methods are not applicable for this method call, hence there would be a *message-not-understood-error*.

By explicitly checking for top-method, we prevent *message-not-understood-error* at run-time. Although not documented in the papers, we tried such an example in MultiJava to check if it takes the same approach as

```
Point2D p1=new Point2D(1,2,3,2005);
Point2D p2=new Point2D(1,2,3,2006);
Point2D p3=new Point2D(1,2,3,2007);

compare(p1,p2,p3);
```

Figure 5.21: Comparing Multiple-Dispatch Algorithms

$\mathcal{F}ickle_{MR}$. Not surprisingly, MultiJava gave a compile error as we expected.

Multiple Dispatch Algorithm

The best way to compare our multiple-dispatch algorithm with that developed by Chambers et al is by an example. We modify the Points example given before to have external multimethod `compare` which takes three Point arguments and checks if they are equal. We assume that while comparing two Points of different dimensions, it is equal if their common dimensions are the same. The `equals` method simply performs equality check on its three arguments. Point4D extends Point3D and Point3D extends Point2D as before. In the Test method, three Point4D instances are created with static type Point2D. These points have the same value for `x`,`y` and `z` dimension but a different value for `time`. Then we call the compare multimethod twice: once assuming Chambers et al's multiple dispatch algorithm and once assuming $\mathcal{F}ickle_{MR}$ multiple dispatch algorithm.

In Chambers et al's approach, the run-time system calls the `top method` (`method 1`) for the multi-method compare because neither `method2` or `method3` is more specialized than each other. This is because their `run-time-more-specific-than` relation is the same as our `compile-time-more-specific` relation $\leq_{spec}$ in which an argument-type AT is more specific than $AT^{'}$ if *all* the individual types in AT are pointwise more-specific ($\leq$) than the types in $AT^{'}$. In their system, if the run-time system cannot resolve the *most-specific* method, then it calls the top-method i.e. method1. This is obviously not the most correct method implementation because it checks only the x and y components of the Point4D objects. There are better method implementations applicable for the method call. On the other hand, $\mathcal{F}ickle_{MR}$ would select method3 for the same method call because of the $\leq_{spec-d}$ `run-time-more-specific-than` relation. As described in previous section, $\leq_{spec-d}$ looks at the *number* of argument places an argument-type is more specific than others. In this case it clearly is method3, so $\mathcal{F}ickle_{MR}$ dispatches the method call to `method3` and this returns `false` as points `p1` and `p3` differ in their `time` components.

We believe our multiple dispatch algorithm is more flexible and efficient than the ones in Tuple, MultiJava and Dubious. Having a flexible multiple dispatch algorithm gives more flexibility to $\mathcal{F}ickle_{MR}$ it turn.

Unifying Dynamic Dispatch with Multiple Dispatch

```
public class Point2D{   public int x,y; }

public class Point3D extends Point2D{ public int z; }

public class Point4D extends Point3D{ public int time; }
//method 1
bool compare(Point2D p1,Point2D p2,Point2D p3){
        return equals(p1.x,p2.x,p3.x) && equals(p1.y,p2.y,p3.y);
}
//method 2
bool compare(Point2D@Point3D p1, Point2D@Point4D p2, Point2D@Point3D p3){
        return equals(p1.x,p2.x,p3.x) && equals(p1.y,p2.y,p3.y) && equals(p1.z,p2.z,p3.z) ;
}
//method 3
bool compare(Point2D@Point4D p1, Point2D@Point3D p2, Point2D@Point4D p3){
        return equals(p1.x,p2.x,p3.x) && equals(p1.y,p2.y,p3.y) && equals(p1.z,p2.z,p3.z)
                                              && equals(p1.time,p3.time);
}
public class Test{
        public static void main(String param[]){
                Point2D p1=new Point4D(1,2,3,2005);
                Point2D p2=new Point4D(1,2,3,2006);
                Point2D p3=new Point4D(1,2,3,2007);

                //Chambers et al multiple dispatch
                compare(p1,p2,p3); //returns true

                //FickleMR multiple dispatch
                compare(p1,p2,p3): //returns false
        }
}
```

Figure 5.22: Comparing Multiple-Dispatch Algorithms

Chapter 6

Relationships

The inspiration for extending $\mathcal{F}ickle_{MR}$ with relationships comes from database modelling languages like Entity Relationship model and system modelling languages like Unified Modelling Language (UML). In these modelling languages, relationship refers to the context dependent behaviour of an object in relation to other objects in the system. For example, a `Teacher teaches` *many* `Students` where the relationship `teaches` binds the objects `Teacher` and `Student` in a *one-to-many* relationship. Relationships are also referred to as *roles*, so the same example can be stated as: a `Teacher's` *role* is to `teach` *many* `Students`. Please note that we refer to an instance of a relationship as a relation.

Relationships have the following jargon associated with them:

- *Cardinality*: Number of objects involved in the relationship. This can be of form *one-to-one* (1:1), *one-to-many* (1:N), *many-to-one* (N:1), *many-to-many* (M:N), where $M, N \in \{0, 1, 2, \ldots\}$. With a little complexity, we can enforce relationships of cardinality (1:5) or (3:2).
- *Degree*: The number of entities/classes that are involved in the relationship. For example consider a scenario where a supplier supplies parts for a project. We could model this situation by a relationship `SUPP-PROJ-PART` that binds the classes `Supplier`, `Project` and `Part`. Most of the relationships in databases have *degree* 2 and are called *binary* relationship.
- *Participation* may be optional or mandatory.
- *Relation*: An instance of a relationship.

For this project, I have taken a formal approach to describe the semantics of relationships in $\mathcal{F}ickle$. In this chapter, we begin by arguing the need for relationships in object-oriented programming languages. We discuss the advantages and complications of extending $\mathcal{F}ickle_{MR}$ with relationships. We develop formal semantics for relationships on the lines of RelJ calculus by G.Bierman and A.Wren [15].

6.1 Need for First Class Relationships

Relationships allow easy modelling of a software system and would be an extremely useful feature in object-oriented languages. In languages without relationships, programmers are forced to take various complicated approaches for designing systems with relationships. This pollutes the software design, making implementation, debugging and reusability harder.

Since the aim of object-oriented languages is to make it easy for us to model the real world semantic notion directly into code, we aim to extend the expressiveness of $\mathcal{F}ickle_{MR}$ with roles. We stress the need for languages to give first-class support to relationships. We say *first-class* for language constructs that can be stored in variables, pointers, passed as arguments, returned from function etc. In this sense, we allow relationship instances to be stored in variables and the compiler type-checks all the statements in context with the class and relationship types. As a result addresses in the heap can point to objects and relationships.

6.1.1 How $\mathcal{F}ickle$ Fits In With Relationships

We take the idea of reclassification beyond objects to relationships. We will see later that relations are much like objects: they have fields and methods, so it is quite natural for them to reclassify, like objects in $\mathcal{F}ickle$. With the feature of *relationship reclassification,* $\mathcal{F}ickle_{MR}$ becomes very interesting for modelling dynamic real world scenarios where objects change regularly.

In database systems, we often come across scenarios where an object's properties changes and hence its relationships with other objects have to be updated. For example, in Figure 1.1 if a `student` is given a lecturer post, then he becomes a `teacher` and now `teaches` students. In database systems, this is typically achieved by deleting the `student` record and creating a new `teacher` record. Along with the records, the `teaches` relationship needs to be created with every student the teacher teaches. In an object-oriented language like $\mathcal{F}ickle_{MR}$, the same could be achieved by reclassifying student to a teacher and by deleting/adding relationships or simply reclassifying relationships.

