Transmigration of Object Identity

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Pascal Costanza

aus

Siegburg

Bonn 2004

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn. Diese Dissertation ist auf dem Hochschulschriftenserver der ULB Bonn http://hss.ulb.uni-bonn.de/diss_online elektronisch publiziert.

1. Referent: Prof. Dr. Armin B. Cremers
2. Referent: Prof. Dr. Clemens A. Szyperski

Tag der Promotion: 2. Dezember 2004

Transmigration of Object Identity

Pascal Costanza
Kölnstr. 57
53111 Bonn

June 23, 2004

Abstract

The object-oriented paradigm is one of the central programming paradigms of our time. The following description is a generally accepted characterization of that paradigm: "An object has state, behavior, and identity" [9]. The concept of object identity plays an important role here insofar it is the only characteristic element that is not available in purely declarative programming languages without further effort. Purely declarative languages also incorporate behavior via functions or rules, and state via immutable values that are passed around to such functions and rules. Objects localize state and behavior, and the single means to access state and behavior of objects are their identity. In other words, the major achievement of object-oriented programming languages is to provide constructs for unambiguously mapping object identities to storage locations and procedures that act on those storage locations.

Early discussions of the notion of object identity have found strong connections between changes of state and equality predicates for objects. Object identity lies at the center of such discussions: when the state of an object is changed by way of its identity the new state is (re)observable via that same identity; when the same object identity is stored in two different variables it is always the same state that is observable via those variables [75]. Later on, various authors have implicitly or explicitly kept that same basic idea.

Now, this thesis shows that an alternative perception of object identity is possible when an analysis starts from a description of the usage scenarios for object identity. These are *reference* on the one hand – an object is able to refer to other objects – and *comparison* on the other hand – two variables may refer to the same or to different objects. These usage scenarios can be separated both on the conceptual level of an object model as well as on the practical level of the implementation of a programming language and run-time environment. From this modified view on object identity new operations can be derived. Especially dynamic object replacement is probably the most intriguing operation that is enabled by the approach taken in this thesis. This operation has the potential to address usage scenarios from the emerging field of unanticipated software evolution. This shows that this thesis is not only of a theoretical nature but also gives insight into possible practical applications. Still, the traditional notion of object identity is

kept as a special case of the broader conceptual framework presented in this thesis.

This thesis presents a historical perspective on the concept of object identity by illustrating central notions through summaries of seminal publications on the topic. It then develops the essential ingredients of the GILGUL model – one of the important results of this thesis – and demonstrates these ingredients as extensions of the Java programming language. Furthermore, the intricacies of the replacement of active objects are analyzed – objects with methods executing at the time of their replacement – and extensions of the GILGUL language are presented that help to deal with these intricacies. Finally, the implementation of the GILGUL language and run-time environment are discussed and evaluated, and usage examples are given.

Acknowledgements

I am grateful to Professor Armin B. Cremers for his supervision and the excellent working conditions he has created at the Institute of Computer Science III. Furthermore, I thank Tom Arbuckle, Michael Austermann, Ferrucio Damiani, Richard Gabriel, Paola Giannini, Peter Grogono, Arno Haase, Günter Kniesel, Thomas Kühne, Sven Müller, James Noble, Markku Sakkinen, Daniel Speicher, Oliver Stiemerling, Clemens Szyperski, Dirk Theisen and Kris De Volder for their critical comments on early stages of the work presented in this thesis, which led to substantial improvements.

Einen besonderen Dank an Beatrix, Calogero und Rouven Costanza, sowie Markus Brodesser, Felix Schaefer und Steffi Schmid, für die langjährige Unterstützung auch in schwierigen Zeiten.

Contents

1 Introduction **9**

2 History of Object Identity Concepts **11**
2.1 Steele, Sussman (1978) 12
2.2 MacLennan (1982) 13
2.3 Khashafian, Copeland (1986) 14
2.4 Atkinson et al. (1990) 14
2.5 Kent (1991) 15
2.6 Beeri (1993) 15
2.7 Wieringa, de Jonge (1995) 16
2.8 Summary 21

3 The Gilgul Model **23**
3.1 Dynamic Object Replacement 24
3.2 Gilgul 27
3.2.1 Operations on Referents 27
3.2.2 Operations on Comparands 28
3.2.3 Reuse of Existing State 29
3.3 Wieringa/de Jonge Requirements Revisited 30
3.4 Smalltalk's become: 34
3.5 A Mental Model for Comparands 35
3.6 The Name Gilgul 37

4 The Programming Language Gilgul **39**
4.1 Basic Language Constructs of Gilgul 39
4.1.1 Operations on Referents 40
4.1.2 Operations on Comparands 40
4.1.3 Operations on References and Objects 41
4.2 Replacement of an Object with its Wrapper 42
4.3 Initializing Referent Assignments 43
4.4 Control Facilities 45
4.5 Type Issues 47
4.5.1 Additive and Subtractive Replacement 47

4.5.2 Typeless Classes 49
4.5.3 Cast Expressions 49
4.5.4 The with Statement 50
4.5.5 Relation to Java's Interfaces 50
4.5.6 Relation to Software Evolution 51
4.5.7 Type Soundness 52

5 Replacement of Active Objects 59
5.1 Default Semantics 59
5.1.1 Replacement of this 60
5.1.2 Augmented return/throw versus try-finally 62
5.1.3 Advanced Requirements 62
5.1.4 Recalls 64
5.1.5 Combination with Referent Assignment 65
5.1.6 Relation to Java's Thread Model 66
5.1.7 Replacement of Long-Running Methods 67

6 Implementation 69
6.1 Architecture of the Java Virtual Machine 70
6.2 Architecture of the Gilgul Virtual Machine 71
6.2.1 The primitive data type comparand 71
6.2.2 Control Facilities 71
6.2.3 Protected Activities 72
6.2.4 Recalls 74
6.2.5 System Classes 75
6.3 New Virtual Machine Instructions 75
6.3.1 Operations on Comparands 76
6.3.2 Operations on References 76
6.3.3 Operations on typeless Classes 77
6.4 Representation of Objects 77
6.4.1 Minimization of Indirection Penalties 77
6.5 Activity Control and Multithreading 78

7 Evaluation 81
7.1 Qualitative Evaluation 81
7.1.1 The Stock Ticker Example 81
7.1.2 Using Referent Assignments to Introduce New Roles . 85
7.1.3 Constant Folding in the Interpreter Pattern 89
7.2 Quantitative Evaluation 90
7.2.1 Memory Overhead 91
7.2.2 Performance Overhead 92
7.2.3 Activity Control 93
7.2.4 Summary 94

8 Conclusions and Future Work 97
8.1 Results . . . 97
8.2 Possible Extensions . . . 99

A The Gilgul Language Specification 101
A.1 Introduction . . . 101
A.2 Notation . . . 101
A.3 Lexical Structure . . . 101
A.3.1 Keywords . . . 101
A.3.2 Operators . . . 102
A.4 Types, Values, and Variables . . . 102
A.4.1 Primitive Types and Values . . . 102
A.4.2 Reference Types and Values . . . 103
A.4.3 Where Types Are Used . . . 103
A.4.4 Variables . . . 104
A.5 Conversions and Promotions . . . 104
A.5.1 Kinds of Conversions . . . 104
A.5.2 Assignment Conversion . . . 105
A.5.3 Casting Conversion . . . 106
A.6 Classes . . . 106
A.6.1 Class Declaration . . . 106
A.6.2 Field Declarations . . . 107
A.6.3 Constructor Declarations . . . 107
A.7 Interfaces . . . 109
A.7.1 Interface Declarations . . . 109
A.8 Exceptions . . . 109
A.8.1 The Causes of Exceptions . . . 110
A.8.2 Compile-Time Checking of Exceptions . . . 110
A.8.3 Handling of Exceptions . . . 110
A.9 Blocks and Statements . . . 111
A.9.1 Local Variable Declaration Statements . . . 111
A.9.2 Statements . . . 111
A.9.3 The `return` Statement . . . 112
A.9.4 The `throw` Statement . . . 112
A.9.5 The `try` Statement . . . 113
A.9.6 The `with` Statement . . . 113
A.10 Expressions . . . 114
A.10.1 Normal and Abrupt Completion of Evaluation . . . 114
A.10.2 Primary Expressions . . . 114
A.10.3 Class Instance Creation Expressions . . . 115
A.10.4 Array Creation Expression . . . 116
A.10.5 Comparand Creation Expressions . . . 116
A.10.6 Equality Operators . . . 116
A.10.7 Assignment Operators . . . 117

A.11 Definite Assignment 121

B The Comparand Pattern **123**

B.1 Thumbnail 123
B.2 Example 123
B.3 Context 125
B.4 Problem 125
B.5 Solution 127
B.6 Implementation 128
B.6.1 The “Right” Comparison Semantics 128
B.6.2 Comparison of Clones 128
B.6.3 Which Classes Are Comparable To Each Other? . . . 129
B.6.4 Boundary Conditions of a Given System 130
B.6.5 Reuse of an Existing Attribute 131
B.6.6 Execution of Comparison Operations 131
B.6.7 Comparands in Distributed Environments 132
B.7 Consequences 136
B.8 Known Uses 137
B.8.1 Java Platform Debugger Architecture 137
B.8.2 Remote Method Invocation 137
B.8.3 CORBA Relationship Service 138
B.8.4 Enterprise JavaBeans 138
B.8.5 Ginko 139
B.8.6 Related Patterns 139
B.9 Conclusion 139
B.10 Acknowledgements 140

C Curriculum Vitae **141**

Chapter 1

Introduction

The object-oriented paradigm is one of the central programming paradigms of our time. The following description is a generally accepted characterization of that paradigm: "An object has state, behavior, and identity" [9]. The concept of object identity plays an important role here insofar it is the only characteristic element that is not available in purely declarative programming languages without further effort. Purely declarative languages also incorporate behavior via functions or rules, and state via immutable values that are passed around to such functions and rules. Objects localize state and behavior, and the single means to access state and behavior of objects are their identity. In other words, the major achievement of object-oriented programming languages is to provide constructs for unambiguously mapping object identities to storage locations and procedures that act on those storage locations.

Early discussions of the notion of object identity have found strong connections between changes of state and equality predicates for objects. Object identity lies at the center of such discussions: when the state of an object is changed by way of its identity the new state is (re)observable via that same identity; when the same object identity is stored in two different variables it is always the same state that is observable via those variables [75]. Later on, various authors have implicitly or explicitly kept that same basic idea.

Now, this thesis shows that an alternative perception of object identity is possible when an analysis starts from a description of the usage scenarios for object identity. These are *reference* on the one hand – an object is able to refer to other objects – and *comparison* on the other hand – two variables may refer to the same or to different objects. These usage scenarios can be separated both on the conceptual level of an object model as well as on the practical level of the implementation of a programming language and run-time environment. From this modified view on object identity new operations can be derived. This shows that this thesis is not only of a theoretical nature but also gives insight into possible practical applications.

Still, the traditional notion of object identity is kept as a special case of the broader conceptual framework presented in this thesis.

Dynamic object replacement is probably the most intriguing operation that is enabled by the approach taken in this thesis. This operation has the potential to address usage scenarios from the emerging field of unanticipated software evolution: although some changes in requirements can typically be anticipated by software developers, unanticipated changes occur repeatedly in practice, and by definition techniques like parameterization or application of design patterns cannot tackle them. To obviate such problems, programming languages and run-time environments should include features that allow for manipulation of program internals without permanently modifying their source code. This latter requirement is especially important in the context of component-oriented software engineering: components are usually deployed using a compiled format and their source code is not available for modifications. Even if the source code can be accessed in some cases, destructive modifications are still not feasible since they would not automatically be incorporated into new versions of a component.

Essentially, unanticipated software evolution can take place at two points in time. It can occur before a program is being linked into its final form, or it can happen at run time. Changes to software that are carried out before linktime can be made effective only by stopping an old version of a program and starting the new one. This results in downtimes that induce a high cost and possibly determines an application's success or failure. Alternatively, run-time systems should be provided with features that allow for subsequent unanticipated evolution of already active programs. Dynamic object replacement is an especially desirable feature in this regard.

This thesis is organized as follows: Chapter 2 presents a historical perspective on the concept of object identity by illustrating central notions through summaries of seminal publications on the topic. Chapter 3 develops the essential ingredients of the GILGUL model – one of the important results of this thesis. In Chapter 4, these ingredients are demonstrated as extensions of the Java programming language – in that chapter, the (type sound) operator for dynamic replacement is already discussed. In Chapter 5, the intricacies of the replacement of active objects are analyzed – objects with methods executing at the time of their replacement – and extensions of the GILGUL language are presented that help to deal with these intricacies. In the following two chapters, the implementation of the GILGUL language and run-time environment are discussed and evaluated, and usage examples are given. Chapter 8 concludes and hints towards future work.

The appendix includes the GILGUL language specification as an addendum to the Java language specification. Furthermore, Appendix B reprints the Comparand pattern that has been jointly written with Arno Haase and makes some of the results of the GILGUL approach available for other "traditional" object-oriented programming languages.

Chapter 2

History of Object Identity Concepts

This chapter gives summaries of relevant historical papers that characterize and discuss important facets of the concept of object identity. These papers are presented in chronological order. They mainly consist of papers that consider the relevance of object identity with regard to programming languages. Of course, the concept plays also an important role in the context of database systems, and therefore relevant literature from that area is also taken into account. Here is a list of the papers that are discussed in this chapter:

- In “The Art of the Interpreter or, the Modularity Complex (Parts Zero, One, and Two)” [75] (1978), Guy L. Steele and Gerald J. Sussman analyze the connection between side effects and equality.

- In “Values and Objects in Programming Languages” [56] (1982), Bruce J. MacLennan discusses the distinction between values and objects.

- The paper “Object Identity” [43] (1986) by Setrag N. Khoshafian and George P. Copeland is the most cited one with regard to the topic of object identity, and gives a thorough presentation of the concept and its implementation in programming languages and database systems.

- In “The Object-Oriented Database System Manifesto” [4] (1990), Malcolm Atkinson et al. list requirements that are to be fulfilled by object-oriented databases. Of course, object identity plays an important role here.

- “A Rigorous Model of Object References, Identity and Existence” [42] (1991) by William Kent describes a model for object identity that, among other things, aims at a facility for merging objects and their identities after the fact.

- Catriel Beeri criticizes a too strong focus on object-oriented features in the context of databases in "Some Thoughts on the Future Evolution of Object-Oriented Database Concepts" [8] (1993), and sketches a reduction of the concept of object identity to pure value-based properties in relational databases.

- Roel Wieringa and Wiebren de Jonge present a detailed formal model in "Object Identifiers, Keys, and Surrogates – Object Identifiers Revisited" [85] (1995) that captures the essential properties of object identity and can be interpreted as the common denominator of previous approaches.

2.1 Steele, Sussman (1978)

In "The Art of the Interpreter or, the Modularity Complex (Parts Zero, One, and Two)" [75] (1978), Guy L. Steele and Gerald J. Sussman discuss a number of programming language constructs that support the modular decomposition of programs. All these language constructs are presented as incremental extensions of a meta-circular interpreter for a Lisp dialect. For example, local function declarations and the difference between lexical and dynamic scoping are discussed in detail, among other features.

A large part of that paper deals with side effects, their meaning and their importance. The authors argue that for a small program, side effects play a minor role, and that usually such programs can be transformed into variants that do not make use of side effects.[1]

However, for a large program the authors illustrate that it is often important to be able to decompose it into modules with independent state. "If more than one module has state [...] then each may perceive changes in the other's behavior. This [is] the essence of side effect."[2]

The discussion about side effects and state in that paper reveals that the "concept of side effect is inseperable from the notion of equality / identity / sameness. The only way one can observationally determine that a side effect has occurred is when the *same* object behaves in two *different* ways at different times. [...] Conversely, the only way one can determine that two objects are the same is to perform a side effect on one and look for an appropriate change in the behavior of the other." (p. 40) "Thus the ability

[1] In fact, the side effects can be encapsulated inside of the interpreter of the language, and this makes the program appear to be free of side effects. This approach has been further investigated by other authors, and recently lead to the discovery of monads and arrows. The details of simulation of side effects in pure applicative approaches are not taken further into account in this thesis.

[2] As a historical sidenote, Sussman and Steele at that time had incorporated object-oriented constructs into Lisp, resulting in the first version of the language Scheme. This is probably the reason why their arguments in that paper resemble those typically hold in favor of object-oriented programming.

to decide whether two objects are the same is directly correlated with the ability to perform side effects on them." (p. 43)

As one result of the discussion, the adequate semantics of an equality predicate are discussed, that are now taken for granted in most programming languages (`eq` in Common Lisp, `==` in Java, C# and C++, `=` in Pascal, Modula-2, Oberon, etc.). Essentially, this equality predicate distinguishes each object from all other objects, i.e. it tests for object identity.

In Gilgul – the approach presented in this thesis – the `==` operatorhas the same granularity by default but can be widened by comparand assignment – see Section 3.2.2 for details. In other words, comparands introduce equivalence classes into the language. One consequence of comparand assignments is that a side effect on an object may not be observable in another object that is nonetheless considered the same under Gilgul's `==` operator. The implications thereof are discussed in Section 3.5.

2.2 MacLennan (1982)

In "Values and Objects in Programming Languages" [56], Bruce J. MacLennan characterizes both values and objects by contrasting and comparing them. Values are described as applicative, atemporal, abstract[3], and uncountable. A motivation for this characterization is given as follows: "[...] it is not common to treat compound data values, such as complex numbers or sequences, as values. If done, this would eliminate one source of errors, namely, updating a data structure that is unknowingly shared [...]".

In order to characterize objects, that paper states that two objects might be different even if they have the same values because they occupy different locations. So here, comparing objects is basically understood as determining if the same location is occupied. This is unified with the concept of a reference to an object: "In general we can say that the uniqueness of an object is determined by its external relations and is independent of its internal relations and properties", where "external relations" are the references to other objects and "internal relations" are the values (the state) of an object. This uniqueness is what is later called "identity".

Based on this concept, several properties of objects are derived. For example, objects "can be created, destroyed, copied, shared and updated". These notions are indeed widely available in object-oriented programming languages. For example in Java, creation is represented by the `new` statement and constructors, destruction by the `finalize` method, copying by the `clone` method, and sharing and updating by assignment.

[3] in the sense that they cannot exist independently of something they describe

2.3 Khashafian, Copeland (1986)

[43] focuses the discussion on the properties of object identity rather than the differences in values. It stresses that "identity is internal to an object. Its purpose is to provide a way to represent the individuality of an object independently of how it is accessed." The focus on comparison is also manifested in the discussion of "operators with object identity" which are, among others, identity equality, shallow equality, deep equality, assignment (making two references equal with regard to identity), shallow copying and deep copying. Several approaches for implementing object identity are described:

Identity Through Physical Address This implementation approach is based on real or virtual addresses as represented by the CPU or the MMU of a computer.

Identity Through Indirection Instead of using direct machine addresses, some languages use an object table that store those addresses and make objects refer to each other indirectly via pointers into such a table. This provides for some advantages with regard to object relocation and remote objects.

Identity Through Structured Identifiers This approach can be used for remote objects and uses compound references that consist of a description of the remote address space where an object resides together with the actual reference within that address space.

Identity Through Identifier Keys In relational databases, entries are usually referenced by user-supplied identifier keys.

Identity Through Tuple Identifiers That paper gives examples of database systems that use tuple identifiers that are used internally for technical reasons but have no conceptual external representation. A way how to use those tuple identifiers to implement identity is sketched.

Identity Through Surrogates Surrogates are globally unique, system-generated identifiers, and are described as the most powerful implementation technique for object identity.

Some of these approaches relate to variants of the Comparand pattern as described in Appendix B.

2.4 Atkinson et al. (1990)

In "The Object-Oriented Database System Manifesto" [4], Malcolm Atkinson et al. list constituent features of object-oriented databases. They are complex objects, object identity, encapsulation, types and classes, class or

type hierarchies, overriding, overloading and late binding, computational completeness, extensibility, persistence, secondary storage management, concurrency, recovery, an ad hoc query facility, and furthermore some optional features: multiple inheritance, type checking and type inferencing, distribution, design transactions and versions.

In that paper, objects are described as having an existence independent of their value as a consequence of object identity. Two implications are outlined, namely object sharing – two objects can refer to the same third object – and object updating, as supported by operations for assigning, copying, and tests for object identity and object equality.

2.5 Kent (1991)

In "A Rigorous Model of Object References, Identity and Existence" [42], William Kent also discusses various aspects of object identity. He argues that object identity is not about comparing objects but about referencing them. The motivating example is given as follows: "A predicate of the form Identical(O_1, O_2) is hard to explain if O_1 and O_2 are meant to be the objects themselves. Can the same object actually be present in the first operand and also the second? This question is best recast into determing whether two references are to the same object." Consequentially, "[d]eciding which things are the same is very carefully excluded from the model. [...] Such questions are decided by the system implementors or data administrators." This means that the actual equality predicates need to be implemented explicitly, only a test for identity of object references is offered by the system.[4]

That paper also introduces the concept of synonymous handles[5], that are essentially different identifiers referencing the same object. For example, an operation MakeSameObject(...) is sketched.

2.6 Beeri (1993)

In [8], Catriel Beeri criticizes the notion of object identity from the perspective of relational databases. He starts from giving a distinction of values and objects, similar to the one made in [56], but then proceeds to argue that most properties of the concept of object identity can already be achieved by the notion of *keys* in relational databases. He argues that objects are actually not identified by their object identities, in the sense that object

[4] For example, Java follows this approach by offering the `==` operator for object identity tests and the method `equals` method that needs to be redefined for all other kinds of equivalence. See [35] for an opposing view in which a programming language designer is advised to offer only one comparison operation per object – either object identity or a different equality predicate, as required by the application domain.

[5] "A reference is an occurrence of a handle in the system" [42].

identities are not explicitly used by programmers to determine an object or a set of object with specific properties. Instead, the retrieval of objects from an object-oriented database is actually accomplished by way of association with values. In other words, "[a]n object is uniquely identifiable if there is a query that retrieves it, and no other object." This argument is illustrated with an example in which an ambiguous query can only be turned into an unambiguous one by adding more value-based constraints.

Object identity still has a place in Catriel Beeri's conceptualization, but only as a generalization and simplification of the notion of *foreign keys*: Object identity extends the relational model by adding sharing and cyclic structures in a straightforward way.

2.7 Wieringa, de Jonge (1995)

In "Object Identifiers, Keys, and Surrogates – Object Identifiers Revisited" [85], Roel Wieringa and Wiebren de Jonge present a formal model of what they call *object identification schemes*. That paper is discussed in more detail here because it must be regarded as the most important influence on this thesis.

The essential understanding of object identification is given as follows in that paper: "Basically, an oid is a proper name of an object such that the connection between the oid and the object is one-one and fixed." In order to achieve a formalization of this notion, the authors take the following three steps.

First, they distinguish between *symbol occurrences*, *symbols* and *values*: A symbol occurrence denotes a specific symbol but a symbol occurrence is always only equal to itself. So for example, two occurrences of the letter "E" are different when regarded as symbol occurrences but the same when regarded as symbols. A procedure for unambiguously determining the symbol from a symbol occurrence is taken for granted and not further discussed in that paper.

A *notation system* is defined as a partial function that maps symbols to values. The same symbol always yields the same value under the same notation system but might yield different values under different notation systems. Different notation systems may map different symbols to the same values.

In the next step, *naming schemes* are introduced. A naming scheme is defined as a function $N : \Sigma \rightarrow \wp(V \times O)$ where Σ is the set of all possible states of the world[6], V is a set of values that might be names of objects, O is the set of all objects that might be named, and $\wp(V \times O)$ is the set of possible *naming relations*, the powerset of $V \times O$. ($N_\sigma \subseteq V \times O$ holds for each $\sigma \in \Sigma$.) It is important to note that this is not supposed to be an

[6] A "world" is the set of all possible objects of interest.

object identification scheme yet, so each value can name many objects and each object can be named by many values.

The domain and range of a naming relation are defined as follows:

$$dom(N_\sigma) = \{v \mid \exists o \in O : \langle v, o \rangle \in N_\sigma\}$$
$$range(N_\sigma) = \{o \mid \exists v \in V : \langle v, o \rangle \in N_\sigma\}$$

In the third and final step, a set of requirements is identified that a naming scheme must satisfy in order to be an *object identification scheme.* In the rest of this thesis, they will be referred to as the *Wieringa/de Jonge requirements.* They are as follows.

Singular reference. In each state $\sigma \in \Sigma$, each name in $dom(N_\sigma)$ must name exactly one object in O. (These names are called *oids.*)

Singular naming. In each state $\sigma \in \Sigma$, each object in $range(N_\sigma)$ must be named by exactly one oid in V.

Monotonic designation. For all successive states of the world σ_1 and σ_2, $N_{\sigma_1} \subseteq N_{\sigma_2}$ must hold.

The authors give the following rationale for those restrictions. The singularity requirements ensure that a name never refers to more than one object, and that an object never has more than one name. Therefore "in each single state of the world, we can count objects by counting their oids". In order to preserve this countability property across all possible states of a world, the monotonicity requirement is needed. The fact that objects can be distinguished by their oids even if they have the same state is presented as another consequence of those requirements.

Relation to Other Concepts In the subsequent sections of that paper, the authors compare their notion of an object identification scheme to concepts mainly from the field of relational databases – keys on the one hand and surrogates / internal identifiers on the other hand.

They compare keys and oids in seven regards.

1. Keys are tied to databases whereas oids are claimed to be more generally applicable.

2. Keys may represent information about objects they identify. The authors argue that because of the singularity and monotonicity requirements placed on oids, they are more restricted: candidates for properties that can act as oids are required to be unique and unchangeable, and the authors claim that it is impossible to find such properties. They name fingerprints as example candidates that can nevertheless be manipulated.

3. Keys are updatable whereas oids are not. Even when keys are made non-updateable, an update operation can always be accomplished by deleting a tuple and inserting a modified one that is supposed to represent the same object.

4. A key is required to be unique in each single state of a database whereas an oid is additionally required to be unique across all possible states of the world.

5. Keys and oids are claimed to have a different level of suitability for solving the information transfer problem. Without naming a specific reference, Wieringa and de Jonge state that oids are considered by authors to be a solution to that problem. In contrast, they argue that this is only a gradual difference: while a set of related keys is typically bound to a specific database, oids are claimed to have a broader scope. So the information transfer problem is considered to be less pressing with oids because it is relatively easy to exchange information between information systems that use the same object identification scheme whereas it is not so clear whether two different databases use the same representation for their keys.

   Wieringa and de Jonge argue for the fact that this is only a gradual difference by stating that "a global oid scheme is unattainable in practice". Obviously, they are not aware of the concept of globally unique identifiers (GUIDs or UUIDs, see [10]) that can serve as a global identification scheme without the need for global synchronization.[7]

6. Keys are usually assigned by database users while oids are typically assigned by "authorities higher than the database user". The authors give the governmental tax department in the Netherlands that assign social security numbers as an example, and they state that this is also only a gradual difference.

7. The authors say that oids should be visible to the user, just like keys. Again, they cite social security numbers as an example.

These comparisons can be criticized as follows. The fundamental mistake Wieringa and de Jonge make is that they compare the implementational level of keys to the conceptual level of oids. For example, whether keys represent information about the objects they identify or not is an implementation aspect that needs to be decided upon in the design phase of an information system. If the strict requirements as expressed for oids are detected to be important for the problem at hand as well, a database designer should of course make the same considerations as the authors of that paper make for oids. It is, of course, perfectly possible to use GUIDs as the sole content of

[7] See also Appendix B for some details about GUIDs.

(primary) keys in a database and thus give keys all the properties that are claimed to be characteristic of oids. When the requirements are detected to be less strict during analysis, then the database designer is free to relax the restrictions on the implementation and leverage the expressive power of keys. Still, that section is illuminating because it sheds light on the fact that the authors have a very general concept in mind when they define the concept of an object identification scheme.

Oids are further compared to surrogates and internal identifiers. Surrogates are objects (tuples) internal to a database that represent conceptual entities, and internal identifiers are identifiers for such surrogates. The important fact here is that surrogates and internal identifiers play a role exclusively on the level of the implementation of a database system and are completely invisible from the outside. The comparison by Wieringa and de Jonge highlight some differences that can all be derived from this fundamental observation, but the details are not interesting here.

In the following sections, Wieringa and de Jonge mention some examples that are either examples for object identification schemes or not. (Passport numbers do not qualify because a person can have more than one passport; employee numbers can qualify as oids when they identify employee roles; libraries typically employ object identification schemes for books and other publications; credit card numbers cannot be used as oids for persons because an account might be used by more than one person; unix process numbers violate the monotonic designation requirement; ethernet addresses are claimed to not qualify as oids because manufacturers have not agreed upon the object space – they are either ethernet boards or machines that carry ethernet boards; internet domain names are not oids because domain names can change.)

The rest of that paper discusses several technicalities, like assignment of oids, borrowing of oids and information transfer that are not of interest here.

Relation to Programming Languages Although the authors cite some papers that discuss object identity in the context of programming languages, they do not discuss the relation of their model of object identification in that context. Here is a possible reconstruction of the steps they take to define oids with regard to programming languages.

