Universe Type System for Eiffel

Annetta Schaad

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

SS 2006

Supervised by:
Dipl.-Ing. Werner M. Dietl
Prof. Dr. Peter Müller

Software Component Technology Group

inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

The Universe type system partitions the object store into contexts to control aliasing and dependencies. It exists already an Universe type system which was developed for Java. This report describes how the existing Universe type system can be adapted for Eiffel. Mainly we developed possible treatments for the special Eiffel constructs expanded types and agents that have no counterpart in Java.

Contents

1 Introduction **7**
1.1 Introduction . . . 7
1.2 Existing Universe Type System for Java . . . 7
1.2.1 Overview . . . 7
1.2.2 Ownership Modifiers . . . 7
1.2.3 Type Combinator . . . 8
1.2.4 Subtyping . . . 8
1.2.5 Overriding and Overloading . . . 8
1.2.6 Arrays . . . 8
1.2.7 Instance and Static Methods . . . 8
1.2.8 Pure Methods . . . 8
1.2.9 Static Fields . . . 9
1.2.10 Object Creation and Constructors . . . 9
1.2.11 Generics . . . 9

2 Eiffel Languages Constructs **11**
2.1 Overview . . . 11
2.1.1 Overview . . . 11
2.1.2 Eiffel . . . 11
2.1.3 Java . . . 12
2.2 Classes and Types . . . 12
2.2.1 Classes . . . 12
2.2.2 Types . . . 12
2.2.3 Expanded Types . . . 12
2.2.4 Root Types ANY and NONE . . . 13
2.2.5 Deferred Classes . . . 14
2.3 Genericity, Arrays, and Tuples . . . 14
2.3.1 Genericity . . . 14
2.3.2 Arrays . . . 15
2.3.3 Tuples . . . 15
2.4 Inheritance . . . 15
2.4.1 Multiple Inheritance . . . 15
2.4.2 Overloading and Overriding . . . 16
2.4.3 Polymorphism and Dynamic Binding . . . 16
2.5 Assertions . . . 16
2.5.1 Design-by-Contract . . . 16
2.5.2 Inheritance and Assertions . . . 16
2.6 Agents . . . 18
2.6.1 Agents . . . 18
2.7 Special Features . . . 19
2.7.1 Once Routines . . . 19
2.7.2 Create Procedures . . . 20

2.7.3 Constant Attributes 20
2.8 Exceptions 24
2.8.1 Exceptions 24
2.9 Concurrent Programming 24
2.9.1 Concurrent Programming 24
2.10 Conclusion 24

3 Universe Type System for Eiffel 25
3.1 General Solution 25
3.1.1 Overview 25
3.1.2 Syntax 25
3.1.3 Rules 26
3.1.4 Command-Query Separation 26
3.1.5 Once Routines 26
3.2 Expanded Types 26
3.2.1 Overview 26
3.2.2 Solution 1: Like Reference Types 26
3.2.3 Solution 2: As Part of the Object 27
3.2.4 Solution 3: Always rep 29
3.2.5 Evaluation 29
3.2.6 Detailed Analysis of Solution 2 29
3.3 Agents 37
3.3.1 Overview 37
3.3.2 Agent Types 37
3.3.3 Target, Open and Closed Operands 38
3.3.4 Pure Agents 40
3.3.5 Passing around 43
3.3.6 Other Solutions 43

4 Conclusion 45
4.1 Conclusion 45
4.2 Future Work 45

Chapter 1

Introduction

1.1 Introduction

In object-oriented programs an object can reference any other object and modify its fields through field access or through method call. Programs with arbitrary object structures are difficult to understand, to maintain and to reason about.

Ownership organizes objects into contexts. Each object is owned by at most one other object, called its owner. A context is the set of all objects with the same owner.

One ownership model that enables modular verification is based on the so called owner-as-modifier property. As described in [3], this model distinguishes between read-write and read-only references. It says that only reference chains that pass through the owner of an object X can modify X. On the other hand owners can control modifications of owned objects, but not read access. Ownership properties can be checked statically by type systems.

Subsequently in this chapter we will describe shortly the already existing Universe type system for Java. In chapter 2 we discover the Eiffel language constructs and explain which constructs differ between Eiffel and Java. In the third chapter we describe how a Universe type system for Eiffel can look like. For the constructs of expanded types and agents we do a longer analysis of their treatment. In the last chapter you can read the future work to do with the Universe type system for Eiffel.

1.2 Existing Universe Type System for Java

1.2.1 Overview

Werner Dietl and Peter Müller present in [3] a lightweight ownership model for the Java Modeling Language (JML) called "Universes". It follows the owner-as-modifier property. To check ownership statically, they use the Universe type system. To compensate for the resulting weaker static guarantees, they combine ownership type annotations with specifications in JML [3].

The following short introduction into the Universe type system is based on the two papers "Universes: Lightweight Ownership for JML" [3] and "Universe Type System - Quick-Reference" [4].

1.2.2 Ownership Modifiers

The classification of references is done by using an extended type system. There are three keywords, called the ownership modifiers, to modify the types of declaration or definition:

- **peer** denotes a reference to an object in the same context. This is the default modifier.
- **rep** denotes a reference from an object into the context it owns.

- **readonly** denotes a reference that is read-only and might point to objects in any context.

These modifiers can be used in front of the standard Java type.

1.2.3 Type Combinator

To determine the owner of an object referenced by x.f - and, thus, the type of the field access x.f - one has to consider the ownership modifiers of both x and f. The rules can be expressed as a type combinator that takes two ownership modifiers and returns the resulting ownership modifier. This type combinator is used to determine the type of field accesses, method call parameters and results. It is defined by the table 1.1:

*	peer	rep	readonly
peer	peer	readonly	readonly
rep	rep	readonly	readonly
readonly	readonly	readonly	readonly

Table 1.1: Table of the type combinator (first argument: left-most cell of the rows, second argument: top-most cell of the columns)

1.2.4 Subtyping

Two **peer**, **rep** or **readonly** types are subtypes if the corresponding classes or interfaces are subtypes in Java. In addition every **peer** and **rep** type is a subtype of the **readonly** type with the same class, interface or array element type.

A downcast can be used to change a read-only type into a read-write type. To be sure that a cast will not fail at runtime **instanceof** can be used to check the type.

1.2.5 Overriding and Overloading

Overloading of methods is forbidden if the signatures only differ in their ownership modifiers. For overriding methods, it is allowed that a non-pure method becomes pure.

1.2.6 Arrays

Arrays of reference types need two ownership modifiers: one for the array object and one for the type of the reference they store. Arrays of primitive types need only one modifier: the one for the array object.

1.2.7 Instance and Static Methods

Methods are executed in the context that contains the receiver object. In the same way the arguments and the return type are relative to the context of the receiver. If any argument type has a **rep** ownership modifier, the method can only be called on **this** as receiver.

Static methods cannot use **rep** types in their signature, because there is no receiver object.

1.2.8 Pure Methods

Methods can be marked with the keyword **pure** if they do not modify existing objects. The only methods that can be called on readonly references are pure methods. All parameter types of pure methods have a readonly ownership modifier. They can only be overridden by pure methods. Within pure methods, new objects can be created only by pure constructors; all field updates are forbidden; only pure methods can be called.

1.2.9 Static Fields

Since types are interpreted relatively to whoever uses the static field and with static fields there is no receiver object, all static fields should be **readonly**.

1.2.10 Object Creation and Constructors

Only peer and rep types are allowed for new expressions, because objects need to have an owner when created and **readonly** does not determine an owner. Constructors cannot use rep types in their signature, because the new object does not own objects that could be passed to the constructor.

1.2.11 Generics

In the draft of ”Generic Universe Types” [2] describe Dietl, Drossopoulou, and Müller a solution for the new generics mechanism of Java 5. In front of type variables (unqualified identifiers) no universe modifier can appear. It is only allowed to use universe modifiers in front of concrete types.

Chapter 2

Eiffel Languages Constructs

2.1 Overview

2.1.1 Overview

Eiffel and Java are both object-oriented programming languages, but there are some language constructs that differ between these two languages. Expanded types and agents are Eiffel constructs that are unknown in Java. Expanded types are values which are not references to objects but the objects themselves. Agents are objects that represent routines ready to be called.

In this chapter we will describe shortly the Eiffel language constructs and identify the constructs which differ between Eiffel and Java. The parts that are the same in both languages are only specified shortly, because there is probably no need for new ideas for the Universe type system. It should be possible to adapt the already developed solutions from Java to Eiffel. The interesting parts with differences between Eiffel and Java are described in more details. For two of these constructs we developed possible solutions for the Universe type system. In the "Design by Contract"-part, we also do a comparison between Eiffel and the "Java Language Specification" JLS from Sun and JML from jmlspecs.org.

We will first describe each Eiffel construct and then show how this is solved in Java and where the differences are. In the Eiffel part, we use the Eiffel language conventions like features or routines. In the Java part, we use Java conventions like methods and attributes.

The Eiffel part of this section is based on Bertrand Meyers "ECMA standard: Eiffel Analysis, Design and Programming Language" [5] and his book "Object-Oriented Software Construction" [12]. For the Java part, where version 5.0 is used, a textbook written by Manuel Oriol [15] and the Java API [17] are used.

2.1.2 Eiffel

Eiffel was originally designed, as a method of software construction and a notation to support that method, in 1985 [5]. As described in [6] the aim of Eiffel is to improve the quality of software systems and the productivity of the development process. It particularly promotes the production of software that has the following qualities: reliability (absence of bugs), extendibility (ease of change), reusability (reliance of libraries of packaged components), and portability (adaptability on many platforms with full source compatibility). Eiffel also makes it possible to produce compilers which generate extremely efficient code.

As written in [5], today, Eiffel is particularly suited for mission-critical developments in which programmer productivity and product quality are essential. In addition Eiffel is a popular medium for teaching programming and software engineering in universities.

2.1.3 Java

As written in [9] the goal of Java technology is to enable the development of secure, high performance, and highly robust applications on multiple platforms in heterogeneous, distributed networks. Primary characteristic is the simple, and object oriented language which is designed for creating highly reliable software. It provides extensive compile-time checking, followed by a second level of run-time checking. For being able to operate on multiple platforms in heterogeneous networks the Java programming language must be architecture neutral, portable, and dynamically adaptable. To accommodate the diversity of operating environments, the Java Compiler^{TM} product generates bytecodes, an architecture neutral intermediate format. These bytecodes will run on any platform and are executed by the so called Java virtual machine.

2.2 Classes and Types

2.2.1 Classes

A class in Eiffel is an implementation of an abstract data type and describes a set of run-time objects. In contrast to Java there are no special classes like inner classes or nested classes. In Eiffel it is possible to specify classes as **deferred** (see section 2.2.5 deferred routines), **expanded** (see section 2.2.3 expanded types) and **frozen**. Because the Universe type system focuses on types, there was no need to analyze classes any further.

