PROGRAMMING LANGUAGE ABSTRACTIONS FOR MOBILE CODE

THÈSE N^{O} 4515 (2009)

PRÉSENTÉE LE 23 OCTOBRE 2009

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE MÉTHODES DE PROGRAMMATION 1
SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Stéphane MICHELOUD

acceptée sur proposition du jury:

Prof. A. Wegmann, président du jury
Prof. M. Odersky , directeur de thèse
Prof. M. Merro, rapporteur
Prof. C. Petitpierre, rapporteur
Dr J. H. Spring, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2009

This document was processed using T_EX Live 2007 for Linux
Schemas were created using Dia 0.95 for Linux
Screenshots were captured using ImageMagick 6.2.4 for Linux

to Inês and David

Contents

Abstract **xx**

Acknowledgments **xxiv**

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Scope . . . 8
- 1.3 Background . . . 8
  - 1.3.1 The Scala Language . . . 8
  - 1.3.2 The Java Platform . . . 10
- 1.4 Contributions . . . 11
- 1.5 Outline . . . 12

2 Multi-paradigm Programming **13**
- 2.1 Functional Programming . . . 14
  - 2.1.1 Higher-Order Functions . . . 14
  - 2.1.2 Lexical Closures . . . 26
- 2.2 Distributed Programming . . . 30
  - 2.2.1 Historical Survey . . . 30
  - 2.2.2 Remote Procedure Calls . . . 31
  - 2.2.3 Remote Evaluation . . . 33
  - 2.2.4 OMG CORBA . . . 36
  - 2.2.5 Java RMI . . . 37
  - 2.2.6 .NET Remoting . . . 38
- 2.3 Distributed Programming in Java . . . 40
  - 2.3.1 Distributed Objects . . . 40
  - 2.3.2 Distributed Exception Handling . . . 41
  - 2.3.3 Distributed Garbage Collection . . . 41
- 2.4 Discussion . . . 43

3 Code Mobility **45**
3.1 Forms of Code Mobility 47
3.1.1 Weak Mobility 47
3.1.2 Strong Mobility 47
3.2 Security 48
3.3 Code Mobility in Java 50
3.3.1 Java Applet 50
3.3.2 Dynamic Class Loading 54
3.3.3 Object Serialization 58
3.3.4 Security 59
3.4 Discussion 60

4 State of the Art **61**
4.1 Calculi 62
4.1.1 λ_{marsh} Calculus 62
4.1.2 Calculus of Module Systems 64
4.2 Languages 66
4.2.1 Obliq 66
4.2.2 ML3000 69
4.2.3 MobileML 70
4.2.4 PLAN 71
4.3 Discussion 72

5 Programming Examples **73**
5.1 Marshalling Example 74
5.2 Basic Example using Typed Channels 77
5.3 Basic Example using Remote Actors 85
5.4 Compute Server Example 87
5.5 Bank Account Example 89
5.6 Calendar Example 92
5.7 Discussion 94

6 Programming Abstractions **95**
6.1 Closures as Control Abstraction 96
6.1.1 Lexical Closures 97
6.1.2 Detached Closures 98
6.2 Generalized Data Binding Mechanism 99
6.2.1 Module Objects 99
6.2.2 Import Clauses 100
6.3 Programming Model 102
6.3.1 Detach Calculus 102

6.3.2 Formalization 104
6.4 Discussion 110

7 Implementation **111**
7.1 Closure Conversion 112
7.1.1 Lexical Closures 112
7.1.2 Detached Closures 114
7.2 Extending Scala 144
7.2.1 Scala Compiler 144
7.2.2 Detach Phase 150
7.2.3 Scala Run-time 152
7.2.4 Scala API 152
7.3 Java Platform 154
7.4 Discussion 155

8 Conclusion **159**
8.1 Achievements 160
8.2 Open Issues 161
8.3 Future Work 161

Abbreviations **163**

Glossary **167**

Index **177**

Bibliography **195**

About the Author **199**

Appendix A: Distributed Application Samples **201**

List of Figures

2.1 RPC architecture. 32
2.2 REV architecture. 34
2.3 CORBA architecture. 36
2.4 RMI architecture. 38
2.5 .NET Remoting architecture. 39

3.1 Java applet using RMI. 54
3.2 Class loader architecture. 57

4.1 λ_{marsh} extended language syntax. 63
4.2 λ_{marsh} typing rules. 63
4.3 Value transmission in Obliq. 67
4.4 Closure transmission in Obliq. 68

5.1 Marshalling application (consoles). 76
5.2 Basic C/S application (consoles). 82
5.3 Basic C/S application (consoles, option `info`). 82
5.4 Basic C/S application (server console, option `info,lib`). . . . 83
5.5 Basic C/S application (client console, option `info,lib`). . . . 84