At first, one is tempted to first interpret the definitions of symbol occurrences, symbols and values as the variable names, variables and values stored in variables of programming languages. However in the next step, those values are interpreted as names for objects, and so this interpretation does not seem to fit.

As already described above in the discussion of the paper by Khoshafian and Copeland, there are mainly two approaches for implementing object

identifiers in programming languages, namely Identity Through Physical Address and Identity Through Indirection. They are similar in that reference/pointer variables store addresses/handles that serve as object identifiers. In order to match the concept of an object identification scheme by Wieringa and de Jonge, those addresses/handles need to be considered as the names for objects. So in turn, symbol occurrences and symbols are representations of such addresses/handles – for example, they might be visualized as hexadecimal values in debuggers. (Here the notation system interprets/presents character strings as hexadecimal representations of integer values.)

The requirements imposed on this naming scheme in order to make it an object identification scheme can be interpreted as follows.

Singular Reference. Indeed, if a physical address names an object within a running program it names exactly one object. The same holds for typcial implementations of Identity Through Indirection because each table entry typically consists of a pointer to exactly one object.

Singular Naming. Each object is located at a unique machine address. For Identity Through Indirection, the Singular Naming Requirement needs to be enforced in order to not have the same object represented in two different table cells at the same time.[8]

Monotonic Designation. If a machine address / table pointer refers to one object at some point in time, it will refer to the same object at any point in time thereafter. This can trivially be accomplished under the following assumptions: physical memory is never reclaimed once it has been captured by an object, even if the object is deleted or could be garbage collected; and objects never change their location in physical memory.

Of course in practical implementations, unused storage is usually reclaimed and therefore machine addresses might name different objects at different points in time. Furthermore, some compacting garbage collectors even relocate objects in physical memory. However, such implementations take great care to keep the illusion that object identities never change in time; indeed, all these implementation techniques can be regarded as mere optimizations of the simplified model above, even if they require collaboration by the programmer in languages like C++.

It is important to note that such optimizations are applied under the assumption that the machine addresses / handles themselves are not

[8] Or alternatively, partitioning the table into active and inactive cells. Then, the requirements needs to be enforced only for all the active cells at the same time. However, this is only an optimization and conceptually equivalent.

used as first class entities, say as integer values. This is in contrast to the statement by Wieringa and de Jonge that oids should be visible for users. In the realm of programming languages, constructs that allow determining the machine address of an object and referencing storage locations via machine addresses are considered low level, and it is generally accepted wisdom that they require great care by programmers and should only be used for very low level programming tasks.

2.8 Summary

The latter comparison reveals that Wieringa's and de Jonge's object identification schemes are trivially implemented by physical machine addresses and still relatively easily by object tables. Indeed, implementations of object tables incorporate restrictions that make pointers into such tables still behave more or less like physical addresses. The contribution of Wieringa's and de Jonge's work is that their requirements on naming schemes indeed characterize the common understanding of object identity, as for example exemplified in the other papers discussed in this chapter.

However, there already exist alternative conceptualizations of object identitiy. One example is the diploma thesis "Datenraumbasierte Formulierung der Objektidentität" by Harald Schmidt [72] that takes the criticism by Catriel Beeri as a starting point and develops an alternative model based on data spaces [22].

This thesis takes yet another route. Instead of taking the characteristics of physical addresses as object identifiers for granted, the question is raised whether the requirements formulated by Wieringa and de Jonge can be altered and turned into something more flexible. This idea is explored in detail in the rest of this thesis.

Chapter 3

The Gilgul Model

This chapter introduces the GILGUL model. Beforehand, an example is given that motivates important ingredients of the GILGUL model. As already pointed out in the introductory chapter, the goal of GILGUL is to provide both a new foundation for the concept of object identity and well-motivated uses of this new foundation.

After giving an outline of the GILGUL model together with a resolution of the motivating example, this chapter presents a comparison of GILGUL and the text "Object Identifiers, Keys and Surrogates – Object Identifiers Revisited" by Wieringa and de Jonge, as discussed in the previous chapter [85]. It turns out that the GILGUL model can be regarded as a deconstruction [24] of that text: as already discussed in the previous chapter, the traditional view of object identity is based on a particular implementation scheme, and abandoning that implicit assumption results in the opportunity to relax all the requirements imposed on object identity by that text.

The requirements are reformulated in the following, and it is shown how these relaxed requirements enable the specification of new language constructs. Namely, object identity is separated into *referents* and *comparands*. This allows the introduction of *referent assignment* and *comparand assignment* along the traditional reference assignment.

These language extensions are already sketched as extensions of the Java programming language. Specifically Smalltalk's `become:` operator is compared to GILGUL's referent assignment because of their similarity, and problems of `become:` are discussed that are avoided in GILGUL.

The GILGUL model is not a minimal model in a mathematical sense but pragmatic aspects are considered more important. This is especially important in the rationale for the inclusion of comparands into the GILGUL model that is presented in this chapter.

Last but not least, the name GILGUL is also explained and motivated.

3.1 Dynamic Object Replacement

In principle, unanticipated evolution can always be dealt with by manual redirection of references. If one knows the reference to an object and wants to add or replace a method or change its class, one can simply assign a new object with the desired properties. The new object can even reuse the old object by some form of delegation [47] so that a transition of the old state is not needed.

For example, [47] gives an illustrating example within the context of the European Union's transition to the Euro currency. Assume you have developed a Java class that represents, for example, the former German currency in an application some years ago, as follows.

```
public class DEM ... {
  ...

  int amount() { return ...; }
}
```

In order to make the (unanticipated) transition to the Euro currency, you can write the following wrapper class.

```
public class EuroWrapper ... {

  DEM parent;

  // constructor
  EuroWrapper (DEM dem) { this.parent = dem; }

  // changed method
  int amount() { return parent.amount() / 1.96; }

  ...
}
```

This wrapper class can be used to adapt existing DEM instances. In [47] this example has been given to illustrate the *self problem* [53] and its solution in the Lava programming language. The self problem occurs when messages are sent to this within a wrapped object that should in fact be sent to the wrapper. In essence, the solution in [47] is to have means at the language level to bind this to the wrapper in forwarded messages (without breaking type soundness).

In this thesis, we take that solution for the self problem for granted, but instead concentrate on the actual introduction of wrappers at run time.[1] On

[1] Examples that use a combination of the Lava and GILGUL approaches are presented in the evaluation chapter of this thesis.

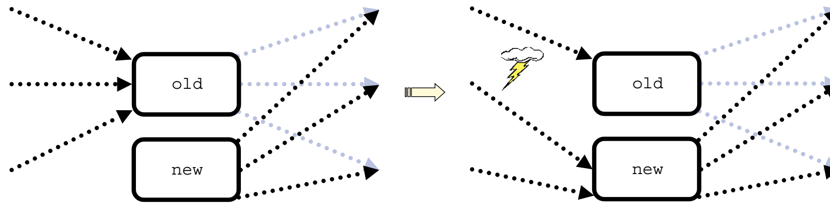


Figure 3.1: If an object is replaced by manual redirection of references, messages may be sent to both objects during the replacement, probably leading to an inconsistent state.

Figure 3.2: The concept of object identity combines the notion of object reference...

the conceptual level, the approach to manually redirect references from the old object to the new wrapper involves two consistency problems. Firstly, if there is more than one reference to the old object, they all must be known to the programmer in order to consistently redirect them. Secondly, even if all references are known, they have to be redirected to the new object one by one. This approach is likely to lead to an inconsistent state of the objects involved if message are sent via these references during the course of the redirections (for example within another thread; see Figure 3.1).

So for example, when references to an existing DEM instance are redirected to its EuroWrapper object, it must be ensured that each attempt at changing the amount property is consistently executed, depending on whether the old or the new version of this object is addressed.

It would be much simpler if we could just "replace" an object with another one without changing the references involved. Such a replacement would be an atomic operation and hence would avoid the consistency problems shown above. However, this would also conflict with the traditional notion of object identity, as is explained in the following paragraphs.

The reason for this conflict is that the concept of object identity in fact combines two distinct notions of object reference which permits object correlation and access to objects' internal states (Figure 3.2), and object comparison which permits the decision if two variables actually refer to the same object (Figure 3.3). In the following paragraphs we discuss the central idea of the Gilgul approach, the strict separation of the notions of reference and comparison. Eventually, this allows us to introduce means for dynamic object replacement into a programming language that solve the consistency

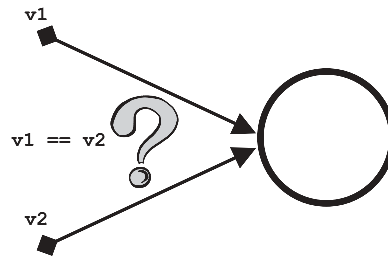


Figure 3.3: . . . and the notion of comparison.

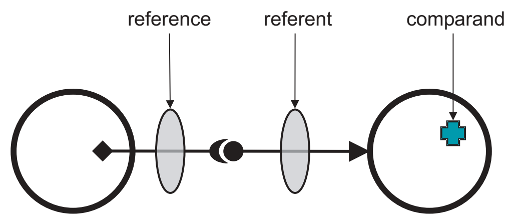


Figure 3.4: GILGUL's model: references hold referents that represent objects, and each object stores a comparand.

problems shown above.

This dissection of object identity concerns is illustrated in Figure 3.4. By default, all object identities are split into *references*, *referents* and *comparands*. A reference refers to a referent which is a representation of the actual object, and each object is supplemented with an attribute that stores a comparand.

In our approach, neither references nor referents are ever compared. Instead, comparands are exclusively used for comparison. They are system-generated, globally unique values that cannot be manipulated by a programmer. So the comparison of two variables v1 == v2 always means the comparison of the comparands stored in the objects being referred to: v1.comparand == v2.comparand. However, they are never used for referencing.

Figure 3.5 illustrates why this strict separation of reference and com-

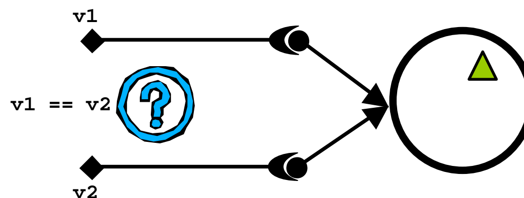


Figure 3.5: Comparison of references and referents may yield different results, so comparands are used for comparison to avoid conceptual ambiguity.

parison is needed. Assume that you want to compare v1 and v2. In this situation, comparison of variables without the use of comparands is ambiguous on the conceptual level, since comparison of the references would yield false, whereas comparison of the referents would yield true. The decision for one or the other option would be arbitrary and cannot be justified other than by technical considerations only. Therefore, we opt for comparison of properties stored inside of the objects involved and thus make comparison of variables unambiguous.[2]

Note that we have avoided to use terms like "key", "identifier", "OID", "surrogate" and so on that are generally used in the literature for object identity and similar concepts. Instead, we have intentionally opted for *reference* ("the referring entity"), *referent* ("what is being referred to") and the artificial word *comparand* ("what is being compared") in order to stress the tasks of the concepts behind these terms.

Essentially, GILGUL's model boils down to the use of double indirection and the use of a default attribute for comparison which are both not breathtakingly new. However, what is new in our approach is the inclusion of these concepts into a programming language in a semantically clean way so that they are open to manipulation in unanticipated contexts.

3.2 Gilgul

Based on this scheme, we outline the operations introduced in GILGUL in the following sections. They are sketched as extensions of Java and introduce means to manipulate referents and comparands.[3]

3.2.1 Operations on Referents

In GILGUL, the *referent assignment operator* #= is introduced to enable the proposed replacement of objects. The referent assignment expression demInstance #= euroInstance replaces the referent of the variable demInstance with the referent of euroInstance without actually changing any references. Effectively, this means that all other variables which hold the same reference as currency refer to the object euroInstance as well. Consider the following statement sequence.

```
demInstance = new DEM();
demAlias = demInstance;
demAlias #= euroInstance;
```

[2] A more complete rationale is given later in this chapter.

[3] More details are given in subsequent chapters and Appendix A.

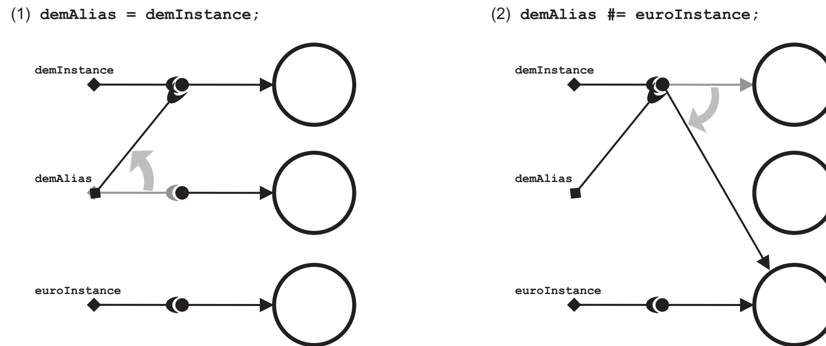


Figure 3.6: Referent Assignment: After execution of demAlias #= euroInstance, all three variables refer to the same object. Since demInstance holds the same reference as demAlias, it is also affected by this operation.

After execution of the referent assignment, all three variables are guaranteed to refer to the same object euroInstance, since after the second assignment, demInstance and demAlias hold the same reference (see Figure 3.6).

Note that the referent assignment operator #= is a reasonable language extension due to the fact that the standard assignment operator = copies the reference from the right-hand operand to the left-hand variable, but not the referent.

3.2.2 Operations on Comparands

Technically, it is clear that comparands may be copied freely between objects. There are in fact good reasons on the conceptual level to allow the copying of comparands. For example, decorator objects usually have to "take over" the comparand of the decorated object so that comparison operations that involve "direct" references to a wrapped object yield the correct result.

Comparands are introduced in Gilgul by means of a new basic type comparandtype which can be used to create new comparands via comparand creation expressions (new comparand). By default, the definition of java.lang. Object includes an instance variable of this type, as follows.

```
public class Object {
  public comparandtype comparand;

  ...
}
```

The equality operators == and != that are already defined on references in Java are redefined in Gilgul to operate on comparands, such that v1 ==

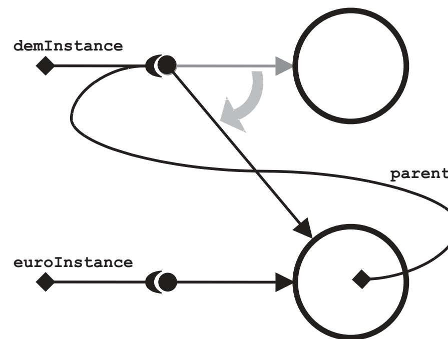


Figure 3.7: Naive application of demInstance #= euroInstance may result in an unwanted cycle. When euroInstance.parent holds the same reference as demInstance beforehand, it will also refer to euroInstance afterwards.

v2 means the same as v1.comparand == v2.comparand, and v1 != v2 means the same as v1.comparand != v2.comparand.

Given these prerequisites, we can let a wrapper "take over" the comparand of a wrapped object in order to make them become equal by simply copying it as follows.

```
wrapper.comparand = wrapped.comparand;
```

3.2.3 Reuse of Existing State

Returning to our given problem, we are now able to apply the new operations to achieve the desired replacement of objects atomically. We can apply demInstance #= euroInstance to let euroInstance replace demInstance consistently for all clients that have references to the original demInstance.

However, one has to be careful when euroInstance wants to delegate messages to the original demInstance. Regard the following naive sequence of operations.

```
euroInstance.parent = demInstance;
demInstance #= euroInstance;
```

This would be erroneous, because afterwards euroInstance.parent would refer to euroInstance, since it contains the same reference as demInstance according to the first assignment. This results in a cycle and therefore, to non-terminating loops for messages that are delegated by euroInstance (see Figure 3.7). The following statement sequence however is correct (see Figure 3.8).

```
// let a fresh reference refer to demInstance
ICurrency tmp #= demInstance;
```

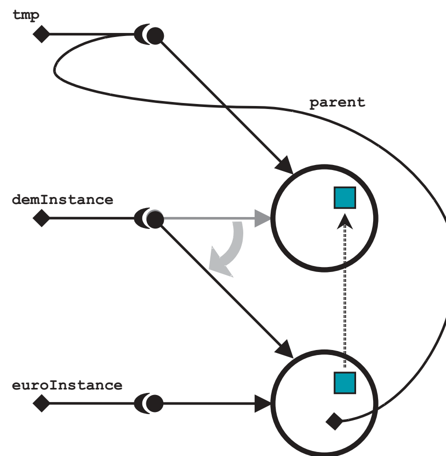


Figure 3.8: Correct application of demInstance #= euroInstance. When euroInstance.parent holds a different reference to the same object as demInstance beforehand, it will still refer to the original demInstance afterwards, since the temporary reference is not affected. In this way, the state of replaced objects can smoothly be reused.

```
// use tmp instead of demInstance for delegation
euroInstance.parent = tmp;

// ensure that equality behaves well
euroInstance.comparand = demInstance.comparand;

// tmp and so euroInstance.parent remain unchanged
demInstance #= euroInstance;
```

The actual replacement of demInstance is initiated by the last operation, and thus is indeed atomic. Further note that the temporary reference can be used to revert the replacement by application of demInstance #= euroInstance.parent.

However, this "Replacement by Wrapper" idiom of using an additional reference for delegation is only needed when newObject actually needs to reuse oldObject. Otherwise, a "simple" replacement is sufficient. In the latter case, reversal of a replacement can also be achieved by the use of an additional reference, but it is not needed for delegation.

3.3 Wieringa/de Jonge Requirements Revisited

In the following section the practicable operations on referents are contrasted with the Wieringa/de Jonge requirements [85], namely Singular Reference,

Singular Naming and Monotonic Designation. We restate each requirement in terms of our approach and show that each can be abandoned because of the differentiation between reference and comparison introduced in our model. We then describe how the referent assignment operation is enabled as a consequence of these relaxations.

Monotonic Designation

The Monotonic Designation requirement demands that, if a name refers to a particular object at a given time (and vice versa), this name is bound to that object at any point in time thereafter. (Recall that in this context, a name means a physical address or a reference to an object table, and not the name of a variable.)

The abandonment of this requirement can be stated as follows:

Variable Designation: If a name refers to a particular object at a given time, it may refer to a different object at another time. Accordingly, if a particular object is referred to by a name, it may be referred to by a different name at another time.

The referent assignment operator #= just "implements" Variable Designation. An assignment demInstance #= euroInstance replaces the former demInstance object with the euroInstance object. All references to demInstance remain unchanged but nevertheless refer to the euroInstance object afterwards.

Singular Naming

The Singular Naming requirement demands that each object is referred to by exactly one name. By abandoning this requirement we get:

Multiple Naming: Each object may be referred to by an arbitrary number of names.

We have already depicted an example of Multiple Naming without comment: everytime a referent assignment is executed, two different names refer to the same object afterwards. So for example in the "Replacement by Wrapper" idiom shown above, after execution of tmp #= demInstance two different names refer to the same object. This characteristic is utilized in that idiom because it helps to set the parent reference to the correct object – if more than one name can refer to the same object, this fact can be used to refer to specific names in a referent assignment.

Hence, the referent assignment operator introduces both Variable Designation and Multiple Naming.

Singular Reference

The Singular Reference requirement demands that each name refers to exactly one object at a given time. Again, by abandoning this requirement we get:

Multiple Reference: Each name may refer to an arbitrary number of objects at a given time.

When calling "The President of the USA"'s phone number, you may actually talk to a representative who might be in a position to act autonomously in the name of his employer in particular well-defined areas. (The representative can be seen as a kind of "wrapper person".) If you have managed to convince him to reach a certain decision, you may rightfully be said to have "The President"'s word without ever having heard it. This is an (admittedly artificial) example of "The President of the USA" referring to more than one person at a time.

A somewhat more natural example of Multiple Reference is a phone number identifying The White House as a whole, including all its employees, so in this way referring to more than one person at the same time. The different employees might then be reached by different extensions.

Variations of Multiple Reference Multiple Reference can be realized as follows. Instead of each referent referring to a concrete object, they consist of simple data structures. The first variation of Multiple Reference can be realized as an array or a linked list, which stores references to the proper objects. When sending messages via name, it is sent to one of these objects providing a method for this message in its interface. When more than one object understands the message, the list may provide the order in which the objects are to be searched.

The second variation, a collection of objects supplemented with "extensions" or "port" numbers (like in internet addresses), can be realized as a hash table, with port numbers as keys and references to the proper objects as hash-table slots. Both kinds of Multiple Reference can even be combined via Red-Black Trees, that have both properties, mapping keys to slots and keeping these mappings in a programmer-defined order.

These examples for Multiple Reference illustrate that there must be a resolution mechanism that decides which object receives a message when it is being sent via a reference that refers to more than one object. In the first kind the decision can be automatically based on the order in which the objects are stored within a linear data structure. This resembles several models of forwarding or delegation, for example [12] and [48], where the runtime system automatically decides which object of a list of specially linked objects receives a particular message.

In the second kind the sender of the message has to augment the message with additional information (a port number). This resembles several role or view models [7] as well as the COM component model [10], where the sender may select a distinguished perspective (role, view, or interface, respectively) on an object before it sends a message.

From this we can see that there are many proposals for how to decide which object receives a particular message, and none of them covers all cases. For this reason, our model carefully excludes the choice of a single one resolution mechanism. It also excludes a standard mechanism for deciding which comparands are to be compared when references to more than one object are involved in a comparison operation. Instead, these choices are deferred to the programmer, who can either define a special resolution mechanism or select from predefined ones.

Multiple Reference through Wrappers So instead of introducing new operators in order to express Multiple Reference, Gilgul has been designed to enable the reuse of several features of the Java platform to achieve the same expressiveness.

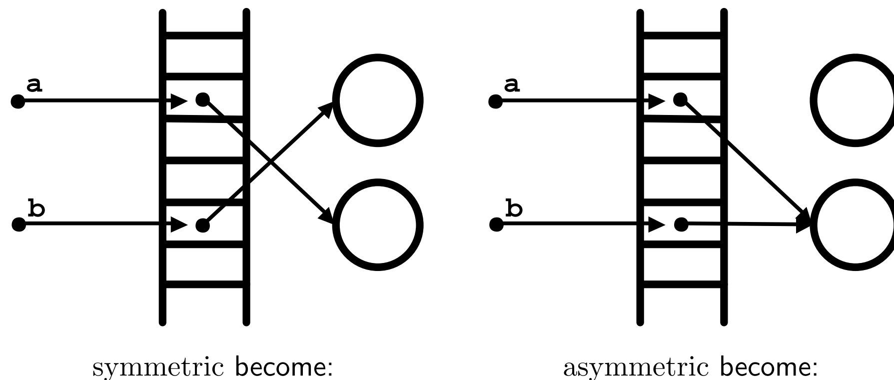
By using the given reference assignment operator #=, it is possible to let a variable that has referred to a single object before a certain point in time refer to a wrapper object that delegates messages to several destination objects afterwards. The following example is based on that idea.

```
MyClass wrapper = new MyClass() {
  void m() {wrapperObject.m();}
  void n() {visualObject.n();}
}

visualObject #= wrapper;
```

Inclusion of methods for manipulating the wrapper's contents enables the inclusion and removal of decorated objects afterwards. A variation of a role or view model can also be expressed by dispatching method calls in a similar way. An example is given in Section 7.1.2.

It is important to note that a programmer is not meant to program wrapper classes explicitly for each case of Multiple Reference. Instead, a class library can be provided to handle the different cases, and a programmer simply reuses this library. To be truely generic, such a library may employ the feature of dynamic proxy classes introduced in version 1.3 of the Java 2 SDK ([78], "Summary of New Features"), or other means of reflective metaprogramming.



symmetric become: asymmetric become:

Figure 3.9: The symmetric version of Smalltalk's become: swaps two objects without changing their references. The asymmetric version lets one reference refer to the same object as another refence, but not vice versa.

3.4 Smalltalk's become:

The programming language Smalltalk provides an operation become: that enables the programmer to (symmetrically) "swap" two objects without actually changing their references. Early Smalltalk implementations were based on object tables, and so this operation was straightforward to implement. Modern Smalltalk implementations that do not use object tables either take considerable effort to implement become: correctly, or reduce become: to an asymmetrical operation [74].

In Smalltalk, the wrapping of a demInstance can be implemented as follows.

```
demInstance become: euroInstance.
demInstance setParent: euroInstance.
```

This works for both the symmetric and asymmetric version of become: but does not carry out the replacement atomically, because the parent field is only set *after* the actual replacement. Furthermore, the need to seemingly set the demInstance's parent instead of euroInstance's parent may be regarded as counterintuitive.

In order to achieve the desired atomicity of the replacement, the "Replacement by Wrapper" idiom presented in this chapter can also be used in Smalltalk with the symmetric version of become:, as follows.

```
tmp become: demInstance.
euroInstance setParent: tmp.
demInstance become: euroInstance.
```

This idiom works only because it relies on the fact that references are not "merged" by this operation. However, references are in fact merged by

the asymmetric version of become: and we are not aware how atomicity can be achieved in this case. Further note that for the symmetric version, this sequence also affects euroInstance which refers to tmp afterwards.

There are still some other serious drawbacks of Smalltalk's become:. It is not type-safe because it does not check for compatible layouts of the objects involved. Furthermore, it does not pay any attention to methods currently executing on the objects, but just lets them continue to execute on the "swapped" objects.

Because of these obscurities, [74] states that "[t]he become: message is dangerous since it is easy to make a mess with it if you don't fully understand what it does." In contrast, GILGUL offers a clean and explicit model of reference and comparison that avoids these obscurities; furthermore, its referent assignment operator respects Java's type system without sacrificing flexibility (see Chapter 4); and it introduces means to correctly and explicitly deal with active objects (see Chapter 5).

3.5 A Mental Model for Comparands

As already mentioned in the introduction to this chapter, the GILGUL model is not a minimal model. This is evident in Figure 3.4: We introduce a comparand for comparison purposes because we do not otherwise want to choose between the reference and the referent to compare. However, because comparands are stored in objects, and objects are potentially replaced by referent assignments, it could be argued that comparands are strictly not necessary because they effectively lead to a comparison of referents unless comparand assignment is made use of in a program. After all, the Java programming language already includes a way to influence equivalence relationships by way of overriding the equals method that is defined for all objects.

Nevertheless, we are convinced that comparands provide a more approachable way to influence the result of comparison operations that should suit the programmer's mental model better. See Figure 3.10 as an illustration: It shows a complex relationship between three members of a family. The right person plays the role of a wife for the left person, the role of a mother for middle person, and at the same the role of the school principal of the school that the middle person attends and in which the left person works as a teacher. Likewise, the left person plays the role of a husband for the right person and the roles of both a father and a teacher for the middle person. Depending on the context in which each of the three persons interacts with another, a different *role object* of that person is used and interacts with a specific role object of the other person. (See Section 7.1.2 for some more details about how role models can make use of GILGUL constructs.)

Although different objects are used to represent the same person in this example, they all describe just an aspect of a larger conceptual entity, i.e. a

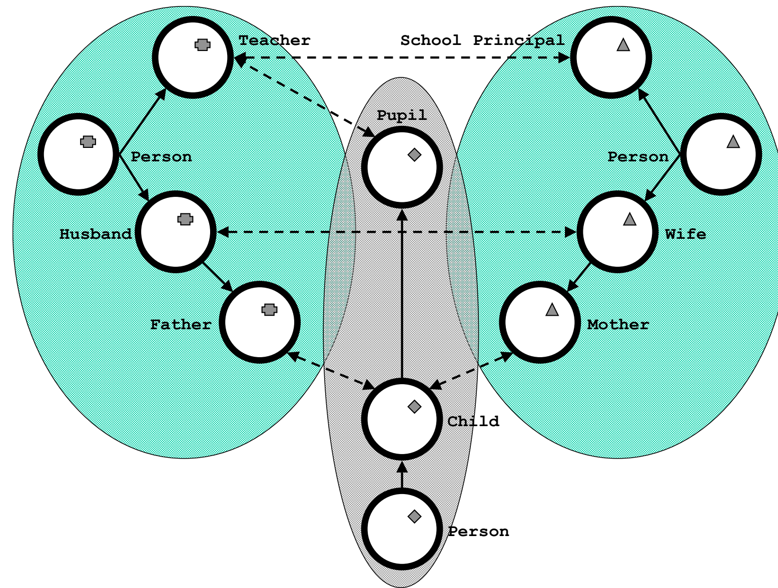


Figure 3.10: An example of a complex relationship between three different persons, realized by way of roles. The different role objects are marked by comparands to indicate to which conceptual entity they belong.

person. Therefore, these different objects should appear to be the same when being compared. This can be accomplished by overriding equals and delegating it to the core object which performs the actual comparison. However, this can lead to complex interactions, especially because overriding equals also leads to the requirement to override the hashCode method accordingly. (See Section 4.1.3 in the following chapter for more details about interactions between comparisons and hashCode.)

Compare this with the obvious and straightforward approach taken in Figure 3.10: The different objects that belong to a conceptual entity are just marked by the same symbol. This is exactly how uses of comparands should be thought of: They mark the objects that belong to a conceptual entity, and in turn the implementation of a programming language and run-time environment can take care of establishing the right dependencies (comparison operations, hash codes) that follow from these marks. In other words, comparands define *equivalence classes* on objects. The comparand assignment operation just allows changing these equivalence classes at run time, which leads to an important contribution to the field of unanticipated software evolution. See Appendix B for some examples and already known uses (without explicit programming language support) of such a notion of comparands.