2.2.2 Types

Eiffel is strongly typed for readability and reliability. Every entity is declared of a certain type, which may be either a reference type or an expanded type [5, p. 10]. Reference type means that the values of a certain type are references to objects, not the objects themselves. Expanded types are described in detail in section 2.2.3.

In Java, primitive types and reference types exist. Reference types are the same in both languages. But instead of primitive types like in Java, Eiffel handles the basic types **INTEGER** etc. with expanded types.

2.2.3 Expanded Types

For expanded types the values are not references to objects but the objects themselves. Expanded types are used for improving efficiency, providing better modeling and supporting basic types [12, p. 256]. Basic types are a closed set of types like **INTEGER**, **REAL**, **DOUBLE**, **CHARACTER**, and **BOOLEAN** which are defined in the standard libraries.

As mentioned in figure 2.1, there are two ways to declare a variable of an expanded type. It is possible to set a whole class E expanded by putting the keyword **expanded** before the declaration of class E. Then all variables declared of type E are automatically of type **expanded** E. On the other hand you can declare a variable of a non-expanded type **expanded** by using the keyword **expanded** in front of the type at the variable declaration.

The difference between reference types and expanded types affects the semantics of using an instance as source of an attachment: assignment or argument passing. This difference is called distinction between reference semantics and copy semantics. An assignment between two expanded types means copying the value of one object to the other object, while assignment between two reference types means copying the reference so that both references point at the same object. If the source type of an assignment is a reference type and the target type is an expanded type, the fields of the source object are copied to the target object. This assignment is only possible if the source object is not **Void**. In the other case, where the source type is an expanded type and the target type is a reference type, the target will be attached to a clone of the source object.

Variant 1:
Class declaration:

```
expanded class E
        ...
end
```

Variable declaration:

```
e: E
```

Variant 2:
Class declaration:

```
class C
        ...
end
```

Variable declaration:

```
c: expanded C
```

Figure 2.1: The two variants to declare a variable of an expanded type

The same difference is for comparison operations like a=b. In the case of reference types, it is a comparison of the references. If at least one of the two objects is of an expanded type, it is a comparison of the object contents.

The instances of a type **expanded** C are exactly the same as the instances of C. The only difference affects declarations using these types: an entity of type C denotes a reference which may become attached to an instance of C; an entity of type **expanded** C directly denotes an instance of C [12, p. 254]. An object O is said to be composite if one or more of its fields are themselves objects - called subobjects of O [12, p. 254]. The little example in figure 2.2 illustrates the difference between expanded types and reference types. The declaration of ”ref” means that every instance of COMPOSITE ”knows about” an instance of C (unless ref is **Void**). Otherwise the declaration of ”sub” means that every instance of COMPOSITE ”contains” an instance of C. An important difference between these two is that the ”contains” relation as provided by expanded types does not allow sharing of contained elements, whereas the ”knows about” relation allows two or more references to be attached to the same object [12, p. 256].

Another property of expanded entities is that they can never be **Void**. Expanded types cannot contain cycles (for example it is not permitted for a class C to have an attribute of type expanded D if class D has an attribute of type expanded C).

Java knows no expanded types, but there is a fixed set of primitive types (**byte**, **short**, **int**, **long**, **float**, **double**, **char**, **boolean**). For the user it is not possible to declare more primitive types, while in Eiffel the user can write own expanded types.

2.2.4 Root Types ANY and NONE

On top of the inheritance tree in Eiffel is the class **ANY**. There is also a class at the bottom of the inheritance structure called **NONE**. This class is only theoretical existent, but serves two practical purposes. First **Void**, which denotes the void reference, is by convention of type **NONE** and can be assigned to an entity of any reference type. The second use of **NONE** is exporting a feature to **NONE** which means that the feature is secret and unavailable to any other class and also other instances of the same class.

In Java the class ”Object” is on top of the inheritance tree which is comparable with **ANY**. But in contrast to Eiffel there is no common bottom type. The two described advantages of **NONE** are solved otherwise in Java. For denoting references, that aren’t attached to an object,

```
class COMPOSITE feature
        ref: C
        sub: expanded C
end
```

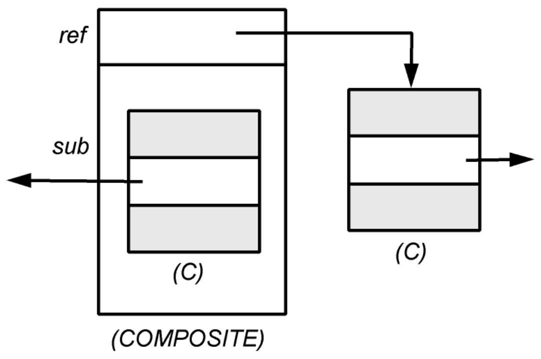


Figure 2.2: Difference between references and expanded types [12, p. 255]

the keyword **null** of the special null type is used. The null reference can always be cast to any reference type [8]. For making a member unavailable in Java, you can use the keyword **private**. But in contrast to Eiffel, **private** members are available for instances of the same class.

2.2.5 Deferred Classes

Deferred classes are abstract classes that are not fully implemented. A class is **deferred** if it has at least one deferred routine. If a routine is declared as **deferred** in a class C, it means that there is no implementation of this routine in this class. The routine should be implemented by the subclasses. You cannot instantiate objects of a deferred class, but you can assign an instance of a non-deferred descendant to a variable of a deferred type. The advantages of deferred classes are that you can put pre- and postconditions for a deferred routine and invariants for deferred classes, which must be followed by the subclasses.

In Java, the concept of abstract classes and abstract methods are equal to the deferred classes and routines in Eiffel.

2.3 Genericity, Arrays, and Tuples

2.3.1 Genericity

Genericity is a mechanism for defining parameterized module patterns, whose parameters represent types [12, p. 96]. With genericity you are able to define a single module pattern of the form **LIST**[G] where G is called formal generic parameter and represents an arbitrary type. By using a type as actual generic parameter instead of the formal generic parameter G, as in **LIST**[**INTEGER**], you achieve a generically derived type.

Eiffel allows generic classes in two different forms as you can see in figure 2.3. In the unconstrained form you can use any arbitrary type as actual generic parameter. The constrained form of genericity allows giving a constraint for the actual generic parameter. It has to be a descendant of the type declared with the formal generic parameter.

Eiffel allows the use of multiple formal generic parameter in one generic class, for example HASH_TABLE[G, KEY]. It is also possible to use a generic class as actual generic parameter like VECTOR[VECTOR[G]]. In Eiffel, LIST[B] is a descendant of LIST[A] if B is a descendant of A

Unconstrained form:

```
class LIST[G] ...
 ...
 my_int_list : LIST[INTEGER]
my_account_list: LIST[ACCOUNT]
```

Constrained form:

```
class VECTOR[G−>ADDABLE] ...
 ...
my_int_vector: VECTOR[INTEGER]
my_real_vector: VECTOR[REAL]
```

Figure 2.3: The two different forms of generic classes

[5, p.84]. On the other hand LINKED_LIST[A] is a descendant of LIST[A], if LINKED_LIST[G] is a descendant of LIST[G] where G denotes the formal generic parameter.

In Java generics exists since version 5.0. As written in [1], it is possible to constrain the formal generic parameter by a supertype like in Eiffel. You can also use a generic class as actual generic parameter. In contrast to Eiffel, it is not the case that List<B> is a subtype of List<A>, if B is a subtype (subclass or subinterface) of A.

2.3.2 Arrays

In Eiffel arrays are handled with genericity. An array is just a container object, an instance of a generic class which is called ARRAY.

In Java you can create a new array type by adding " [] " after a type. Array types are covariant. This means that if the class Sub is a subclass of another class Super then "Sub[]" is a subclass of "Super[]" too. On top of this array inheritance tree is "Object[]" which inherits finally from "Object". Because of this covariance the Java Virtual Machine has to do runtime checks when storing objects in arrays to guarantee type safety. Arrays in Eiffel are covariant too as explained in section 2.3.1.

2.3.3 Tuples

A tuple is a simple extension to the notation of classes. Instances of tuples have sequences of elements of which each has a type. Tuples provide a simpler alternative to classes when you don't need specific features, just a sequence of values of given types.

Java 5 supports generics which is not as powerful as tuples. A generic class has a fixed number of generic parameters while you can use the tuples in Eiffel like a generic class with a variable number of parameters [7]. This means that Java has no construct similar to tuples.

2.4 Inheritance

2.4.1 Multiple Inheritance

Eiffel allows multiple inheritance. It is possible to distinguish between two cases of inheritance. Conforming inheritance means subclassing, which is both code reuse and subtyping. Non-conforming inheritance means only code reuse. Thus it is not possible to assign values of the new type to variables of the parent type. Due to name conflicts by multiple inheritance, it is possible to rename features (see example in figure 2.4).

Java knows only single inheritance but you can implement multiple interfaces. The concept of interfaces is not known by Eiffel. Renaming of methods or fields is not possible in Java.

```
class
        C
inherit
        A
                rename
                        foo as fooa
                end
        B
                rename
                        foo as foob
                end
feature
        ...
```

Figure 2.4: Multiple inheritance in Eiffel with renaming

2.4.2 Overloading and Overriding

In Eiffel ”overloading” (multiple methods with the same name as long as the signatures differ) is not possible. But overriding (redefining the implementation of an inherited method) is possible with the keyword **redefine**. Java supports both overloading and overriding.

2.4.3 Polymorphism and Dynamic Binding

Polymorphism means the ability to take several forms. In object-oriented development what may take several forms is a variable entity or data structure element, which will have the ability, at run time, to become attached to objects of different types, all controlled by the static declaration [12, p. 467].

The rule known as dynamic binding implies that the dynamic form of the object determines which version of the operation to apply [12, p. 480]. Polymorphism and dynamic binding are integrated in both languages.

2.5 Assertions

2.5.1 Design-by-Contract

In Eiffel you have the possibility to use assertions. These are routine preconditions, routine postconditions, and class invariants. Preconditions are requirements that clients must satisfy whenever they call a routine. Postconditions express conditions that the routine guarantees on return, if the precondition was satisfied on entry. The invariant must be satisfied by every instance of the class whenever the instance is externally accessible. In figure 2.5 you can see an example of the Eiffel Design-by-Contract.

Since J2SE 1.4 Java supports assertions too. You can everywhere in your code put an assert-statement which tests the current state. With these assertions you can simulate preconditions, postconditions and invariants [11]. Otherwise Design-by-Contract is integrated in JML. Like in Eiffel there are method preconditions and postconditions as well as type invariants [10].