Reclassifying objects and relations is a simple and elegant approach for users to express changes in a system. In addition to the simplicity, it is less error-prone. For these reasons, we believe $\mathcal{F}ickle_{MR}$ is more expressive than its counterparts. Clearly, $\mathcal{F}ickle_{MR}$ would offer significant advantages when modelling database design.

6.2 Extended Syntax and Definition

In [15], the authors introduce RelJ, a subset of Java, with support for relationships. RelJ provides means to define relationships between objects and relationships. Relationships are defined by them as a class-like structures containing attributes. Their syntax of relationship looks like:

```
relationship r extends r'
    from n to n' {FieldDecl*}
```

This defines a new relationship with a number of type/field name pair, FieldDecl*. To simplify things, RelJ does not allow relationships to have methods. The relationship is between *n* and *n'* where *n, n'* range over classes and relationships. This provides means for relationship instances to participate in further relationships. This feature is known as *aggregation* in E/R modelling.

In addition to features from RelJ, we allow relationships to have methods($MethDecl^*$) and *reclassification directives* $ReclDir^*$. Reclassification directive is a novel feature which allows user-described operations on reclassification. It will be discussed in more details later. The relationship must optionally state if it is a `root` or `state` class, like classes in $\mathcal{F}ickle_{||}$. This feature allows relationships to reclassify, like classes in $\mathcal{F}ickle_{MR}$ (for more details on `root` and `state` classes, please refer to Chapter 3). Figure **??** gives the complete syntax of $\mathcal{F}ickle_{MR}$.

```
[root|state] relationship r extends r'
    from n to n' {FieldDecl* MethDecl* ReclDir*}
```

Figure 4.1, gives the full syntax of the language. The set of `types` in the language now include `relationships` and a `set` of generics (set¡n¿). The idea of having set of generics was introduced in RelJ to facilitate the processing of relationships. For the Teacher-Student example in Figure 1.2, a statement of the type

- $teacher.Teaches$ will return a set of all the instances of `Student` that are related to teacher by `Teaches` relationship
- $teacher : Teaches$ will return a set of all Teaches relations (relationship instances), i.e. set¡*Teaches*¿
- $teaches.from$ will return the `from` part of the relation, which in this case is of type *Teacher* (assuming `teaches` is an instance of `Teaches`)
- $teaches.to$ will return the `to` part of the relation
- $teacher.Teaches+ = student1$ will create a new instance of `Teaches` relation between `teacher` and `student1`
- $teacher.Teaches- = student1$ will remove `teacher`'s instance of `Teaches` relation with `student1`

6.2.1 A Simple Example of Relationships in $\mathcal{F}ickle_{MR}$

In Figure 6.2, a simple $\mathcal{F}ickle_{MR}$ program is presented to show the basics of relationship declaration and usage. The program has a `root` class `Staff` and two state classes: `TeachingStaff` and `ResearchStaff`. This means that `TeachingStaff` can reclassify to `ResearchStaff` and vice-versa. A UML diagram for this example is given in Figure 6.1.

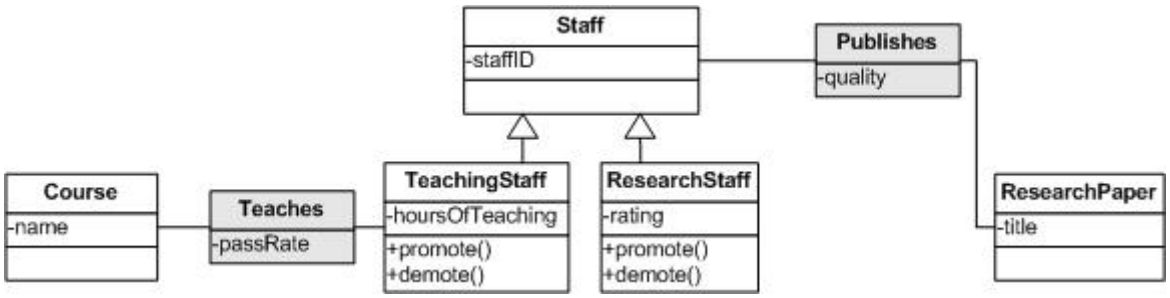

Figure 6.1: UML Diagram for Staff Example in Figure 6.2

We assume that the `teaching hours` of a `TeachingStaff` decreases each time they are `promoted`. Finally they become `ResearchStaff` and are not required to teach. `ResearchStaffs` have `rating` which increases each time they are `promoted` and decreased each time they are `demoted`. When the rating becomes 0, the `ResearchStaff` become `TeachingStaff`. There are two more classes: `Course` and `ResearchPaper`. There are two relationships: `Teaches` from `TeachingStaff` to `Course`, and `Publishes` from `Staff` to `ResearchPaper`. We allow only `TeachingStaff` to teach a course, but both `TeachingStaff` and `ResearchStaff` to `publish ResearchPaper`.

Line 40 declares a `TeachingStaff` *mathew*. Line 42 creates a new course named `programming` (the constructor is eliminated to keep the example short). A new `Teaches` relation is created between `mathew` and `prog` in line 43. Line 44 shows how to access member fields (and methods) from relations, which is the same as accessing member fields (and methods) from objects. In line 45, a new `ResearchPaper fickle` is created and placed in `Publishes` relation with `mathew` in line 46. The last line reclassifies `mathew` from `Teachingstaff` to `ResearchStaff`. It is interesting to ask what happens to `mathew`'s relationships, after `mathew` reclassifies. We defer this talk to Section 6.4, as we would like to introduce formal semantics before we proceed onto relationship reclassification.

6.3 Class Inheritance vs Relationship Inheritance

Figure 6.3 shows an example of `student-course` classes with `attends` and `reluctantlyAttends` relationships. There are four classes: `Student` and its sub-class `LazyStudent` and `Course` and its sub-class `HardCourse`. Relationship `attends` is a base-class for `reluctantlyAttends`. Wren and Bierman [15] noticed that the application of standard class-based inheritance to these 'relationship classes' does not adequately capture the intuitive semantics of relationship inheritance. In RelJ, the authors implement relationship inheritance as a restricted form of delegation, as found in Self[] and δ[]. Consider an instance of Figure 6.3: let `alice` be an instance of `student` and `programming` be an instance of `course`.

```
1  abstract root class Staff extends Object{
2          int staffId;
3  }
4
5  state class TeachingStaff extends Staff{
6          int hoursOfTeaching;
7
8          void promote(){Staff}
9          {
10                 hoursOfTeaching--;
11                 if (hoursOfTeaching==0){this⇓ResearchStaff;rating=1;}
12         }
13         void demote(){ }{hoursOfTeaching++;}
14 }
15
16 state class ResearchStaff extends Staff{
17         int rating;
18
19         void promote(){} {rating++;}
20         void demote(){Staff}
21         {
22                 rating--;
23                 if(rating==0){this⇓TeachingStaff;hoursOfTeaching=1}
24         }
25 }
26
27 class Course{String name;}
28 class ResearchPaper{String title;}
29
30 relationship Teaches
31         from TeachingStaff to Course{
32                 int passRate;
33 }
34 relationship Publishes{
35         from Staff to ResearchPaper{
36                 int quality;
37 }
38
39 class Test{
40         Staff mathew := new TeachingStaff();
41         mathew.hoursOfTeaching=1;
42         Course prog := new Course("Programming");
43         Teaches rel_prog := (mathew.Teaches += prog);
44         int rate= tel_prog.passRate;
45         ResearchPaper fickle := new ResearchPaper("Fickle");
46         Publishes rel_fickle := (mathew.Publishes += fickle);
47         mathew.promote(); //what happens mathew's relationships?
48 }
```