3.6 The Name Gilgul

This dissertation's title "Transmigration of Object Identity" is inspired by a concept in Jewish mystical thought known as "gilgul" which in turn has been chosen as the name for the model and language extension described in this thesis. "Gilgul" is a variation of a belief in transmigration of souls in which not a soul in its entirety, but a "nitzotz" or spark of soul can reincarnate, so an individual person can possess several such sparks of people who have lived previously. This has been the idea behind the concepts of Variable Designation, Multiple Naming and Multiple Reference: The comparands of objects may "reincarnate" when being copied to other objects, and references can possess "sparks" of other objects by letting them refer to more than one object at a given time. For example, see [66] for a discussion of "gilgul".

Chapter 4

The Programming Language Gilgul

The previous chapter has already given an outline of the fundamental constructs of the programming language GILGUL: comparands, comparand assignment and referent assignment. In this chapter, we introduce, define and describe these constructs in more detail, together with appropriate rationales. Furthermore, it introduces control facilities for the new assignment operations as well as the so-called *typeless* classes. This is accompanied by a reasoning about the type soundness of the language GILGUL, especially with regard to related approaches.

The programming language GILGUL has been carefully designed not to compromise compatibility with existing Java sources. However, it introduces new keywords, like `comparand`, `typeless`, and so on, that are needed to declare respective properties of program parts. Such keywords are not available anymore to the programmer as names for variables, methods, classes, and so on, and need to be renamed accordingly should they occur in legacy Java code. Apart from that, existing Java program will run without noticeable changes in functionality.

4.1 Basic Language Constructs of Gilgul

There are four levels that can be manipulated when dealing with variables: the reference and the object level that already exist in Java, and the referent and the comparand level that are new in GILGUL. A class instance creation expression (new MyClass(...)) results not only in the creation of a new object, but also in the creation of a new reference, a new referent and a new comparand.

4.1.1 Operations on Referents

The referent assignment operator #= is an assymetric object replacement operator, as introduced in the previous chapter. (See Section 3.2.1.)

Since the null literal does not refer to any object, the referent assignment is prevented from being executed on null. The expression null #= *expression* is rejected by the compiler, and v #= *expression* throws a GilgulRestrictionException when v holds null. This ensures that a programmer is not able to erroneously redirect all variables that hold null to a non-null object. Note, however, that v #= null is valid when v does not hold null and redirects all variables that have the same reference as v to null.

The GilgulRestrictionException is an unchecked exception, so this case is similar to the throw of a NullPointerException when attempting to access the properties of a variable that holds null. Both kinds of exception can be avoided by testing against null beforehand. See Section A.10.1 for more details about the GilgulRestrictionException.

4.1.2 Operations on Comparands

Comparands and comparand assigment are introduced in the programming language GILGUL along the lines of the notions as described in the previous chapter. (See Section 3.2.2.)

Ensuring the uniqueness of a single object is always possible by assigning a freshly created comparand as follows: v.comparand = new comparand.

In Java, the equality operator == is only accepted by the compiler if one operand can be cast to the type of the other. The language thereby excludes meaningless comparisons of arbitrarily-typed references [34]. Consequently, a comparand assignment of the form expression1.comparand = expression2.comparand is only accepted at compile-time if expression2 can (potentially) be cast to the type of expression1. This restriction can be lifted by an explicit cast as follows: expression1.comparand = ((Object)expression2).comparand. Other forms of comparand assignment are always accepted. There are no restrictions imposed on comparand assignment at run time.

Since null does not have a comparand an attempt to access null.comparand is rejected by the compiler, and v.comparand throws a GilgulRestrictionException when v holds null. This ensures that testing equality against null is guaranteed to be unambiguous.

The actual implementation of comparands is hidden from programmers. In particular, GILGUL prevents comparands from being arbitrarily cast and, for example, does not allow arithmetic operators to be executed on comparands.

Since comparands cannot be manipulated directly, there are no limitations on how they are implemented in a concrete virtual machine. The only requirement they have to fulfil is that if v1.comparand and v2.comparand

have been generated by the same (different) class instance or comparand creation expression, then v1.comparand == v2.comparand yields true (false), and v1.comparand != v2.comparand yields false (true).

One reasonable and efficient implementation of comparands are 64-bit unsigned integers with comparand creation being accomplished by increment of a global counter. This scheme provides for approximately 10 billion unique comparands per second for half of a century. (On the other hand, 32-bit values are usually not big enough to ensure uniqueness for long-running applications. At a rate of 1000 comparands per second, they wrap around after roughly 6 weeks.)

4.1.3 Operations on References and Objects

Besides GILGUL's new operations on referents and comparands, the operations on references and objects are still available as a matter of course. However, there are some interesting interdependencies between the standard methods equals(...) and hashCode() and the ability to copy comparands between objects.

Note that the standard definition of equals(...) relies on the definition of the equality operators == and !=, and therefore is affected by their redefinition to operate on comparands instead of references. Hence, it yields true iff the comparands of the corresponding objects are the same. As a consequence, the standard definition of hashCode() has been changed to return a hash code value for an object's comparand, since the contract of hashCode() is based on equals(...) – [78] states that if "two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce the same integer result." A comparand's hash code value fulfils the same requirement, except that it is based on the equality operator on comparands (==) instead of on the method equals(...).

From this perspective, comparands can be seen as an alternative way to redefine the method equals(...) by just copying them between objects. This complies with the mental model for comparands as illustrated in Section 3.5. Furthermore, comparands relieve the programmer of the requirement to remember to override hashCode() accordingly whenever he or she is about to change the equality semantics of an object via comparand assignment.

Another consequence is that the equality operators == and != and the method equals(...) are always redefined in a uniform way by copying of comparands, unless equals(...) is explicitly overridden by the programmer. This complies with the suggestion that there should be only one comparison

operation per object, as is stated for example in [35][1] or [51][2].

4.2 Replacement of an Object with its Wrapper

Recall the object replacement example given in the previous chapter. (See Section 3.2.3.) It shows that in general, an object replacement consists of four consecutive steps. We repeat the steps here:

```
// let a fresh reference refer to the original object
Object tmp #= orgObject;

// use tmp instead of orgObject for delegation
newObject.parent = tmp;

// ensure that equality behaves well
newObject.comparand = orgObject.comparand;

// tmp and so newObject.parent remain unchanged
orgObject #= newObject;
```

These four steps are sufficiently complex to be remembered by an application programmer so in principle, this would call for another language extension in order to simplify the replacement operation. Fortunately, this complexity can be hidden in the class of the wrapper as follows.

```
class Wrapper ... {

    Target parent; // the object being delegated to

    // constructor
    public Wrapper(Target orgObject) {
        Target tmp #= orgObject;
        this.parent = tmp;
        this.comparand = orgObject.comparand;
    }

    ...
}
```

[1]It states that a programming language should provide "only one copy method and one comparison method for each class. The designer of the class, rather than its clients, should choose appropriate semantics for these methods."

[2]"Use method equals instead of operator == when comparing objects. [...] Rationale: If someone defined an equals method to compare objects, then they want you to use it. Otherwise, the default implementation of Object.equals is just to use ==." From this perspective, the decision to include two different ways to compare objects into the Java programming language may be considered questionable.

Now, the constructor consists of the first three steps of the object replacement idiom given above. In this way, the replacement operation is reduced for the application programmer to the following simple statement: orgObject #= new Wrapper(orgObject).

4.3 Initializing Referent Assignments

Without further notice, we have introduced another language construct in the above example, the *initializing referent assignment*: The left-hand side of a referent assignment expression is not only allowed to be an already initialized variable, such that all variables that contain the same reference are also affected, but it can also be an uninitialized variable as well, as in the first line of the constructor for the class Wrapper above.

Since an uninitialized variable does not hold anything that could be replaced, the semantics are that a fresh reference is created for the referent designated by the right-hand side of the referent assignment expression.

Note that this is a different situation from that in which the variable of the left-hand side contains null which is generally forbidden by GILGUL (see above). This gives rise to another distinction that needs to be specified for the language GILGUL: Java allows the separation of declaration and initialization of local variables. This in turn allows for complex initialization schemes, for example as follows.

```
MyClass object;

if (condition) {
  object = initialization1;
} else {
  object = initialization2;
}
```

Here, it is guaranteed that exactly one assignment takes place, either that of initialization1 or that of initialization2.

GILGUL generalizes this notion and allows for similar initializing referent assignments, as follows.

```
MyClass object;

if (condition) {
  object #= initialization1;
} else {
  object #= initialization2;
}
```

In order to avoid ambiguities and give initializing referent assignments reasonable semantics, the Java's concept of *definite assignment* is reused (see Chapter 16 in [34]). We give the following definitions:

A referent assignment is said to be *definitely initializing* if it can be statically proven that the variable on the left-hand side is definitely unassigned on all execution paths up to this referent assignment. It is said to be *definitely non-initializing* if it can be statically proven that that variable is definitely assigned on all execution paths up to this referent assignment.

If a referent assignment is definitely initializing then a compiler for the language Gilgul generates a code sequence that implements initializing referent assignment as described above. If it is definitely non-initializing then a Gilgul compiler generates a code sequence that implements normal referent assignment as described above. If a referent is neither definitely initializing nor definitely non-initializing, because various paths up to this referent differ in this regard, then a Gilgul compiler must reject that referent assignment at compile time.

So for example, the following code is invalid.

```
MyClass object;

if (condition) {
  object = anInitialization;
}

object #= replacement;
```

Here, if condition would hold at run time, the referent assignment would need to be non-initializing, otherwise it would need to be initializing.

Note however, that the following code is acceptable.

```
if (condition) {
  object = initialization1;
} else {
  object #= initialization2;
}
```

In the latter case, both paths are unambiguous with regard to the initialization status of object.

Rationale The semantics of an initializing referent assignments differs strongly from that of a non-initializing referent assignment with regard to the effects these variants potentially have on other variables in the running system. It seems well-justified and in line with aims of the design of the Java programming language to enforce consistent use of these two different

uses of referent assignment. Furthermore, a strict separation of these two variants allows for a cleaner and more efficient implementation as is shown in Chapter 6.

Since the terms *definitely initializing referent assignment* and *definitely non-initializing referent assignment* are well-defined in terms of *definite assignment* and *definite unassignment*, it is possible to reuse the rules stated for Java source code in Chapter 16 of [34] without any changes. Therefore, we do not repeat them in this thesis.

4.4 Control Facilities

Gilgul offers facilities for controlling what operations are valid on concrete referents and comparands. Since references and comparands are created at the same time as their initially corresponding objects via class instance creation expressions, these restrictions have to be given in constructor declarations as follows.

```
class SecurityManager {
  // constructor
  SecurityManager() with fixed referent, bound comparand {...}

  ...
}
```

The possible restrictions are fixed, bound or none for comparands, and fixed, bound or none for referents.

Restricitions on Comparands If no restriction is declared for a comparand, it may be copied or replaced freely. If a comparand is declared as fixed, it cannot be replaced with another comparand, but it may be copied elsewhere. If a comparand is declared as bound, it may neither be replaced nor copied, which means that a bound comparand is implicitly fixed.

The rationale behind this implication is that if a programmer declares a comparand as bound, he/she wants to guarantee that there does not exist a copy of this comparand elsewhere. However, if a bound comparand could be replaced with a comparand of another object, this guarantee would be violated, because the other object could not be prevented from using the latter comparand.

Restrictions on Referents Similarly, if no restriction is declared for a referent, it may be copied or replaced freely. If a referent is declared as fixed, it cannot be replaced with another one, but it may be copied elsewhere. If a referent is declared as bound, it may neither be replaced nor copied, which

means that a bound referent is implicitly fixed. The rationale is the same as for comparands.

These constructs allow the flexible declaration of detailed restrictions on comparands and referents, ranging from the allowance of all operations introduced in the previous sections to the reduction to the "classic" approach of dealing with object identity. For example, consider the following class declaration.

```
class TraditionalObject {
  // constructor
  TraditionalObject() with bound referent, bound comparand
  { ... }

  ...
}
```

Instances of this class can neither have their referents nor comparands replaced nor copied.

Since the use of wrappers in conjunction with GILGUL's new operations is so salient, we have opted for allowing comparands to be initialized in the constructor head as follows.

```
class EuroWrapper ... {
  // constructor
  EuroWrapper(DEM dem) with fixed comparand = dem.comparand
  { ... }

  ...
}
```

This is accepted by the compiler because the constructor's parameters are visible in the scope of the restriction declaration and can be used to initialize the (fixed or unbound) comparand. Especially in the case of fixed comparands, this both allows the comparand to be set to a given comparand once and ensures that it is never changed afterwards. This kind of comparand initialization is carried out before a call to a super constructor in order to ensure consistency.

Note that in contrast to the standard access modifiers of Java (public, protected, private), the restrictions on comparands and referents are not attached to variables and consequently cannot be checked statically in the general case. Therefore, comparand assignments (v1.comparand = v2.comparand) and referent assignments (v1 #= v2) may throw instances of GilgulRestrictionException. The restrictions imposed on the null literal in Sections 4.1.1 and 4.1.2 can be restated as if null's "constructor" had been declared

with fixed referent, bound comparand, so the presumed exceptional cases for null are direct consequences of this "declaration".

Alternatively to the scheme presented above, it would have been possible to allow for declaration of restrictions either at the class level (class SecurityManager with fixed referent ...), or at each instance creation (new SecurityManager(...) with fixed referent, ...). However, declaration of restrictions at the class level is most probably too coarse-grained in practice because an implementor of a class might want to declare private and/or protected unrestricted constructors. On the other hand, declaration of restrictions at the instance level is likely to be too fine-grained because it makes it harder to prevent clients from instantiating "loose" objects.[3]

It turns out that declaration of restrictions at the constructor level accomodates all possible usage scenarios, ranging from complete prevention of comparand/referent assignment operations up to complete openness to modification of object identity. For example, the possible options in between include restricted use of comparand/referent assignment for clients via public constructors and open use of comparand/referent assignment for subclasses via protected constructors at the same time.

However, note that GILGUL's flexibility comes at the price of an increased complexity of contracts since it still must be determined which kinds of referent assignment and comparand assignment are valid for concrete classes or objects. The possible restrictions on comparands and referents just help to make these contracts more explicit, but they do not reduce their complexity at the conceptual level as is the case for the standard access modifiers [49].

4.5 Type Issues

The referent assignment operator #= respects Java's type system. In the following sections we explore what this actually means and especially show that this may lead to unnecessarily restricted situations. An extension of Java's type system is presented afterwards that allows "pure" implementation classes to be declared that do not define new types. This is a novel approach which to our knowledge is not available in any previous language. This extension allows the restrictions to be resolved which are associated with the type-sound use of the referent assignment operator.

4.5.1 Additive and Subtractive Replacement

The referent assignment operator allows an object to be replaced always with another one that implements at least the same types (*additive replacement*). A special case is the replacement with an object that is an instance of exactly

[3] The only solution would be to prevent clients from using constructors at all, and require them to go through factory methods [31].

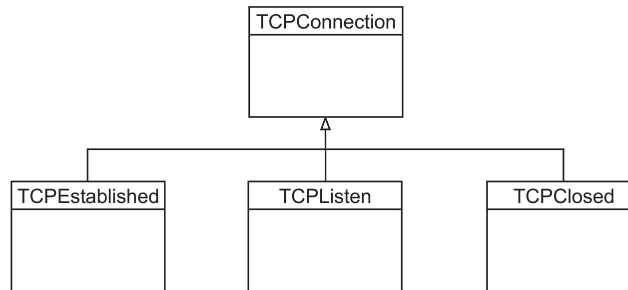


Figure 4.1: An example of the State pattern. Usually, only TCPConnection is used as a type.

the same class, but this statement naturally also includes objects that are instances of any of the old object's subclasses. This is a direct consequence of the Liskov Substitution Principle [55].

The situation becomes more complex when you want to use GILGUL's constructs in a setting where *subtractive replacement* is inevitable. Assume that you want to implement the State pattern [31] without the proposed use of forwarding. Instead, GILGUL's referent assignment operator is to be used to consistently express state transitions for all clients that refer to a shared state object.

So for example, an instance of the State pattern is illustrated in Figure 4.1 (adapted from [31]). The state transition of a TCP connection can be expressed in GILGUL as follows: connection #= new TCPEstablished().

However in general, type safety is not ensured when a reference to the previous state object exists that has an incompatible type, as follows.

```
TCPListen listen = new TCPListen();
connection #= listen;

... // after some time
connection #= new TCPClosed();
```

Just before the last step of this code sequence, the listen variable might still refer to the previous state, and so the referent assignment operator cannot statically guarantee type safety.

In this and similar situations, before replacement of an object with another one, it must be checked dynamically that the old object is referenced only by variables that expect it to implement the intersection of the types implemented by both the old and the new object. Therefore, the run-time system has to keep track of all references to an object and their respective types. However, in order to ensure that subtractive replacements are always possible whenever needed, an extension of GILGUL's type system is inevitable.

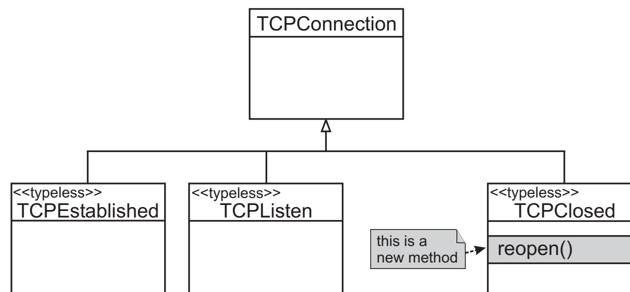


Figure 4.2: How can a new method in a typeless class be called from the outside?

4.5.2 Typeless Classes

On closer examination of the TCPConnection example, it can be noticed that classes TCPEstablished, TCPListen and TCPClosed are most likely never needed as actual types of their own. Instead, all clients that are in need of a TCPConnection will in fact always use this general class type. This can be expressed explicitly in GILGUL by adding the modifier typeless to the classes that are not needed as types (all except for TCPConnection) as follows.

```
typeless class TCPEstablished extends TCPConnection {
  ...
}
```

Afterwards, these classes can still be used as any other class in most respects, for example within instance creation expressions. However, they cannot be used as types anymore, in the sense that variables must not be declared as being of a type of a typeless class. Consequently, instances of these classes (TCPEstablished, etc.) can always be replaced with instances of any class of the given hierarchy, since they all implement the same set of types (which only consists of the type of TCPConnection and its supertypes) by definition. This property results in the desired applicability of subtractive replacement.

4.5.3 Cast Expressions

Assume that TCPClosed additionally defines a public method reopen() which is not defined in the other classes of the TCPConnection hierarchy (see Figure 4.2). How can this method ever be called from outside of class TCPClosed? As we already know, the only way instances of this class can be used is via references of type TCPConnection. Therefore, sending the message reopen() to such references would result in compile-time errors since this method is not defined in TCPConnection. However, message reopen() can be sent to a connection by casting it to the TCPClosed class beforehand as follows.

```
((TCPClosed)connection).reopen();
```

Class cast expressions do not conflict with the original goal of typeless classes, that is to allow for subtractive object replacement. Variables which are cast to a typeless class still cannot be assigned to variables that are declared to be of this very class type, because the restriction still holds that it must not be used as a type. Therefore, in the very moment of applying the referent assignment operator to an instance of a typeless class, it is still ensured that there are no variables in the running system that expect the new object to implement the old object's class type.

4.5.4 The with Statement

In order to conveniently express cascaded method calls to the same variable in the presence of typeless classes, Gilgul introduces a with statement that is reminiscent of the similar statement in the programming language Oberon [70]. In Gilgul, it can be used as follows.

```
with (object instanceof aClass) {
  ...
}
```

Its effect is that object is regarded as an instance of the respective class for the scope of the following (block) statement. The left-hand side of the with condition can be any interface or class, including typeless classes. Given this statement, a call to a method defined in a typeless class can be expressed as follows.

```
with (connection instanceof TCPClosed) {
   connection.reopen();
   ...
   connection.otherExclusiveMethods();
}
```

Note that whereas in Oberon the with statement is introduced to allow for compiler optimization, such that code for the with condition is emitted only once, in Gilgul the with statement is syntactic sugar only. Since the object referred to by the variable in the with condition can always be replaced with another object (for example, within another thread), the condition might not hold for the following block completely, possibly resulting in a class cast exception at any place within that block where the variable is actually used.

4.5.5 Relation to Java's Interfaces

Apart from the confined use of typeless classes as types, they do not differ from usual classes. Especially, they are allowed to implement any interface, as follows.

```
typeless class C extends D implements I {
  ...
}
```

Note, however, that this declaration introduces a new type into the class hierarchy if interface I is not implemented by any of C's superclasses. As soon as a variable of type I refers to an instance of class C, this instance can be replaced only with objects that simultaneously are instances of any subclass of D and implement I.

Yet we have not chosen to disallow typeless classes to implement interfaces since this feature can be utilized for a clean separation of types and implementations as follows.

```
interface ICurrency {
  ...
}

typeless class Euro implements ICurrency {
  ...
}
```

Although the main purpose of Java's interfaces is the definition of "pure" types, we have also chosen to allow for the declaration of *typeless interfaces*. For example, this might be useful in order to group related methods into a typeless interface without declaring a new type. This is similar to Smalltalk's concept of categories. Typeless interfaces could also help to avoid the declaration of new types just to introduce application-wide constants, or can be used as marker interfaces to indicate specific class-related properties, like Cloneable or Serializable, which are also usually not intended to be used as types.

4.5.6 Relation to Software Evolution

The introduction of typeless classes fits perfectly to the goal of widening the range of unanticipated software evolution. It is always possible to extend an existing class hierarchy by additional typeless classes and thus allow for both additive and subtractive object replacements. There is no need to change existing classes, so this feature can be used for third-party components without further effort.

The possibility to introduce new methods into a typeless class without restricting the replaceability of its instances also improves adaptability. Since typeless classes can be used in class cast expressions, these new methods can be called within unrelated classes that yet are aware of these new methods. Still, the existing class hierarchy does not need to be changed for this purpose.

The fact that casts can only be checked dynamically and therefore might raise class cast exceptions may be regarded as a disadvantage of this proposal. However, this is only the flipside of the possibility to declare *optional methods*, which in turn is a benefit that otherwise cannot be expressed easily.

Typeless classes effectively decouple declarations of optional properties on the one hand, that virtually can be added to and removed from objects, and the actual use of such optional properties on the other hand, which of course may result in the temporary absence of these properties.

Other approaches that allow for optional properties insist on their introduction into the existing class hierarchy in order to allow for static checks of their sound use, but in this way they simultaneously narrow the range of unanticipated evolution of third-party components. See for example the concept of *empty methods* in Component Pascal [80] which are similar to abstract methods but default to empty method bodies in order to avoid exceptions at run time.

Note that the goal of separating interfaces and classes has been proposed explicitly by Cook et al. [14] and, for example, has been addressed in Emerald [40], Sather [81] and Java [34]. However, whereas it is possible to declare pure interfaces/types in one or the other way in all of these approaches, the declaration of classes still implies the accompanying (implicit or explicit) declaration of interfaces in order to use newly declared properties from the outside. In GILGUL, it is possible to declare "pure" implementation classes that must never be used as types. In this way, GILGUL "completes" the separation of types and classes that has been initiated with the former approaches.

Typeless classes are not only useful in conjunction with referent assignments, but also with other programming language constructs, like Generic Wrappers [12] or Delegation [47]. For example, Generic Wrappers could be enabled to dynamically change their wrappees to instances of the wrappee's superclass, if the wrappee is an instance of a typeless class. Currently, Generic Wrappers do not allow for the subtractive exchange of wrappees.

Newer approaches that head for subtractive object replacement, and modify the type system for this purpose in a similar way, are Fickle [28] and Wide Classes [73]. However, these approaches still do not allow for declaration of classes that must not be used as types.

4.5.7 Type Soundness

Java is intended to be a type-sound programming language, and type soundness has been proven for non-trivial subsets of Java. For example, see [29, 64, 79] in [1]. With regard to type soundness, GILGUL is a conservative extension of Java. The following sections list and discuss the features of GILGUL that affect Java's type system.

Comparands

Comparands do not affect Java's type system. Since the comparand field is publicly defined for the class java.lang.Object which is the top-most superclass of all classes, it is guaranteed that it is always accessible in all objects.

One implication of making the comparand field a definition of the class Object is that this field also becomes defined in all interfaces, since the type Object is also a supertype of all interface types. In a strict sense, this conflicts with Java's restriction that interfaces may not contain (non-static) field definitions. However, GILGUL does not generalize interfaces to include field definitions in order to allow for the proper definition of the comparand field. Instead, it treats comparands as a special exceptional case.

In its current definition, GILGUL treats `comparand` as a keyword that cannot be used by programmers for definitions of their own. This justifies the inclusion of comparands in interfaces as a special case, and it also ensures that the comparand field cannot be hidden by programmers in their own classes.

This design is compatible with possible future extensions of Java/GILGUL that may allow for the general inclusion of non-static fields in interfaces.

Referent Assignment

In general, a referent assignment cannot be checked statically in order to ensure run-time type safety without further run-time checks. For example, assume the referent assignment left #= right with right denoting an object of compile-time type R and left denoting an object of compile-time type L. The following cases may occur:

a) R may be the same type as L,

b) R may be a subtype of L,

c) R may be a supertype of L,

d) R and L may be unrelated types.

It seems straightforward to accept case a) and b) at compile-time. However at run time, right might actually refer to an object of a less specific run-time type than that of left, so the referent assignment would not be safe. Likewise it seems straightforward to reject case c) and d) at compile-time. However at run time, right might actually refer to an object of a more specific run-time type than that of left in case c), so the referent assignment would be safe. The rejection is justified in case d) only because Java does not allow for multiple inheritance; otherwise, even in case d) a referent assignment might succeed at run time.

The reason for the disparity of compile-time and run-time type checks for referent assignment is the fact that referent assignments do not operate

on variables but on objects: Whereas a simple assignment in Java stores references to objects in variables, a referent assignment redirects references without directly affecting the contents of variables. In other words, a referent assignment strictly does not deal with compile-time entities.

Still, the Gilgul language specifies the same compile-time type checks for referent assignments as those for simple assignments in Java. This is to ensure the same familiar degree of programmer guidance by the type system as in Java. Case c) above might seem as an unnecessary restriction because Gilgul strictly rejects code that might succeed at run time. However, a referent assignment can always be forced to be accepted by the Gilgul type checker by casting either side to an appropriate type.

In any case, all referent assignments generally need to be checked again at run time, taking into account the actual run-time types of left and right. If right's class is not a subclass of any of the non-typeless superclasses of left's class, then a ClassCastException is thrown. This ensures that a referent assignment never violates the expected type of a variable in any part of a running Gilgul program.

The grammar productions for referent assignments do not allow for chains of referent assignments of the form `a #= b #= c ....` (See Section A.10.7.) This would have seriously complicated the type system with no obvious gain in expressivity. Still, referent assignment uses the form of an expression instead of a statement in order to accomodate possible future extensions of the language.

Null References

In Java, the null reference is generally understood as the single reference to "no" object [54]. Gilgul retains this null reference from Java that has a fixed referent in Gilgul by default, so it cannot be redirected to actual objects. However, the fixed restriction allows for other references to be redirected to null, and in this way allows creating *unbound* null references. For example, Object obj #= null creates an unbound null reference that can be redirected to an actual object later on.

There is seemingly a conflict between the applicability of referent assignments to unbound null references and Java's permissive rules for casting null's "type" to any reference type and vice versa. This conflict has been detected by Sven Müller in his diploma thesis [58] and is illustrated there with the following example.

In Figure 4.3, a reference to an instance of class B is stored in two variables a and b of type A and B respectively. Then, a is assigned null via referent assignment, which results in an unbound null reference, and afterwards it is assigned a fresh instance of class A via referent assignment. If the latter assignment were permissible this would result in b referring to an object that is not an instance of type B, i.e. a run-time type error.

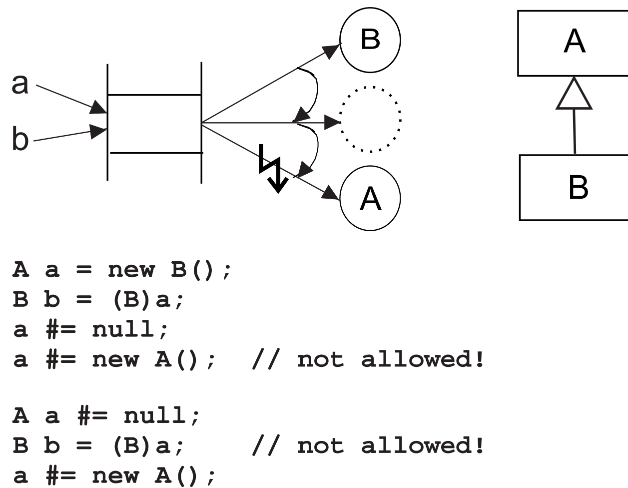


Figure 4.3: In conjunction with referent assignments, unbound null references require additional rules in order to ensure type soundness. (Illustration taken from [58].)