2.5.2 Inheritance and Assertions

Assertions in Eiffel are inherited and must be held by the subclasses. When redefining a routine the precondition of the redefined routine can only be equal or weaker. The postcondition can only be equal or stronger. Invariants can only be equal or stronger.

```
class STACK1 [G] feature -- Access
        count: INTEGER
        capacity: INTEGER
        item: G is
                require
                        stack_not_empty: not empty
                do ... end
feature -- Status report
        empty: BOOLEAN is
                        do ... end
        full : BOOLEAN is
                        do ... end
feature -- Element change
        put (x: G) is
                require
                        stack_not_full : not full
                do
                        ...
                ensure
                        stack_not_empty: not empty
                        x_inserted_at_top : item = x
                        count_incremented: count = old count + 1
                end
        remove is
                require
                        stack_not_empty: not empty
                do
                ...
                ensure
                        stack_not_full : not full
                        count_decremented: count = old count - 1
                end
invariant
        count_non_negative: 0 <= count
        count_bounded: count <= capacity
end
```

Figure 2.5: Code example Design-by-Contract in Eiffel

The assertions in Java are not inherited. But in JML, a subclass inherits specifications such as preconditions, postconditions, and invariants from its superclasses and interfaces that it implements [10].

2.6 Agents

2.6.1 Agents

The section about agents is based on [13]. An agent is a way to define an object that represents a certain routine, ready to be called. You can pass the agent object around to other software elements, which can use this object to execute the operation whenever they want. The concept of agent separates the place of an operation's definition from the place of its execution. The construction time of an agent object is the time of evaluation of the agent expression defining it. Its call time is when a call to its associated operation is executed. For a normal routine call the two moments are the same. For an agent we will have zero or one construction time and zero or more call times. The definition of an agent can be incomplete, since you can provide any missing details at the time of any particular execution.

If you have a procedure like this:

```
f(a, b, c)
```

The corresponding agent can easily be built by putting the keyword **agent** before the function call:

```
s := agent f(a, b, c)
```

As mentioned above the arguments of an agent can be set at definition time (called closed operands) or provided at the time of each actual call (called open operands). Here is an example of a definition of an agent with one open operand (symbolized with "?") and the two closed operands b and c:

```
t := agent f(?, b, c)
```

This agent calls the procedure "f" of the current object. Instead of calling a routine on **Current**, it is also possible to specify the target object:

```
u := agent obj.f(a, ?, ?)
```

The target object can also be a so called open target. For this you can put the target type in braces instead of the target:

```
v := agent {T}.f(a, ?, ?)
```

This means that for a call of this agent the target object must be passed as an argument and it must be of type T.

The type of agent objects is **ROUTINE**, **PROCEDURE**, or **FUNCTION**. The class diagram of these generic classes is illustrated in figure 2.6. The formal generic parameters are needed for making the agent mechanism statically type-safe.

If the declaration of the procedure f in the examples above is as follows:

```
class T
feature
        f(a: A; b: B; c: C) is
                do
                        ...
                end
end
```

The types of the agent objects are as follows:

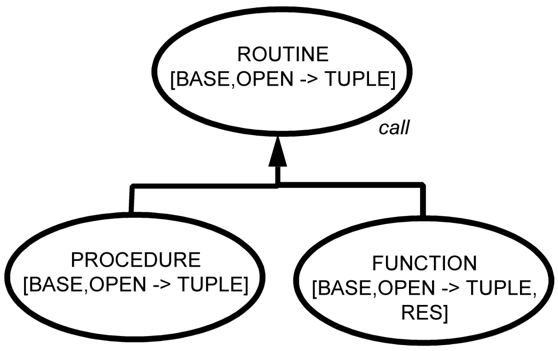


Figure 2.6: Class diagram of the three agent classes with their formal generic parameters. BASE means the type of the target object. OPEN is the tuple of the types of the open operands and RES is the result type.

```
s: PROCEDURE[T, TUPLE]
t: PROCEDURE[T, TUPLE[A]]
u: PROCEDURE[T, TUPLE[B, C]]
v: PROCEDURE[T, TUPLE[T,B,C]]
```

The class **ROUTINE** has a feature "call" that allows to call agents. The call to the agent is written as follows:

```
my_agent.call([open_arguments])
```

As mentioned above the arguments must contain the target object if it is not specified at construction time. The call of the examples above looks as follows:

```
obj1: T; val1: A; val2: B; val3: C
s.call ([])
        -- f(a, b, c)
t.call ([val1])
        -- f(val1, b, c)
u.call ([val2, val3])
        -- obj1.f(a, val2, val3)
v.call ([obj1, val2, val3])
        -- obj1.f(a, val2, val3)
```

There are some iterators available for all traversable structures as for example "for_all". This boolean-valued function determines whether a certain property, passed in form of a call expression with one open argument, holds for every element of a sequential structure. Two examples with "for_all" are given in figure 2.7.

The construct of agents has no counterpart in Java.

2.7 Special Features

2.7.1 Once Routines

The first time a once routine is called during a system's execution, it executes its body. Every subsequent call executes no instruction at all, but terminates immediately returning the result computed the first time around. Once routines can be redefined by any descendant into once or non-once routines. Once routines are like constants that at the first invocation are executed and afterwards cannot be changed any more.

In Java there is no direct construct like once routines, but it is possible to simulate once routines with static methods and static fields as you can see in the example in figure 2.8. Simple examples

```
class C
feature
        is_positive (x: INTEGER): BOOLEAN is
                do
                        Result := (x > 0)
                end
        ...
        my_integer_list : LIST[INTEGER]
        ...
        all_positive := my_integer_list . for_all (agent is_positive)
        -- same as: all_positive := my_integer_list . for_all (agent is_positive (?))
        ...
end


class EMPLOYEE
feature
        is_married: BOOLEAN is
                ...
        my_employee_list: LIST[EMPLOYEE]
        all_married := my_employee_list. for_all (agent {EMPLOYEE}.is_married)
```

Figure 2.7: Two examples for using agents with the iterator for_all

can be solved with a static final field. If the once routine is more complex, it can be simulated with a private static field that stores the return value and a public method that at the first invocation executes the once routines body, stores the result in the private field and afterwards returns the value of this field. This is similar to a singleton pattern.

In these two examples we use static methods to simulate once routines. A difference is that once routines are bound dynamically, but static methods are bound statically. It is possible to solve this problem by using a dynamically bound method that returns the value of a static field as you can see in the example in figure 2.9 (Eiffel part) and 2.10 (Java part).

2.7.2 Create Procedures

In Eiffel you can define multiple creation procedures which can have names like all other procedures. To decide which constructor is taken for a create-statement, the language rely on the name of the procedure. In Java all constructor methods must have the name of its class and different signatures.

2.7.3 Constant Attributes

Constant attributes in Eiffel are defined in a class, have a fixed value and are the same for all instances of this class. It is only possible to declare constant attributes of basic types. These are the types like **INTEGER** or **REAL** which are defined in the standard libraries. For constants of other types, including all self defined expanded types, you must use once methods (see section 2.7.1). A constant attribute is declared as follows:

```
Var: INTEGER is 1
```

Static variables in Java are class variables. They are the same for all instances of this class. Final variables must be assigned a value at declaration. In the case of primitive types the value cannot be changed later. This means that static final variables for primitive types are class constants and comparable with the constant attributes in Eiffel.

Example 1: Eiffel:

```
i: COMPLEX is
        once
                !! Result.make(0,1)
        end
```

Java:

```
public class Complex {
        public Complex (int r, int i) {
                y = i;
                x = r;
        }
        private int y, x;
        public static final Complex i = new Complex(0,1);
}
```

Example 2: Eiffel:

```
once_res: COMPLEX is
        once
                // body of the once routine
        end
```

Java:

```
private static Complex once_res;
// current class have to ensure that once_res is modified only by the method once
public static Complex once() {
        if (once_res == null) {
                // body of the Eiffel once routine
                // save result in once_res
        }
        return once_res;
}
```

Figure 2.8: These two examples (a simple and a more complex one) show how once routines can be simulated in Java.

```
class A
create
        make_a
feature
        foo: STRING is
                once
                        Result := "A"
                end
        make_b is do ... end
end


class B inherit
        A
                redefine foo end
create
        make_b
feature
        foo: STRING is
                once
                        Result := "B"
                end
        make_b is do ... end
end

class ROOT_CLASS
create
        make
feature
        a: A
        b: B
        make is
                do
                        create a.make_a()
                        io.put_string(a.foo())
                        create b.make_b()
                        io.put_string(b.foo())
                        a := b
                        io.put_string(a.foo())
                        // output "B"
                end
end
```

Figure 2.9: Eiffel part of the example that shows how the dynamic binding of once routines can be simulated with Java methods. The Java part is in figure 2.10.

Java:

```
public class A {
        public A() {}
        private static String once_result;
        public String foo() {
                if (once_result == null) {
                        once_result = "A";
                }
                return once_result;
        }
}

public class B extends A{
        public B() {}
        private static String once_result;
        public String foo() {
                if (once_result == null) {
                        once_result = "B";
                }
                return once_result;
        }
}

public class Main {
        public static void main(String[] args) {
                A a;
                B b;
                a = new A();
                b = new B();
                System.out.println(a.foo ());
                System.out.println(b.foo ());
                a = b;
                System.out.println(a.foo ());
                // output "B"
        }
}
```

Figure 2.10: Java part of the example that shows how the dynamic binding of once routines can be simulated with Java methods. The Eiffel part is in figure 2.9.

2.8 Exceptions

2.8.1 Exceptions

A routine fails because of some specific event (arithmetic overflow, assertion violation...) called exception that interrupts its execution. If the cause of the exception is an assertion violation, it is called a failure. A routine in Eiffel can have a rescue clause that attempts to bring the current object to a stable state after the appearance of an exception. The rescue clause can include the retry instruction to retry to execute the method body.

In Java, the throw statement allows to interrupt execution and follow the call stack until a corresponding catch block is found. If a finally clause exists, it is always executed. After that, the program continues execution.

2.9 Concurrent Programming

2.9.1 Concurrent Programming

With the Simple Concurrent Object-Oriented Programming (SCOOP) Model Eiffel has a high-level concurrency mechanism. With the keyword **separate** you can declare that two objects are handled by different processors. Analogous to preconditions you can define waiting conditions for routines on separated objects.

The concurrent model of Java with threads, synchronization, and monitoring is quite different.

2.10 Conclusion

Eiffel and Java are in many parts similar. Other parts like the once routine can be mapped from Eiffel to Java. The two important constructs, that exist only in Eiffel and are interesting for the Universe type system, are expanded types and agents. For both we will show possible solutions in the next chapter.

Chapter 3

Universe Type System for Eiffel

3.1 General Solution

3.1.1 Overview

In this chapter we describe our developed Universe type system for Eiffel. In the first section we show a possible syntax extension for Eiffel. Subsequently we explain which part of the existing Universe type system for Java can be taken. Afterwards we show possible solutions for the treatment of expanded types and agents.