Figure 6.2: Implementation of Roles in a $\mathcal{F}ickle_{MR}$ Program

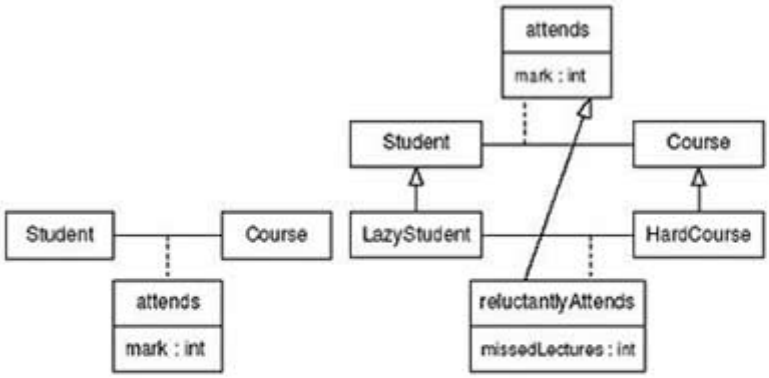

Figure 6.3: Relationship Inheritance example from [15]

When `alice` and `programming` are placed in the `Attends` relationship, an instance of `Attends` is created between those objects. Subsequently, when `alice` and `programming` are further placed in `ReluctantlyAttends`, an instance of `ReluctantlyAttends` is created between `alice` and `programming`, but contains only the `missedLectures field`. If the `ReluctantlyAttends` instance receives a field look-up request for `mark`, it passes(*delegates*) the request to the `Attends`(super) instance that exists between those same objects. To ensure completeness between all instances, we must ensure the following two cases:

- If `alice` and `programming` were placed in the `ReluctantlyAttends` relationship without first having been placed in the `Attends` relationship, then an `Attends` instance would be implicitly created between them.
- If `alice` and `programming` were later placed in the `CompulsorilyAttends` relationship, then its instance and that os `ReluctantlyAttends` would share a common super-instance: the `Attends` instance between `alice` and `programming`.

This change in relationship inheritance model was necessary to reflect semantic correctness of relationships. For example, if `Alice` reluctantly attends a course, even then she is attending and hence will get a mark. Also, if `Alice` is attending lecture, both reluctantly and compulsorily, she will get the same marks. For each pair of related objects, there should be only one instance of each relationship so that relationship properties are consistent.

6.4 Relationship Reclassification

Relationship reclassification is a novel and very powerful idea to revolutionize the way database systems and dynamic object-oriented programs are written. In $\mathcal{F}ickle_{MR}$ we allow reclassification on both: objects and relations as we believe both are equally important. The most interesting part of this project is what to do with an object's relationships when it is reclassified ? Should we simply discard the object's relationships ? In this section we give the various solutions we have discussed for the above mentioned question. In the next sections we give the typing rules and operational semantics for the strategy we use in $\mathcal{F}ickle_{MR}$.

Consider the class-relationship hierarchy presented in Figure 6.4. Let `Person` be the root class for `Author` and `Poet` and `Publication` be the root class for `Book` and `Poem`. This allows `author` to reclassify to `poet` and vice-versa and `book` to reclassify to `poem`(hypothetically!!!) and vice-cersa. *Publishes* is the root-relation for *writes* and *composes*, hence *writes* may reclassify to *composes* and vice-versa. `Person` *publishes* `publication`, `author` *writes* `books` and `poet` *composes* `poems`. We'll assume that $john : Author, lotr : Book, john.Writes+ = lotr$. Now we reclassify `john` to a `Poet` $john \Downarrow Poet$.

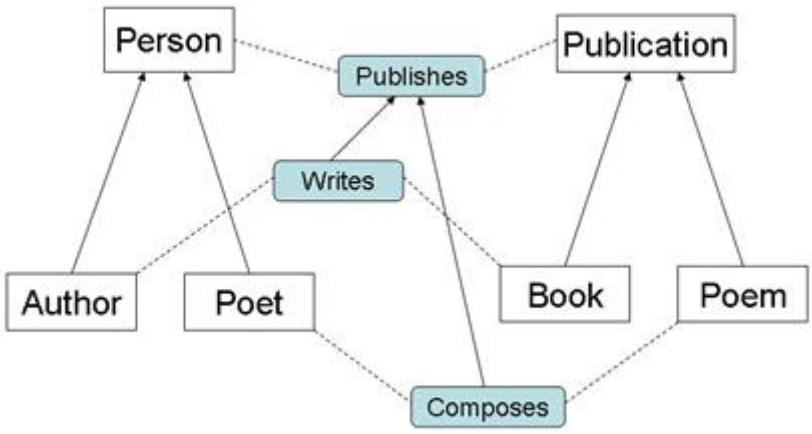

Figure 6.4: UML Diagram for Relationship Reclassification

First option is to reclassify objects as it is done in $\mathcal{F}ickle_{||}$and simply do nothing about the objects' relationships i.e. they are deleted. A programmer would have to specify new relationships for all objects being reclassified. For the example being considered, this would mean that the $Writew$ relation between $john$ and $lotr$ is discarded. This ofcourse is not very elegant but is sometimes necessary in some situations where the class to which an object is being reclassified is not related to the object on

the other side of the relation in any way.

Second option is to reclassify objects like in $\mathcal{F}ickle_{||}$ and automatically reclassify its relations. This is an extremely elegant way to perform reclassification, but on the other hand its behaviour is very complex to specify. Automatic reclassification can only be performed when there is a clear association being the object's classes and relations. The example being discussed in Figure 6.4 is perfect for automatic reclassification. As $john$ reclassifies to $Poet$, the system looks at all of $john's$ relationships, in this case it is only $Writes$. It then looks at the `to` attribute of $Writes$ which is $lotr : Book$. Then it checks the class $Poet$ (to which $john$ is reclassifying to) for all of it's relationships, in this case it is only $Composes$. Then it checks the `to` attribute of $Composes$ which is $Poem$. Once this dependency check has been performed, the system then reclassifies $lotr : Book$ to $Poem(lotr \Downarrow Poet)$ and then reclassifies the $Writes$ relation to a $Composes$ relation.