Such situations are actually prevented by storing additional type information with unbound null references that record the required type to be checked in subsequent referent assignments. The required type is determined by the type of the referent that a new unbound null reference replaces. In the example given above, this is the type B that prevents the type violation in the second referent assignment.

In the case of freshly created references, the required type of an unbound null reference cannot be determined by a previous referent. In these cases, the statically declared type is used. For example in Figure 4.3, the third referent assignment records the type A with the freshly created unbound null reference, which in turn prevents the type violation in the subsequent cast to type B. This leads to the probably surprising situation that casting a null reference might issue the throw of a ClassCastException in Gilgul.

However, it is very important to note that all these additional rules exclusively apply to unbound null references, a concept that does not exist in pure Java. Especially, the semantics of the default null reference, as known in Java, are not changed: null can still be arbitrarily cast and involved in assignment operations. Gilgul does not lose its compatibility with pure Java programs because of unbound null references.

Typeless Classes

Typeless classes do also not affect the type soundness of pure Java. The only effect is that typeless classes are forbidden to be declared in certain

contexts. Apart from that, Java's typing rules do not change: the left-hand side of a simple reference assignment is always a variable, and therefore has the type of a non-typeless class. Therefore, the right-hand side still must be of the same or a more specific type than that of the left-hand side.

Typeless classes only affect the type checks of referent assignments insofar it is not sufficient to check the type of the right-hand side against the (most specific) type of the left-hand side, because the left-hand side might be an instance of a typeless class. Therefore, all the non-typeless superclasses of the left-hand side must be taken into account, and the class of the right-hand side must be a a subclass of each of those. At run time, this is the case when the left-hand side is an instance of a typeless class. At compile time, this is only the case when the left-hand side is the keyword `this`. (In fact, this is the only case in which GILGUL's compile-time type rules for referent assignment deviate from those of Java for simple assignment.)

With Statement and Casts

Consider the general form of a `with` statement.

```
with (myExpression instanceof MyClass) {
   . . .
   ... myExpression ...
   . . .
}
```

The `with` statement is a simple rewrite rule that textually replaces all occurrences of a given expression in its body with the same expression explicitly cast to a given class. This means that the resulting code looks roughly as follows.[4]

```
. . .
... ((MyType)myExpression) ...
. . .
```

No implicit assumptions about the type of the expression are made. In fact, the type of the expression is repeatedly checked at each occurrence in the body of the `with` statement at run time.

When a cast expression is used to access a field or method that is otherwise not accessible – for example `((MyClass)myExpression).m()` – this typically results in two steps at run time: first, it is checked whether the resulting object is really an instance of the given class – this potentially results in a `ClassCastException` – and second, the actual field access or method call is performed.

[4] A more detailed description is given in Section 4.5.4.

Since in GILGUL, this two-step process might potentially be interspersed by a referent assignment in another thread that affects the expression being cast, GILGUL requires the combination of theses two steps into an atomic operation. See Chapter 6 for more details. This requirement also ensures the run-time type safety of cast expressions.

Chapter 5

Replacement of Active Objects

This chapter discusses issues that can occur when active objects are being replaced by way of a referent assignment. In the context of this thesis, an object is said to be active at a specific point in time when a method is being executed at that point in time. There are two cases of active objects: either an object is active in the same thread, or in one or more other threads.

The language GILGUL offers various options to handle referent assignments on active objects: the following sections describe default semantics and the language construct *recall* that allows for controlled deviations from the default semantics.

5.1 Default Semantics of Referent Assignment on Active Objects

Consider Figure 5.1 that depicts an attempt at replacement of object1 that still has a method m() executing on it. In such a situation, it is unclear

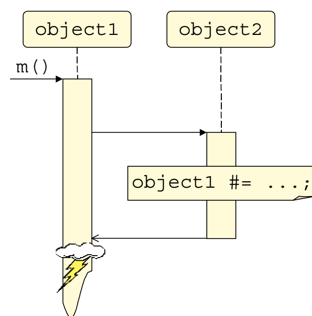


Figure 5.1: Replacement of object1 – what object should m() continue to execute on?

what object the active method m() should continue to execute on afterwards. There are only two options – either it continues to execute on the old object that has been replaced or it chooses the new object. Both options have serious drawbacks. If m() continues to execute on the old object, it does not reflect the programmer's intention to have the object replaced. If it chooses the new object, this may lead to severe consistency problems, for example, if m() is implemented differently in the class of the new object. For example, the method might try to access private instance variables that are not present anymore. This example also illustrates why in the general case, such attempts cannot be detected statically since they might be issued in other objects and especially in totally different source code.

In order to ensure consistency, the default semantics of GILGUL are defined as follows. If the active method m() and the attempt at object replacement are executed by different threads the referent assignment operation blocks until m() (and all other methods that are active on the target object) complete. If they are executed by the same thread (that is, on the same execution stack) the referent assignment operation throws an (unchecked) ReferentAssignmentException. (See Section A.10.1 for more details on ReferentAssignmentException.)

5.1.1 Replacement of this

There are several situations when an object itself knows best when to be replaced. For example, the State pattern gives the advice "to let the State subclasses themselves specify their successor state and when to make the transition" [31]. However, a referent assignment this #= *expression* implies that the method that currently executes on this is active in the same thread by definition (see Figure 5.2).

GILGUL relaxes its restrictions for this case and does not throw a ReferentAssignmentException if it is ensured by a combination of static and dynamic checks (see below) that after the replacement of this, code will no longer be executed on this. For example, in the case of a void method this can be accomplished by just placing the replacement of this at the end of the method, as follows.

```
void close() {
  ...
  this #= new TCPClosed(); }
```

For non-void methods a new construct is introduced.

```
int close() {
  ...
  return expression with this #= new TCPClosed(); }
```

The semantics are as follows. First, the return expression is evaluated. Then the replacement of this is carried out. Lastly, the method completes and returns the result of the evaluation in the first step. Together these steps ensure that the replacement of this is indeed the last statement that gets executed on this. Possibly, there are still other methods active on the current object. In this case, the rules hold that are given in the previous section.

Another variation of this construct occurs in combination with the throw statement: throw *expression* with this #= *replacement*. The semantics are defined accordingly.

In previous versions of GILGUL, we have allowed replacements of this to be placed inside the finally block of a try statement. This would ensure roughly the same semantics (both for return and throw statements). However, the finally block is meant to be used for code that should always execute, even when an exception is thrown. It is more likely that replacements of this should not occur in this case so we have added the with construct to our design with these slightly different semantics.

The checks that ensure that a replace of this is indeed the last statement in a method are as follows. First, we define the term *definite last referent assignment*: a referent assignment is said to be definite last if it can be statically proven that no other code follows after that referent assignment within the same method. As in the case of definite initializing/non-initializing referent assignments in the previous chapter, we reuse the rules for definite assignment/unassignment in Chapter 16 of [34] accordingly.

The static check for definite last referent assignment does not take into account whether the left-hand side of a referent assignment is indeed the keyword this. This would be too restrictive in general, because the following code sequence is also a valid definite last referent assignment on this.

```
Object v = this;
v #= expression;
```

Instead, a GILGUL compiler has to mark all definite last referent assignments in the generated bytecode, independent from the expression on the left-hand side. This also complies with the fact that all assignments in the Java Virtual Machine and consequently the Gilgul Virtual Machine go via the stack: first, the expression on the left-hand side is evaluated, then the expression on the right-hand side, and finally an assignment operation is issued. Hence, a static check for definite last referent assignmen does not need to be more specific than described above. The Gilgul Virtual Machine simply has to disallow referent assignments on all left-hand variables that hold the same reference as this only for the unmarked cases.

5.1.2 Augmented return/throw versus try-finally

There are also some minor technical issues that result from interdependencies between try-catch-finally statements and the with construct for return and throw statements. In the case of a with construct inside a try block covered by a finally block, we have two code sections that are declared to be executed last.

```
try {
  ...
  return ... with ...; // or throw ... with ...;
} finally {
  ...
}
```

This results in a priority conflict that is hard to solve. Therefore, the language GILGUL disallows such nesting of with and finally code at compile time.

In the case of a with construct inside a try block covered by a catch block, the cases for return ... with ... and throw ... with ... differ slightly: In the case of return ... with ..., the referent assignment can only be executed when no exception is thrown beforehand. Therefore, none of the declared catch blocks can potentially be executed after the definite last referent assignment. On the other hand, in the case of throw ... with ..., the exception thrown by that throw statement might be caught by one of the enclosing catch blocks. Therefore, one of the catch blocks might be executed after the definite last referent assignment, rendering its definite last status questionable. There, this case must be forbidden. These considerations result in the following rules:

- A referent assignment in the with construct for a return or throw statement may not be covered by a finally block. A referent assignment in the with construct for a throw statement may also not be covered by a catch block. However, a referent assignment in the with construct for a return statement may be covered by one or more catch blocks.

5.1.3 Advanced Requirements

So far, GILGUL does not offer a completely satisfactory solution for replacements of active objects. For the default case, our main goal was to ensure consistency, and consequently this precludes solutions for the following issues.

- The rule that replacements block until active methods in other threads complete does not by itself handle the situation when the active method

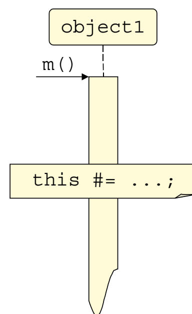


Figure 5.2: Replacement of this – method m() is active by definition.

consists of a non-terminating loop (for example, in the case of daemon threads).

- More often than not, administrators might be willing to trade timeliness for (temporary) inconsistency and so want to have an object replaced immediately rather than having to wait for the completion of other methods.

For example, assume that you have implemented a stock ticker roughly as follows.

```
public class StockTicker {
  ... // some member declarations

  public void run() {
    // mainly a non-terminating loop
    while (true) {
      input.getRequest();
      getQuote();
      output.putResponse();
    }
  }
}
```

Because of the non-terminating loop in method run it is not obvious how to replace an instance of this class without having to shut down the system.[1]

In GILGUL, the concept of *recalls* is introduced to provide support for these cases. In the following sections, we first outline the concept of recalls and afterwards show how they can be combined with referent assignments in order to provide a feasible means to deal with these cases.

[1] We are aware of only one approach that allows for dynamic software evolution and tries to deal with non-terminating loops without stopping them, but rather by letting the programmer define states of loops that allow for quasi-“morphing” to other loops [52].

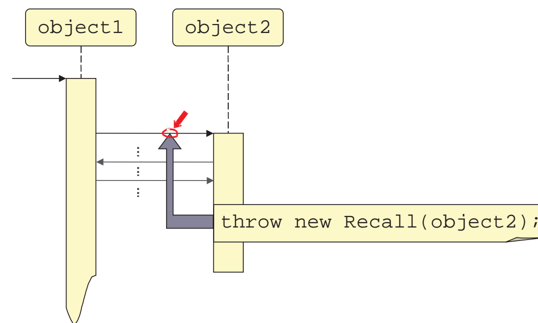


Figure 5.3: A *recall* on object2 unwinds the current stack up to the first method call on object2, and then calls this method again. Just before re-execution of this method call (at the marked spot), the call stack is guaranteed to be clear of object2 as a receiver.

5.1.4 Recalls

Exceptions in Java are a means to step out of the standard flow of control, and they can be caught by dedicated exception handlers, possibly at a higher level in the call stack, to revert to a corrected flow. In other words, the current call stack is *unwound* until a matching exception handler is found [34].

A *recall* in Gilgul mimics the behaviour of Java's exceptions, but instead of relying on the definition of dedicated exception handlers, a return to the standard flow takes place as soon as the call stack is clear of a particular receiver during the process of unwinding. For example, assume the throw of a recall on object2 in Figure 5.3. The current call stack is unwound up to the first method call on object2, and then this particular method is called again on object2.

Note that a recall guarantees that immediately before the point of return to the standard flow of control, the current call stack does not contain any method calls to the specified object. So by definition there is no method active on this object at this point in time (at the marked spot in Figure 5.3). If there is no method active on the target object at the moment of the throw of a respective recall, this recall is simply ignored and evaluates to a non-operation.

Just as in the case of exceptions in Java, Recall is a plain class in Gilgul and instances thereof can be thrown and caught. It is unchecked, like java.lang.RuntimeException, in order to have it smoothly integrated into existing Java code.

A *recall handler* can be used for setting corresponding objects to a consistent state. However, a caught recall should be rethrown in order to guarantee its completion, as follows.

```
try {
  ...                         // do your stuff here
} catch (Recall recall) {
  ...                         // reset your object(s)
  throw recall;               // and rethrow the recall
}
```

Since GILGUL's recalls are not exceptions in the strict sense to indicate that something has gone wrong, Recall is not defined as a subclass of java.lang.Exception, but as a subclass of the more general class java.lang. Throwable. This ensures that a general catch of all exceptions does not accidentally catch a recall as well, as follows.

```
try {
  ...
} catch (Exception e) {
  // recalls are not caught here
  ...
}
```

Further note that when parameters have been passed to the method that gets re-executed by a recall, the (possibly complex) parameter expressions are not reevaluated, but the previously evaluated values are simply reused: In Java bytecode, the call of a method consists of pushing parameters on the operand stack, and then, as a distinct step, execute an invocation of the respective method. It is only this last step that gets re-executed by a recall, and the invocation merely reuses the old state of the operand stack.

5.1.5 Combination with Referent Assignment

Given these prerequisites, GILGUL's referent assignment can be amended by recalls in order to replace even active objects. This is accomplished by annotating the application of the referent assignment operator accordingly, for example as follows.

```
stockTicker #= new StockTicker with global recall;
```

The options are as follows: a *local recall* throws the recall only for the current thread; a *global recall* throws it for all other threads that have a method executing on the target object, but not the current thread. Since the throw of a recall within a thread does not have any effect if the respective call stack is already clear of the specified object as a receiver, this definition is equivalent to the more general one that a global recall is thrown for all other threads.

These options can be combined as in *expression1 #= expression2* with local recall, global recall. Then the recall is thrown for both the current thread and other threads. The combination of a local and a global recall can be abbreviated, as in *expression1 #= expression2* with total recall.

The actual Recall instance that is thrown in this case takes the left-hand side of the referent assignment expression as a parameter. This means that the respective call stacks are unwound up to the first method call on the object referred to by the left-hand side. The actual replacement of this object is deferred to this point in time when the call stack is guaranteed to be clear of methods that are active on this object (as a receiver) and just before the re-execution of the respective method call (at the marked spot in Figure 5.3).

As a consequence, by means of the recall construct the replacement takes place at a point in time when it is safe to carry it out. Afterwards the standard flow of control is reentered and can for example return to a thus temporarily terminated loop. Since recalls may be caught during the unwinding of the call stack it is possible to reset the target object (and possibly dependent objects) to a reasonable, consistent state. However, it is not required to provide for such clearance code because recalls are unchecked. Therefore they can still be thrown even in unanticipated contexts. In the latter case, it is the task of the programmer who wants to replace a specific object to decide if clearance is needed or not and to take the necessary steps.

5.1.6 Relation to Java's Thread Model

There is a close relation of GILGUL's recalls to Java's *interrupts* that are used in order to signal that a specific thread should terminate. In the latter case, no automatic steps are taken by the run-time system to actively stop the thread, but instead all threads are required to regularly check their own interrupt flags and terminate eventually. In order to help programmers to remember to check for interrupts, some standard thread-oriented methods in Java, like wait(...) or Thread.sleep(...), may throw a (checked) InterruptedException when the corresponding flag is set. Note that this is the only default means in Java to support the termination of threads. Up to JDK 1.2, the class java.lang.Thread has included methods stop(...) and suspend(...) for explicit termination, but they have since been deprecated for safety reasons [78].

Essentially, global recalls in GILGUL mimic this behaviour. Instead of directly throwing recalls in other threads, a global recall merely sets a corresponding flag in each thread. This recall flag is checked in the well-known methods that already check for the interrupt status, namely wait(...), Thread.sleep (...) and so on, and they may throw recalls accordingly. Additionally, the recall flag is checked in the equally well-known Thread.yield(). Since platform independent components are expected to at least occasion-

ally call Thread.yield() for compatibility reasons, and generally make use of the other methods mentioned as well, the recall mechanism will most likely be already applicable in most cases without change of existing components.

In Java, the method Thread.yield() is used as a hint to the run-time system to indicate where switches between thread contexts may occur, and is essentially a Thread.sleep(0). The occasional call of Thread.yield() is optional in pre-emptive implementations of Java's thread model, but is required in cooperative ones. See [50] for more details on Thread.yield().

5.1.7 Replacement of Long-Running Methods

We now show how the combined use of the referent assignment operator and the recall construct can be used by sketching a class for consistently replaceable objects with long-running methods.

```
public class StockTicker {
  ... // some member declarations

  public void run() {
    ...

    try {

      while (true) {
        Thread.yield(); // implicitly check for global recalls

        input.getRequest();
        getQuote();
        output.putResponse();
      }

    } catch (Recall rec) {

      // ensure consistency
      if (input.requestsSoFar() < output.responsesSoFar()) {
        output.signalDropouts();
      }

      throw rec; // rethrow the recall
    }
  }

}
```

Now the replacement of a StockTicker instance with an appropriate wrapper (see Section 4.2) looks as follows.

```
stockTicker #= new StockTickerWrapper(stockTicker)
                                        with global recall;
```

Please note the following.

- The try-catch block inside the run method is only required if there is a need for clearance in the presence of recalls. Otherwise, Java's standard exception handling mechanism guarantees the propagation of recalls to higher levels.
- If the run method is the topmost method that has been called for the respective object on the call stack, the throw of a recall is guaranteed to re-execute this method (after object replacement, if thrown in conjunction with a referent assignment). There is no need for the programmer to take explicit steps in order to achieve this behaviour.
- The only requirement that is currently placed on the programmer is to occasionally insert checks for recalls, for example by calling Thread.yield(). A programmer may also choose to explicitly check for global recalls by inspection of the respective recall flag. However, we expect programmers to want to do this seldomly. See Chapter 6 for more details.
- So in general, the referent assignment operator is applicable even in the case of long-running methods.

Chapter 6

Implementation

In this chapter, the implementation of the Gilgul compiler and the Gilgul Virtual Machine is described. The Gilgul compiler is an extension of the original Java compiler, as provided by Sun Microsystems, Inc. The Gilgul Virtual Machine (GVM) is a modification of the Kaffe virtual machine [41], released under the GNU Public License.

The Java execution model differs from typical implementations of other languages that compile source code directly into machine code and link the code into an executable program file. Instead, a Java compiler produces a platform-independent *bytecode* that is stored in *class files*. An implementation of the *Java Virtual Machine* (as specified in [54]) – for example the Kaffe virtual machine – is able to to load such class files, link them on the fly, and eventually execute them. Execution is either accomplished via interpreting the bytecode, or via compiling it into machine code (*just-in-time compilation*), or via a combination of interpretation and compilation (*dynamic compilation*). Since Java is designed for loading and executing programs from potentially untrusted origins, its execution scheme also involves a *bytecode verifier* that rejects malicious code. Java's execution model is illustrated in Figure 6.1.

The Gilgul compiler uses well-known implementation techniques and compiles the specification given in Appendix A into an extension of the Java class file format, specifically customized for Gilgul. The bulk of the implementation is mainly driven by the specification of the extensions of the Java Virtual Machine instruction set and their implementation in the Gilgul Virtual Machine.

We have put our main focus on providing a stable proof of concept in a manageable amount of time and not on industrial-strength performance, given the manpower of just one programmer for the compiler and one for the run-time system. Therefore we have chosen to only modify the pure interpreter part of the Kaffe virtual machine and have not addressed issues of just-in-time or dynamic compilation.

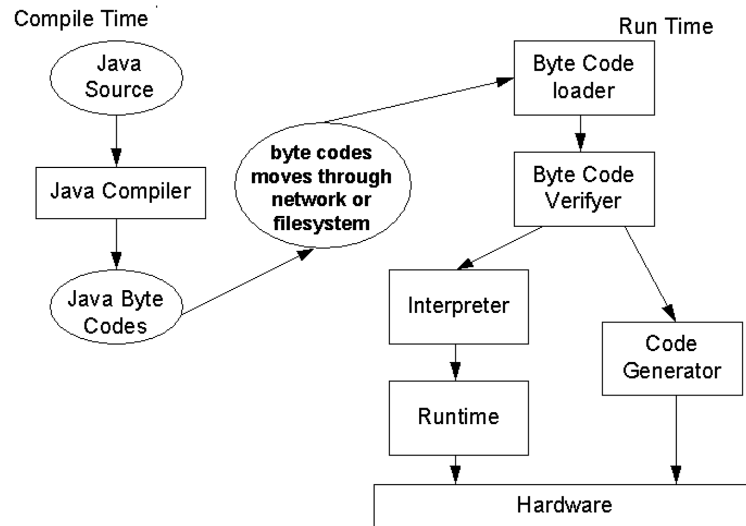


Figure 6.1: The Java execution model

Currently, a beta-quality implementation of the complete GILGUL programming language, as described in this chapter, can be freely downloaded as a compiler and run-time system from the GILGUL homepage [32].

The complete details of the implementation of the Gilgul Virtual Machine are given in [58]. In the remaining sections of this chapter, we give an overview of that work.

6.1 Architecture of the Java Virtual Machine

The Java Virtual Machine combines the use of a stack for passing parameters to methods and managing local variables, and a heap for allocating objects and arrays. Memory management is automatic: Stack-allocated memory is freed upon return from a method and heap-allocated memory is regularly garbage collected (without fixing a specific garbage collection algorithm). The stack and heap are operated on primarily via bytecode instructions.

Like the Java programming language, the Java Virtual Machine distinguishes between primitive data types (like short, int, long, char, and so on) and reference types for objects and arrays. All bytecode instructions exclusively operate on well-specified types. For example, integer additions are not allowed to operate on reference types. Some of these restrictions are (specified to be) statically checked by the bytecode verifier, at load time, while others can only be enforced at run time.

The *class file format* specifies an intermediate representation of Java classes that is generated by the Java compiler. It consists of a *constant pool* that stores constant values, strings, field and method signatures, and so on,

that are used in a class; references to related classes (the superclass, implemented interfaces, inner classes, and so on); field and method definitions; and the *bytecode* for non-abstract methods. Most information, except for references to local variables, is encoded symbolically and resolved at load time the earliest.

The instruction set of the Java Virtual Machine consists of load and store operations for local variables and operands; arithmetic operations; type conversions between primitive data types; operations for allocating and initializing objects and arrays; for accessing object, class and array members; for determining the length of an array; for determining the instanceof relation; for jumping within a method; for invoking and returning values from methods; and for synchronizing threads that execute in parallel.

6.2 Architecture of the Gilgul Virtual Machine

6.2.1 The primitive data type comparand

GILGUL requires the type comparandtype. As noted in Section 4.1.2, it is suggestive to implement it as a 64-bit unsigned integer. However, GILGUL also requires that comparands and numerical values are strictly separated in order to ensure comparand uniqueness. Comparands should only be generated by the comparand creation operation, and not be modified via arithmetic operations. Therefore, the Gilgul Virtual Machine also introduces the type comparandtype at the virtual machine level.

However, comparands can be converted to values of type long, but not vice versa. A comparand always maps to the same long value, and no two comparands can be mapped to the same long value. The conversion of comparand to long values can be used in distributed programming scenarios – see Appendix B.

In the Gilgul Virtual Machine, a comparison of two objects is always performed as the comparison of their respective comparands. However, null references are treated specially: a comparison of a null reference with another reference yields only true when that other reference is also a null reference. This check is performed before the references are attempted to be dereferenced. This effectively means that obj1 == obj2 is not strictly equivalent to obj1.comparand == obj2.comparand when null refereces are involved – if both obj1 and obj2 are null, then the latter throws a NullPointerException while the former yields true.

6.2.2 Control Facilities

GILGUL specifies several control facilities for restricting the use of referent and comparand assignments. These restrictions are declared alongside constructors which are compiled into methods with the name `<init>` that

are treated specially by the Java Virtual Machine. Because of several restrictions that the Java Virtual Machine specification imposes on constructors, we have opted for the introduction of another kind of specially treated `<gilgulinit>` methods that are used instead of `<init>` methods if available, and in turn call `<init>` to behave well with pure Java classes. The restrictions declared on referents and comparands in a GILGUL class are associated with such `<gilgulinit>` methods and handled accordingly by the Gilgul Virtual Machine.

6.2.3 Protected Activities

Referent assignments can only be carried out on references while they are inactive. The diploma thesis [58] presents the following definitions and rules in order to achieve a sound implementation of this requirement.

Definition 6.1 (Compound Activity) *A compound activity on an object reference is a succession of machine operations of the target platform that (potentially) need to refer to that fixed object more than once via that object reference.*

During execution of a method, there can occur arbitrarily many accesses to this that must refer to the same object. A synchronized block is implemented by a pair of monitorenter and monitorexit bytecode instructions, as specified by the Java Virtual Machine, that also must refer to the same object via the same reference. Therefore, method invocations and synchronized blocks are compound activities at the bytecode level. (See below for examples of compound activities below the bytecode level.)

Definition 6.2 (Protected Activity) *A protected activity is a compound activity on a reference during which no other thread is allowed to intervene by performing a referent assignment on that reference.*

By definition, *interventions* cannot occur for atomic operations. Therefore, only compound activities can be referred to as protected activities. According to these definitions, method invocations and synchronized blocks are protected activities. Additionally, the following bytecode instructions qualify as compound activities that must be protected.

checkcast and instanceof Instructions that check for the instanceof relationship between an object reference and a class must first check for null before dereferencing that reference. So these instructions are compound activities that must be protected because they use a given object reference twice.

aastore Each assignment into an array must be preceded by an implicit checkcast at run time because of a subtle hole in Java's static type system. Therefore, it also qualifies as a compound activity.

if_acmpeq **and** if_acmpne Instructions that check for object identity are redefined in GILGUL to operate on the comparands of the involved objects. Since accesses to object comparands must be preceded by null checks, they qualify as compound activities.

invokecast**,** invokeinterfacecast**,** getfieldcast**,** putfieldcast Each method invocation and field access for instances of typeless classes must be preceded by typechecks and subsequent casts to such classes. This is accomplished in the Gilgul Virtual Machine by combining typecheck, cast and access into the listed compound operations. These operations qualify as compound activities.

assignreferent The referent assignment operation is accomplished by a dedicated bytecode instruction that combines several type integrity checks, making it a compound activity.

Field and comparand accesses as such may also qualify as compound activities depending on implementation details of a concrete Gilgul Virtual Machine. Such accesses must at least implicitly check for null in order to correctly issue throws of NullPointerException. However, the memory management facilities of the operating system and the underlying hardware may allow an implementation to avoid explicit null checks in these cases.

Rule 6.1 (Integrity Rule) *A Thread* T *may only perform a referent assignment on a reference while no other thread performs a protected activity on that reference. When necessary, thread* T *must wait until this condition holds. If thread* T *itself currently performs a protected activity on that reference, the Gilgul Virtual Machine must throw a* SelfBlockingThreadException*. An exception for this rule exists for* definite last assignments*.*

Rule 6.2 (Stability Rule) *As long as a thread performs a referent assignment on a reference, no other thread is allowed to start a protected activity on that reference. When necessary, such a thread must wait until this condition holds. This especially means that a Gilgul Virtual Machine is not allowed to performe more than one referent assignment on the same reference at the same time.*

Rule 6.3 (Compatibility Rule) *There are no restrictions on the parallelizability of compound activities on a reference as long as no referent assignment or synchronization on that reference is involved.*

Taken together, these rules ensure that referent assignments behave well. Examples are given in [58].

There exists an exception to the integrity rule: A referent assignment on a reference that is the target of the current method invocation (the reference

stored in this) is accepted by GILGUL as long as it is a definite last assignment. This is mapped to the Gilgul Virtual Machine level by the following conditions under which a SelfBlockingThreadException is not thrown.

- There must be no other protected activity on this in the current thread.
- The corresponding assignreferent instruction must be marked as a *definite last assignment* by the compiler (see Section 6.3.2).
- None of the following instructions may potentially follow in the control flow of the current method after that assignreferent instruction: getfield, putfield, invokevirtual, invokeinterface, invokespecial, invokestatic, checkcast, instanceof, if_acmpeq, if_acmpne, monitorenter, getcomparand, putcomparand, getfieldcast, putfieldcast, invokecast, invokeinterfacecast.

  These are all bytecode instructions that potentially access objects either directly or indirectly and therefore would violate the definite last assignment status of the assignreferent instruction. An exception is made in the case of monitorexit that also needs to access an object in order to release a lock. However, a monitorexit instruction needs to be matched by a preceding monitorenter instruction on the same reference. If this happens to be this, the referent assignment would not be valid anyway because of the integrity rule given above. On the other hand, this exception allows for the added flexibility of being able to place a definite last assignments in a synchronized block.

  This last condition must be checked by the bytecode verifier.

6.2.4 Recalls

The basic implementation scheme for recalls is already sketched in Section 5.1.6. The following additional details are noteworthy.

An instance of java.lang.Recall can only be thrown as a result of the checks performed by the assignreferent instruction of the Gilgul Virtual Machine, or as an explicit throw issued by the programmer. When a recall is thrown, the only protected activities that may currently still be executed on the respective reference, except for the issuing assignreferent, are method calls and synchronized blocks. All other protected activities are ensured to be atomically executed by the Gilgul Virtual Machine in this regard.