3.1.2 Syntax

The Eiffel syntax style is described in [5, p. 8] as focusing on readability, not overwhelming the reader with symbols, and using simple keywords, each based on a single English word. Due to this syntax style, it makes sense to take the syntax presented by the Universe type system for Java. **peer**, **rep**, and **readonly** as type modifier keywords as well as **pure** as keyword for side-effect free routines seems to be a good choice. Like in the Universe type system for Java we set the ownership modifier before the type of an object:

a: **peer** COMPLEX
is_in_bound (l: **rep** COMPLEX; u: **rep** COMPLEX; c: **readonly** COMPLEX)

The proposed syntax is also similar to the syntax of detachable types (types which permit void values) in Eiffel (more in [14]), where the declaration is as follows:

a: ? COMPLEX

To let the code be syntactically correct Eiffel code, that can be complied by the standard Eiffel compiler, you have to put the ownership code into comments. Since a special compiler should differ between normal comments and ownership comments, there is a need to mark the ownership comments with a special character, for example "@". In the same way it is done in the Java version where they use the JML specification syntax:

/∗@ **peer** @∗/ Node prev;

Because in Eiffel only line comments and no block comments exist it is necessary to use a new line for each comment. This would look as follows:

a:
−−@ *peer*
COMPLEX

is_in_bound (l:
−−@ *rep*

COMPLEX; u:
−−@ *rep*
COMPLEX; c:
−−@ *readonly*
COMPLEX)

This is very bad readable and eventually for future work on a Universe type system for Eiffel, there is a need for block comments.

In the following examples we will write the ownership modifiers for readability reasons without comments.

3.1.3 Rules

The main concepts between Java and Eiffel are the same as shown in the last chapter. Therefore, the rules from the Java Universe type system as presented shortly in section 1.2 can be taken. The plain rules can be read in [3] and [4]. In addition we will present solutions for expanded types and agents which have no counterpart in Java in section 3.2 and 3.3.

3.1.4 Command-Query Separation

Eiffel differs between command and query features. A command serves to modify objects and is implemented as a procedure. A query returns information about objects and may be implemented either as an attribute or as a function. The command-query separation principle says that functions should not produce abstract side effects. Informally this means that asking a question should not change the answer.

The Universe type system differs between pure and non-pure methods. Pure methods don't modify the existing objects. According to the command-query separation principle, queries in Eiffel should be side-effect free which means that they should be pure routines.

3.1.5 Once Routines

As written in section 2.7.1, Eiffel once routines can be simulated in Java with a dynamically bound method and a static field. Since in the Java Universes static fields must be **readonly**, once routines can only have **readonly** as return type.

3.2 Expanded Types

3.2.1 Overview

We first shortly present three possible solutions for the treatment of expanded types. For every solution we also list the advantages and disadvantages. Subsequently follows an evaluation of all solutions. Afterwards we analyze some special aspects of the best solution like how the assignment could work.

For expanded type solutions we have to include the concept of expanded types that the entities denote not references to instances but instances themselves. In addition we have to consider that expanded types cannot be referred by a reference from outside. Therefore, the aliasing problem, which to solve is the task of the ownership model, doesn't exist with expanded entities.

3.2.2 Solution 1: Like Reference Types

This is the most general solution. It treats expanded types like reference types.

You can supplement each entity of an expanded type with one of the three ownership modifiers **rep**, **readonly** and **peer** as you can see in the example in figure 3.1. Like for reference types

peer is taken as default value. With this solution there is no difference between expanded types and reference types.
Advantages:

- Allows a uniform handling of expanded types and reference types.
- By allowing all modifiers, this solution is very flexible and leaves the programmer all possibilities open.

Disadvantages:

- The concept of expanded types, that they aren't references to objects but the objects themselves, is broken by this solution.

We can split this solution into three different cases: one when the expanded entity is of type **peer**, one when it is of type **rep** and one when it is of type **readonly**.

In the case where it is of type **rep** the expanded entity is owned by the composite object. Therefore, a good encapsulation of the object can be achieved, which means that it is well protected against modifications from outside. A modification of the expanded entity can only be done by the composite object and the rep attributes of the composite object. Since a **rep** reference of the composite object and a **rep** reference of the expanded object don't point into the same context, the possibilities of assignments are quite limited.

When the expanded entity is of type **peer**, which is the default value, the composite object and the expanded entity are in the same context. The expanded entity is bad encapsulated as it is modifiable by the same objects as the composite object. The semantics of expanded types, that they are part of the composite object, is broken by declaring an expanded object as **peer**.

A special case is when the expanded entity is of type **readonly**. Because there is no reference from outside to the expanded entity and the composite object has only a readonly connection to this entity, the subobject is not modifiable.

An extension of this solution could be that we insert a forth possible modifier called "this" which means that the expanded entity is in the same context as the composite object and both are owner of the same context. The expanded entity is like a part of the object and their attributes are handled like attributes of the composite object. The difference to a **peer** expanded object is, that with the modifier "this" the composite object and the expanded object are owner of the same context, while with **peer** both are owner of an own context. With the modifier "this" we consider, like with the modifier **rep**, that the expanded object belongs to the composite object. The difference is the better encapsulation of an **rep** expanded object compared to a "this" expanded object. This can be illustrated with a little example. Assume that we have an object x that is **peer** to the composite object and the expanded entity has a peer attribute a. In the solution with "this" as modifier, x can modify a because they are in the same context. This is not possible in the solution with the modifier **rep**. Because a is **rep** of the composite object and this is peer to x, x has no read-write reference to a.

3.2.3 Solution 2: As Part of the Object

The second solution is to handle every expanded entity as part of the object like with the above introduced modifier "this". Since there is no choice for a modifier there is no need to introduce the modifier "this" and write it.

This solution considers that there is no aliasing problem with expanded types and therefore no need for a complex solution. The attributes of the expanded entity will be handled like they would be directly attributes of the composite object. In the example of figure 3.2 d1 of the subobject is handled like a1, both as peer of the object. More interesting is that the context owned by the composite object is the same as the context owned by the subobject. This means that the references a2 and d2 show into the same context.
Advantages:

```
class COMPOSITE
feature
        a1: peer A
        a2: rep B
        a3: readonly C
        sub_peer: expanded peer D
        sub_rep: expanded rep D
        sub_readonly: expanded readonly D
end

class D
feature
        d1: peer E
        d2: rep F
        d3: readonly G
```

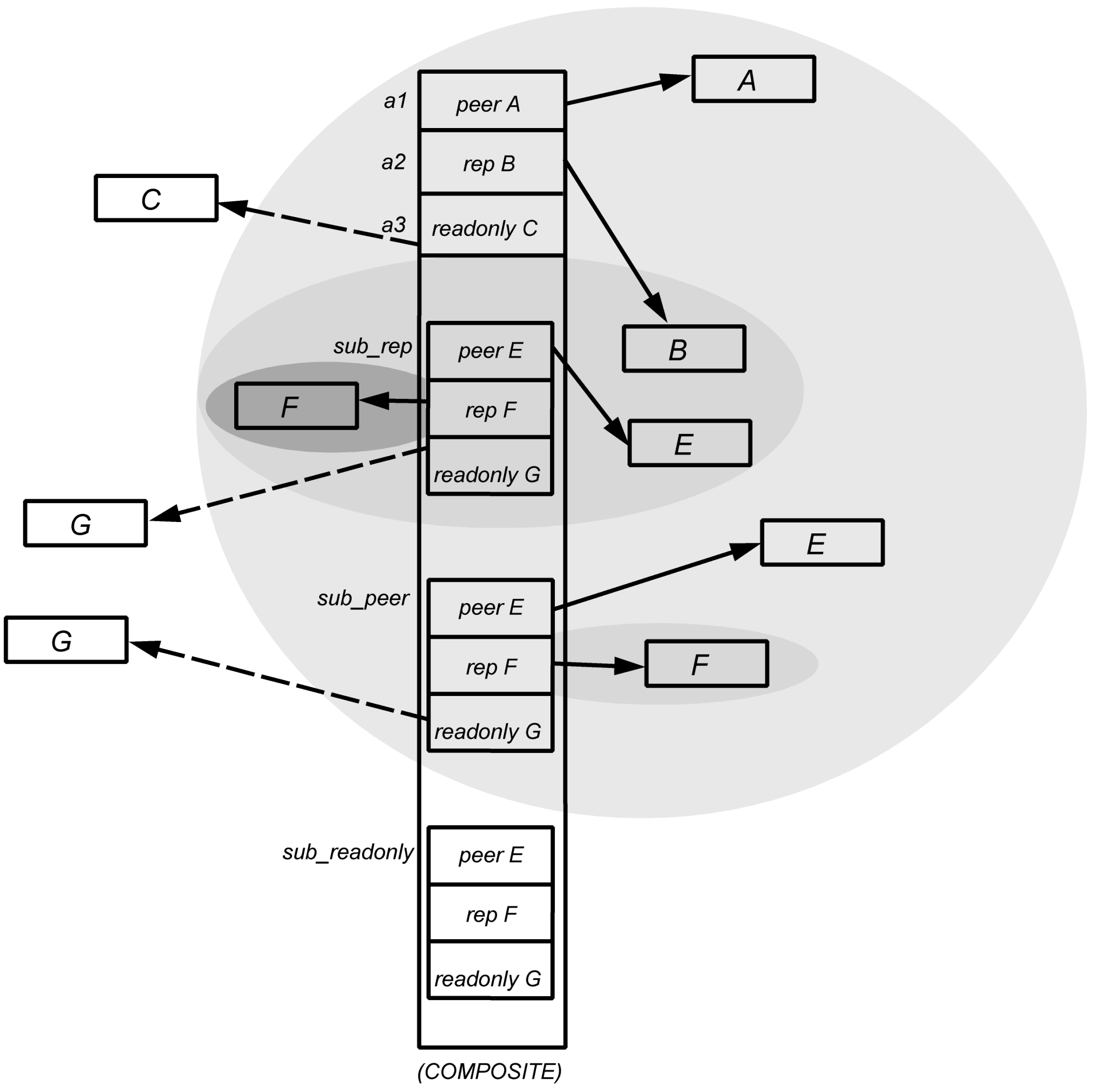


Figure 3.1: Solution 1 for expanded types: Like reference types

- The solution considers the main idea behind expanded types being objects not references to objects.

Disadvantages:

- Since there is no choice of modifiers for expanded types, this solution offers not much flexibility.

A possible extension could be that, like in the solution 1, all modifiers are allowed, but that "is part of the object" is the default value.

3.2.4 Solution 3: Always rep

With this solution all entities of an expanded type are of ownership type **rep**.

Every value of an expanded type is of ownership type **rep** and its owner is the composite object. It is not allowed to specify ownership types before an expanded type. An example using this solution is illustrated in figure 3.3.
Advantages:

- The expanded object is well protected against modifications from outside.