Although automatic reclassification may seem elegant there are several issues with taking this approach. First issue is that automatic reclassification is limited to perfect class-relatonship hierarchy like the one given in above example. For example, what if `book` could not be reclassified to `poem` (as in real world). Then the relation $Writes$ cannot be reclassified to $Composes$. Second issue with it is that the programmer may not have intended to perform extra reclassifications that the system performs, like $lotr$ being changed from $Book$ to $Poem$. Third problem is with typechecking. Formal system for automatic reclassification is very complex to specify and typecheck, as it is hard to say when the class-relationship association is not perfect. Fourth and most interesting issue is of *infinite reclassification chain*. This is an issue for the third approach too, so we discuss it after defining our third strategy for reclassification.

```
class Author extends Person{
 ...
 upon reclassification to n {
  this.Writes⇓Composes;
 }
}

relationship Writes extends Publishes from Author to Book{
 ...
 upon reclassification to Composes {
  this.from⇓Poet;
  this.to⇓Poem;
 }
}
```

Figure 6.5: Reclassification Directives for Example in Figure 6.4

Third option is to have *reclassification directives.* Reclassification directives are user-defined piece

of code which is executed whenever a class or relationship is reclassified. This is simple and elegant approach to reclassification. The reclassification directive for a class is similar to the concept of a destructor in C++ and finalizer in Java which are called before an object is destroyed. The reclassification directive for the example in Figure 6.4 is given in Figure 6.5. When $john : Author$ is reclassified to $Poet$, the reclassification directive is called by *directive lookup function* $\mathcal{DRD}(P, Author, Poet)$ (see Figure 4.9) which takes a program P, the class which is reclassifying $Author$, and the class to which it is reclassifying, $Poet$. The directive in this example would run the code $this.Writes \Downarrow Composes$ which requests the $Write$ relation to reclassify to $Composes$. This in turn call the equivalent lookup function for relationships $\mathcal{DRD}_R(P, Writes, Composes)$.

Note the reclassification directive for reclassifying $Writes$ to $Composes$ has code to reclassify both: its `from` component and `to` component. This is so because instead of reclassifying a class we allow programmers to reclassify relationships directly. Now there is a problem because the code $this.from \Downarrow Poet$ asks for $john$ to be reclassified again. If we take the approach of destructors and finalizers which are executed `before` the object is destroyed, then the execution of this program will cause a cycle. If the directive is run before objects/relations are reclassified then while executing the code $this.from \Downarrow Poet$, the system will still see $john : Author$ so it will run the code $this.from \Downarrow Poet$ which will again call the reclassification directive from class $Author$ and this cycle will continue infinitely. For this reason, we reclassify objects/relations *before* calling the required reclassification directive. This way when the system reaches the code $this.from \Downarrow Poet$, it would see $john : Poet$ and skips.

In $\mathcal{F}ickle_{MR}$ we take the third option as it is simple, user controlled and not very hard to specify formally. The typing rules and the operational semantics for reclassification is given in the next sections.

It is quite intuitive to allow relationships to be reclassified directly in addition to classes. For example, if a database administrator wished to change everyone's role from $Writes$ to $Composes$, he/she could simply reclassify all $Writes$ instances to $Composes$. Relationship reclassification is a little different from classes as discussed before and as demonstrated in Figure 6.5. While reclassification, a relationship must consider what to do with its `from` and `to` attributes both.

Reclassifying can induce a chain of other reclassifications. Until the execution of the whole chain of reclassifications is complete, the heap is in a corrupt state i.e. the type system is broken. For example, in the case above, between reclassifying $john$ to $Poet$ and executing the reclassification directive, the behaviour of $john.Composes$ is undefined until the reclassification directive has finished execution. For this reason reclassification must be an atomic execution. This does not effect $\mathcal{F}ickle_{MR}$ but would surely effect any multi-threaded languages.

6.5 Type System

We extend the type system of $\mathcal{F}ickle_{MR}$ with relationships. Wren et al give a formal system for basic operations in relationships in the form of RelJ calculus [15]. We develop our type system with relationships by extending RelJ with method calls (multiple dispatch) and reclassification operator. Figure 6.6 gives the typing rules from RelJ for `set-types`, `to` and `from` attributes of relationships and relationship `addition` and `deletion`. The sub-relation and subtype rules are given in Figures 4.3 and 4.14 respectively. `SETSubType` says that a set containing entities of type n_1 is a sub-type of a set containing entities of type n_2 if n_1 is a sub-type of n_2. $empty$ is a sub-type of set type. `RELAllTo` says that an expression of the form $e.r$ returns a set-type comprising of the `to` part of the relationship `r`. `RELInst` rule says that an expression of the form $e : r$ returns a set-type comprising of relations (relationship instances) of type `r`. The `FROM` rule says that in an expression of the form $e.from$, if e is a relation then the expression returns the type of the `from` entity of the relationship. Likewise, an expression of the form $e.to$ returns the type of the `to` entity.

$$\frac{P \vdash\ n_1\ \leq\ n_2}{P \vdash\ set\langle n_1\rangle\ \leq\ set\langle n_2\rangle}\ (SETSubType) \qquad \frac{}{P \vdash\ empty\ :\ set\langle n\rangle}\ (SETType)$$

$$\frac{P,\Gamma \vdash\ e\ :\ n[]\Gamma^{'}[]\phi \quad \mathcal{R}(r) = rel\ r\ extds\ r^{'}\ from\ n^{'}\ to\ n^{''} \quad P \vdash n \leq n^{'}}{P,\Gamma \vdash\ e.r\ :\ set\langle n^{''}\rangle[]\Gamma^{'}[]\phi}\ (RELAllTo)$$

$$\frac{P,\Gamma \vdash\ e\ :\ n[]\Gamma^{'}[]\phi \quad \mathcal{R}(r) = rel\ r\ extds\ r^{'}\ from\ n^{'}\ to\ n^{''} \quad P \vdash n \leq n^{'}}{P,\Gamma \vdash\ e : r\ :\ set\langle r\rangle[]\Gamma^{'}[]\phi}\ (RELInst)$$

$$\frac{P,\Gamma \vdash e : r[]\Gamma^{'}[]\phi \quad \mathcal{R}(r) = rel\ r\ extds\ r^{'}\ from\ n\ to\ n^{'}}{P,\Gamma \vdash\ e.from : n[]\Gamma^{'}[]\phi}\ (FROM)$$

$$\frac{P,\Gamma \vdash\ e\ :\ r[]\Gamma^{'}[]\phi \quad \mathcal{R}(r) = rel\ r\ extds\ r^{'}\ from\ n\ to\ n^{'}}{P,\Gamma \vdash e.to : n^{'}[]\Gamma^{'}[]\phi}\ (TO)$$

$$\frac{P,\Gamma \vdash e_1\ :\ n_1[]\Gamma^{'}[]\phi \quad P,\Gamma^{'} \vdash e_2\ :\ n_2[]\Gamma^{''}[]\phi^{'} \quad \mathcal{R}(r) = rel\ r\ extds\ r^{'}\ from\ n_1^{'}\ to\ n_2^{'} \quad P \vdash n_1\ \leq\ n_1^{'} \quad P \vdash n_2\ \leq\ n_2^{'}}{P,\Gamma \vdash\ e_1.r+ = e_2 : r[]\Gamma^{''}[]\{\phi \cup \phi^{'}\}}\ (RELAdd)$$

$$\frac{P,\Gamma \vdash e_1\ :\ n_1[]\Gamma^{'}[]\phi \quad P,\Gamma^{'} \vdash e_2\ :\ n_2[]\Gamma^{''}[]\phi^{'} \quad \mathcal{R}(r) = rel\ r\ extds\ r^{'}\ from\ n_1^{'}\ to\ n_2^{'} \quad P \vdash n_1\ \leq\ n_1^{'} \quad P \vdash n_2\ \leq\ n_2^{'}}{P,\Gamma \vdash e_1.r- = e_2 : r[]\Gamma^{''}[]\{\phi \cup \phi^{'}\}}\ (RELSub)$$