This means that the start of synchronized blocks (i.e., the monitorenter instruction) and all instructions for method invocations are candidates for being the top-most implicit recall handlers. However, we have slightly deviated from this apparently straightforward choice in the following way: When the start of a synchronized block is the top-most protected activity for a recall reference, the immediately surrounding method call is chosen as the entry point for the implicit recall handler. This means that there are no provisions

in the Gilgul Virtual Machine for directly re-executing synchronized blocks. The reasons for this design choice are as follows:

- It is easier for programmers to understand the consequences of a recall, especially with regard to explicit recall handlers.
- The proposed scheme is easier to implement efficiently. The recall of a method requires the run-time system to only restore the actual parameters to the method, whereas the recall of a synchronized block would require restoring all accessible local variables that might have been side-effected by that block in the previous run.

As noted in Section 5.1.6, recalls that several threads issue for each other need to be checked in regular intervals. The low-level method for this purpose is called checkRecall() and is defined in class java.lang.Thread. Since the flags that this method checks are by nature accessed in a multi-threaded fashion, special care must be taken to synchronize those accesses appropriately in a Gilgul Virtual Machine. Again, details are given in [58].

6.2.5 System Classes

The restrictions for several system classes are defined as follows. (See Section 4.4 "Control Facilities".)

- References to (built-in) arrays are bound, as well as their comparands.
- References to instances of the following classes are bound, as well as their comparands: java.lang.String, java.lang.Class, java.lang. SecurityManager, java.lang.reflect.Method, java.lang.reflect.Constructor, java.lang. reflect.Field.
- References to instances of java.lang.Thread *and all of its subclasses* are bound. There are no restrictions on their comparands, except for possibly programmer-defined ones.

Classes like java.lang.SecurityManager and java.lang.Method are restricted because of obvious security concerns. Other classes are restricted because of implementation-dependent details.

6.3 New Virtual Machine Instructions

The Gilgul Virtual Machine introduces 18 new instructions at the bytecode level that reflect the new operations introduced by GILGUL.

6.3.1 Operations on Comparands

A major portion is dedicated to operations involving the new "primitive" data type comparand, and they are designed in a similar fashion like those for other primitive data types in the Java Virtual Machine. They comprise kload and kstore for copying comparands between operand stack and local variables; kaload and kastore for copying comparands out of and into arrays; if_kcmpeq and if_kcmpne for comparing comparands; kreturn for returning comparands from methods; k2l for converting comparands to long values; getcomparand and putcomparand for accessing an object's comparand; and newcomparand for generating fresh comparands.

Note that the bound restriction of an object's comparand is not yet checked when it is read via getcomparand in order not to prevent client code to use its value as a basis for determining hash codes, for example. Instead, the comparand values themselves carry flags that indicate whether they are acceptable as comparands for other objects or not.

6.3.2 Operations on References

There exist three operations for dealing with newly introduced operations in Gilgul:

assignreferent This operation performs referent assignments. Apart from the two operands, this operation can be further refined with three flags.

> WITH_LOCAL_RECALL, WITH_GLOBAL_RECALL These flags reflect the with local, with global and/or with total recall annotations given by the programmer.
>
> WITH_RELEASE_THIS This flag indicates whether this assignreferent is a candidate for a definite last assignment. This flag is implicitly set by the Gilgul compiler as the result of a simple control flow analysis.[1]

newreference_unbound, newreference_fixed These instructions create a fresh reference for an object passed as an operand. They reflect the Gilgul construct for initializing referent assignments (see Section 4.3). The newreference_fixed is used when the annotation with fixed referent is given by the programmer. The newreference_bound is used otherwise, and must be provided with the static type of the initializing referent assignment (i.e., the left-hand side of the assignment). That type is used in subsequent run-time checks involving null references (see Section 4.5.7). Because such type information is not required in the with fixed referent case, initializing referent assignments are divided into two operations.

[1] It has to be re-checked by the Gilgul Virtual Machine for security reasons.

6.3.3 Operations on typeless Classes

Accesses to fields and invocations of methods of typeless classes require checking that an object is indeed an instance of the expected class beforehand. However, other threads may potentially intervene the check and the actual access / invocation. In order to be able to treat a check and an access / invocation as a protected activity, the Gilgul Virtual Machine provides the following four instructions that simply execute a checkcast and the respective "unchecked" operation, as already provided by the Java Virtual Machine: invokecast, invokeinterfacecast, getfieldcast and putfieldcast.

6.4 Representation of Objects

The presentation of the GILGUL model in Chapter 3, and especially its comparison with the Smalltalk object model in Section 3.4, suggests an implementation of object references in the Gilgul Virtual Machine as pointers into a central object table that in turn contains pointers ("referents") to the actual objects. While this is certainly a feasible implementation technique, as has been demonstrated by early Smalltalk and also Java implementations, it is generally known as a very inefficient approach and therefore has been replaced by implementations with direct pointers both for Smalltalk and Java.

Since the referent and comparand assignment operators are the central contribution of the work presented in this thesis, we cannot afford to impurify the semantics of those operations in order just to be able to switch to direct pointers. Instead, we have to find a more reasonable resolution between double indirection and direct pointers that does not restrict the applicability of referent assignment. The implementation of the Gilgul Virtual Machine, as presented in [58], presents a feasible approach.

6.4.1 Minimization of Indirection Penalties

It is clear that additional indirections cannot be avoided in the Gilgul Virtual Machine. However by default, when objects are newly created, there exists an unambiguous mapping between references and referents that is only disassociated by application of the referent assignment operation. Therefore, it can be expected that there is always a reasonably sized subset of mappings between references and referents, whose size depends on the frequency of the referent assignment operation. In the general case, it can vary over time whether a specific object belongs to that subset or not, except for objects with bound referents which always belong to that subset. (It is not sufficient that a reference's referent is fixed because there might still exist more than one reference with that referent.)

These observations have led to the following two essential optimizations in the Gilgul Virtual Machine.

- Objects that are referenced with bound referents can always be accessed via direct pointers. This already affects a major portion of regularly used Java classes, for example the classes for immutable data types provided by the Java core API, like java.lang.String, java.lang.Integer, and so on. This effectively means that a large number of object accesses can be implemented as efficiently as in pure Java.
- For indirectly addressed objects, an implementation technique is reused that is also employed in some Smalltalk implementations. Instead of relying on a central object table, so-called "hidden pointers" are used that are placed immediately before the objects to which they refer [74]. Such a hidden pointer refers to the following object as long as no operation is performed that redirects it to a different object. With regard to modern random access memory architectures, such an approach is especially efficient: Of the two accesses that are needed with any access that uses double indirection, the first access results in caching the surrounding memory area, and it can be expected that the second access can be very efficiently resolved in cache.

These two steps result in several additional implementation details, like adding flags to object headers that indicate direct or indirect addressing, and so forth. These details are described in [58]. An overview of the quantitative evaluation (performance benchmarks) performed in that thesis is given here in Section 7.2.

6.5 Activity Control and Multithreading

In the Java Virtual Machine, monitors are used to synchronize concurrent accesses to shared resources. Since these monitors are associated with objects, but the Gilgul Virtual Machine needs monitors for references in order to synchronize concurrent referent assignments on shared references, it introduces such *reference monitors*. Reference monitors are used to coordinate referent assignments and protected activities so that they do not overlap with each other. Since it must be allowed to have overlaps between protected activities as such, in the absence of referent assignments, reference monitors must not be held during whole activities. Instead, activities are only *registered* by way of activity counters embedded in reference monitors.

In [58], a *Simple Registration Protocol* is presented that fulfils the requirements for synchronization of referent assignments, but is very inefficient in terms of book keeping and frequent use of heavy-weight synchronization operations at the machine level. Furthermore, these penalties are incurred whether referent assignments are involved or not.

Therefore, an additional *Adaptive Registration Protocol* is described that reduces the overhead to an essential minimum. That protocol distinguishes between a light-weight and a heavy-weight registration phase with regard to a reference. During the light-weight registration phase, the costly access to registration monitors is completely avoided, but instead only minimal, purely local auxiliary information is recorded. The transition to the heavy-weight registration phase is only needed when a referent assignment is actually performed, and then also only temporarily. Just before that transition, a global analysis of the call stacks of all threads is performed in order to determine the actual activity status of a reference after the fact.

Furthermore, the Adaptive Registration Protocol is also a suitable basis for an implementation of recalls in the Gilgul Virtual Machine. This involves determining the top-most activity on a recall reference, and the necessary communication between threads to determine an appropriate point in time for re-executing method invocations.

Chapter 7

Evaluation

This chapter presents an evaluation of the GILGUL model and language on two levels. On the one hand, we present a qualitative evaluation by presenting three different examples that show that the concepts introduced in GILGUL can indeed be used in a wide range of usage scenarios. The first example (Section 7.1.1) presents how a live update of a server that provides data from a stock ticker can be accomplished. The second example (Section 7.1.2) illustrates how role models can benefit from referent assignments. Finally, the third example (Section 7.1.3) presents constant folding by way of referent assignment as an optimization for the Interpreter pattern.

All these examples focus on uses of referent assignment. Usage scenarios for comparands are presented in detail in the Comparand pattern in Appendix B, and are not repeated here.

On the other hand, a quantitative analysis has been performed as part of [58], and we give an overview of that work in Section 7.2. Its purpose is to quantify the performance overhead that the implementation of comparands and referents described in the previous chapter incur.

7.1 Qualitative Evaluation

7.1.1 The Stock Ticker Example

We present a client-server scenario in which a client repeatedly inquires the value of a stock ticker from a server. This scenario is depicted in Figure 7.1: StockClient consists of one simple thread that repeatedly reads a value from a socket stream and displays that value for the user. The server consists of three loosely coupled loops:

- StockServer listens to incoming connection requests from clients and produces threads to handle such requests. The handlers communicate with StockTicker to get the current stock value to be delivered to a client.

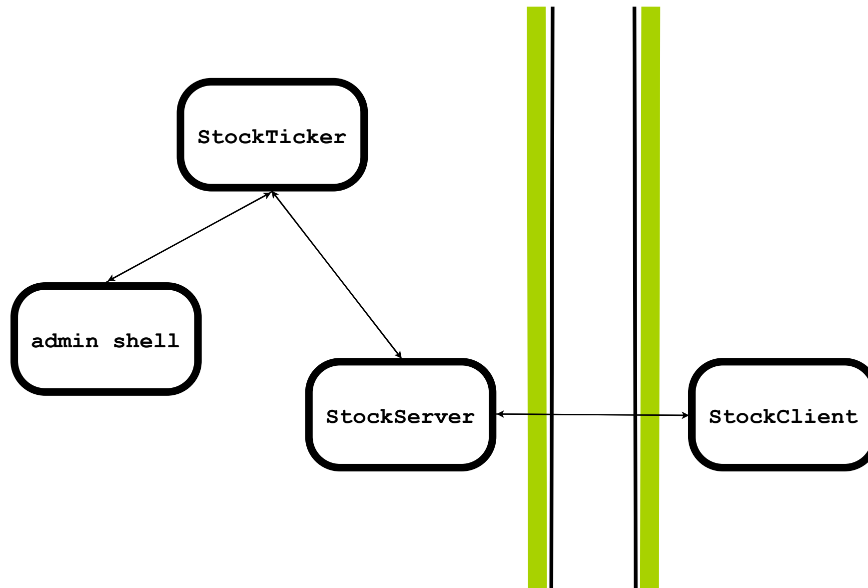


Figure 7.1: The basic architecture of the stock ticker example.

- StockTicker consists of a loop that repeatedly refreshes its value.
- The server also provides an *admin shell* that allows interacting with the run-time environment that executes the whole server.

In our example, we demonstrate how a switch from the former German DEM currency to the European EURO currency can be accomplished, which was an unanticipated change for most European countries a few year ago. In order to understand the issues that such a switch involves, we sketch the source code for the StockTicker class.

```
class StockTicker implements Runnable {

  // (1) The StockTicker is a singleton.
  static private StockTicker instance = new StockTicker();
  static public StockTicker getInstance() {return instance;}

  // (2) The stock value.
  // In our example, the currency is fixed.
  protected double currentStockValue;
  protected final String stockCurrency = "DEM";
```

```
  // (3) In a multi-threaded environment, accesses to
  // the stock value must be synchronized.

  public synchronized double getCurrentStockValue() {
    return currentStockValue;}

  protected synchronized void setCurrentStockValue(double d) {
    currentStockValue = d;}

  public synchronized String getStockCurrency() {
    return stockCurrency;}

  // (4) This produces a representation for the client.
  protected String printStockValue() {
    return ((int)getCurrentStockValue()) + " "
           + getStockCurrency();}

  // (5) For our example, the value change is randomized.
  protected synchronized void refreshCurrentStockValue() {
    setCurrentStockValue(getCurrentStockValue()
                         +java.util.Math.random()*60);}

  // (6) The main endless loop.
  public void run() {
    try {
      while (true) {
        Thread.sleep(2000);
        refreshCurrentStockValue();
      }
    } catch (InterruptedException e) {}
  }
}
```

Here are more details to the comments given in the source code above.

1. The StockTicker is a singleton class (see [31]). (We do not lazily initialize the singleton reference here because the instance must be initialized very early on anyway.)
2. The currency is fixed, so the change to a different currency cannot be accomplished by a mere assignment to the currency field.
3. Since the fields are accessed purely locally (in the methods printStockValue and refreshCurrentStockValue), it would strictly not be necessary to provide "getter" methods. However, we need to synchronize those accesses, so therefore we need the "getter" methods.

4. The printed representation for the client includes the currency.
5. The randomization of changes in the stock value is only included to have an interesting effect for demonstration purposes.
6. The main loop of the stock ticker presents the main challenge in this example: We need to break out of this loop in order to replace the stock ticker with a new one that reflects the change to the new EURO currency.

Of course, the change to the new currency can be performed by way of a referent assignment in conjunction with a recall (see Section 5.1.4). Here is the code of the EuroStockTicker.

```
class EuroStockTicker extends StockTicker {

  // (1) We delegate calls to the original stock ticker.
  private StockTicker parent;

  // (2) The new currency.
  public String getStockCurrency() {return "Euro";}

  // (3) The stock value converted to / from EURO.
  public synchronized double getCurrentStockValue() {
    return parent.getCurrentStockValue()/1.95583;}

  protected synchronized void setCurrentStockValue(double d) {
    parent.setCurrentStockValue(d*1.95583);}

  protected synchronized void refreshCurrentStockValue() {
    setCurrentStockValue(getCurrentStockValue()
                         -java.util.Math.random()*80);}

  // (4) The constructor prepares for the
  // replacement of the original stock ticker.
  protected EuroStockTicker(StockTicker st) {
    StockTicker old #= st;       // create second reference...
    this.parent = old;           // ...and store it as parent
  }

  // (5) The static initializer performs the replacement.
  static {
    StockTicker st = StockTicker.getInstance();
    st #= new EuroStockTicker(st) with global recall;
  }
}
```

Comments:

1. We use a simple simulation of delegation in order to redefine some methods of the original stock ticker class and leave others unchanged. Here, the use of a language extension for proper delegation would be appropriate (for example, see [48]).

2. It is sufficient to simply replace the "getter" method for the currency field in order to introduce the new EURO currency.

3. The "getter" and "setter" methods for the stock value are redefined with proper conversion methods that otherwise delegate to the original accessors.

4. The constructor follows the idiom presented in Section 4.2.

5. The actual replacement of the stock ticker takes place in the static initializer of the EuroStockTicker class. The static initializer is executed when the class is loaded into a running Java / Gilgul Virtual Machine. Therefore, it suffices when the admin shell for the server allows loading of the EuroStockTicker class. The replacement is annotated with a global recall in order to ensure that the endless loop of the StockTicker is exited before the replacement and re-executed after the replacement (see Section 5.1.4).

7.1.2 Using Referent Assignments to Introduce New Roles

In this example, we illustrate how referent assignments allow for the global introduction of new roles, given a suitable role model. A role model extends basic object-oriented language constructs with a notion of additional views to an object that show a different behavior than the core object itself. Still, the core object together with its views present themselves as one single conceptual identity. So for example, a Person object may act as an Employee in some contexts and as a Parent to a child in others, and so forth.

The notion of roles is discussed for example in [46], and [48] shows how a role model can be implemented with a proper language extension for delegation. A similar notion is the concept of *perspectives*, as for example presented in [69]. An important difference between roles and perspectives on the conceptual level is that roles can be acquired and dropped during the lifetime of an object (a Person may get unemployed, and again employed later on), while a perspective is attached to an object for its whole lifetime. For example, [69] gives examples from the field of architecture in which building materials present themselves differently in a simulation depending on what properties an architect is currently interested in. Nevertheless, the building materials have the different modeled physical properties during their whole lifetime.

Another interesting aspect of the work on perspectives presented in [69] is the fact that the presented language extension allows for the dynamic introduction of new perspectives into a running program while maintaining the “illusion” that those perspectives have existed since the creation of the respective objects.

In the following, we present step by step how some of the notions induced by role and perspective models can be simulated with the help of referent assignment.

The basic idea is to start from a role model as presented in [46, 48], with a simplified approach for implementing delegation as in the previous section, for presentation purposes. A class Person adheres to an interface IPerson as follows:

```
public interface IPerson {
  String getName();
  . . .
}

public class Person implements IPerson {

  String name;

  public Person(String name, . . .) {
    this.name = name;
    . . .
  }

  public String getName() {
    return name;}

  . . .
}
```

The interface IPerson is used to factor out the essential methods of a Person instance so that roles of a Person can have the proper type:

```
public class Employee implements IPerson {

  IPerson delegatee;
  String employer;

  public Employee(IPerson person, String employer) {
    this.delegatee = person;
    this.employer = employer;}
```

```
  public String getEmployer() {
    return employer;}

  // delegated methods
  public String getName() {
    return delegatee.getName();}

  . . .
}
```

Now, there are essentially two different ways to make an existing Person instance acquire a new role. A *local activation* simply acquires the role purely for the local context:

```
Employee emp = new Employee(somePerson);
```

Other contexts do not see this newly introduced role. In order to achieve a *global activation* of a new role, one can use a referent assignment:

```
somePerson #= new Employee(somePerson);
```

However, this approach might not lead to desirable results because other contexts may also try to introduce a new role by way of referent assignment, so this might lead to a mismatch between globally activated roles. In order to avoid such mismatches, there is another option to introduce a role globally without necessarily affecting other already activated roles: One can first obtain a reference to the core object that is wrapped by various role objects and replace that core object. In order to be able to do this, we introduce a method getCoreObject into the IPerson interface and implement it in the Person class to return this, and in all role classes to delegate it to the delegatee. Now, the global activation of a role looks as follows:

```
somePerson.getCoreObject() #=
  new Employee(somePerson.getCoreObject());
```

The two basic activation schemes (local and global) are depicted in Figure 7.2: A local activation places a new role in front of an existing conceptual entity (core object + roles) while a global activation places a new role in front of the core object and after the already existing roles.

A final issue is the question how to find already existing roles in the path from an "outer" role to a core object. This can be important when one needs to ensure that a role only exists once but is only created when needed. This is, for example, the case when we want to simulate a perspective concept as presented in [69] and allow for the dynamic introduction of new role / perspective classes into a running system. We introduce a RoleManager class for this purpose that maps role classes to role instances for a given core object.

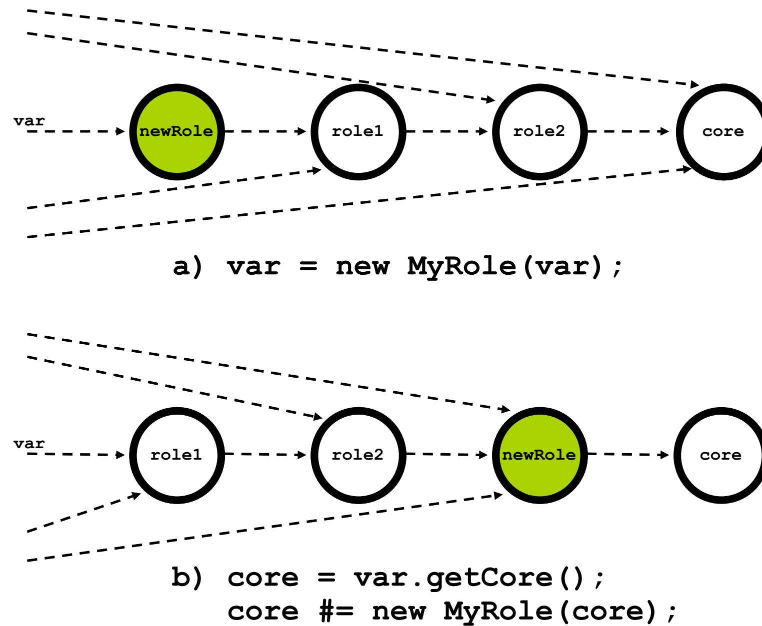


Figure 7.2: The two basic idioms for introducing roles.

```
public interface RoleCreator {
  public Object createRole(Object delegatee);
}

public class RoleManager {

  java.util.HashMap roles = new java.util.HashMap();

  public Object getRole(Class clazz, RoleCreator creator) {
    if (roles.containsKey(clazz))
      return roles.get(clazz);
    else {
      Object role = creator.createRole(this);
      roles.put(clazz,role);
      return role;
    }
  }
}
```

The RoleCreator interface is provided to allow passing a closure to the getRole method in the RoleManager class for lazily creating a new role instance in case it does not already exist in the hash map. The RoleManager

is intended to be used as a superclass for core object classes, for example Person.[1]

Now, the basic activation scheme looks as follows.

```
Employee emp = (Employee)
  somePerson.getRole
    (Employee.class,
     new RoleCreator() {
       public Object createRole(Object delegatee) {
         return new Employee((IPerson)delegatee);}
     });
```

Again, this can be combined with referent assignments in order to distinguish between local and global activations.

7.1.3 Constant Folding in the Interpreter Pattern

The Interpreter pattern is one of the classic patterns described in [31]. Its intent is to "define a representation for [a language's] grammar along with an interpreter that uses the representation to interpret sentences in the language". This can mainly be used to embed a "scripting" language into an application for tasks that need to be flexibly specified at run time. The pattern description in [31] gives a language for regular expressions as an example.

An aspect that is not covered in that book is how to optimize the execution of interpreted languages. A straightforward optimization is a technique known as constant folding: Assume a language for arithmetic expressions that involves both variables and constant numbers, as illustrated in Figure 7.3. Given an expression new AddExpr(new NumberExpr(6), new NumberExpr(7)), a naive implementation of the interpret method will perform the addition in each run, although the result will never change from being 13. The referent assignment operation allows us to perform constant folding during the interpretation of the arithmetic expression, for example in the class AddExpr, as follows.

```
public Expression interpret(. . .) {
  Expression leftResult = left.interpret(. . .);
  Expression rightResult = right.interpret(. . .);

  Expression result = new NumberExpr
    ( ((NumberExpr)leftResult).value +
      ((NumberExpr)rightResult).value );
```

[1] In cases where the single superclass of a core object class is already used for a different purpose, for example aspect-oriented language extensions like AspectJ [45] can be used to mix in the desired role manager behavior.

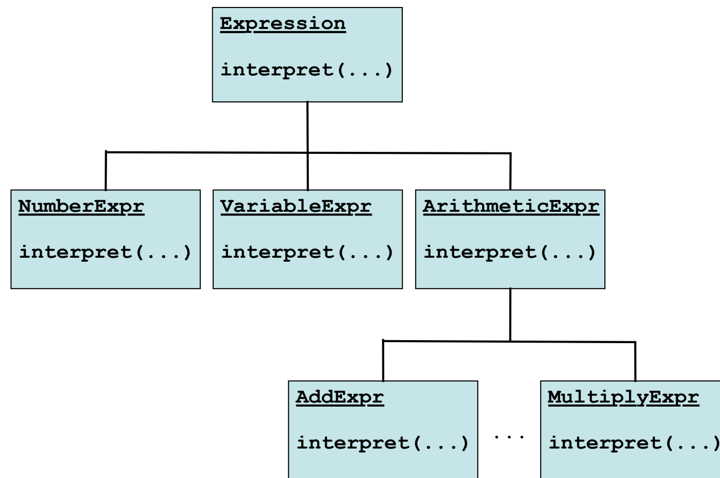


Figure 7.3: The Interpreter pattern.

```
  if ((left instanceof NumberExpr) &&
      (right instanceof NumberExpr))
    return result with this #= result;
  else
    return result;
}
```

The referent assignment associated with the return statement ensures that the current arithmetic expression is replaced with a constant number expression in case both operands are in turn constant numbers. The instanceof checks of the left and right references are performed after interpret has been executed on them so that their potential (recursive) replacements are correctly taken into acount in the current interpret method. It is important to note that all subclasses of Expression need to be typeless so that instances of ArithmeticExpr can indeed be replaced by instances of NumberExpr (see Section 4.5.2).

7.2 Quantitative Evaluation

As described in the previous chapter of this thesis, a prototypical implementation of the Gilgul Virtual Machine has been implemented as an extension of the bytecode interpreter that is part of the Kaffe virtual machine [41]. The diploma thesis [58] describes that implementation and a quantitative

analysis of its performance characteristics. This section presents an overview of that analysis.

Two different kinds of benchmarks have been performed. On the one hand, *microbenchmarks* have been written that allow for isolated measurements of single instructions, like method calls, field accesses and comparison operations. On the other hand, *macrobenchmarks* are “natural” real-world programs, or simulations thereof, that allow for analysis of whole program performance characteristics that put overheads incurred by single instructions into perspective.

As a basis for the microbenchmarks, parts of the Java Grande Forum’s benchmark suite are used [37]. This suite has been extended to cover additional operations that are of special interest in our context: reference comparisons, instanceof checks, comparand creation and referent assignment. In order to evaluate concurrent referent assignments and other protected activities, a specialized *GilgulBench* benchmark has been written because the issues involved are by definition out of scope of standard Java benchmarks.

The macrobenchmarks consist of widely used Java benchmarks that correspond to real-world Java programs with a fixed set of input data [3, 27, 83]. Apart from VolanoMark they consist of javasrc, a program for generating HTML documentation from Java source code; rhino, an interpreter for the scripting language JavaScript; and gj, a compiler for the Java programming language.

All tests have been performed on a PC with an AMD Athlon processor (800 MHz) and 384 MByte RAM, an top of Linux Kernel 2.4. The heap settings for the virtual machine have been modified by standard environment variables in order to ensure that runs of the garbage collector do not distort the results. The tests have been performed several times, and the results have been averaged over these runs.

7.2.1 Memory Overhead

The design of the Gilgul Virtual Machine incurs the following memory overhead for each object:

- Each object stores a comparand that spans 64 bits.
- If a reference to an object is represented explicitly, another 64 bits are needed for the additional reference. (References with bound referents do not need such additional references – see Section 6.4.1.)

The actual memory overhead in the benchmark programs ranges between 0% (for a heapsort benchmark that purely uses large arrays) and over 20%, depending on the types of objects that are actually used.

7.2.2 Performance Overhead

In all macrobenchmarks, the Gilgul Virtual Machine has performed less than 5% slower than the unmodified Kaffe virtual machine, given that the adaptive registration protocol is used (see Section 6.5). In the case of rhino and gj, the Gilgul Virtual Machine even performs slightly better than the unmodified Kaffe virtual machine. We suspect that this is due to optimizations that the C compiler can perform based on the changed memory layout of objects because of the additional comparands. (Optimizations on the micro level are generally very sensitive to such minimal changes. This is an indication that the Kaffe virtual machine itself is not yet fine tuned with regard to performance characteristics.)

The results of the microbenchmarks are as follows.

Comparands Comparand creation adds 5 to 10% performance overhead to instance creation. This overhead can be considerably reduced by a factor of 0.5 by introducing thread-local comparand buffers. Such buffers assign a range of comparands to a single thread at a time, instead of requiring threads to synchronize accesses to a single comparand counter on each comparand creation. Only when a comparand buffer runs out of comparands, synchronizing threads because of comparands become inevitable. Implementing comparand buffers is straightforward and does not require considerable memory and performance overhead itself, so this technique is definitely worthwhile to employ. A moderate number of comparands per request of about 100 to 1000 comparands is sufficient to gain considerable performance improvements. A more sophisticated scheme, however, that keeps track of comparands that become unused due to garbage collection is unlikely to make sense in the face of a virtually unlimited total number of comparands.

Comparison of references are about 25% slower due to the additional indirection via explicit references and the use of comparands for comparisons.

Referents The additional indirection layer induces a performance overhead of 4% at most, as determined in microbenchmarks that do not involve a control of concurrent activities. In order to determine whether this very low overhead is due to the use of local indirections, as described in Section 6.4.1, an alternative implementation of the Gilgul Virtual Machine was used in further test runs that strictly keeps references and their referents on different memory pages. Only under pessimistic pathological boundary conditions, when a very large number of objects is accessed in a random fashion, this separation of references and referents lead to a considerable performance overhead of 10 to 20%. The more realistic macrobenchmarks, however, can only be improved by about 1.5 to 4% by using local indirections. It has been a surprise to us that indirections do not incur a high overhead, contrary to widely held assumptions.