Disadvantages:

- Since there is no choice of modifiers for expanded types, this solution offers not much flexibility.
- There are limited possibilities for an assignment to the expanded object.

As in the second solution, a possibility is to extend it, so all modifiers are allowed and **rep** is the default value.

3.2.5 Evaluation

Solution 1 is more flexible than solution 2, but it is also very complicated. Since there is no aliasing problem with expanded objects, this solution gives no real advantages in comparison to the second solution. Therefore, it is better to use the less complex solution 2.

Solution 3 is similar to solution 2. In solution 3 the expanded entity is better encapsulated because it is **rep** to the composite object. But the protection of solution 2 is enough. In solution 3 the expanded entity can only be modified by the composite object and the rep objects of the composite object. In solution 2 a modification of the expanded object is also possible by the owner of the composite object and all objects that are in the same context as the composite object. Since the expanded entity can protect its attributes by using the modifier **rep**, it is no problem to use the less restrict solution 2.

The comparison with the other possibilities shows, that solution 2 is the best. But there are also semantic reasons for this solution. The semantics of expanded types, to be an object instead of a reference, is important. Because a subobject cannot be referenced from outside the problem of aliasing doesn't exist with expanded types. Furthermore, as explained in the next section, the assignment for this solution is easy.

3.2.6 Detailed Analysis of Solution 2

In this section we will consider some part of solution 2 in more detail. First we show how assignments of expanded entities can be solved.

Remember that assignment with expanded types is always done by copying the values of one object to the other object. When you assign new values to an object the ownership types must be correct. This means that for an assignment with expanded types that, in case of rep or peer attributes, the owner of the source and the target attribute must be the same. What this means

```
class COMPOSITE
feature
        a1: peer A
        a2: rep B
        a3: readonly C
        sub: expanded D
end

class D
feature
        d1: peer E
        d2: rep F
        d3: readonly G
```

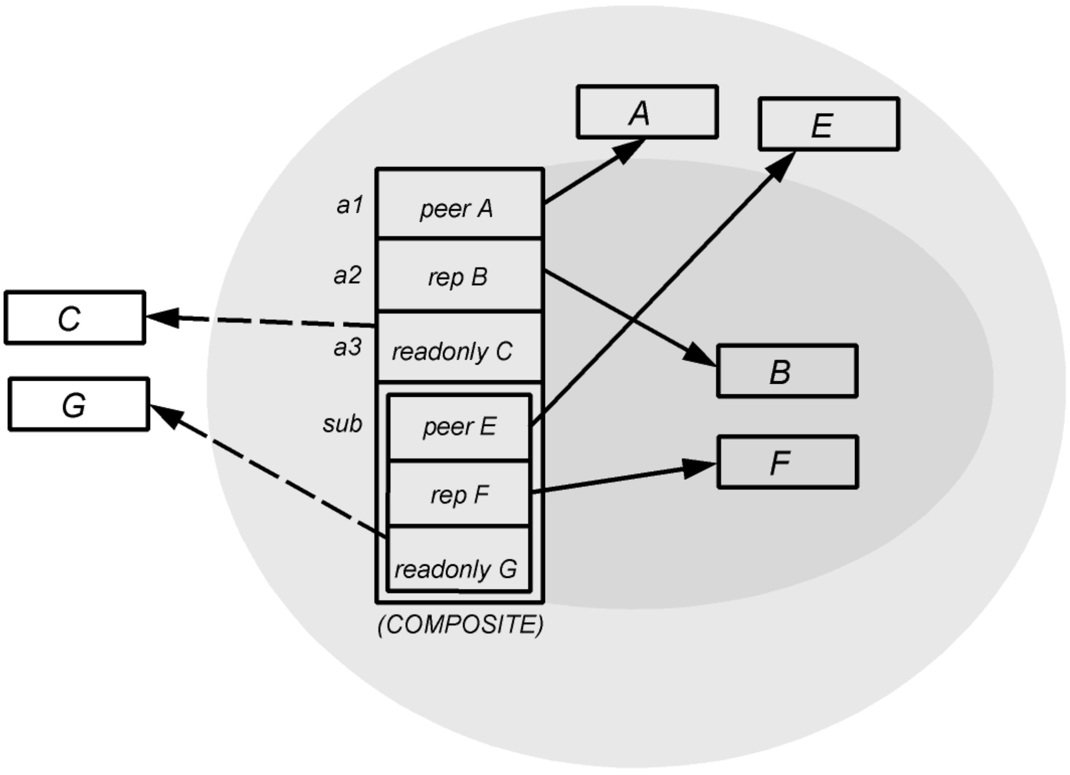


Figure 3.2: Solution 2 for expanded types: As part of the object

```
class COMPOSITE
feature
        a1: peer A
        a2: rep B
        a3: readonly C
        sub: expanded D
end

class D
feature
        d1: peer E
        d2: rep F
        d3: readonly G
```

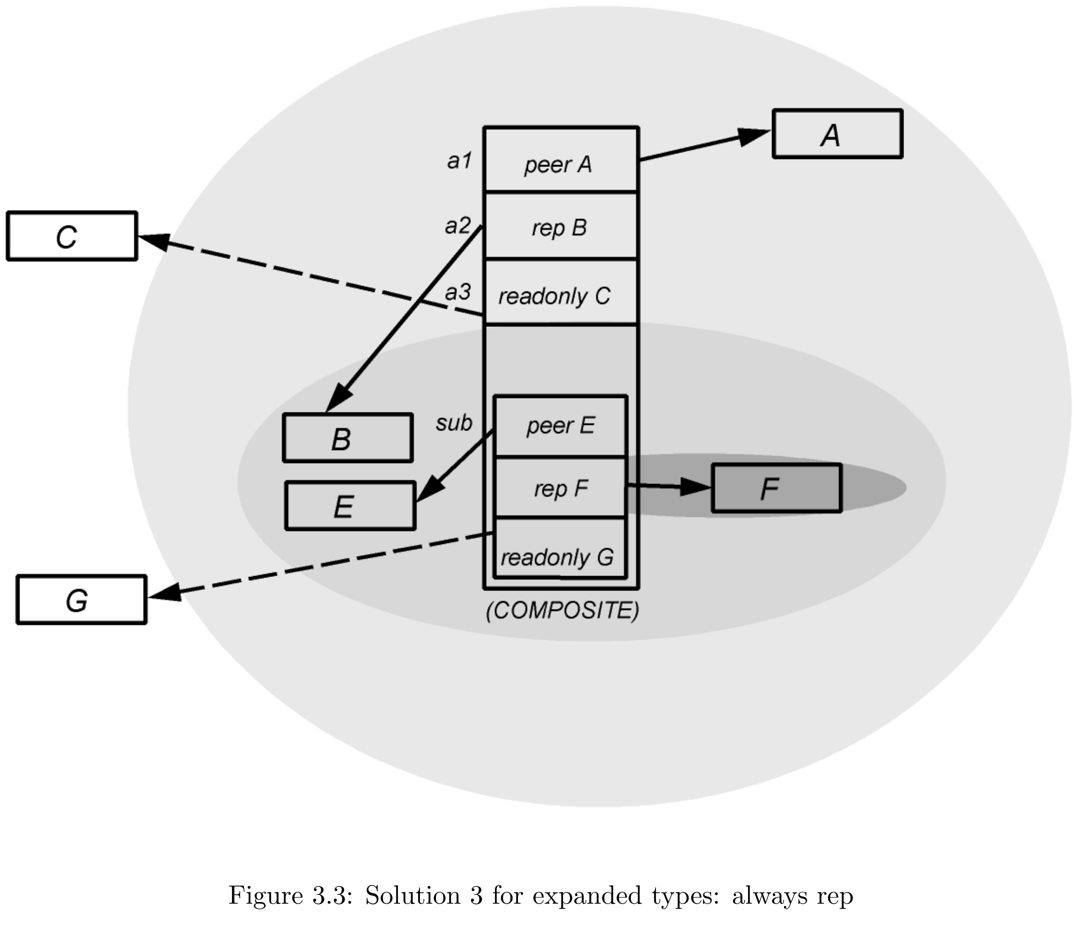


Figure 3.3: Solution 3 for expanded types: always rep

will be shown with an example. We handle three different cases with each of the three ownership modifiers. If you have an expanded entity with attributes of different ownership types (this means a composition of the following three examples), the assignment must be correct for each of this.

The starting position of the example is shown in figure 3.4. The class WORKSTATION has three expanded entities. These three entities each have an attribute of one of the three ownership types. In the following part we describe how the assignment of the expanded entities can be solved.

At first we study the assignment of the expanded entity with a readonly attribute. This assignment is easy. Due to the fact that the target of the assignment has only a readonly attribute, the source reference can have any arbitrary ownership type (because **readonly** is a supertype of the two other ownership types as long as the corresponding classes are also in a subtype relation). It also doesn't matter which object is the owner of the source attribute. The following source code shows the assignment:

> w1: **peer** WORKSTATION
> w2: **peer** WORKSTATION
> w1.cpu := w2.cpu

The values of the marked entity in object w2 are copied to the target entity of object w1 (see in figure 3.5).

The assignment of expanded entities with peer attributes needs the attention that both peer attributes must be in the same context. Otherwise the assignment is not allowed. The code of the example is the following:

> w1: **peer** WORKSTATION
> w2: **peer** WORKSTATION
> w1.networkcard := w2.networkcard

In figure 3.6 you see, that the reference of the marked source object w2 was in the same context as the reference of the target object w1. This allows the assignment. After the assignment the peer reference of the target object points at an object in the same context which is correct.

More problematic is the assignment of expanded types if they have rep attributes. The problem is that the values of the source object should be in the same context as the values of the target object. Therefore, the following assignment is invalid:

> w1: **peer** WORKSTATION
> w2: **peer** WORKSTATION
> w1.keyboard := w2.keyboard

The correct assignment requires that the rep attributes of the source and the target object are in the context which is owned by the target object. On the other hand the rep attribute of the source object is in the context owned by the source object (see figure 3.7). The only legal possibility is, that the source and the target object are the same. An example for this configuration is shown in figure 3.8.