Figure 6.6: Type System for Relationships

6.5.1 Typechecking Reclassification Expressions

Section 4.14 discussed the three different strategies for reclassification. The third option with reclassification directives will be used in $\mathcal{F}ickle_{MR}$. Figure 6.7 gives the typing rules for reclassification of objects and relations. The RELRecl rule is for relationship reclassification where id is a relationship variable which is being reclassified to relationship $r^{'}$. The judgement $P \vdash r^{'} \diamondsuit_{rt}$ (see Figure **??**) says that $r^{'}$ is a reclassifiable type. $\Gamma(id)$ returns the type of id which is r. The root classes of r and $r^{'}$ must be the same, which is checked by the $\mathcal{R}oot_R$ function (see Figure 4.10). As discussed in section 4.14, the reclassification (environment update $\Gamma^{'} = \Gamma[id \mapsto r^{'}]$) must occur before calling the related directive. The directive is type-checked in the updated environment $\Gamma^{'}$. The last statement $\Gamma^{''}(id) = r^{'}$ is important, because it ensures that the reclassification directive does not update $\Gamma^{'}$ to $\Gamma^{''}$ where $\Gamma^{''}(id) \neq r^{'}$ otherwise it would result in incorrect types and undefined behaviour at run-time. The effects resulting from typechecking the reclassification directives is combined with $r \Downarrow r^{'}$ and returned as result of typechecking the main $id \Downarrow r^{'}$ expression. The reclassification for classes is same as relationships. Rules RELReclSkip and OBJReclSkip skip a reclassification statement $id \Downarrow r$ and $id \Downarrow c$ if variable `id` is of dynamic type `r` and `c` respectively.

$$\frac{\begin{array}{l}(RELRecl)\\ P \vdash r^{'} \diamondsuit_{rt}\\ \Gamma(id) = r\\ \mathcal{R}oot_R(P, r) = \mathcal{R}oot_R(P, r^{'})\\ \Gamma^{'} = \Gamma[id \mapsto r^{'}]\\ \mathcal{DR}_R(P, r, r^{'}) = e^{'}\\ P, \Gamma^{'} \vdash e^{'} : void[]\Gamma^{''}[]\phi\\ \Gamma^{''}(id) = r^{'}\end{array}}{P, \Gamma \vdash id \Downarrow r^{'} \ : \ r^{'}[]\Gamma^{''}[]\{\phi \cup (r \Downarrow r^{'})\}} \qquad \frac{\begin{array}{l}(OBJRecl)\\ P \vdash c^{'} \diamondsuit_{rt}\\ \Gamma(id) = c\\ \mathcal{R}oot(P, c) = \mathcal{R}oot(P, c^{'})\\ \Gamma^{'} = \Gamma[id \mapsto c^{'}]\\ \mathcal{DR}(P, c, c^{'}) = e^{'}\\ P, \Gamma^{'} \vdash e^{'} : void[]\Gamma^{''}[]\phi\\ \Gamma^{''}(id) = c^{'}\end{array}}{P, \Gamma \vdash id \Downarrow c^{'} \ : \ c^{'}[]\Gamma^{''}[]\{\phi \cup (c \Downarrow c^{'})\}}$$

$$\frac{\begin{array}{l}(RELReclSkip)\\ \Gamma(id) = r \ where \ r = r^{'}\end{array}}{P, \Gamma \vdash id \Downarrow r^{'} \ : \ r^{'}[]\Gamma[]\{\ \}} \qquad \frac{\begin{array}{l}(OBJReclSkip)\\ \Gamma(id) = c \ where \ c = c^{'}\end{array}}{P, \Gamma \vdash id \Downarrow c^{'} \ : \ c^{'}[]\Gamma[]\{\ \}}$$

Figure 6.7: Typing Rules for Reclassification

6.6 Operational Semantics for Relationships

To give first class support to relationships, we allow addresses to point to relations. Like objects in $\mathcal{F}ickle_{MR}$, relations are stored in the heap. The structure of stack and heap with relations is given in Figure 4.18. The stack maps `this` to an address and local variables to a value. The heap maps adresses to objects *and* relations. The values in $\mathcal{F}ickle_{MR}$ are `true`, `false`,`null` and addresses. Objects consist of field-value pairs. Relations are like objects with additional `from` and `to` addresses.

Introducing relations has a deep impact on the operational semantics of $\mathcal{F}ickle_{||}$. Now since addresses can point to objects and relations, the rules must take both into consideration. For each heap access, we would need to determine if the address points to an object or relation. Most of the rules for relationships are similar to the ones for objects and we gave some thought over unifying objects and relations. This could not be achieved for two reasons. The *first* one being that relations have extra `from` and `to` fields. The *second* and more significant one is that relationships use delegation for inheritance as discussed in Section 6.3 which makes the relation reclassification very different from object reclassification evident from the rules later on. We talk more about it in Chapter 8.

6.6.1 Object Reclassification

First, lets consider object reclassification. The rule for reclassifying objects has been given in Figure 6.8. In the reclassification expression $e \Downarrow c^{'}$, e is first evaluated to an address ι which must point to an object of class c. The function $\mathcal{R}oot(P, c)$ in Figure 4.10 returns the root class of class c. The function $\mathcal{F}s(P, \mathcal{R}oot(P, c))$ is used to retreive the values of all fields in the root class and its superclasses. When the new object of class $c^{'}$ is created these fields values are simply copied to the object while the rest of the fields are initialized with the initial values of the fields. After the reclassification, the directive lookup function is called and the body of the reclassification directive is executed. The second reclassification rule skips a reclassification statement if the reclassification source and target classes are the same.

Just like type-checking reclassification expression, the directive is executed after reclassification. As discussed in Section 6.4 this eliminates the possibility of having a cycle in the reclassification chain. This however does not guarantee that the reclassification chain is finite. For example, consider the UML diagram in Figure 6.9. If we reclassify an object of class A1 to A2, its reclassification directive reclassifies relation R_A1_B1 to R_A2_B2 which to which will reclassify class B1 to B2 and so on. Assuming that no programs contain infinite number of classes and relationships, we can safely conclude that reclassification operation is not non-terminating.

$$\frac{\begin{array}{l} e, \sigma, \chi \rightsquigarrow_p \iota, \chi^{'} \\ \chi^{'}(\iota) = [[\ c || \ldots\ ]] \\ \mathcal{F}s(P, \mathcal{R}oot(P, c)) = \{f_1, \ldots, f_r\} \\ v_l = \chi^{'}(\iota)(f_l)\ \forall l \in \{1, \ldots, r\} \\ \mathcal{F}s(P, c^{'}) \backslash \{f_1, \ldots, f_r\} = \{f_{r+1}, \ldots, f_{r+q}\} \\ v_l\ initial\ for\ \mathcal{F}(P, c^{'}, f_l)\ \forall l \in \{r+1, \ldots, r+q\} \\ \chi^{''} = \chi^{'}[\iota \mapsto [[c^{'} | f_1 : v_1, \ldots, f_{r+q} : v_{r+q}]]\ ] \\ \mathcal{DR}(P, c, c^{'}) = e^{'} \\ e^{'}, \sigma, \chi^{''} \rightsquigarrow_p void, \chi^{'''} \end{array}}{e \Downarrow c^{'}, \sigma, \chi \rightsquigarrow_p \iota, \chi^{'''}}$$

$$\frac{\begin{array}{l} e, \sigma, \chi \rightsquigarrow_p \iota, \chi^{'} \\ \chi^{'}(\iota) = [[c^{'} | \ldots]] \end{array}}{e \Downarrow c^{'}, \sigma, \chi \rightsquigarrow_p \iota, \chi}$$