7.2.3 Activity Control

The Gilgul Virtual Machine implements two variants of a Registration Protocol in order to prevent replacements of active objects (see Section 6.5). On the one hand, the influence of these protocols on the execution of standard Java programs has been analyzed. Naturally, the Adaptive Registration Protocol leads only to a minimal performance overhead since its very purpose is to shift as much overhead as possible to the referent assignment operation. However, the Simple Registration Protocol incurs a very high overhead especially in the case of instanceof checks and comparisons (about 3 to 4.5 times slower than without any activity control).

On the other hand, it is important to be able to predict what overhead activity control will incur in "real-world" GILGUL programs. Due to the lack of a considerable number of GILGUL programs because of its status as a research prototype, we have opted for writing a macrobenchmark GilgulBench in which the relative number of various operations can be influenced by parameters, but otherwise behaves in a random fashion. The operations whose frequency distribution can be determined by those parameters are as follows.

- Referent assignments.
- Method calls with a nesting depth of five method calls.
- Synchronized method calls, again nested.

The parameters are as follows.

- t determines the number of threads.
- a determines the number of operations per thread.
- o determines the number of objects.
- r determines the number of referent assignments.
- s determines the number of synchronized method calls.

Furthermore, GilgulBench has been executed on different configurations of the Gilgul Virtual Machine: with only a Simple Registration Protocol, with an Adaptive Registration Protocol, and for comparison purpose without any activity control. (In the latter case, the behavior of the program is semantically wrong because active objects can be arbitrarily replaced. However, since GilgulBench is already a synthetic program for pure analysis purposes this does not matter much.)

The test runs lead to the following conclusions.

- The Adaptive Registration Protocol induces a very low performance overhead in comparison to the Gilgul Virtual Machine without activity control, given that the rate of referent assignments is low ($r <= 1$).
- The Simple Registration Protocol induces a very high performance overhead in any configuration (25% and more).
- The performance of the Adaptive Registration Protocol depends on the number of threads. This is clear given the fact that it needs to analyze the call stack of each thread in the event of a referent assignment. Nevertheless, it is interesting that the performance overhead is linear in the number of threads, which indicates a good scalability at least when the number of referent assignments is low.
- The Simple Registration Protocol becomes better than the Adaptive Registration Protocol as late as the rate of referent assignments and the number of threads becomes very high ($r = 10, t > 50$).
- Both protocols do not seem to suffer from a higher number of concurrent accesses to objects since the parameter o has no significant effect on the test results.
- The criticality of synchronized blocks is not very high. When s is set to 50% the tests run slower than with s set to 0%. However, the penalty is only a constant of about 10% that is independent from t.

The parameter r (rate of referent assignments) has a considerable effect on the test results. In order to get an understanding what that parameter means, the following considerations can be taken into account.

- When the use of the referent assignment is restricted to scenarios in which it is used for dynamic software evolution, or only selected design patterns, the rate of referent assignmenst is likely to be very low (r falls in the range of 0.1 to 1%). In that case, the activity control does not have a significant effect on the performance of a program.
- When the referent assignment is regarded as an essential part of the programming language, it is conceivable that the rate of referent assignments rises to 5 to 10%. In combination with a high number of threads ($t > 50$), the Simple Registration Protocol becomes preferable: Although it induces a high performance overhead, that overhead is independent from the number of threads.

7.2.4 Summary

The memory overhead induced by comparands and explicit references can be as high as 20% or more while the performance overhead is generally less

than 5%. This performance overhead covers using comparands for comparison operations, an additional level of indirection for accessing objects, and the coordination of protected activities. When observed in isolation, the performance overhead of single operations ranges from 25% for comparisons due to comparands, over 10% for comparand creation, down to less than 5% for additional indirection and activity control. Comparand creation can be considerably improved by way of thread-local comparand buffers. The localization of indirections (by placing explicit references close to the objects to which they refer) can considerably improve performance in certain artificial circumstances, but additional indirections incur a surprisingly small overhead by default. With regard to activity control, the Adaptive Registration Protocol should generally be preferred over the Simple Registration Protocol. Only when the number of referent assignments is considerably high, the cost of analyzing the call stacks after the fact makes the Adaptive Registration Protocol infeasible. Apart from that, both variants display a good scalability with regard to various parameters, especially with regard to the number of threads and the number of `synchronized` statements.

As a final note, it must be stressed that the results must be put into perspective because of the fact that the tests have been performed on purely interpreted virtual machines. In a compiled environment, it is likely that the performance of most of the standard Java bytecode instructions can be considerably improved while operations that deal with thread synchronization are by their very nature very expensive operations at the hardware level. Since especially the activity control relies heavily on thread synchronization, it is likely that the extensions needed for the Gilgul Virtual Machine will paint a less optimistic picture than the one we have found here. It is an open question what the actual performance characteristics of Gilgul are in a compiled run-time environment. Some design alternatives are sketched in [58].

Chapter 8

Conclusions and Future Work

This chapter summarizes the essential results of this thesis. The GILGUL model provides a new alternative conceptualization of the notion of object identity; the GILGUL programming language provides a consistent and backwards-compatible extension of the Java programming language that offers flexible operations based on the GILGUL model; the implementation consists of a compiler and a run-time system for the GILGUL programming language that achieves a promising level of performance. This thesis also provides a number of application examples from very different fields that make use of the new operations, but their power shows foremostly when used as a basis for dynamic unanticipated software evolution.

Possible future work includes an implementation of GILGUL as a compiler that translates the new constructs into instructions understood natively by the hardware, for example as part of a dynamic or just-in-time compilation framework, and extensions of the notion of atomic object replacement towards atomic replacement of whole networks of objects, and of class objects at the meta level.

8.1 Results

This thesis starts from an exploration of traditional notions of object identity. It is clear that the notions of reference and comparison are essential ingredients of such traditional notions, but this thesis presents the first approach that strictly separates these notions to the best of our knowledge.

The GILGUL model and the GILGUL programming language introduce the new basic type `comparandtype` together with means to copy comparands between objects, and the referent assignment operator #=. It also changes the definition of the existing equality operators == and != according to the GILGUL model.

This model is a generalization of what can be expressed in terms of object identity in current object-oriented programming languages. GILGUL offers flexible means for declaring restrictions on which operations are valid on specific referents and comparands, for example in order to prevent the replacement of sensible objects. These restrictions can range from the unrestricted applicability of GILGUL's new operations to the reduction to the traditional stringent restrictions placed on object identity.

We have shown how GILGUL's new operations can be applied for the purpose of dynamic object replacement without the need to deal with consistency problems. We have then investigated various effects of GILGUL's model on other aspects of the language, most notably the type system and the handling of active objects.

We proposed an extension of Java's type system that allows "pure" implementation classes to be declared that must not be used as types. This novel approach "completes" previous efforts to separate pure interfaces from classes that, however, still define types of their own. It effectively widens the range of both anticipated and unanticipated adaptations.

By default, GILGUL deals with dynamic replacement of active objects both in multi-threaded and single-threaded contexts in a way that preserves consistency. If programmers are willing to trade consistency for timeliness, or even need to break consistency in order to be able to replace objects at all in the case of non-terminating loops, they can take advantage of GILGUL's advanced facilities for these cases.

We have outlined the concept of *recalls* that has been introduced in GILGUL. Like exceptions, recalls unwind the call stack, and can be thrown and caught. Unlike exceptions, the return to the standard flow of control is guaranteed as soon as the call stack is clear of a specific object as a receiver of a method call. At this point in time, the very first call to the specified object is simply re-executed.

The referent assignment operator can be annotated with several variants of recalls, which means that the actual replacement is deferred until the corresponding call stack is clear of the object to be replaced. Just before re-execution of the first method call to this object, the actual replacement takes place.

Although this combination of the referent assignment operator and a recall might break consistency, target objects are still able to react to the throw of recalls by providing recall handlers for clearance purposes, just like exception handlers in Java. However, even if recall handlers have not been provided, replacements can still be carried out ensuring timeliness, and replaceability in the presence of non-terminating loops. This might sometimes be the last resort before a system shut-down becomes inevitable.

We have sketched some of the properties of the implementation of the GILGUL compiler and run-time system. The most notable achievements in this area include the surprising result that double indirection, which can

only partly be avoided when implementing referent assignments, does not impose a serious performance penalty, and a sophisticated registration protocol for compound activities that move the overhead for synchronizing referent assignments among multiple threads almost completely to the referent assignment operation itself. This leads to usable performance characteristics of the Gilgul Virtual Machine that does not impose serious performance penalties, neither on "pure" Java programs nor on programs that make use of GILGUL's new capabilities. Only in corner cases, the registration protocols required for multithreading considerably decreases the run-time performance of a GILGUL program.

We have shown a considerable number of usage scenarios, in Section 7.1 for the referent assignment operation and in Appendix B for comparands and comparand assignment. This proves that GILGUL provides language constructs that have the potential to be of practical relevance in real-world applications.

8.2 Possible Extensions

GILGUL is one of the approaches of the TAILOR Project at the University of Bonn, whose purpose has been to explore opportunities for unanticipated software evolution in programming languages and run-time systems. For this reason, GILGUL has been intentionally designed to be as flexible as possible in order to fit to these goals, while putting up with certain run-time exceptions (for example, the GilgulRestrictionException, or the ClassCastException in the case of the with statement for typeless classes). Although we are certain that our approach is not too extreme, it can still be a fruitful task to investigate a more "gentle" approach that aims for a higher degree of static checkability in the future.

Future work also clearly includes an integration of GILGUL's concepts into a just-in-time or dynamic compiler at the implementation level. The work presented in [58] already sketches some possible ingredients of such an implementation.

At the conceptual level, the notion of referent assignment provides two obvious potential paths for further exploration. Both paths can be characterized as a widening of the granularity of referent assignment: In the GILGUL model, exactly one object can be replaced at a time. This object-centric granularity can be extended either by focusing on whole nets of objects and trying to achieve atomicity of the replacement of such nets. This would provide a natural extension from object-based replacement towards component-based replacement. On the other hand, several languages provide *metaobject protocols* [44] that model classes and other aspects of a programming language as objects themselves. These languages include for example Common Lisp and Smalltalk, and Java also provides a limited

form of a metaobject protocol in its reflection API. In principle, it would be straightforward to allow a referent assignment operation to be applied to such meta-level objects. However, the replacement of a class metaobject immediately leads to the question how to consistently reflect the changes of the class metaobject in its instances at the base level. Both paths, replacements of nets of objects and replacements of class metaobjects, open up a new dimension of complexity with regard to the various issues that are discussed in this thesis for replacements of base-level objects, like control facilities, type issues and replacements of active objects.[1]

[1] Exactly because these issues are outside of the scope of this thesis, Section 6.2.5 define instances of java.lang.Class to have bound referents. The issues that future work on opening up referent assignments for meta-level objects needs to deal with can be formulated as the question what it exactly means to define instances of java.lang.Class to have unbound referents.

Appendix A

The Gilgul Language Specification

A.1 Introduction

The following sections define the programming language GILGUL in the style of “The Java Language Specification” [34]. They only list changes and additions to that specification. All the other definitions are exactly like those given in that specification.

In ambiguous cases, the specifications given here override those given in the Java Language Specification. For example, Java specifies a call to another constructor to be the very first action in any constructor except for the class `Object`, whereas in A.10.3 it is specified for GILGUL that comparand initializations are performed before any call to other constructors. In this case, the rules for GILGUL take precedence over those given for Java.

A.2 Notation

The notational conventions are like those given in Chapter 2 of [34]. Differences to that specification are underlined. Like in that book, the grammar is presented here in a way that is better for expositon rather than as a basis for a parser.

At the beginning of each of the following sections, a short note in *italics* indicates the Chapter and Section of [34] to which the subsequent sections refer to.

A.3 Lexical Structure

A.3.1 Keywords

Section 3.9

The list of keywords is augmented, resulting in the following production.

Keyword: one of

```
abstract      continue  goto        private    synchronized
boolean       default   if          protected  this
bound         do        implements  public     throw
break         double    import      recall     throws
byte          else      instanceof  referent   total
case          extends   int         return     transient
catch         final     interface   short      try
char          finally   local       static     typeless
class         fixed     long        strictfp   void
comparand     float     native      super      volatile
comparandtype for       new         with       while
const         global    package     switch
```

A.3.2 Operators

Section 3.12

The list of operators is augmented, resulting in the following production.

Operator: one of

```
=   >   <   !   ~   ?   :   #=
==  <=  >=  !=  &&  ||  ++  --
+   -   *   /   &   |   ^   %   <<   >>   >>>
+=  -=  *=  /=  &=  |=  ^=  %=  <<=  >>=  >>>=
```

A.4 Types, Values, and Variables

A.4.1 Primitive Types and Values

Section 4.2

The list of primitive types is augmented, resulting in the following production.

PrimitiveType::
  NumericType
  `boolan`
  `comparandtype`

There are no requirements on how comparands are represented internally. The programming language Gilgul does not provide any means to access the internal representation of a comparand.

The operations that are defined on comparands are the comparand equality operators `==` and `!=`, which result in a value of type `boolean` (A.10.6), and the comparand creation expression `new comparandtype`, which results in a fresh value of type `comparandtype` (A.10.5).

Furthermore, the reference equality operators == and != are redefined to operate on the comparands of the involved objects (A.10.6), and the class instance creation expression is redefined to create a fresh comparand for a newly created object (A.10.3).

A.4.2 Reference Types and Values

The Class Object

Section 4.3.2

The class Object is augmented with the field comparand of type comparandtype, which is by default initialized with a fresh comparand when a new instance is created. The definition of that field can therefore be roughly summarized as follows.

```
package java.lang;

public class Object {
   public comparandtype comparand = new comparandtype;

                . . .
}
```

Remark: This code is strictly not legal GILGUL code since comparand is listed as a keyword of the language in A.3.1, and keywords are not available as user-definable identifiers. However, the comparand of an object can still be understood as one of its fields. The treatment of comparand as a keyword also allows a comparand to be accessed via an interface type as a special case. Since Object is the only type whose definitions are implicitly available via interface types, GILGUL's treatment of the comparand field as a special case does not give rise to generally allow instance field declarations in interfaces.

A.4.3 Where Types Are Used

Section 4.4

GILGUL introduces the notion of typeless classes and interfaces (A.6.1, A.7.1). Of the kinds of usage of types listed in Section 4.4 in the Java Language Specification, the types defined by such classes and interfaces can never be used in declarations, except for import declarations, but are always available in expressions, except for array creation. This means that the types of typeless classes and interfaces are not available as types for field declarations, method parameter types, method result types, constructor parameter types, local variable types, exception handler parameter types, and array creation expressions; the types of typeless interfaces are available as types for import declarations, casts and the instanceof operator; the types of typeless classes are additionally available for class instance creations.

A.4.4 Variables

Section 4.5

If the type of a variable is a reference type, it must not be the type of a typeless class.

Initial Values of Variables

Section 4.5.5

If a class variable, instance variable or an array component is of type `comparandtype`, its default value is a freshly created comparand, as per comparand creation expression (A.10.5). The GILGUL programming language does not define any other meaningful default value for comparands. The semantics of GILGUL ensure that it is not possible to read a non-initialized comparand.

Types, Classes, and Interfaces

Section 4.5.6

Types that are introduced by typeless classes or interfaces can never play the role of compile-time types of variables. If an object is a direct instance of a typeless class, it is still possible to refer to that class as its "run-time type".

Historically, the Java programming language had a stricter distinction between the terms "types" and "classes", as is described in Section 4.5.5 of the first edition of the Java Language Specification [33]. GILGUL's term "typeless" was coined on the background of that stricter distinction.

A.5 Conversions and Promotions

A.5.1 Kinds of Conversions

Identity Conversions

Section 5.1.1

The identity conversion from `comparandtype` to `comparandtype` is permitted.

Narrowing Conversions

Section 5.1.3

The narrowing conversion from `comparandtype` to `long` is permitted. For each comparand, each conversion to `long` always produces the same value. Furthermore, for two ore more different comparands within a program run, the values produced by conversion to `long` are distinct.

Forbidden Conversions

Section 5.1.7

Except for the identity conversion from `comparandtype` to `comparandtype` and the narrowing conversion from `comparandtype` to `long`, no other conversions are permitted that involve the type `comparandtype`. Especially, there is no conversion from `long` to `comparandtype`.

A.5.2 Assignment Conversion

Section 5.2

Referent assignment of a value of compile-time reference type S (source) to a variable of compile-time reference type T (target) is checked in the same way as for reference assignments. However, since the run-time type of an object referred to by T might be a subtype of the compile-time type of T, this check cannot guarantee that the referent assignment will be executed without an exception, in contrast to the check for reference assignment. This is because a referent assignment potentially affects variables other than T whose actual types cannot be determined at compile-time in the general case.

There is an additional case that is covered by referent assignments: The left-hand side T of a referent assignment is allowed to be `this` under certain circumstances (A.10.7). Since the class of `this` can be a typeless class, the right hand-side S of the referent assignment does not need to be the same class or a subclass thereof. Instead, the *set* of all compile-time types of T is determined, and the referent assignment check is performed potentially more than once, that is for each of those types being considered as the type of T. The referent assigment check to `this` is successful if and only if the referent assignment checks for all of these types are successful. Consider the following example.

```
class TCPConnection {
  . . .
}

typeless class TCPOpen extends TCPConnection
                       implements Serializable {
  . . .
  void listen() {
    this #= new TCPListen();
  }
}

typeless class TCPListen extends TCPConnection
                         implements Serializable {
```

```
  . . .
  void close() {
    this #= new TCPClosed();
  }
}

typeless class TCPClosed extends TCPConnection {
                         // does not implement Serializable
  . . .
  void open() {
    this #= new TCPOpen();
  }
}
```

The set of compile-time types for both the classes TCPOpen and TCPListen consists of Object, TCPConnection and Serializable; the set of compile-types for the class TCPClosed consists of Object and TCPConnection. Therefore, the referent assignment checks yield the following results:

- For method listen(), the referent assignment is accepted because TCPListen can statically be determined to be a subtype of each Object, TCPConnection and Serializable.
- For method close(), the referent assignment is not accepted because TCPClosed cannot statically be determined to be a subtype of Serializable.
- For method open(), the referent assignment is accepted because TCPOpen can statically be determined to be a subtype of each Object and TCPConnection.

A.5.3 Casting Conversion

Section 5.5
There are no changes to this section. Especially, typeless classes and interfaces can be used in cast expressions without any restrictions.

A.6 Classes

A.6.1 Class Declaration

Class Modifiers

Section 8.1.1
The list of class modifiers is augmented, resulting in the following production.

ClassModifier: one of
`public` `protected` `private`
`abstract` `static` `final` `strictfp`
`typeless`

The type of a typeless class is not available as a type for field declarations, method parameter types, method result types, constructor parameter types, local variable types, exception handler parameter types, and array creation expressions; the type of a typeless class is available as a type for import declarations, class instance creations, casts and the `instanceof` operator.

There are no restrictions on the possible combinations of the `typeless` modifier with other class modifiers.

A.6.2 Field Declarations

Section 8.3

It is possible to initalize a newly declared field by way of a referent assignment. This means that during initialization, a fresh reference is created for the object yielded by the right-hand side and the left-hand side is assigned this fresh reference. This reference will not be affected by subsequent referent assignments to already existing references to that same object. Apart from these semantics for initialization by referent assignment, such initialization follows the same rules as standard Java field initialization.

The definition of *VariableDeclarator* is changed as follows.

VariableDeclarator:
VariableDeclaratorId
VariableDeclaratorId `=` *VariableInitializer*
Identifier `#=` *Expression WithFixedReferent*$_{opt}$

WithFixedReferent:
`with fixed referent`

Initialization by referent assignment is only available for non-array reference types. An additional `with fixed referent` declaration results in a reference whose referent cannot be changed by subsequent referent assignments. The exact rules for fixed referents are given in A.10.7.

A.6.3 Constructor Declarations

Section 8.8

It is possible to declare restrictions on the use of comparands and referents together with a constructor definition. The modified/new productions are as follows.

ConstructorDeclaration:
  ConstructorModifiers$_{opt}$ *ConstructorDeclarator Throws*$_{opt}$
    <u>*WithRestrictionDeclaration*</u>$_{opt}$ *ConstructorBody*

<u>*WithRestrictionDeclaration*</u>*:*
  `with` *ComparandRestriction*
  `with` *ReferentRestriction*
  `with` *ComparandRestriction* `,` *ReferentRestriction*
  `with` *ReferentRestriction* `,` *ComparandRestriction*

ComparandRestriction:
  FixedOrBound `comparand`
  `fixed`$_{opt}$ `comparand =` *Expression*

ReferentRestriction:
  FixedOrBound `referent`

FixedOrBound:
  `fixed`
  `bound`

When a new object is created and initialized, the selected constructor determines the resulting restrictions on comparands and referents, which are just those declared with that constructor. Declarations of other constructors, whether called or not, are ignored.

A fixed comparand cannot be assigned a new value, unless an explicit assignment is given with the restriction declaration. In that case, the comparand is assigned exactly once and cannot be changed afterwards. A comparand initialization, explicit or implicit, is always the first action performed exactly once by the first constructor in a constructor chain, before any call to another constructor, such that all constructor code sees correctly initialized comparands. Only an explicit comparand initialization given with the selected constructor is performed instead of an implicit comparand initialization. All other explicit comparand initializations of other constructors are always ignored, no matter whether indirectly called or not. A comparand that is assigned in the method header does not need to be fixed. In that case, subsequent assignments to the comparand are still possible.

A bound comparand is fixed and additionally cannot be copied to comparands of other objects. A bound comparand cannot be explicitly initialized, but always gets a fresh comparand value determined by the run-time system.

A fixed referent cannot be modified by subsequent referent assignments. A bound referent is fixed and additionally cannot be assigned to other references by subsequent referent assignments. Referents can never be initialized in constructor headers.

When no comparand or referent restrictions are declared, they are respectively considered to be unbound. This is, for example, the case for the

implicitly created default constructor when no other constructor declaration is given in a class.

The exact rules how comparand and referent restrictions are checked are given in Section A.10.7.

A.7 Interfaces

A.7.1 Interface Declarations

Interface Modifiers

Section 9.1.1

The list of interface modifiers is augmented, resulting in the following production.

InterfaceModifier: one of
`public` `protected` `private`
`abstract` `static` `strictfp`
`typeless`

The type of a typeless interface is not available as a type for field declarations, method parameter types, method result types, constructor parameter types, local variable types, exception handler parameter types, and array creation expressions; the type of a typeless interface is available as a type for import declarations, casts and the `instanceof` operator.

There are no restrictions on the possible combinations of the `typeless` modifier with other interface modifiers.

A.8 Exceptions

Chapter 11

GILGUL adds another kind of exception, namely the class Recall that is a subclass of java.lang.Throwable. A recall is always defined in conjunction with an object that must be passed in a constructor of the class Recall. The (public) definition of that class is as follows.

```
package java.lang;

public class Recall extends Throwable {

  public Recall (Object obj) {
    . . .
  }
}
```

The (explicit) throw of a recall results in the same behavior as that of other exceptions, except for the following difference: The call stack of the thread in which the recall is thrown is unwound until the very first method call to the object on which the recall is defined. A recall may be caught by appropriate exception handlers before the run-time system reaches that point. This means that that point might never be reached for a specific throw of a recall. However, if it is reached, the process of unwinding the call stack is stopped at that point and that very first method call is re-executed, without reevaluation of the arguments passed to that method.

A.8.1 The Causes of Exceptions

Section 11.1

Recalls may be explicitly thrown, and such explicit recalls always refer to the thread in which they are thrown. They may also be implicitly thrown in conjunction with referent assignments (A.10.7). In that case, they may also apply to other threads. A GILGUL virtual machine implementation may choose to use subclasses of Recall to distinguish between different variants of implicit or explicit recalls.

A.8.2 Compile-Time Checking of Exceptions

Section 11.2

Like Java's run-time exceptions and errors, but unlike Java's other exceptions, recalls are not statically checked. This is because static checks of recalls would interfere with the goals of unanticipated software evolution. Another reason is that recalls might stem from other threads, and it is inherently not possible to statically determine how Java threads interact with each other in that regard.

A.8.3 Handling of Exceptions

Section 11.3

Since the main purpose of recalls is to allow for object replacement in situations that would otherwise result in a deadlock, it is strongly recommended to rethrow a recall that is caught in an exception handler, and to only use exception handlers for recalls to reset computations to a consistent state. For this reason, Recall is a direct subclass of Throwable so that existing Java code that already catches exceptions and errors without rethrowing them – which is the standard practice in Java – behaves gracefully with regard to recalls.

See Section 5.1.7 in this thesis for an example of a recall handler.

Handling Asynchronous Exceptions

Section 11.3.2

Recalls always occur synchronously, never asynchronously. A Gilgul virtual machine implementation must ensure that this is always the case, especially in the case of global recalls that can be thrown for other threads (A.10.7).

A.9 Blocks and Statements

A.9.1 Local Variable Declaration Statements

Section 14.4

The semantics of local variable declaration statements are changed in the same way as those of field declarations (A.6.2). The description is repeated here to make the presentation clearer.

It is possible to initalize a newly declared local variable by way of a referent assignment. This means that during initialization, a fresh reference is created for the object yielded by the right-hand side and the left-hand side is assigned this fresh reference. This reference will not be affected by subsequent referent assignments to already existing references to that same object. Apart from these semantics for initialization by referent assignment, such initialization follows the same rules as standard Java local variable initialization.

The definition of *VariableDeclarator* is changed as follows.

VariableDeclarator:
  VariableDeclaratorId
  VariableDeclaratorId `=` *VariableInitializer*
  Identifier `#=` *Expression* *WithFixedReferent*$_{opt}$

WithFixedReferent:
  `with fixed referent`

Initialization by referent assignment is only available for non-array reference types. An additional `with fixed referent` declaration results in a reference whose referent cannot be changed by subsequent referent assignments. The exact rules for fixed referents are given in Section (A.10.7).

A.9.2 Statements

Section 14.5

The list of possible statements (*Statement* and *StatementNoShortIf*) is augmented as follows.[1]

[1] The *...NoShortIf* variants are only needed for resolving ambiguities resulting from potentially dangling `else` branches. See Section 14.5 of the Java Language Specification for further details.

Statement:
StatementWithoutTrailingSubstatement
LabeledStatement
IfThenStatement
IfThenElseStatement
WhileStatement
ForStatement
WithThenStatement
WithThenElseStatement

StatementNoShortIf:
StatementWithoutTrailingSubstatement
LabeledStatementNoShortIf
IfThenElseStatementNoShortIf
WhileStatementNoShortIf
ForStatementNoShortIf
WithThenElseStatementNoShortIf

A.9.3 The `return` Statement

Section 14.16

ReturnStatement:
`return ;`
`return` *Expression* *WithRefAssign*$_{opt}$ `;`

WithRefAssign:
`with` *ReferentAssignment*

A `return` statement can be augmented with a referent assignment to `this`. The semantics are as follows: First, the return expression is evaluated; then, the referent assignment is carried out; finally, the result of the first step is returned. This is to enable referent assignments to `this` to be the definite last operation in a method (A.10.7).

If the referent assignment completes abruptly, then the return statement completes abruptly for the same reason. In all other respects, such a return statement behaves like Java return statements.

The GILGUL compiler does not require the left-hand side of the referent assignment to be the keyword `this`.

A.9.4 The `throw` Statement

Section 14.17

ThrowStatement:
`throw` *Expression* *WithRefAssign*$_{opt}$ `;`

A `throw` statement can be augmented with a referent assignment to `this`. The semantics are as follows: First, the throw expression is evaluated; then, the referent assignment is carried out; finally, the result of the first step is thrown. This is to enable referent assignments to `this` to be the definite last operation in a method (A.10.7).

If the referent assignments completes abruptly, then the throw statements completes abruptly for the same reason. In all other respects, such a throw statement behaves like Java throw statements.

The Gilgul compiler does not require the left-hand side of the referent assignment to be the keyword `this`.

A.9.5 The `try` Statement

Section 14.19

Gilgul does not permit a `return` statement or a `throw` statement augmented with a referent assignment to `this` to be placed inside the `try` block of a `try-finally` statement, and also does not permit a `throw` statement augmented with a referent assignment to `this` to be placed inside the `try` block of a `try-catch` statement, and rejects attempts to do so at compile-time. See Section 5.1.2 in this thesis for a detailed rationale.