The solution 2 could also be achieved by saying that the keyword **expanded** is mandatory at the declaration also for attributes of expanded classes and **expanded** would be like a forth ownership modifier. Then the type **expanded** is handles like the **Current** reference. The extended type combinator table is shown in table 3.1:

*	expanded	peer	rep	readonly
Current / expanded	Current	peer	rep	readonly
peer	peer	peer	readonly	readonly
rep	rep	rep	readonly	readonly
readonly	readonly	readonly	readonly	readonly

Table 3.1: Table of the extended type combinator (first argument: left-most cell of the rows, second argument: top-most cell of the columns)

Class declarations:

```
class WORKSTATION
feature
        cpu: expanded CPU
        networkcard: expanded NETWORKCARD
        keyboard: expanded KEYBOARD
end

class CPU
feature
        producer: readonly PRODUCER
end

class NETWORKCARD
feature
        connected_to: peer WORKSTATION
end

class KEYBOARD
feature
        keys: rep KEY_LIST
end
```

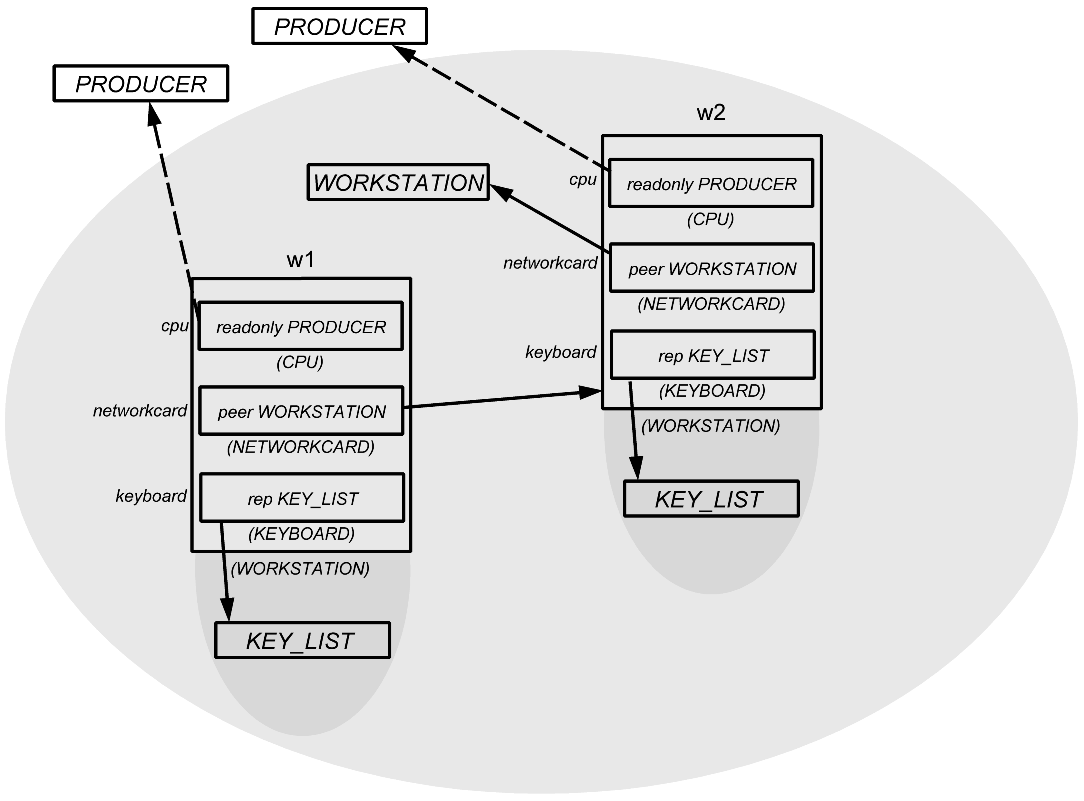


Figure 3.4: Starting position of assignment example with solution 2 for expanded types

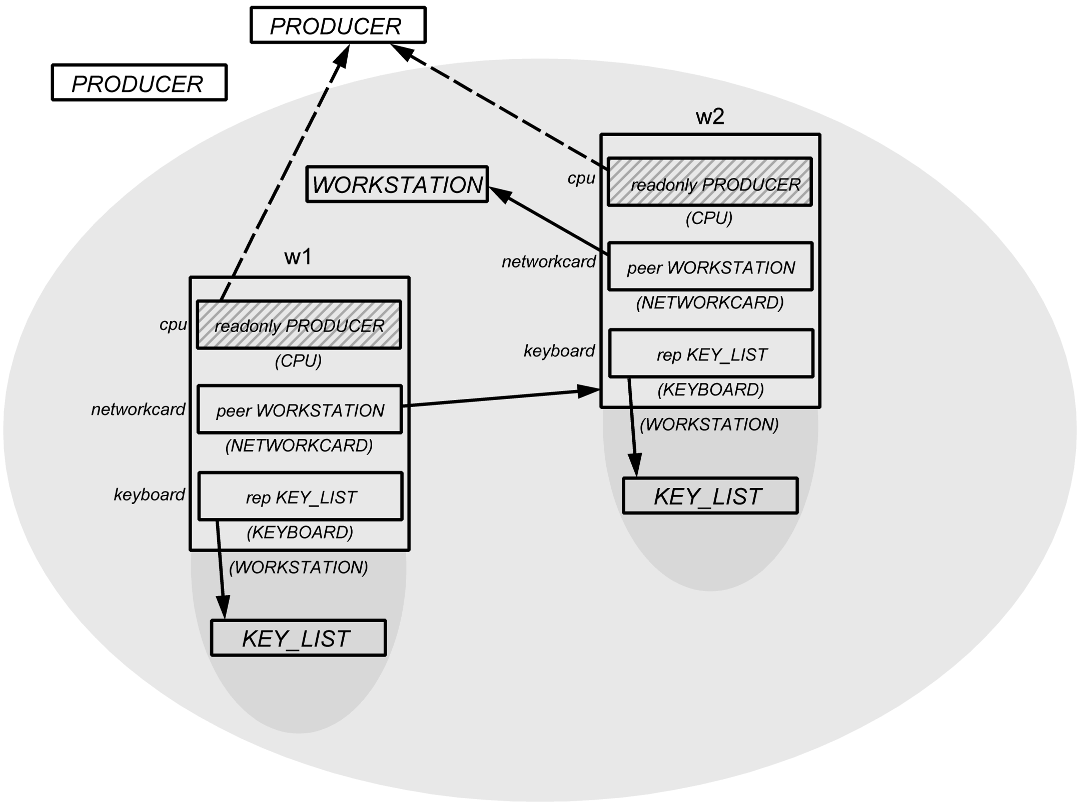


Figure 3.5: Example assignment of expanded entities with readonly attributes: After the assignment the values of the marked object on the right have been copied to the other one.

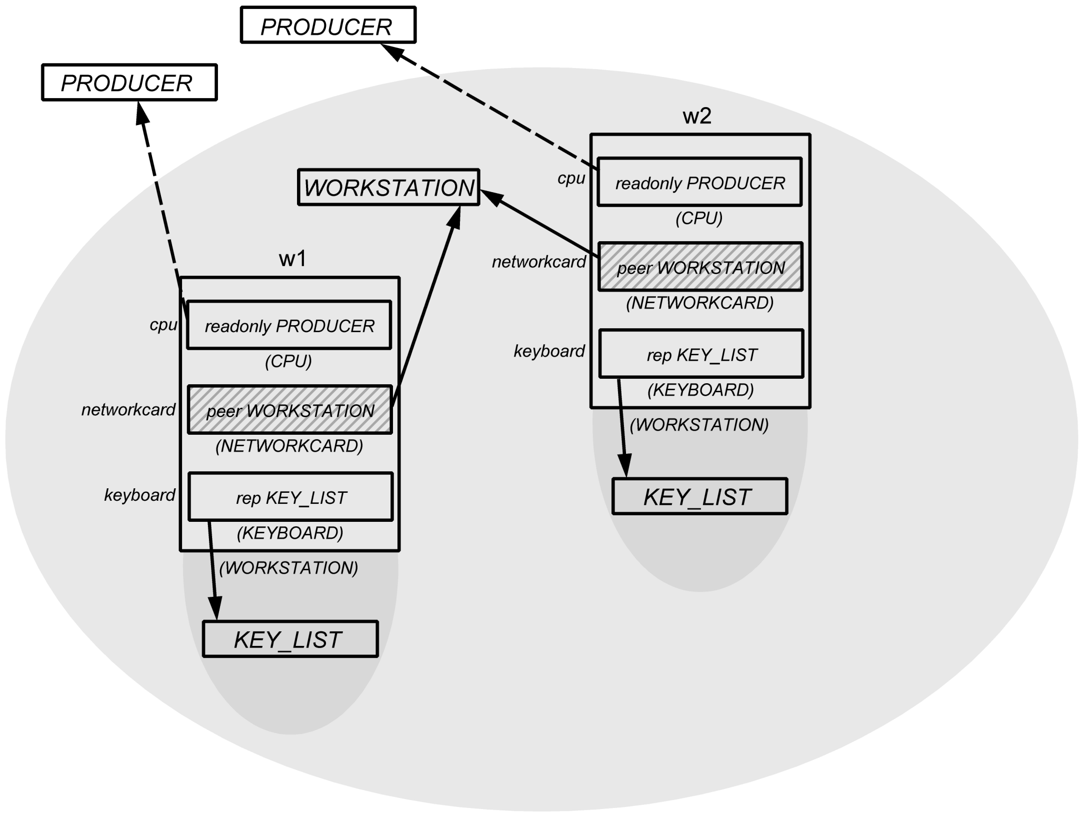


Figure 3.6: Example assignment of expanded entities with peer attributes: After the assignment the values of one marked object have been copied to the other one.

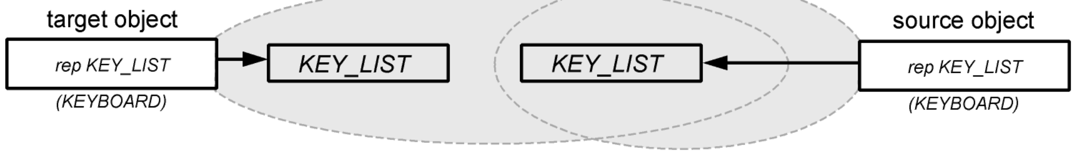


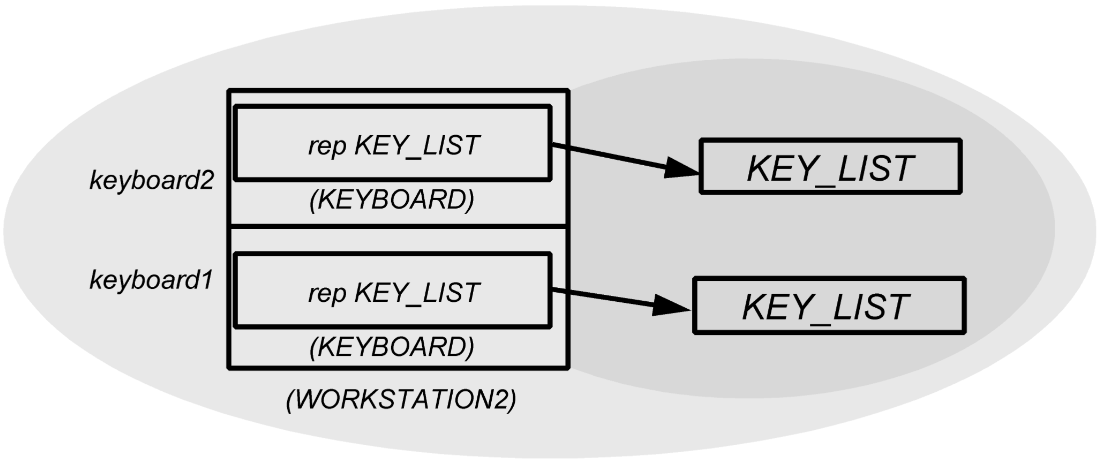
Figure 3.7: Problem with the assignment of expanded entities with rep attributes: the attribute of the source object should be in two different contexts which is not allowed.

```
class WORKSTATION2
feature
        keyboard2: expanded KEYBOARD
        keyboard1: expanded KEYBOARD
end

class KEYBOARD
feature
        keys: rep KEY_LIST
end

 ...

w1: peer WORKSTATION2
w1.keyboard1 := w1.keyboard2
```