Figure 6.8: Operational Semantics for Object Reclassification

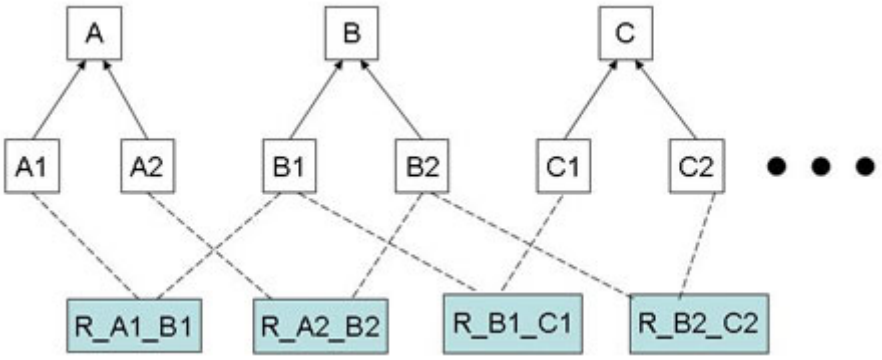

Figure 6.9: Infinite Reclassification Chain

6.6.2 Relationship Reclassification

Relationship reclassification is quite different to object reclassification because relationship uses delegation for inheritance. In object reclassification we copied the fields of the root class and its superclasses onto an object of the target class of the reclassification. The rest of the fields are initialized with their initial values.

$$createRel(r, \iota_r, \iota_1, \iota_2, r_{rt}, \iota_{r_{rt}}, \chi) = \begin{cases} \chi[\iota_r \mapsto \langle\langle r || (\iota_{rt}, \iota_1, \iota_2, \{f_1 : v_1, \ldots, f_r : v_r\})\rangle\rangle] & if\ r^{'} = r_{rt} \\ \quad \forall i \in \{1, \ldots, r\} : \mathcal{F}_R(P, r, f_i) \neq \mathcal{U}df \\ \quad \forall i \in \{1, \ldots, r\} : v_i\ initial\ for\ f_i \\ \\ \chi^{''} & otherwise \\ \quad \chi^{'} = createRel(r^{'}, \iota^{'}, \iota_1, \iota_2, r_{rt}, \iota_{r_{rt}}, \chi) \\ \quad where\ \iota^{'} \neq \iota\ and \\ \quad \chi^{''} = \chi^{'}[\iota \mapsto \langle\langle\ r\ ||\ (\iota^{'}, \iota_1, \iota_2, \{f_1 : v_1, \ldots, f_r : v_r\})\rangle\rangle] \\ \quad \forall i \in \{1, \ldots, r\} : \mathcal{F}_R(P, r, f_i) \neq \mathcal{U}df \\ \quad \forall i \in \{1, \ldots, r\} : v_i\ initial\ for\ f_i \end{cases}$$

Figure 6.10: Functions for Relationship Reclassification

$$\mathcal{RA}(\chi, \iota, R) = \begin{cases} \iota & if\ r = R \\ \mathcal{RA}(\chi, \iota^{'}, R) & otherwise \\ & where\ \chi(\iota) = \langle\langle r | (\iota^{'}, \_, \_, \_)\rangle\rangle \end{cases}$$

$$\mathcal{RA}(\chi, \iota, Relation) = \begin{cases} \iota & if\ r = Relation \\ \emptyset & otherwise \\ & where\ \chi(\iota) = \langle\langle Relation | (\iota^{'}, \_, \_, \_)\rangle\rangle \end{cases}$$

Figure 6.11: Common Root-Address Function

In relationship reclassification we do not need to copy fields of root relation and its super-relations onto the target relation as we do for objects. With delegation, we need to create the relationship hierarchy from the target relation to the direct sub-relation of the root which is delegated to the root relation. We do this by tail recursion where the super-relation is created and its address is returned, the sub-relations are then created and delegated to the super-relations by the returned address. Function $createRel$ does this in Figure 6.10. In the function $createRel(r, \iota_r, \iota_1, \iota_2, r_{rt}, \iota_{r_{rt}}, \chi)$, r is the

target relationship, ι_r is the address at which to create the target relation, ι_1 and ι_2 are `to` and `from` attributes for r respectively, r_{rt} is the root relationship and $\iota_{r_{rt}}$ is the address in heap where the root relation is stored. $createRel$ function assumes $\mathcal{R}(r) = relationship\ r\ extends\ r^{'} \ldots$. The function is recusive which creates a super-relation before its sub-relation. The updated heap and address of its super-relation are returned for sub-relations to delegate to them.

$$\frac{\begin{array}{l} e, \sigma, \chi \rightsquigarrow_p \iota, \chi^{'} \\ \chi^{'}(\iota) = \langle\langle r^{'} || (\iota^{'}, \iota_1, \iota_2, \{\ldots\})\rangle\rangle \\ \mathcal{R}oot_R(P, r) = r_{root} \\ \mathcal{RA}(\chi^{'}, \iota, r_{root}) = \iota_{root} \\ \chi^{''} = createRel(r^{'}, \iota, \iota_1, \iota_2, r_{root}, \iota_{root}, \chi^{'}) \\ \mathcal{DR}_R(r, r^{'}) = e \\ e, \sigma, \chi^{''} \rightsquigarrow_p void, \chi^{'''} \end{array}}{e \Downarrow r^{'}, \sigma, \chi \rightsquigarrow_p \iota, \chi^{'''}}$$

$$\frac{\begin{array}{l} e, \sigma, \chi \rightsquigarrow_p \iota, \chi^{'} \\ \chi^{'}(\iota) = \langle\langle r^{'} | \ldots\rangle\rangle \end{array}}{e \Downarrow r^{'}, \sigma, \chi \rightsquigarrow_p \iota, \chi}$$

Figure 6.12: Operational Semantics for Relationship Reclassification

The function $\mathcal{RA}(\chi, \iota, r_{root})$ given in Figure 6.11 takes a heap χ, ι address of the relation being reclassified and the root relationship, and returns the `address` of the root relation. The operational semantics for relationship reclassification is given in Figure 6.12. The rule first finds the root relationship r_{rt} and its address ι_{rt}. It calls the $createRel$ funtion to create the target relation $r^{'}$ and delegate to the root relation r_{rt}. The reclassification directive is evaluated after the reclassification as usual.