A.9.6 <u>The `with` Statement</u>

<u>*WithThenStatement*</u>:
  `with (` *Expression* `instanceof` *ReferenceType* `)` *Statement*

<u>*WithThenElseStatement*</u>:
  `with (` *Expression* `instanceof` *ReferenceType* `)` *StatementNoShortIf*
    `else` *Statement*

<u>*WithThenElseStatementNoShortIf*</u>:
  `with (` *Expression* `instanceof` *ReferenceType* `)` *StatementNoShortIf*
    `else` *StatementNoShortIf*

The `with-then` and `with-then-else` statements behave like Java's `if-then` and `if-then-else` statements with the following differences: The only boolean expression allowed to be checked is an `instanceof` expression. In the lexically visible code that makes up the `then` branch of a `with` statement, all occurrences of the left-hand side of the `instanceof` expression are embedded into casts to the right-hand side of that `instanceof` expression. Consider the following example.

```
with (this.field instanceof MyClass) {
   . . .
   this.field.m();
   . . .
}
```

This is translated into the following equivalent code.

```
if (this.field instanceof MyClass) {
   . . .
   ((MyClass)this.field).m();
   . . .
}
```

It is generally not possible to ensure that the expression of the left-hand side of the `instanceof` expression always refers to the same conceptual entity througout the body of a `with` statement, especially if it would be an arbitrary expression. The intended usage of the `with` statement is for local variables or for fields of `this`. In order to cast this into a manageable usage restriction, Gilgul only permits an *ExpressionName* to be the left-hand side of an `instanceof` expression being the condition of a `with` statement. All other kinds of expressions are rejected by the compiler in these specific places. See Section 6.5 of the Java Language Specification for details on the meaning of names in Java.

A.10 Expressions

A.10.1 Normal and Abrupt Completion of Evaluation

Section 15.6

Gilgul adds the following run-time exceptions for predefined operators:

- A simple assignment (A.10.7) of one value's comparand to another value's comparand throws a GilgulRestrictionException when either value is null, or when the left-hand object's comparand has been created as a fixed or bound comparand, or when the right-hand object's comparand has been created as a bound comparand.

- A referent assignment (A.10.7) throws a GilgulRestrictionException when the left-hand side is null, or when it has been created with a fixed or bound referent, or when the right-hand side has been created with a bound referent. It throws a ReferentAssignmentException if at least one method is active on the object referred to by the left-hand operand and no `local recall` or `total recall` declaration is given with the referent assignment.

A.10.2 Primary Expressions

Section 15.8

PrimaryNoNewArray:

Literal
Type `. class`
`void . class`
`this`
ClassName `. this`
`(` *Expression* `)`
ClassInstanceCreationExpression
<u>*ComparandCreationExpression*</u>
FieldAccess
MethodInvocation
ArrayAccess

A.10.3 Class Instance Creation Expressions

Run-time Evaluation of Class Instance Creation Expressions

Section 15.9.4

Since Gilgul does not define a distinct default value for comparands, the comparand field of an object is always initialized either with the result of evaluating the explicit comparand initialization expression (A.6.3) given in the constructor chosen for the particular class instance creation expression, or else implicitly with a fresh comparand if no comparand initialization expression is given, as per comparand creation (A.10.5). Only the selected constructor is used to determine whether an explicit or an implicit comparand initialization takes place. Other constructors that might be called by the selected constructor are completely ignored in this regard.

Comparand initialization (implicit or explicit) takes place immediately after the actual arguments to the constructor have been evaluated. An explicit constructor initalization expression can access those arguments via the constructor parameter names. If the comparand initialization expression completes abruptly, the class instance creation expression completes abruptly for the same reason.

Class instance creation proceeds with the invocation of the selected constructor, as specified in the Java Language Specification.

Anonymous Class Declarations

Section 15.9.5

It is not possible to define any comparand or referent restrictions on instances of anonymous classes. This might be a possible extension of a future version of Gilgul, but the need for this did not arise as yet. As one consequence, the comparands of instances of anonymous classes are always implicitly initialized.

A.10.4 Array Creation Expression

Section 15.10

A single-dimensional array with `comparandtype` as its component type has each of its components initialized with a fresh comparand, as per comparand creation (A.10.5).

A.10.5 Comparand Creation Expressions

ComparandCreationExpression:

```
new comparandtype
new comparand
```

A comparand creation expression creates a fresh comparand that is unique for the currently executing virtual machine. Comparand creation determines the result of the comparand equality operators `==` and `!=` (A.10.6).

The compile-time type of a comparand creation expression is `comparandtype`. The expressions `new comparandtype` and `new comparand` are synonymous; the latter is just provided for convenience.

A.10.6 Equality Operators

Reference Equality Operators == and !=

Section 15.21.3

In GILGUL, references cannot be compared. The reference equality operators are redefined as follows.

At run time, the result of `==` is `true` if the operand values are both `null`, or both refer to arrays or objects with the same comparand; otherwise, the result is `false`.

The result of `!=` is `false` if the operand values are both `null`, or both refer to arrays or objects with the same comparand; otherwise, the result is `true`.

Comparand Equality Operators == and !=

If the operands of an equality operator are both of type `comparandtype`, then the operation is comparand equality.

At run time, the result of `==` is `true` if the operand values have been created by the same comparand creation expression; otherwise, the result is `false`.

The result of `!=` is false if the operand values have been created by different comparand creation expressions; otherwise, the result is `true`.

A.10.7 Assignment Operators

Section 15.26

AssignmentExpression:
  ConditionalExpression
  Assignment
  ReferentAssignment

AssignmentExpressionNoRefAssign:
  ConditionalExpression
  Assignment

Assignment:
  LeftHandSide AssignmentOperator AssignmentExpressionNoRefAssign

Simple Assignment Operator =

Section 15.26.1

If the types of the right-hand operand and the variable involved in the simple assignment are reference types and the operand is also a variable, then subsequent referent assignments to either variable assigns the other as well. This also holds transitively for assignments of either variable to a third variable, and so on.

It is possible to determine whether both the variable and the right-hand operand denote comparands of objects. This is always the case when the name comparand is used as a selector (*expression*`.comparand`).[2] When this is the case, i.e. when both sides denote comparands of objects, then a compile-time error occurs if it is impossible to convert the type of either object to the type of the other object by a casting conversion. These objects would not be comparable under the given types following the Java rules for equality operators, so it is reasonable to warn the programmer in this case. This restriction can be lifted by casting at least one of the involved objects to type Object.

All other kinds of comparand assignment are handled liberally with regard to static typing, including the cases in which only one side is the comparand of an object.

At run time, the left-hand comparand is checked whether it was created with a `fixed comparand` or `bound comparand` declaration. If this is the case, the assignment expression completes abruptly by throwing a GilgulRestrictionException. Likewise, the right-hand comparand is checked whether it was created with a `bound comparand` declaration. Again, if this is the case, the assignment expression completes abruptly by throwing a GilgulRestrictionException.

[2] Because comparand is defined as a keyword in GILGUL, a programmer can not use the name comparand to hide the comparand field of class Object.

Referent Assignment Operator #=

ReferentAssignment:
  LeftHandSide `#=` *AssignmentExpressionNoRefAssign WithRefAssignMods*$_{opt}$

WithRefAssignMods:
  `with` *RefAssignMods*

RefAssignMods:
  RefAssignMod
  RefAssignMod , *RefAssignMods*

RefAssignMod:
  `fixed referent`
  `local recall`
  `global recall`
  `total recall`

A compile-time error occurs if the type of the right-hand operand cannot be converted to the type of the variable by assignment conversion. A compile-time error occurs if the left-hand operand is a definitely unassigned variable and one or more of the declarations `local recall`, `global recall` or `total recall` is used. A compile-time error occurs if the left-hand operand is a definitely assigned variable or `this`, and the declaration `fixed referent` is used. A compile-time error occurs if the left-hand side is neither definitely assigned nor definitely unassigned, or if it is both definitely assigned and definitely unassigned. A compile-time error occurs if the left-hand operand is `this` and the referent assignment cannot possibly be the definite last operation of the method in which it occurs.

At run time, the expression is evaluated in one of three ways. If the left-hand operand expression denotes a definitely unassigned variable, then the following steps are required:

- A fresh reference is created for the object yielded by the right-hand side and the left-hand side is assigned this fresh reference. This reference will not be affected by subsequent referent assignments to already existing references to that same object. Apart from these semantics for initialization by referent assignment, such initialization follows the same rules as standard Java variable initialization.

- Initialization by referent assignment is only available for non-array reference types. An additional `with fixed referent` declaration results in a reference whose referent cannot be changed by subsequent referent assignments. The exact rules for fixed referents are given below.

Otherwise, if the left-hand operand expression is not an array access expression, then it is a referent assignment to a definitely assigned variable and the following steps are required:

- First, the left-hand operand is evaluated to produce a variable. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the right-hand operand is not evaluated and no assignment occurs.

- Otherwise, the reference stored in that variable is checked whether it was created with a `fixed referent` or a `bound referent` declaration. If this is the case, the assignment expression completes abruptly by throwing a GilgulRestrictionException; the right-hand operand is not evaluated and no assignment occurs.

- Otherwise, if at least one method is active on the object referred to by the left-hand operand, and no `local recall` or `total recall` declaration is given with the referent assignment, the assignment expression completes abruptly by throwing a ReferentAssignmentException and no assignment occurs.

- Otherwise, the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.

- Otherwise, the reference of the right-hand operand is checked whether it was created with a `bound referent` declaration. If this is the case, the assignment expression completes abruptly by throwing a GilgulRestrictionException and no assignment occurs.

- Otherwise, the run-time type of the right-hand operand is checked against the type of the object *obj* referred to by the left-hand operand. If the run-time type of the right-hand operand is not a subclass of any of the non-typeless classes and interfaces that *obj* is an instance of, the assignment expression completes abruptly by throwing a ClassCastException and no assignment occurs.

- Otherwise, if a `local recall` or `total recall` declaration is given with the referent assignment, and the current thread has at least one active method on the object *obj* referred to by the left-hand variable, an instance of Recall on object *obj* is thrown in the current thread and the subsequent steps are performed immediately before the first method call to *obj* is re-executed. Additionally, if a `global recall` or `total recall` declaration is given with the referent assignment, an instance of Recall on object *obj* is thrown in all other threads than the current one that have at least one active method on *obj*, and the subsequent steps are performed immediately before the first method calls to *obj* are re-executed in each of those threads. In other words, the subsequent steps must wait for all recalls to wait for re-execution

of the corresponding first methods. Recalls that are thrown in other threads are probably delayed in order to ensure synchronicity (A.8.3).

- Then, the actual referent assignment is performed so that the left-hand variable refers to the same object referred to by the right-hand operand. Subsequent referent assignments to either side of the assignment expression do not affect the other. If the object referred by the right-hand operand evaluates to null, a fresh unbound null is created that remembers the set of non-typeless classes and interfaces of the object referred to by the left-hand operand, and that fresh unbound null is stored as a referent for the reference stored in the left-hand operand instead. This set of non-typeless classes and interfaces is used for the dynamic type check of a subsequent referent assignment to the reference stored in the left-hand operand.

Otherwise, if the left-hand operand expression is an array access expression, then the following steps are required:

- First, the array reference subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the index subexpression (of the left-hand operand array access expression) and the right-hand operand are not evaluated and no assignment occurs.

- Otherwise, the index subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and the right-hand operand is not evaluated and no assignment occurs.

- Otherwise, the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.

- Otherwise, if the value of the array reference subexpression is null, then no assignment occurs and a NullPointerException is thrown.

- Otherwise, the value of the array reference subexpression indeed refers to an array. If the value of the index subexpression is less than zero, or greater than or equal to the length of the array, then no assignment occurs and an ArrayIndexOutOfBoundsException is thrown.

- Otherwise, the value of the index subexpression is used to select a component of the array referred to by the value of the array reference subexpression. This component is a definitely assigned variable. Referent assignment proceeds as above.

The grammar productions for referent assignments do not allow for chains of referent assignments of the form `a #= b #= c ...`. This would have seriously complicated the type system with no obvious gain in expressivity. Still, referent assignment uses the form of an expression instead of a statement in order to accomodate possible future extensions of the language.

A.11 Definite Assignment

Chapter 16

Java's rules for definite assignments are augmented in GILGUL by simply regarding referent assignments as an additional kind of assignment. This means that a local variable can be assigned by referent assignment (A.9.1) as well as blank final variables, for example fields (A.6.2). In both cases, a referent assignment is considered an initialization if and only if the left-hand side is a definitely unassigned variable. Otherwise, the left-hand side must be a definitely assigned variable or `this`. It is a compile-time error if the left-hand side is neither definitely assigned nor definitely unassigned, or both definitely assigned and definitely unassigned.

The rules for definite assignment for `with` statements are the same as those for `if` statements – see Section 16.2.7 of the Java Language Specification.

Appendix B

The Comparand Pattern

This Appendix includes the text of the paper [19] that describes how some of the results of this thesis that involve the notion of comparands can be used in pure Java and other programming languages that do not provide them by default. This paper was jointly written with Arno Haase.

B.1 Thumbnail

The COMPARAND pattern provides a means to interpret different objects as being the same for certain contexts. It does so by introducing an instance variable in each class of interest – the comparand – and using it for comparison. Establishing the sameness of different objects is needed when more than one reference refers to conceptually the same object. In distributed systems, the COMPARAND pattern provides for efficient comparison of (possibly) remote objects.

B.2 Example

Suppose you want to implement the Java Platform Debugger Architecture (JPDA), a specification of a debugging framework for the Java Virtual Machine (JVM).

The JPDA consists of three levels: the Java Virtual Machine Debug Interface (JVMDI), an API that is to be implemented in native code, at the level of the JVM; the Java Debug Wire Protocol (JDWP), that allows debuggers to remotely employ the capabilities offered by the JVMDI; and finally, the Java Debug Interface (JDI), a high-level Java API that abstracts from the details of the other levels and thus allows for the implementation of a concrete debugger in a pure object-oriented fashion (see fig. B.1).

This architecture expects a debugger to be executed on an instance of the JVM which is different from that of the target application. Therefore the target application's objects cannot be directly referred to in the debugger

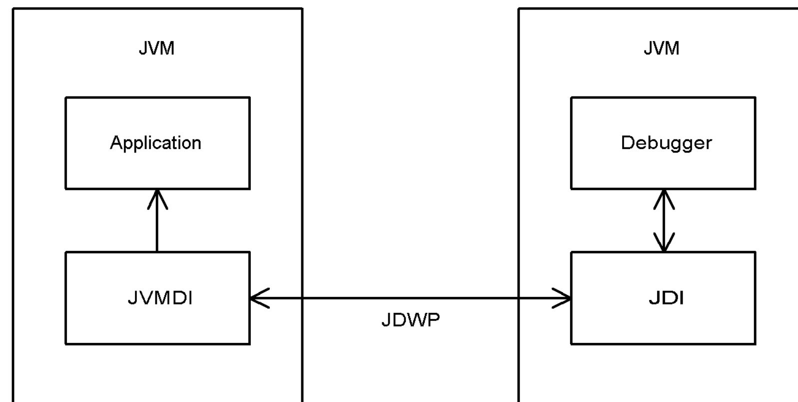


Figure B.1: The Java Platform Debugger Architecture.

by references as offered by the Java Programming Language. Instead they have to be represented as objects that act as remote references.

If a debugger needs to compare variables holding such remote references in order to determine if they refer to the same remote object, care has to be taken to do so correctly. Different remote references might refer to the same remote object, since they can be created independently, for example by consecutive retrieval operations. Therefore, if comparison of remote references yields false, it is not guaranteed that they actually represent different remote objects.

The straightforward solution is to execute an operation on the remote system that determines the correct answer. However, beyond the performance penalty that this solution incurs, it also interferes with the goal of the Java Platform Debugging Architecture which is to isolate the debugger from the target application as far as possible in order to avoid potential side effects.

The COMPARAND pattern solves this problem by adding an attribute to the remote references, the so-called comparand. The comparand of a particular reference is assigned a value that uniquely identifies its remote object.

Consequently, only a comparison of comparands is needed in order to determine sameness or difference of the respective remote objects. They can therefore be used to carry out the comparison operation efficiently. Interferences with the execution of the target application are reduced to the actual creation of comparands inside the JVM of the target application.

B.3 Context

Comparison of objects with reference semantics without comparing their references.

B.4 Problem

Object comparison is usually taken to mean either comparison for sameness (object identity) or comparison of state. The first of these approaches corresponds to so-called *reference semantics*, usually based on comparison of references or pointers; the second approach corresponds to *value semantics*, using all or a subset of the attributes.[1]

However, neither of these approaches is sufficient when reference semantics is to be maintained but reference comparison does not ensure sameness. This is the case when there are different references that can refer to conceptually the same object. In the motivating example, remote references are represented as objects on their own: for this reason, the target object together with its remote references form a conceptual entity that should be indistinguishable from the outside. So the issue is not how to change reference semantics to value semantics but how to have reference semantics with a comparison operation that does not simply compare the references. As another example, particularly in distributed systems, several proxies [31] in the same address space refer to either the same or to different remote objects (see fig. B.2). In this case, the fact that a lack of a direct reference mechanism for remote objects has to be overcome results in potentially ambiguous references.

Another example is an implementation of the DECORATOR pattern [31], where not only different decorators may be applied to the same core object, but they can even decorate each other since decorators and decorated objects have the same interfaces in general. (See [31] for examples.) Here, comparison of references might not reveal that they actually refer to the same core object, but there is a need to establish sameness for decorator objects that are strictly different.[2]

In such circumstances, the following forces have to be balanced:

- A comparison of object state does not yield the intended result since reference semantics is desired.
- A comparison of references cannot be relied upon since one wants to consider different objects to be the same. These objects might even be instances of different types.

[1] Other semantics for object comparison include more complex equality operations that, for example, take structural equivalence into account. See [5] and [35] for discussions on the range of possible equality semantics.

[2] The latter is also known as an example of a split object [6].

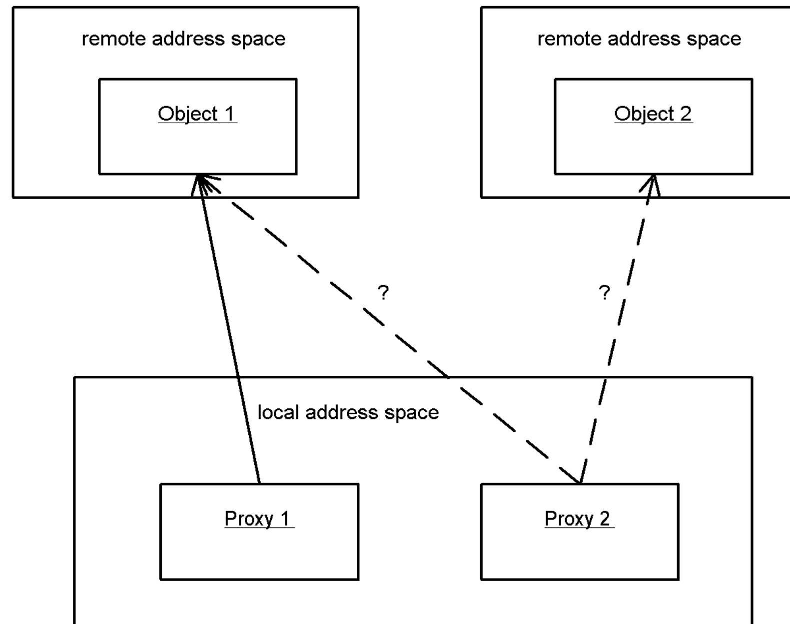


Figure B.2: Do the two proxies refer to the same remote object or not?

- The sameness of objects in general depends on the context. Objects that are considered the same in one context can be different in another.

- If an object is copied[3], care must be taken how to define comparison between original and copy. There are cases where comparison between the two should yield `true` – leaning more towards value semantics – but others where they must be distinguished.

- Sometimes comparison of objects of different types must yield `true`, for example different decorators of the same object, especially decorators of decorators.

- A system may want detailed control of the possible results of object comparisons. For example, when the cost of object creation is to be lowered by introducing a recycling mechanism, the expected result of comparison even changes over time.

- Comparison must be a cheap operation in terms of run-time overhead if it is performed frequently. This requires particular attention in distributed systems.

[3]For the purposes of this paper, the `clone` method is just one means of allowing an object to be copied. Therefore the two are used interchangeably except where implementation details are discussed.

- In a distributed system, it is usually non-trivial to determine sameness of object references locally and executing a remote call for comparison introduces a significant performance overhead.

- The additional memory overhead associated with achieving the desired comparison behavior often needs to be small, especially if many objects are involved.

B.5 Solution

Introduce an instance variable in each class of interest – the comparand – that does not belong to the conceptual state of their respective objects, and compare objects by comparing their comparand values.[4]

```
public class MyClass {

   protected static long comparandCounter = 0;

   protected long comparand = comparandCounter++;

   public boolean equals(Object obj) {
      if (obj instanceof MyClass) {
         MyClass that = (MyClass)obj;
         return this.comparand == that.comparand;
      }
      return false;
   }

   // rest of class body
   ...

}
```

The comparands stored in the objects under consideration can be either values of a primitive type or instances of a compound type. Primitive values of 64 bits are large enough to allow 10 billion unique comparands per second to be created for half a century which is good enough for almost

[4]We have chosen the artificial name COMPARAND for this pattern to stress that this instance variable is a passive entity that is not used for referencing, but within comparison operations only. Elsewhere, names like "key" and "identifier", or acronyms like "OID" and "id" are used for this concept, but these names are used ambiguously and with overloaded meanings throughout the literature. Many brainstorming sessions have not revealed a better name, so we have opted for COMPARAND.

all applications.[5] In this case, unique comparands can always be created efficiently by just increasing a global counter. Therefore, in a local context that allows the management of comparands to be centralized, there is no reason to use compound comparands with the associated performance and memory overhead.[6]

B.6 Implementation

There are some subtle issues when applying the COMPARAND pattern, which are discussed in the following sections.

B.6.1 The "Right" Comparison Semantics

It is important to thoroughly understand what exactly object comparison is supposed to mean in the context at hand. The COMPARAND pattern is applicable only if the intended behavior is that of reference semantics, but not that of value semantics or even of some intermediate semantics.[7]

Sometimes different contexts require different comparison semantics. For example, after application of the DECORATOR pattern the core object and its decorators represent the same conceptual entity. However, certain clients expect the comparison of decorator objects to determine if their respective core objects are the same, whereas other clients need to differentiate between the decorators. The introduction of more than one comparison operation (for example `equals` and `equalsDecorator`) is advisable under these circumstances.

B.6.2 Comparison of Clones

If an object can be copied or cloned, typically afterwards both objects have exactly the same state, but they are not identical. The COMPARAND pattern offers the flexibility to define any desired degree of sameness. There is a free choice to assign the copy the original comparand or a new one which can even be based on dynamic properties of the environment. However, then one must consider the question what the correct behavior should be in a given context. In the general case, a new comparand should be assigned by

[5] 32 bit values, on the other hand, are usually not big enough to ensure uniqueness for long-running applications. At a rate of 1000 comparands per second, they wrap around after roughly $1\frac{1}{2}$ months. In the rare cases when even 64 bits are insufficient, two or more long integers can easily be combined in a customized value type with a larger range of numbers.

[6] This need only arises in the case of distributed applications. See *Comparands in Distributed Environments* in the Implementation section for further details.

[7] Sometimes, comparison of objects needs to take sophisticated aspects into account, for example structural equivalence of complex object types. A thorough overview of these issues is given in [35].

default, as illustrated in the following example, since clones can usually be regarded as independent instances.

```
public class MyClass implements Cloneable {

   // comparand and equals() as above
   ...

   public Object clone() {
      try {
         MyClass myClone = (MyClass)super.clone();
         myClone.comparand = comparandCounter++;
         return myClone;
      } catch (CloneNotSupportedException e) {
         // since MyClass implements Cloneable
         // this exception cannot occur
         throw new InternalError();
      }
   }
}
```

However, an example of a system that may need to treat objects and their clones as equal is one that offers transactional services. It creates copies of objects to operate on them instead of the original ones, so that a rollback operation is easily implemented by just discarding these copies. From a system programmer's point of view, the disambiguation of copies from original objects is clearly needed, but from an application programmer's point of view it is not desirable to distinguish between them. Again, the introduction of more than one dedicated comparison operation (with different access rights, if applicable) may solve this problem.

B.6.3 Which Classes Are Comparable To Each Other?

Another important issue is the determination of the classes that are supposed to be comparable. If it is required to potentially establish identity for any two objects of arbitrary type, further effort is needed. For example, in Java an interface can be introduced that otherwise unrelated classes can implement, allowing their objects to be compared as follows.

```
public interface Comparable {
   public long getComparand();
}
```

Since in this case the creation of comparands does not naturally belong to one of the comparable classes anymore, it should be factored out into a

class of its own.[8]

```
public class ComparandFactory {

   private static long comparandCounter = 0;

   public long getNewComparand() {
      return comparandCounter++;
   }
}

public MyClass implements Comparable {

   protected long comparand =
      ComparandFactory.getNewComparand();

   public long getComparand() {
      return this.comparand;
   }

   public boolean equals(Object obj) {
      if (obj instanceof Comparable) {
         Comparable that = (Comparable)obj;
         return this.comparand == that.getComparand();
      }
      return false;
   }

   // rest of class body
   ...

}
```

Provided that each Comparable class implements equals in this way, any two Comparable objects can be made the same by assigning their comparands the same value.

B.6.4 Boundary Conditions of a Given System

A good understanding of the properties and the “feel” of the environment at hand is important. Are there standard ways to establish and determine sameness? For example, in C++ sameness is usually determined via the

[8] Note that the getNewComparand() method must be synchronized in the presence of multi-threading.

operator==, so it is advisable to redefine it accordingly, whereas in Java the == operator cannot be redefined, but instead the equals method has to be overridden and used.

What kinds of comparison and guarantees of uniqueness are provided or required by the programming language and the libraries and frameworks to be used?

For example, libraries for collections usually expect comparison operations to behave well, as in the case of Java's Collection Framework that requires the standard hashCode method to return the same result for two objects that are equal in terms of the standard equals method. In fact, the comparand should be used as a hash code value for this reason, as shown in the following code fragment.[9]

```
public class MyClass {
   // comparand and equals() as above
   ...

   public int hashCode() {
      return (int)this.comparand;
   }
}
```

B.6.5 Reuse of an Existing Attribute

There are cases where there is no need to define and create comparands specifically. For example, in frameworks that map objects to table entries in relational database systems, primary keys are good candidates for comparands, especially when they are created by some kind of sequence number generator inside the database system. However, care must be taken to ensure that the preexisting attribute exactly reflects the intended comparison semantics. There are deceptive cases where an attribute "accidentally" reflects the intended semantics without being conceptually bound to it, in which case it is better to introduce a dedicated attribute.

B.6.6 Execution of Comparison Operations

There are two options in this dimension of variance. On the one hand, the objects that hold the comparands can offer methods to carry out the comparison, hiding the fact that the COMPARAND pattern is used for this purpose (*internal comparison*). This allows one to change the implementation later on and base it on a technique other than the COMPARAND pattern as required. The implementation can be scaled down to even a comparison

[9] See the JDK documentation on hashCode() in java.lang.Object for further details [77].

of plain references as offered by the programming language when the reasons for an advanced solution have vanished.[10]

On the other hand, objects can allow one to access the comparands and perform the comparison directly (*external comparison*). This variant may be opted for when comparands offer additional functionality and there is therefore already a need to access them. For example, comparands can also serve as keys for later retrievals of the same object. In this case, comparison of two objects looks like follows. (Note that casts to the Comparable interface are not always necessary.)

```
if ((obj1 instanceof Comparable) &&
     (obj2 instanceof Comparable)) {
   Comparable comp1 = (Comparable)obj1;
   Comparable comp2 = (Comparable)obj2;
   if (comp1.getComparand() == comp2.getComparand()) {
      ....
   }
}
```

The two options are not mutually exclusive: an object can offer both internal and external comparison. However, in this case, the specific advantage that internal comparison hides the implementation details of the COMPARAND pattern vanishes, and therefore, pure internal comparison is a better alternative in the general case.

B.6.7 Comparands in Distributed Environments

Especially in the case of distributed systems, the COMPARAND pattern can significantly reduce the runtime overhead of comparison operations. When the comparand of a remote object is cached within each of its remote references[11], comparisons do not require any remote execution at all (see fig. B.3). Instead of allowing various remote references for the same remote object to coexist, a system can choose to unify remote references as soon as they enter an address space. Since this guarantees the uniqueness of remote references they can directly be compared as such.

However, in order to check if an old reference must be reused or a new one must be created, the system has to keep a table that maps comparands, which are determined via the underlying communication mechanism, to the actual remote references.[12]

[10] Other details of the specific implementation are also encapsulated and therefore easily exchanged, like the issues of primitive types vs. compound types, and so on. See *Comparands in Distributed Environments* for further details on compound comparands.

[11] thus making it a simple instance of the CACHE PROXY Pattern [71]

[12] Note that comparands should always be implemented with value semantics rather than reference semantics, since only values can be copied across machine boundaries.

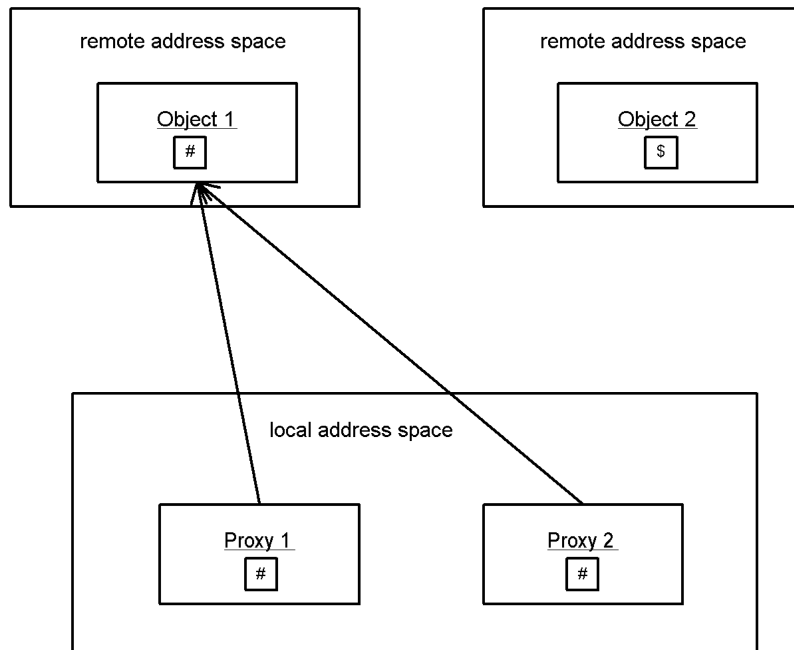


Figure B.3: The sameness of a remote object can be determined locally by comparing the comparands.