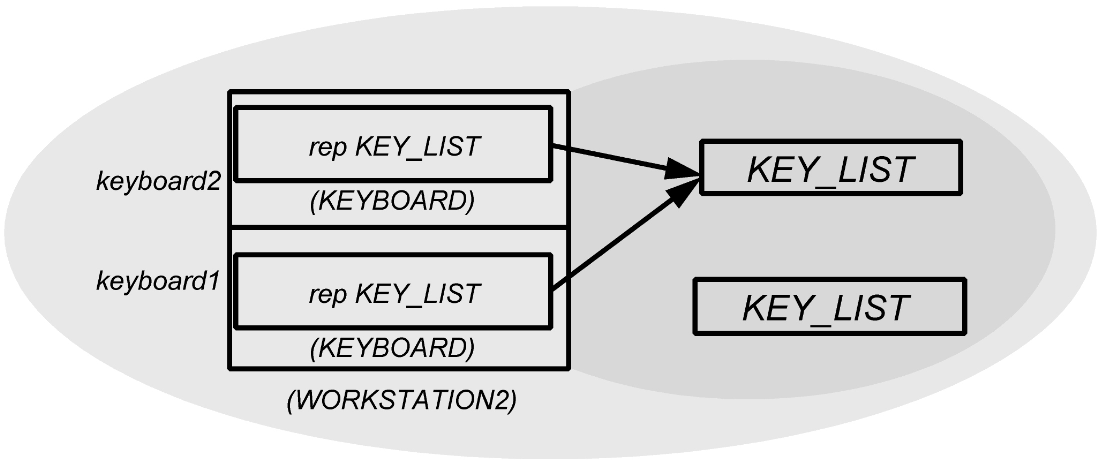


Figure 3.8: Possibility for assignment with expanded entities with rep references: The upper image shows the situation before the assignment, the lower image shows the situation after the assignment.

3.3 Agents

3.3.1 Overview

Agents is the method to save the arguments of a feature call to call them later. This is done by packing all needed informations into an object called the agent object. It is important for the treatment of agents to consider that an agent is an object. This means that they are like all other objects in a context and have at most one owner.

The keyword **agent** creates a new agent. Therefore, it is quite similar to the keyword **create**. Both creates a new object. This gives some inputs about the handling of agents in the Universe type system. In addition the ownership solution should be similar to the one of procedure calls.

3.3.2 Agent Types

As mentioned in section 2.6, agents are of one of these three types:

```
ROUTINE[BASE, OPEN −> TUPLE]
PROCEDURE[BASE, OPEN −> TUPLE]
FUNCTION[BASE, OPEN −> TUPLE, RES]
```

PROCEDURE and FUNCTION inherit from ROUTINE. The class ROUTINE contains the fundamental features for the agent mechanism. The most important are the following:

```
deferred class
        ROUTINE [BASE, OPEN −> TUPLE]

feature
        operands: OPEN is ...
                        −− Open operands.
        target: ANY is ...
                        −− Target of call.
        call (args: OPEN) is ...
                        −− Call routine with operands 'args'.
        internal_operands: TUPLE
                        −− All open and closed arguments provided at creation time
```

A little example shows how to deal with agents as objects. First the example without ownership types is shown.

```
        class
                BOUND
        create make
        feature
                lower: COMPLEX
                upper: COMPLEX
                is_in_bound_agent: FUNCTION[BOUND,
                        TUPLE[COMPLEX,COMPLEX],BOOLEAN]

                make (l: COMPLEX; u: COMPLEX) is
                        do
                                lower := l
                                upper := u
                                is_in_bound_agent := agent is_in_bound(lower, ?, ?)
                        end

                is_in_bound (l: COMPLEX; u: COMPLEX; c: COMPLEX):
                        expanded BOOLEAN is
```

```
class
        BOUND
create make
feature
        lower: rep COMPLEX
        is_in_bound_peer_agent: peer FUNCTION[peer BOUND,TUPLE[
                readonly COMPLEX,readonly COMPLEX],expanded BOOLEAN]
        is_in_bound_rep_agent: rep FUNCTION[peer BOUND,TUPLE[
                readonly COMPLEX,readonly COMPLEX],expanded BOOLEAN]

        make (l: readonly COMPLEX; u: readonly COMPLEX) is
                do
                        create rep lower.make_cartesian (l.x, l.y)
                        is_in_bound_peer_agent := peer agent is_in_bound(lower, ?, ?)
                        is_in_bound_rep_agent := rep agent is_in_bound(lower, ?, ?)
                end

        is_in_bound (l: readonly COMPLEX; u: readonly COMPLEX;
                        c: readonly COMPLEX): expanded BOOLEAN is
                do
                        Result := ((l.greater(c)) and (c.greater(u)))
                        -- greater is a pure function
                end
end
```

Figure 3.9: Handling of agents as objects: code (without concept of pure agents)

```
                do
                        Result := ((l.greater(c)) and (c.greater(u)))
                end
end
```

Now we insert the ownership modifiers. For better illustrating the difference between a peer and a rep agent object, two agents of both types are included. The code and the corresponding ownership diagram is shown in figure 3.9 and 3.10. Consider that in the example the call of ”is_in_bound_rep_agent” would not be correct since there are some missing keywords **pure**. The concept of pure agents is explained in section 3.3.4 where you can also find the final version of this example.

Regarding to the the similarity between the keywords **agent** and **create** some additional rules can be derived. First when creating a new agent, the wanted ownership modifier needs to be specified, since it is possible to create a peer agent when you have declared a readonly agent.

```
is_in_bound_peer_agent := peer agent is_in_bound(lower, ?, ?)
```

In the existing Universe type system it is only allowed to create peer and rep objects because objects need to have an owner when created. By the same token it is only possible to create peer and rep agents.

3.3.3 Target, Open and Closed Operands

The types of the arguments, of the target object and of the return value are stored in the agent object relative to this object. But at the declaration place the formal generic parameters of the class ROUTINE are written relative to the current object. This is a rule declared for generics in [2]. This means that the declaration of the peer agent in example 3.9 is:

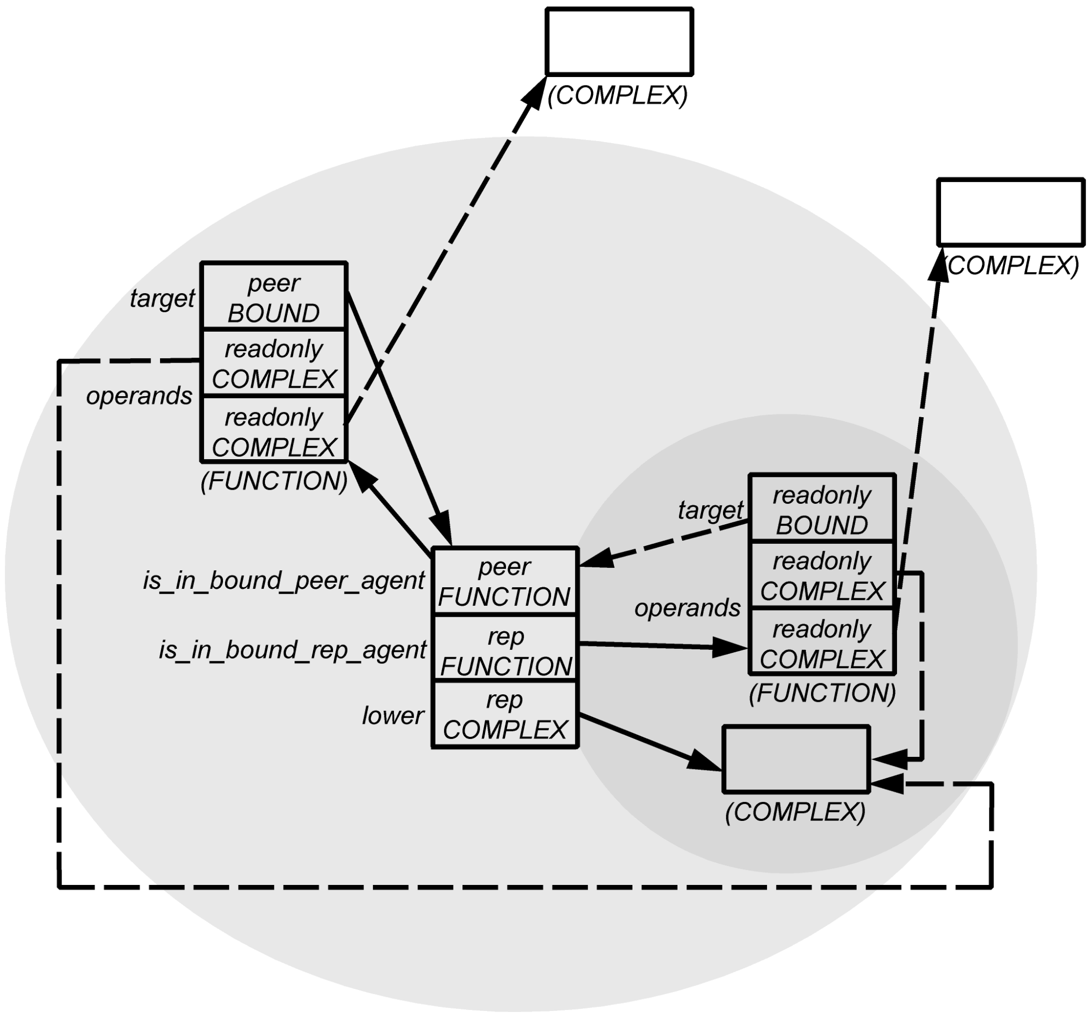


Figure 3.10: Handling of agents as objects: ownership diagram (without concept of pure agents)

```
        is_in_bound_rep_agent: rep FUNCTION[peer BOUND,
                TUPLE[readonly COMPLEX, readonly COMPLEX],expanded BOOLEAN]
```

In this declaration the type of the target object is **peer** BOUND because the target object (**Current**) is relative to the current object **peer**. In the agent object itself the target object is stored as **readonly** BOUND because relative to the agent object the target object is **readonly**.