6.6.3 Other Rules

Wren et al use a relationship table $\rho : (RelName \times Address \times Address) \rightarrow Address$ in their configuration. $\rho(r, \iota_1, \iota_2)$ returns the address in their store where the instance of relationship r is stored between ι_1 and ι_2. We do not use this in our configuration to keep the operational semantics simple. We get the address of a relation by looking in the heap. The rules for operational semantics are given in Figure 6.13. The FROM rule says that an expression of the form $e.from$ returns the `from` attribute of the relation that e evaluates to. Likewise, an expression of the form $e.to$ returns the `to` attribute of the relation that e evaluates to. The ALLTo rule says that an expression of the form $e.r$ returns all the objects/relations that are related to object/relation resulting from evaluation of expression e. In

$$\frac{e, \sigma, \chi \rightsquigarrow_p \iota, \chi' \quad \chi'(\iota) = \langle\langle r \mid\mid (\iota', \iota_1, \iota_2, \{\ldots\}) \rangle\rangle}{e.from, \sigma, \chi \rightsquigarrow_p \iota_1, \chi'} \; (FROM)$$

$$\frac{e, \sigma, \chi \rightsquigarrow_p \iota, \chi' \quad \chi'(\iota) = \langle\langle r \mid\mid (\iota', \iota_1, \iota_2, \{\ldots\}) \rangle\rangle}{e.to, \sigma, \chi \rightsquigarrow_p \iota_2, \chi'} \; (TO)$$

$$\frac{e, \sigma, \chi \rightsquigarrow_p \iota_1, \chi' \quad S = \{\iota_2 \mid\mid \chi'(\iota) = \langle\langle r \mid\mid (\iota', \iota_1, \iota_2, \{\ldots\}) \rangle\rangle \}}{e.r, \sigma, \chi \rightsquigarrow_p S, \chi'} \; (ALLTo)$$

$$\frac{e, \sigma, \chi \rightsquigarrow_p \iota_1, \chi' \quad S = \{\iota \mid\mid \chi'(\iota) = \langle\langle r \mid\mid (\iota', \iota_1, \iota_2, \{\ldots\}) \rangle\rangle \}}{e : r, \sigma, \chi \rightsquigarrow_p S, \chi'} \; (ALLRelInst)$$

$$\frac{e, \sigma, \chi \rightsquigarrow_p \iota_1, \chi' \quad e', \sigma, \chi' \rightsquigarrow_p \iota_2, \chi'' \quad (\iota', \chi''') = addRel(r, \iota_1, \iota_2, \chi'')}{e.r+ = e', \sigma, \chi \rightsquigarrow_p \iota', \chi'''} \; (ADD)$$

$$\frac{e, \sigma, \chi \rightsquigarrow_p \iota_1, \chi' \quad e', \sigma, \chi' \rightsquigarrow_p \iota_2, \chi'' \quad \chi''' = delRel(r, \iota_1, \iota_2, \chi'')}{e.r- = e', \sigma, \chi \rightsquigarrow_p void, \chi'''} \; (DEL)$$

Figure 6.13: Other Operational Semantics Rules for Relationships

other words it returns objects/relations that appear as `from` attribute of relationship r. ALLRelInst rule says that an expression of the form $e : r$ returns a set of relationship instances or relations r that relates the object/relation resulting from evaluation of e to other objects or relations. The ADD rule says that an expression of the form $e.r+ = e'$ creates a new r relation between the objects/relations resulting from e and e'. The $addRel$ function is given in Figure 6.14.

The DEL rule is for deleting a relation r between two objects/relations resulting from evaluating e and e'. This rule is interesting because Wren et al return the address of the deleted relation after evaluating $e.r- = e'$. They update the relationship table ρ mentioned earlier for deleting the relationship instance, but the heap still maintains the relationship instance between objects/relations from e and e'. As a result a reference to the deleted relation could be kept and used to access the relation after deletion. The garbage collector would not sweep the deleted relation from the heap because a reference to it is maintained. This could cause `memory leaks` in C++ and `out-of-memory-error` in Java. We take a different approach by deleting the actual relation from the heap. The expression $e.r- = e'$ returns void.

Figure 6.14 defines the $addRel$ and $delRel$ functions for ADD and DEL rules. The $addRel$ rule given here is simpler for of that given by Wren et al in [15]. The function is recursive and returns a tuple of address and heap. This function creates a super-relation first, then returns its address for

$$
addRel(r, \iota_1, \iota_2, \chi) = \begin{cases} (\iota, \chi') & if\ r = Relation \\ \quad \chi' = \chi[\iota \mapsto \langle\langle Relation || (null, \iota_1, \iota_2, \{\ \})\rangle\rangle] & \\ & \\ (\iota, \chi'') & otherwise \\ \quad where\ \mathcal{R}(r) = rel\ r\ extds\ r' \ldots & \\ \quad (\iota', \chi') = addRel(R', \iota_1, \iota_2, \chi) & \\ \quad \chi'' = \chi'[\iota \mapsto \langle\langle r || (\iota', \iota_1, \iota_2, \{f_1 : v_1, \ldots, f_r : v_r\})\rangle\rangle\ ] & \\ \quad \forall i \in \{1, \ldots, r\} : \mathcal{F}_R(P, r, f_i) \neq \mathcal{U}df\ and & \\ \quad \forall i \in \{1, \ldots, r\} : v_i\ initial\ for\ f_i & \end{cases}
$$

$$
delRel(r, \iota_1, \iota_2, \chi) = \begin{cases} \chi[\iota_i \mapsto null] & \\ \quad \forall i : \chi(\iota_i) = \langle\langle\ r'\ ||\ (,\iota_1, \iota_2,)\ \rangle\rangle & \\ \quad and\ r' \leq r & \end{cases}
$$

Figure 6.14: Variant of *addRel* and *delRel* function from RelJ

sub-relations to delegate to it. All relations extend `Relation` which has no fields. The $delRel$ rule simply deletes the relation r between ι_1 and ι_2 from heap i.e frees the memory location ι. It also deletes any sub-relations of r that relate object/relation at ι_1 to object/relation at ι_2.

6.7 Evaluation of Relationships in $\mathcal{F}ickle_{MR}$

Chapter 7

Unifying Roles, Multimethods and Reclassification

Implementing roles, multimethods and reclassification in $\mathcal{F}ickle_{||}$ would be extremely advantageous as it would allow easy modelling of a real world scenario into a software system. For example consider the UML diagram in Figure 7.1. `Employees` may be `Programmers` or `Managers`. They work on `Projects`, which may be `Technical` projects or `Management` projects. Relation `WorksWellOn` inherits from relation `WorksOn`. All Employees `work on` projects, but Programmers `work well` only on technical projects. The dashed lines show the relationship between objects.

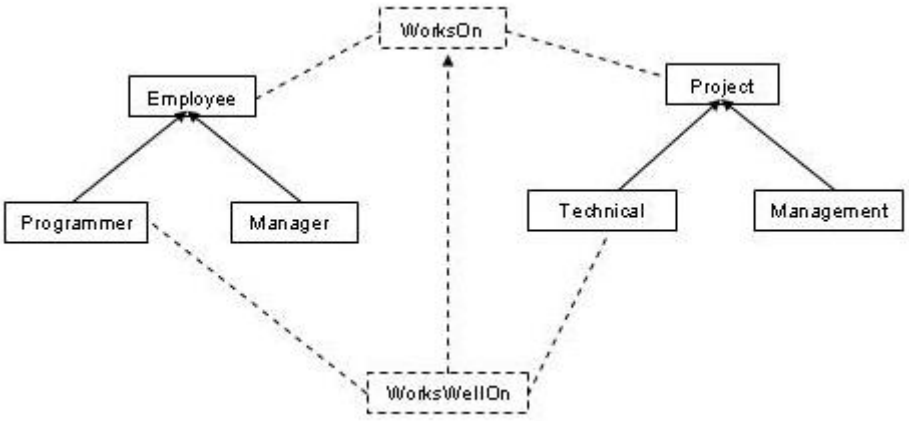

Figure 7.1: UML diagram for a Employee-Project Example

In a real world scenario, entities change and so does their roles. Programmers become Managers after years of experience but their basic identity remains the same, for example their `name`. We achieve

this in $\mathcal{F}ickle_{||}$ by reclassifying objects from one state class to another. Multiple dispatch allows us to dispatch methods on the dynamic types of the entities (Employees). Roles allow us to model relationships between objects and relantionships. In Figure 7.2, we write a $\mathcal{F}ickle_{||}$ code assuming multiple dispatch (with MultiJava syntax) and roles (with RelJ syntax), to demonstrate how we would code the scenario in Figure 7.1.