In distributed systems, the goal of unique comparands can be achieved only at great expense. Uniqueness can be complicated even further when a heterogeneous application has to be built which consists of independently developed subsystems. The following variants of the COMPARAND pattern offer different solutions for this problem.

Ambiguous Comparands Instead of trying to achieve the goal of globally unique comparands, the requirements can be relaxed by letting all participating subsystems independently create potentially overlapping sets of comparands.

In this case, two objects might have equal comparands by accident. Therefore, one needs to know whether these comparands stem from the same subsystem in order to definitely determine sameness. As a last resort, the comparison operation has to be executed remotely. However, two objects that have different comparands are guaranteed to be different. The aim of avoiding remote invocations is not fully achieved, but the looser coupling of the systems involved outweighs this loss of performance, depending on the frequency of comparison operations.

Compound Comparands Instead of sacrificing uniqueness of comparands, compound comparands can store identifiers for the process in which the respective objects live. Comparand creation is then a process that involves several steps, such as the creation of a unique number within a server and incorporating a server identifier into comparands within clients.

The basic implementation scheme for compound comparands is as follows.

```
public class Comparand {

   protected java.net.URL remoteSystem;
   protected int processNo; // identifies an address space
   protected long remoteComparand;

   public Comparand(java.net.URL remoteSystem,
                    int processNo,
                    long comparand) {
      this.remoteSystem = remoteSystem;
      this.processNo = processNo;
      this.remoteComparand = comparand;
   }

   public boolean equals(Object obj) {
      if (obj instanceof Comparand) {
         Comparand that = (Comparand)obj;
         return this.remoteSystem.equals(that.remoteSystem) &&
                (this.processNo == that.processNo) &&
                (this.remoteComparand == that.remoteComparand);
      }
      return false;
   }

   public int hashCode() {
      return (int)remoteComparand;
   }

   // note: no redefinition of clone()!

}
```

A class for remote references that uses compound comparands looks as follows.

```
public class MyRemoteReference {

   protected Comparand comparand;

   public MyRemoteReference(java.net.URL host,
                            int processNo,
                            long comparand) {
      this.comparand =
         new Comparand(host, processNo, comparand);
   }

   public boolean equals(Object obj) {
      if (obj instanceof MyRemoteReference) {
         MyRemoteReference that = (MyRemoteReference)obj;
         return this.comparand.equals(that.comparand);
      }
      return false;
   }

   public int hashCode() {
      return this.comparand.hashCode();
   }

   // note: no redefinition of clone()!

}
```

Note that there are some fundamental differences between this implementation and the example that is given for non-distributed applications earlier in this paper. Firstly, remote references do not request the creation of a totally new comparand but let a comparand be initialized with given values that identify an existing remote object. This information must be determined via the underlying communication mechanism (for example IP). Secondly, the clone() method is not redefined since a clone of a remote reference refers to the same remote object by definition.

Computed Comparands Comparands may be computed by an algorithm that takes considerable effort to ensure global uniqueness. For example, GUIDs in the Microsoft Component Object Model (COM) can be used as 128 bit comparands. Again, counters that are global for the current machine are taken into account, together with the local machine's network address and the current time in order to ensure (world-wide) global uniqueness [10]. Since GUIDs store all this information in a standardized way, they may still be regarded as a special case of compound comparands. However,

since GUID creation imposes a significant runtime overhead, in the general case ambiguous comparands and compound comparands are preferable.

Coordinated Comparands Another viable alternative for ensuring unique comparands is the assignment of non-overlapping sets of comparands to each node of a distributed application. Then each node is responsible for providing objects with unique comparands from the range of permitted comparands. This implies the need for a central comparand server that coordinates the creation of these non-overlapping sets and their assignment to the respective nodes. A possible disadvantage of this approach is the dependency on the availability of the comparand server. On the other hand, the access rate can be scaled by the number of comparands that are granted on each request.[13]

B.7 Consequences

Using COMPARANDS to compare objects yields the following benefits.

Flexibility The use of comparands makes it easy to define sameness of objects in an arbitrary way. It is even possible to change sameness at run time without affecting the objects state. In addition, it is possible to make objects of different types equal, for example different decorators wrapping the same object.

Comparison is cheap Comparison using primitive comparands is about as cheap as possible in terms of performance overhead.

Comparison of remote objects Proxies of remote objects can cache comparands locally, allowing remote references to be compared without the need for network traffic. This provides an efficient way to implement unification of remote references.

There are however several liabilities.

Complexity As is often the case, flexibility comes at the cost of increased complexity. The use of comparands makes it more difficult to understand which objects are the same by looking at their implementation. The code that determines equality of objects can be part of objects other than those being compared, scattering the definition of sameness across several classes.

[13] There is no completely satisfactory solution to this problem because of the inherent unreliability of distributed applications. For example, see [25].

Collections If the default comparison mechanism of the language (equals in Java, operator== in C++) is implemented with comparands, care must be taken when container classes are used. Many container implementations rely on comparison of the contained objects, and if several objects have the same comparands, unexpected behavior can result.

Memory overhead The COMPARAND pattern relies on the introduction of an additional attribute, incurring some memory overhead. This can be an issue if the number of objects is large or compound comparands are used.

B.8 Known Uses

B.8.1 Java Platform Debugger Architecture

The JPDA [38] does not only include a set of specifications, as introduced above, but also a standard implementation of all key components. The implementation of the Java Debug Interface uses comparands extensively to compare general ("user-defined") objects, strings, arrays, class loaders, and threads as well as reified types, fields and methods.

The comparands are implemented as long integer values (field ref in class com.sun.tools.jdi.ObjectReferenceImpl). In principle, the implementation allows a debugger to connect to more than one virtual machine at the same time. For this reason, objects that have the same comparands are not necessarily the same. Therefore, a representation of the originating virtual machine is also taken into account during comparison. Consequently, the comparands can be created independently by their respective hosts.

Although the Java Debug Interface offers methods to retrieve the comparands of remote references, these comparands cannot be used to carry out comparison operations because of their ambiguity. Therefore, dedicated comparison methods are offered in addition.

See [36] for the source code of JDK 1.3, which also includes the sources of the standard implementation of the Java Platform Debugger Architecture.

B.8.2 Remote Method Invocation

In Java RMI [39], remote objects are represented by objects that implement the java.rmi.server.RemoteRef interface. In the standard implementation of RMI (as of JDK 1.3), this interface is implemented by the sun.rmi.server.UnicastRef class. This class uses the COMPARAND pattern to compare remote objects by comparing the field ref of type sun.rmi.transport.LiveRef, that is defined for this class. This field consists of a representation of a server ("Endpoint"), a unique address space within that server ("UID") and a unique long integer value corresponding to an object within that address space.

This representation of remote objects allows each server to create their own respective sets of values representing actual objects. Since remote references always record the execution context of their remote objects, they are the same if and only if they have the same comparands.

The java.rmi.server.RemoteRef interface does not allow the retrieval of the comparands of remote references, but it completely hides the fact that comparands are used in the standard implementation. Instead, a remoteEquals method is offered to carry out the comparison operation.

Again, see [36] for the source code of JDK 1.3, which also includes the sources of the standard implementation of RMI.

B.8.3 CORBA Relationship Service

In principle, CORBA does not provide any means to compare components. However, the Relationship Service Specification [62] defines the CosObjectIdentity module which includes an IdentifiableObject interface. It defines a long integer attribute as a comparand ("ObjectIdentifier").

Since this value is not guaranteed to be unique, two objects that have the same comparands are not necessarily the same. In order to definitively determine if two component references refer to the same component, an is_identical operation is also defined that has to be carried out remotely.

The "ObjectIdentifier"-comparands are explicitly meant to be used as keys in hash tables. Therefore they can be accessed directly as readonly attributes.

B.8.4 Enterprise JavaBeans

In Enterprise Java Beans [76], the so-called entity beans offer primary keys which can be obtained by getPrimaryKey methods. For example, they can be used to retrieve or remove the components they represent and they can also be used as comparands.

Again, comparison of such primary keys does not completely determine whether two references refer to the same component. If they are equal, it must be determined whether they are obtained from the same execution context (the so-called "home") or otherwise an isIdentical method has to be invoked remotely.

Whereas primary keys are technically realized as instances of possibly user-defined primary key classes, these classes are restricted to be legal Value Types in RMI-IIOP [61]. These Value Types are constrained in a way that essentially leads to classes with value semantics rather than reference semantics. For example, they are required to redefine Java's standard equals method accordingly.

B.8.5 Ginko

Ginko [63] is an email client for the Apple Macintosh (including Mac OS X), and is implemented in Objective-C. One of its features is the unified handling of different copies of the same email. Emails are represented as objects and can be stored into more than one folder whilst keeping the same set of attributes, such as status information and priority markers. The repeated receipt of the same email is also detected by Ginko.

Different instances of the same email are identified by comparison of the standard MESSAGE-ID, as specified by the Internet Request For Comments document number 822 [23]. As RFC 822 states, the "uniqueness of the message identifier is guaranteed by the host which generates it". Therefore in Ginko, these MESSAGE-IDs are used as comparands and they are equal if and only if the corresponding emails are the same.

B.8.6 Related Patterns

Several of the standard patterns from [31] employ some kind of delegation to let methods of one object operate on behalf of another. If a multitude of objects delegate to a single object, implementations of these patterns can apply the Comparand pattern instead of delegating requests for comparison to the respective target objects. The patterns that can take advantage of the Comparand pattern in this way are Adapter, Bridge, Decorator and Proxy.[14]

The OID pattern from [11], which can be regarded as a special case of the Comparand pattern, is restricted to the context of integrating objects and relational database systems. It discusses only primitive types (integer or strings) as candidates for comparands, and favors the use of sequence number generators, which are built into some relational database systems, as sources for comparand creation.

B.9 Conclusion

There are several techniques for implementing object comparison, depending on the desired semantics and the context of its use, with reference comparison being built into almost all programming languages and therefore being most widely employed. An interesting distinction between the Comparand pattern and reference comparison is the following asymmetry. With the Comparand pattern, two objects are guaranteed to be the same if their

[14]Other patterns from [31] that also use delegation are State and Strategy. However, they are not candidates for the application of the Comparand pattern, since in these cases, the respective target objects do not play an "identifying" role, so it makes no sense to compare them at all.

comparands are equal; with reference comparison, two objects are guaranteed to be equal if their references are the same. The latter case is often utilized to optimize otherwise complex comparison operations.

We believe that these considerations could be extended into a useful pattern language covering the realm of object comparison. Other sources that should be taken into account are [5] and [35], which discuss various aspects of object comparison, and the Extrinsic Properties of [30], which can also be used as a means to determine object equality, to name just a few.

B.10 Acknowledgements

The authors thank James Noble for shepherding this paper; the other members of this paper's Writers' Workshop at EuroPLoP 2001 – Fernando Lyardet, Juha Pärssinen, Gustavo Rossi, Dietmar Schütz and Sherif Yacoub; and furthermore, Tom Arbuckle, Michael Austermann, Peter Grogono, Axel Katerbau, Günter Kniesel, Thomas Kühne, Markus Lauer, Oliver Stiemerling, Clemens Szyperski, Dirk Theisen and Kris De Volder for many fruitful discussions on earlier drafts and related publications which led to substantial improvements.

Pascal Costanza's contribution to this work has been carried out for the TAILOR project at the Institute of Computer Science III of the University of Bonn. The TAILOR project is directed by Armin B. Cremers and supported by Deutsche Forschungsgemeinschaft (DFG) under grant CR 65/13.

Appendix C

Curriculum Vitae

Name:	Pascal Costanza
Address:	Kölnstraße 57 53111 Bonn Germany
Telephone:	+49/228/9653601
Email:	costanza@cs.uni-bonn.de
Web:	http://www.cs.uni-bonn.de/∼costanza/

Educational Background

University of Bonn 1998 Master's degree (Diplom) in Computer Science

Städtisches Gymnasium Troisdorf-Sieglar 1990 Qualification for university entrance (Abitur)

Past Experience

1/8/1999 - 31/3/2004	University of Bonn, Institute of Computer Science III, TAILOR Project, Bonn, Germany
15/2/1999 - 31/7/1999	University of Bonn, Institute of Computer Science III, teaching and preparation of TAILOR Project, Bonn, Germany
1/2/1999 - 1/4/1999	DLR, German Aerospace Center, Department: Microgravity User Support Center (MUSC), Project "Medizin Telematik Zentrale" (Medicine Telematics Center), Cologne, Germany
16/9/1998 - 31/12/1998	University of Bonn, Institute of Computer Science III, POLITeam Project, Bonn, Germany
1/4/1998 - 31/7/1998	GiKOM Pro, programming / hospital information system, Bonn, Germany
16/1/1998 - 15/9/1998	University of Bonn, Institute of Computer Science III, teaching and project acquisition, Bonn, Germany
1/9/1995 - 31/12/1997	M & T Online Service GbR, programming / project lead, Troisdorf, Germany
1/4/1995 - 31/8/1995	comet GbR, programming / project lead, Troisdorf, Germany
1/7/1993 - 31/3/1995	comet GbR, co-owner, Troisdorf, Germany
1992 - 1993	Heynmöller Informatik GmbH, programming, Bonn, Germany
1989 - 1992	infill Computer GmbH, programming, Troisdorf, Germany

Notable Experience

Founder: comet – Costanza, Merklinghaus und Trapp GbR, 1993[1]

Industrial experience: Was the project lead for several websites, including sites for several Nestlé brands and for Hagemann Verlag, a major German publisher for educational books for schools. Was the project lead for a web shopping system written in Java.

Project work: Primary research assistant for the TAILOR Project, funded by Deutsche Forschungsgemeinschaft (DFG) and directed by Prof. Dr. Armin B. Cremers, 1999 - 2004. This project has been very successful

[1] Left in 1995. Later on, comet became M & T Online Service – Merklinghaus und Trapp GbR, and subsequently evolved to the now-current tro:net GmbH and tro:media GmbH. Pascal Costanza continued to work for the company until he has obtained his master's degree.

and has given rise to two journal special issues, four journal publications, three PhD theses, ten diploma theses, six conference papers, twelve workshop papers, four workshop reports, three posters at conferences, and one book chapter.

Teaching: Contributed to and held university courses; given advanced training to industrial programmers; supervised master theses

Major Accomplishments

Co-designed and implemented the first compiler for Lava, an integration of delegation into a strongly-typed class-based programming language. Contrary to object-oriented programming languages that base inheritance mechanism on a class hierarchy, there are so-called prototype-based languages that implement inheritance exclusively on objects, via delegation. Such languages do not only allow sound overriding of methods of a "parent" object in a "child" object, but parent objects can also be dynamically replaced resulting in a dynamic change of an inheritance hierarchy. This enables important kinds of Unanticipated Software Evolution. The design and implementation of the Lava compiler was a major part of Pascal Costanza's master thesis that was supervised by, and based on an original model by Günter Kniesel. Pascal Costanza's major results were a seamless integration into the Java programming language, and the detection and fix of a hole in the static type system of the original design. Subsequently, Lava has been reimplemented several times using different implementation strategies, but the original design has largely remained the same which is the first working integration of class-based and prototype-based features into a strongly-typed language. Lava is mainly used in academia.

Originated the ClassFilters package, a framework for load-time transformation of Java classes. This has been rewritten as JMangler under his supervision (joint work with Günter Kniesel) which is now a widely used framework, both in academia and industry. The major results of that work are two-fold. On the technical level, JMangler is the first load-time transformation framework that is neither implemented as a custom Java Virtual Machine implementation nor as a custom Java class loader. This ensures applicability in a much wider range of scenarios than other approaches. On the conceptual level, JMangler is the first and currently only framework that allows transformations of class files to be expressed as independently extensible transformer components. This means that the developer of a transformer component does not need to be aware of the complete transformation process and/or other transformer components in order to express desired features to be added to a set of Java classes. As a consequence, JMangler is a powerful and convenient platform-independent framework for the introduction of systematic properties into a Java program. This has

been effectively taken advantage of in several scenarios, for example as a basis for an alternative implementation of Lava that enables delegation to pure Java classes, or as a basis for an efficient customized code-coverage tool that has been used in large industrial projects.

Other Noteworthy Accomplishments

Introduced Internet technologies (WWW, http, HTML) to his company in 1994, making it one of the first companies in Germany to develop commercial websites.

Introduced the Java programming language to his company in 1996, making it one of the first German companies to develop software in Java.

Was accepted for the OOPSLA Doctoral Symposium in 2001.

Helped forming the German AOSD community by co-organizing workshops. Adapted and introduced the Writers' Workshop format to that community. German AOSD workshops have sticked to that format since then.

Established the notion of Unanticipated Software Evolution (jointly with Günter Kniesel) by organizing workshops and a special issue of the Journal of Software Maintenance and Evolution, published by John Wiley & Sons. Other researchers have started to adopt this term.

Has written a guide to the Common Lisp programming language and published it at his website. This has been widely received as excellent introductory material and is linked prominently from many sites, including as recommended links by both commercial and open-source vendors of Common Lisp implementations. As of February 2004, a Korean translation of this guide has been published on the web.

Bibliography

[1] J. Alves-Foss (ed.). *Formal syntax and semantics of Java.* Springer LNCS 1523, 1999.

[2] K. Arnold and J. Gosling. *The Java Programming Language, Second Edition.* Addison-Wesley, 1998.

[3] Ashes project homepage. http://www.sable.mcgill.ca/ashes/.

[4] M. Atkinson, F. Bancilhon, D. DeWitt, K. R. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In: *Proceedings of the 1st Internation Conference on Deductive and Object-Oriented Databases 1989.* Elsevier, 1990.

[5] H. Baker. Equal Rights for Functional Objects or, The More Things Change The More They Are The Same. ACM OOPS Messenger 4, 4, October 1993.

[6] D. Bardou and C. Dony. Split Objects: a Disciplined Use of Delegation within Objects. *OOPSLA '96*, Proceedings, 1996.

[7] D. Bardou. Roles, Subjects and Aspects: How do they relate?. Aspect Oriented Programming Workshop, ECOOP '98, Brussels, Belgium, July, 1998.

[8] C. Beeri. Some Thoughts on the Future Evolution of Object-Oriented Database Concepts. In: *Datenbanksysteme in Büro, Technik und Wissenschaft.* GI-Fachtagung, Braunschweig 3.-5. März 1993. Springer (Informatik Aktuell), 1993.

[9] G. Booch. *Object Oriented Design with Applications.* Addison-Wesley, 1991.

[10] D. Box. *Essential COM.* Addison-Wesley, 1998.

[11] K. Brown and B. Whitenack. Crossing Chasms: A Pattern-Language for Object-RDBMS Integration. In: J. Vlissides, J. Coplien, N. Kerth (eds.), *Pattern Languages of Programs 2.* Addison-Wesley, 1996.

[12] M. Büchi and W. Weck. Generic Wrappers. In: *14th European Conference on Object-Oriented Programming* (ECOOP 2000). Proceedings, Springer.

[13] A. Cockburn. *Agile Software Development.* Addison-Wesley, 2001.

[14] W. R. Cook, W. C. Hill, and P. S. Canning. Inheritance is not subtyping. In: *17th Annual ACM Symposium on Principles of Programming Languages* (POPL '90). Proceedings, ACM Press.

[15] P. Costanza. Dynamic Object Replacement and Implementation-Only Classes. 6th International Workshop on Component-Oriented Programming (WCOP 2001) at ECOOP 2001, Budapest, Hungary.

[16] P. Costanza. The Programming Language Gilgul. In: Net.ObjectDays Tagungsband, Erfurt, 10.-13. September 2001 (Young Researchers Workshop on Generative and Component-Based Software Engineering).

[17] P. Costanza Transmigration of Object Identity: The Programming Language Gilgul. Doctorial Symposium at OOPSLA 2001, Tampa Bay, Florida, USA.

[18] P. Costanza. Dynamic Replacement of Active Objects in the GILGUL programming language. *First International IFIP/ACM Working Conference on Component Deployment*, Berlin, Germany, June 20 - 21, 2002. Proceedings, Springer LNCS 2370.

[19] P. Costanza and A. Haase. The Comparand Pattern. *EuroPLoP 2001*, Irsee, Germany. Proceedings, UVK Universitätsverlag Konstanz GmbH.

[20] P. Costanza, G. Kniesel, Armin B. Cremers. Lava – Delegation in Java. In: Clemens H. Cap (ed.), *Java-Informations-Tage 1999 (JIT '99).* Springer (Informatik Aktuell), 1999.

[21] P. Costanza, O. Stiemerling, and A. B. Cremers. Object Identity and Dynamic Recomposition of Components. in: *TOOLS Europe 2001.* Proceedings, IEEE Computer Society Press.

[22] A. B. Cremers and T. N. Hibbard. Formal Modeling of Virtual Machines. IEEE Transactions on Software Engineering, 4(5):426-436, 1978.

[23] D. H. Crocker. Standard for the Format of ARPA Internet Text Messages. Internet Request for Comments (RFC) 822, 1982. http://www.ietf.org/rfc/rfc0822.txt

[24] J. Culler. *Dekonstruktion – Derrida und die poststrukturalistische Literaturtheorie.* Rowohlt, 1988, 1999. (German transation of: *On Deconstruction. Theory and Criticism after Structuralism.* Cornell University, Ithaca, New York, 1982.)

[25] P. Deutsch. The Eight Fallacies of Distributed Computing. http://today.java.net/jag/Fallacies.html

[26] M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE) at OOPSLA 2001, Tampa, Florida, USA.

[27] Osvaldo Pinali Doederlein. The Java Performance Report. http://www.geocities.com/ResearchTriangle/Node/2005/jpr/

[28] S. Drossopoulou, F. Damiani, M. Dezani, and P. Giannini. Fickle: Dynamic Object Reclassification. In: *15th European Conference on Object-Oriented Programming* (ECOOP 2001). Proceedings, Springer.

[29] S. Drossopoulou and S. Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In [1], 1999.

[30] M. Fowler. Dealing with Properties. http://www.martinfowler.com, 1997.

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[32] The Gilgul homepage. http://javalab.cs.uni-bonn.de/research/gilgul/

[33] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[34] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition.* Addison-Wesley, 2000.

[35] P. Grogono and M. Sakkinen. Copying and Comparing: Problems and Solutions. In: *ECOOP 2000.* Proceedings, Springer.

[36] Java 2 Platform, Standard Edition (J2SE). Sun Community Source Licensing. http://www.sun.com/software/communitysource/java2/

[37] Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/

[38] Java Platform Debugger Architecture. http://java.sun.com/products/jpda/

[39] Java Remote Method Invocation.
http://java.sun.com/products/jdk/rmi/

[40] E. Jul, R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, and N. C. Hutchinson. Emerald: A General-Purpose Programming Language. In: *Software – Practice and Experience*, 1991, 21(1):91-118.

[41] The Kaffe homepage. http://www.kaffe.org/

[42] W. Kent. A Rigorous Model of Object References, Identity and Existence. In: *Journal of Object-Oriented Programming*, 4(3):28-36, June 1991.

[43] S. N. Khoshafian and G. P. Copeland. Object Identity. In: *OOPSLA '86*. Proceedings, ACM Press.

[44] G. Kiczales, J. des Rivières, D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[45] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold. An Overview of AspectJ. In: *ECOOP 2001*. Proceedings, Springer LNCS 2072.

[46] G. Kniesel. *Objects Don't Migrate – Perspectives on Objects with Roles*. Technical Report IAI-TR-96-11, Institute of Computer Science III, University of Bonn, April 1996.

[47] G. Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. In: *ECOOP '99*. Proceedings, Springer.

[48] G. Kniesel. *Dynamic Object-Based Inheritance with Subtyping*. Ph.d. thesis, University of Bonn, 2000.

[49] G. Kniesel and D. Theisen. JAC – Access Right Based Encapsulation in Java. In: *Software – Practice and Experience*, 2001, 31(6):555-576.

[50] D. Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 1999.

[51] D. Lea. Draft Java Coding Standard, February, 2000.
http://gee.cs.oswego.edu/dl/html/javaCodingStd.html

[52] I. Lee. *DYMOS: A Dynamic Modification System*. Dissertation, University of Wisconsin-Madison, USA, 1983.

[53] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In: *Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '86). Proceedings, ACM Press.

[54] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[55] B. Liskov. Data Abstraction and Hierarchy. In: ACM SIGPLAN Notices, 23, 5, May 1988.

[56] B. J. MacLennan. Values and Objects in Programming Languages. SIGPLAN Notices, 17(12):70-79, December, 1982.

[57] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[58] S. Müller. *Transmigration von Objektidentitäten – Integration der Spracherweiterung Gilgul in eine Java-Laufzeitumgebung*. University of Bonn, Insitute of Computer Science III, diploma thesis, 2002.

[59] P. Naur. Programming as Theory Building. In: *Computing: A Human Activity*. ACM Press, 1992. (reprinted in [13]).

[60] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*. Springer LNCS 2283, 2002.

[61] Object Management Group, Inc. Java Language to IDL Mapping Specification. June 1999. http://www.omg.org/technology/documents /formal/java_language_mapping_to_omg_idl.htm

[62] Object Management Group, Inc. Relationship Service Specification, Version 1.0. April 2000. http://www.omg.org/technology/documents/formal/relationship_service.htm

[63] Objectpark Group. http://www.objectpark.org

[64] D. von Oheimb and T. Nipkow. Machine-Checking the Java Specification: Proving Type-Safety. In [1], 1999.

[65] A. Ohori. Representing Object Identity in a Pure Functional Language. *Proceedings of the Third International Conference on Database Theory*, Springer LNCS 470, 41-55, December, 1990.

[66] Rabbi Zev T. Paretzky. Reincarnation in Judaism. In Judaism Alive, http://www.judaismalive.org/

[67] M. Petre Programming Paradigms and Programming Culture: Expert Practice and Implications for Teaching. Workshop on the Choice of Programming Languages, British Open University, Milton Keynes, September 1993.

[68] M. L. Powell. Objects, References, Identifiers and Equality – White Paper. OMG TC Document Number 93.7.5, Sun Microsystems, Inc., 1993.

[69] W. Reddig. *Perspektiven: Persistente Objekte mit anwendungsspezifischer Struktur und Funktionalität*. Ph.d. thesis, University of Bonn, 2001.

[70] M. Reiser and N. Wirth. *Programming in Oberon – Steps Beyond Pascal and Modula*. Addison-Wesley, 1992.

[71] H. Rohnert. The Proxy Design Pattern Revisited. In: J. Vlissides, J. Coplien, N. Kerth (eds.), *Pattern Languages of Program Design 2*, Addison-Wesley, 105-188, 1996.

[72] H. Schmidt. *Datenraumbasierte Formulierung der Objektidentität*. University of Bonn, Insitute of Computer Science III, diploma thesis, 1994.

[73] M. Serrano. Wide Classes. In: *13th European Conference on Object-Oriented Programming* (ECOOP '99). Proceedings, Springer.

[74] D. N. Smith. Smalltalk FAQ. http://www.dnsmith.com/SmallFAQ/, 1995.

[75] G. L. Steele and G. J. Sussmann. The Art of the Interpreter or, the Modularity Complex (Parts Zero, One, and Two). MIT AI Lab, AI Lab Memo AIM-453, May 1978.

[76] Sun Microsystems, Inc. Enterprise JavaBeans Specification, Version 2.0. October 2000. http://java.sun.com/products/ejb/docs.html

[77] Sun Microsystems, Inc. Java 2 SDK, Standard Edition Documentation, Version 1.3.1. http://java.sun.com/j2se/1.3/docs/

[78] Sun Microsystems, Inc. Java 2 SDK, Standard Edition Documentation, Version 1.4.2. http://java.sun.com/j2se/1.4/docs/

[79] D. Syme. Proving Java Type Soundness. In [1], 1999.

[80] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[81] C. Szyperski, S. Omohundro, S. Murer. Engineering a Programming Language: The Type and Class System of Sather. Technical Report TR-93-064. The International Computer Science Institute, Berkeley, CA, USA, 1993.

[82] The Tailor Project. http://javalab.cs.uni-bonn.de/research/tailor/

[83] VolanoMark Java Benchmarks.
http://www.volano.com/benchmarks.html

[84] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. OOPS Messenger, 1:7-87, August, 1990.

[85] R. Wieringa and W. de Jonge. Object Identifiers, Keys, and Surrogates – Object Identifiers Revisited. In: *Theory and Practice of Object Systems*, 1(2):101-114, 1995.