What this means for the agent declaration is well illustrated in the example 3.9 in the last section.

3.3.4 Pure Agents

The call of an agent produces two routine calls. First, the routine "call" of the agent object is executed. This invokes the real routine which is in the example above the routine "is_in_bound". Since on readonly receivers only pure routine calls are allowed, this should also be considered with agents.

Non-pure agents are only possible if you have a peer or rep agent whose target object is peer relative to the agent object. This configuration is achieved with the agent "is_in_bound_peer_agent" in the example in figure 3.9 and 3.10.

If the target object is **readonly**, the corresponding routine should be **pure**. This works without any difficulty. The case is more complicated, if you have an agent that is referenced by a readonly reference. The routine "call" should also be **pure**, but in the current implementation "call" is not **pure**.

A possible solution is to introduce a second call procedure "pure_call" that is **pure**. For this a detailed look at the routine "call" in the class ROUTINE is needed:

```
deferred class
        ROUTINE [BASE, OPEN −> TUPLE]

feature
        call (args: OPEN) is
                        −− Call routine with operands 'args'.
                require
                        valid_operands: valid_operands (args)
                        callable : callable
                do
                        set_operands (args)
                        apply
                        if is_cleanup_needed then
                                remove_gc_reference
                        end
                end

        apply is
                −− Call routine with 'args' as last set.
        require
                valid_operands: valid_operands (operands)
                callable : callable
        deferred
        end

        frozen is_cleanup_needed: BOOLEAN
                −− If open arguments contain some references, we need
                −− to clean them up after call.
end
```

```
class
        PROCEDURE [BASE, OPEN −> TUPLE]
feature
        apply is
                        −− Call procedure with ‘args’ as last set.
                do
                        rout_obj_call_procedure (rout_disp, $internal_operands)
                end

        rout_obj_call_procedure (rout: POINTER; args: POINTER) is
                        −− Perform call to ‘rout’ with ‘args’.
                external
                        ”C inline use %”eif_rout_obj.h%””
                alias
                        ”rout_obj_call_agent($rout, $args, $$_result_type)”
                end
end
```

For introducing ”pure_call” also the called routines ”set_operands”, ”apply”, ”is_cleanup_needed”, and ”remove_gc_reference” should be pure routines. The procedure ”apply” performs the routine call by invoking ”rout_obj_call_procedure”. For ”pure_call” we need two new versions of this two procedures that are **pure**, called ”pure_apply” and ”pure_rout_obj_call_procedure”. The only needed condition is that ”pure_rout_obj_call_procedure” is side-effect free if the as argument passed routine ”rout” is **pure**.

The other three from ”call” invoked routines are needed because the open operands are stored and not directly passed to apply. Since the procedure ”set_operands”, which saves the open arguments, is not **pure**, it cannot be invoked in ”pure_call”. Therefore, ”pure_call” could only be used for agents with no open arguments where the procedure ”set_operands” is not needed. The same problem is about ”remove_gc_reference”.

If you don’t change the implementation of ROUTINE, this would mean that pure agents cannot have open arguments. Since this is a big constraint, it would be better if the implementation of ”call” would be changed. Conceptual, the passing of arguments for an agent call, that call a pure routine, should be **pure** because it doesn’t have to modify any objects. It should be possible that, instead of the call to ”set_operands”, the open arguments are passed directly as routine arguments to ”apply”. Therefore, a version of ”pure_call” could look like the following:

```
deferred class
        ROUTINE [BASE, OPEN −> TUPLE]

feature
        pure pure_call(args: OPEN) is
                        −− Call routine with operands ‘args’.
                require
                        callable: callable
                do
                        pure_apply(args)
                end
end
```

With this solution, the call of a pure agent would look as follows:

```
my_pure_agent.pure_call([args])
```

Another solution instead of the new routine ”pure_call” is to introduce two new classes called PURE_FUNCTION and PURE_PROCEDURE as you can see in figure 3.11. In these two classes the procedure ”call” is overridden by a pure procedure. This allows a uniform handling of the agent call, because for both pure and non-pure agents ”call” should be invoked.

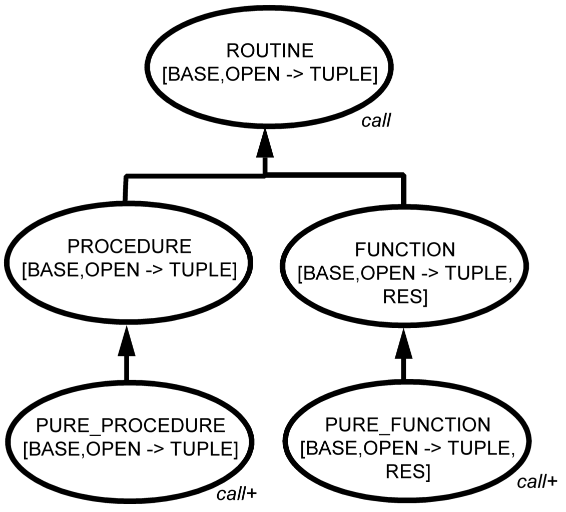


Figure 3.11: The extended agent class diagram with the two new classes PURE_PROCEDURE and PURE_FUNCTION

The difference of the two solutions is, that in the solution with "pure_call" you must know at call time if you have to call "call" or "pure_call". In the solution with the two new classes this decision is done at construction time which has two advantages. First the power of the decision is by the creator of the agent. The second advantage is that the decision at construction time should only be made once, while the decision at call time can be made several times. Another difference is that the first solution needs some implementation changes in the existing classes, while the second solution needs two new classes. In the second solution the difference between pure and non-pure agents is done by different types, which guarantees more static type safety. Therefore, the solution which introduces two new classes is the better.

Analogous to pure routines pure agents do not modify existing objects and are the only agents that can be called on readonly agents. You can only declare an agent **pure** if the routine it invokes is **pure** too.

The only routine that a pure agent can call is a pure routine. Since all argument types of pure routines implicitly have the readonly modifier, the arguments of pure agents must be readonly too. The return type can have any arbitrary modifier.

In the example above the agent "is_in_bound_rep_agent" operates on a readonly target object. Therefore, the agent should be pure. By allowing open arguments, the corrected code looks as follows:

```
class
        BOUND
create make
feature
        lower: rep COMPLEX
        upper: rep COMPLEX
        i: readonly COMPLEX
        is_in_bound_rep_agent: rep FUNCTION[readonly BOUND,TUPLE[
                readonly COMPLEX, readonly COMPLEX],expanded BOOLEAN]

        make (l: readonly COMPLEX; u: readonly COMPLEX) is
                do
                        create rep lower.make_cartesian (l.x, l.y)
                        create rep upper.make_cartesian (u.x, u.y)
                        is_in_bound_rep_agent := rep agent is_in_bound(lower, ?, ?)
                end

        pure is_in_bound (l: readonly COMPLEX; u: readonly COMPLEX;
                        c: readonly COMPLEX): expanded BOOLEAN is
                do
                        Result := ((l.greater(c)) and (c.greater(u)))
                        -- greater is a pure function
                end
end

class
        COMPLEX
feature
        x,y: expanded INTEGER
        pure greater(c: readonly COMPLEX): expanded BOOLEAN is
                do
                        Result := (x < c.x) and (y < c.y)
                end
end
```

3.3.5 Passing around

One concept of agents is that they can be passed around. Thus the observer pattern can be solved with agents as follows:

```
your_button. click_actions .extend(agent your_routine)
```

When you give an agent around, the reference to the agent object will become **readonly** in the majority of cases. Therefore, only pure agents are possible to call. This assures that the owner-as-modifier property is hold. On the other hand it avoids a lot of powerful agent examples (like the observer pattern). This means that with the presented solutions there is a big constraint for agents.

3.3.6 Other Solutions

In this section about possible treatments for agents we handled agent objects like normal objects. As mentioned in the last section this conservative solution causes big constraints. To reduce these constraints a possibility is to handle agent objects not as normal objects but with special rules. For example a solution could be to allow transfer of ownership. Another possibility could be to handle calls on agent objects not like normal calls, but allow to call non-pure routines on readonly receivers. Both is not possible according to the current ownership rules, but it would allow powerful agents like observer patterns. In this report only the solutions where agents are handled as normal objects are considered, but for future work one should consider if the best solution is to introduce special rules for agents.

Chapter 4

Conclusion

4.1 Conclusion

The existing Universe type system for Java is good adaptable to Eiffel. For the special Eiffel construct of expanded types, good solutions can be found that consider the ownership idea and the idea of expanded types. More difficult is the treatment of agents. The strict solution according the current ownership rules allows only a bounded group of agents. It is not possible to allow more powerful agents with the owner-as-modifier property. In addition some changes at the implementation of the agent classes are needed.

4.2 Future Work

To allow powerful agents, a better solution for this construct is needed. The current Universe ownership concepts probably have to be extended. Eventually some interesting inputs can come from the master thesis about ”Universe Type System for Scala” which Daniel Schregenberger is writing [16].

Bibliography

[1] Gilad Bracha. Generics in the Java Programming Language. Sun Developer Network Homepage, July 2004.

[2] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. Preliminary version to appear at http://www.sct.ethz.ch/publications/index.html.

[3] Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[4] Werner Dietl, Peter Müller, and Daniel Schregenberger. Universe Type System - Quick-Reference. August 2005.

[5] ECMA. *ECMA-367: Eiffel Analysis, Design and Programming Language.* ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA:adr, June 2005.

[6] Eiffel Software. *About Eiffel.*

[7] Eiffel Software. *Tuples.*

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*, chapter 4 Types, Values, and Variables. Addison Wesley, 2000.

[9] James Gosling and Henry McGilton. *The Java Language Environment.* Sun Microsystems, May 1996.

[10] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, June 2006.

[11] Qusay H. Mahmoud. Using Assertions in Java Technology. Sun Developer Network Homepage, June 2005.

[12] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition.* Prentice-Hall, 1997.

[13] Bertrand Meyer. Agents, Iteration and Introspection. Draft 5.02.00-0. Extracted from ongoing work on future third edition of ”Eiffel: The Language”., August 2005.

[14] Bertrand Meyer. Attached Types and their Application to Three Open Problems of Object- Oriented Programming. in ECOOP 2005, Springer Verlag, pages 1-32, to appear, July 2005.

[15] Manuel Oriol. Techniques of Java Programming: Java Basics. Handouts of lecture ”Techniques of Java Programming” in summer term 2006 at ETH Zürich, 2006.

[16] Daniel Schregenberger. Universe Type System for Scala.

[17] Sun Microsystems. *JavaTM 2 Platform Standard Edition 5.0 API Specification*, 2004.