In Figure 7.2, each `Project` has a `desirable` method which determines if an `Employee` is desirable for that project. In a general project every employee is desirable.In a `Management` project programmers are desirable only if they have a PhD and in a `Technical` project, managers are desirable only if they have an MBA. The `desirable` method in `Management` class overrides the `desirable` method in `Project` and performs multiple dispatch on `Programmer` dynamic type. Likewise, the `desirable` method in `Technical` performs multiple dispatch on `Manager` dynamic type. When a programmer is promoted, he becomes a `Manager`. This is achieved by reclassifying `Programmer` to `Manager`. The programmer's `employeeId` remains the same, but his salary increases. The instances of `WorksOn` relation between Employee and Project is untouched by the reclassification, but instances of `WorksWellOn` relation between `Programmer` and `Technical` projects are deleted.

Consider the following test case on the code described in Figure 7.2:

We create one instance of a `Technical` project and one instance of a `Management` project. We create two programmers: one with PhD and another one without PhD, and one manager without an MBA. All three employees are desirable for project `cOmega` but only employees `e1` and `e3` are desirable for project `merger`. So the output will look like:

7.1 Case Study A: The Ocean Ecosystem

7.2 Case Study B: DOC Database System

```
1  abstract root class Employee extends Object{
2          int employeeId;
3          int salary;
4
5          abstract int promote(){};
6  }
7
8  state class Programmer extends Employee{
9          bool phd;
10
11         int promote(){Manager}
12         {
13                 this⇓Manager;
14                 salary++;
15                 mba := false;
16         }
17 }
18
19 state class Manager extends Employee{
20         bool mba;
21
22         int promote(){} {salary++;}
23 }
24
25 class Project extends Object{
26         String projectName;
27         int budget;
28
29         bool desirable(Employee e){
30                 true;
31         }
32 }
33
34 class Technical extends Project{
35         int numberOfPhds;
36
37         bool desirable(Employee@Manager manager){
38                 if (manager.mba){true}
39                 else {false}
40         }
41 }
42
43 class Management extends Project{
44         int numberOfMBAs;
45
46         //only programmers who have a PhD can work on management projects
47         bool desirable(Employee@Programmer prog){
48                         if(prog.phd) {true}
49                         else {false}
50         }
51 }
52
53 relation WorksOn
54         from Employee to Project{
55                 int totalHours;
56 }
57
58 relation WorksWellOn extends WorksOn
59         from Programmer to Technical{
60                 bool enjoyed;
61 }
```

```
class Test{
        Project cOmega := new Technical("c omega",10000);
        Project merger := new Management("merger",15000);

        Employee e1 := new Programmer(1,1000,true);
        Employee e2 := new Programmer(2,500,false);
        Employee e3 := new Manager(3,2000,false);
        //assign them an appropriate project
        for (Employee e: e1 to e3){ //Employee e — e in eN
                if (cOmega.desirable(e)){
                        e.WorksOn += cOmega;
                }
                if(merger.desirable(e)){
                        e.WorksOn += merger;
                }
        }
        //output result
        for (Employee e: e1 to e3){
                for (Project c: e.WorksOn){
                        print "Employee " + e.employeeID + " WorksOn "+ c.projectName;
                }
        }
}
```

```
Employee 1 WorksOn c Omega
Employee 2 WorksOn c Omega
Employee 2 WorksOn merger
Employee 3 WorksOn c Omega
Employee 3 WorksOn merger
```

Chapter 8

Conclusion and Further Work

In this project, we showed how multiple dispatch and relationships can be applied to $\mathcal{F}ickle_{||}$. The interesting property of $\mathcal{F}ickle_{||}$to allow objects to reclassify at run-time, can also be applied to relationships as shown in the project.

Appendices

Appendix A

Semantics of $\mathcal{F}ickle_{MR}$

Bibliography

[1] Martin Abadi and Luca Cardelli, *A theory of objects*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. 8, 9, 10

[2] Rakesh Agrawal, Linda G. Demichiel, and Bruce G. Lindsay, *Static type checking of multi-methods*, SIGPLAN Not. **26** (1991), no. 11, 113–128. 15, 53, 57

[3] John Boyland and Giuseppe Castagna, *Parasitic methods: an implementation of multi-methods for java*, OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), ACM Press, 1997, pp. 66–76. 15, 17

[4] Craig Chambers, *Object-oriented multi-methods in cecil*, ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming (London, UK), Springer-Verlag, 1992, pp. 33–56. 6

[5] Craig Chambers and Gary T. Leavens, *Typechecking and modules for multimethods*, ACM Trans. Program. Lang. Syst. **17** (1995), no. 6, 805–843. 43, 45, 57

[6] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein, *Multijava: modular open classes and symmetric multiple dispatch for java*, SIGPLAN Not. **35** (2000), no. 10, 130–145. 6, 42, 44, 57

[7] Ole-Johan Dahl and Kristen Nygaard, *Simula: an algol-based simulation language*, Commun. ACM **9** (1966), no. 9, 671–678. 8

[8] F. Damiani, S. Drossopoulou, and P. Giannini, *Refined Effects for Unanticipated Object Reclassification: Fickle3 (Extended Abstract)*, Theoretical Computer Science: 8th Italian Conference (ICTCS'03. 33

[9] Linda G. DeMichiel and Richard P. Gabriel, *The common lisp object system: an overview*, European conference on object-oriented programming on ECOOP '87 (London, UK), Springer-Verlag, 1987, pp. 151–170. 6

[10] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini, *Fickle: Dynamic Object Re-classification*, ECOOP'01, LNCS, vol. 2072, Springer-Verlag, 2001, pp. 130–149. 22, 24

[11] ———, *More Dynamic Object Re-classification: Fickleii*, ACM Transactions On Programming Languages and Systems **24** (2002), no. 2, 153–191. 22, 23, 24, 25, 27

[12] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid, *Is the Java Type System Sound?*, Theory and Practice of Object Systems **5** (1999), no. 1, 3–24. 43, 46

[13] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach, *Java type soundness revisited*, September 2000. 46

[14] Christopher Dutchyn, Paul Lu, Duane Szafron, Steve Bromling, and Wade Holst, *Multi-dispatch in the java virtual machine (poster session): design and implementation*, OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum) (New York, NY, USA), ACM Press, 2000, pp. 115–116. 12

[15] Bierman G. and Wren A., *First-class relationships in an object-oriented language*, FOOL '05: Proceedings of the The Twelth International Workshop on Foundations of Object-Oriented Languages, 2005. 20, 61, 62, 64, 66, 70, 76

[16] Gary T. Leavens and Todd D. Millstein, *Multiple dispatch as dispatch on tuples*, OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), ACM Press, 1998, pp. 374–387. 12, 15, 17, 45, 57, 58

[17] Todd D. Millstein and Craig Chambers, *Modular statically typed multimethods*, ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming (London, UK), Springer-Verlag, 1999, pp. 279–303. 19, 45, 57

[18] Bjarne Stroustrup, *The design and evolution of c++*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994. 5