6.1 Core language syntax. 105
6.2 Extended language syntax. 105
6.3 Well-formedness. 106
6.4 Typing rules. 107
6.5 Extended typing rules. 107
6.6 Transformation rule (close). 108
6.7 Transformation rule (detach). 108
6.8 Transformation rule (proxy). 108

7.1 Detached closure and outer class. 118
7.2 Detached closure and free variables. 143
7.3 Scala compiler phases. 146

7.4 Detached Scala closure architecture. 153

List of Tables

2.1 `map` function in functional languages. 15
2.2 `map` function in dynamic languages. 16
2.3 `map` function in object-oriented languages. 16
2.4 `map` function in functional languages. 17
2.5 `map` function in dynamic languages. 18
2.6 `map` function in object-oriented languages. 19
2.7 Closures in functional languages. 27
2.8 Closures in dynamic languages. 28
2.9 Closures in object-oriented languages. 29

List of Listings

1.1 Source location "A". 4
1.2 Target location "B". 4
1.3 Agenda pseudo-code (library). 5
1.4 Agenda pseudo-code (client). 6
1.5 Agenda pseudo-code (server). 6
1.6 Agenda pseudo-code (client converted). 6
2.1 Composition of HOFs in C#. 20
2.2 Composition of HOFs in Python. 21
2.3 Composition of HOFs in Ruby. 21
2.4 Composition of HOFs in Scala. 22
2.5 Composition of HOFs in Scheme. 22
2.6 Abstracting over functional behavior in C#. 23
2.7 Abstracting over functional behavior in Python. 24
2.8 Abstracting over functional behavior in Ruby. 24
2.9 Abstracting over functional behavior in Scala. 25
2.10 Abstracting over functional behavior in Scheme. 25
3.1 Java applet using RMI (service). 51
3.2 Java applet using RMI (server). 51
3.3 Java applet using RMI (client). 52
3.4 Java applet using RMI (HTML page). 53
3.5 Service class reloading (interface). 55
3.6 Service class reloading (server). 55
3.7 Service class reloading (client). 56
5.1 Type-safe marshalling in Scala. 74
5.2 Marshalling application (server). 74
5.3 Marshalling application (client). 75
5.4 The `Channel` class (Scala API). 77
5.5 The `ServerChannel` class (Scala API). 77
5.6 Basic C/S application (server). 78
5.7 Basic C/S application (client). 79
5.8 Basic C/S application (server script). 80

5.9 Basic C/S application (client script). 80
5.10 Basic C/S application (policy file). 81
5.11 C/S application with Scala actors (server). 85
5.12 C/S application with Scala actors (client). 85
5.13 A compute server in Obliq. 87
5.14 Compute application (service). 87
5.15 Compute application (server). 88
5.16 Compute application (client). 88
5.17 Bank account application (Account). 89
5.18 Bank account application (server). 89
5.19 Bank account application (client). 90
5.20 Calendar application (agendas). 92
5.21 Calendar application (server). 92
5.22 Calendar application (client). 93
6.1 Scala closure and free variables. 97
6.2 Import clauses at any scope level. 100
6.3 Import clauses opening any object. 101
6.4 Core language example. 109
6.5 Core language example (transformed). 109
7.1 Scala closure inside a class. 112
7.2 Scala closure inside a class (converted). 113
7.3 Detached Scala closure inside a class. 116
7.4 Detached Scala closure inside a class (1/4). 116
7.5 Detached Scala closure inside a class (2/4). 117
7.6 Detached Scala closure inside a class (3/4). 117
7.7 Detached Scala closure inside a class (4/4). 117
7.8 Scala closure inside an object. 119
7.9 Scala closure inside an object (converted). 119
7.10 Detached closure inside an object. 120
7.11 Detached closure inside an object (1/4). 120
7.12 Detached closure inside an object (2/4). 120
7.13 Detached closure inside an object (3/4). 121
7.14 Detached closure inside an object (4/4). 121
7.15 Scala closure inside an inner class. 122
7.16 Scala closure inside an inner class (converted). 122
7.17 Detached closure inside an inner class. 124
7.18 Detached closure inside an inner class (1/4). 124
7.19 Detached closure inside an inner class (2/4). 125
7.20 Detached closure inside an inner class (3/4). 125
7.21 Detached closure inside an inner class (4/4). 126
7.22 Scala closure inside a member function. 127

7.23 Scala closure inside a member function (converted). 127
7.24 Detached closure inside a member function. 128
7.25 Detached closure inside a member function (1/4). 128
7.26 Detached closure inside a member function (2/4). 129
7.27 Detached closure inside a member function (3/4). 129
7.28 Detached closure inside a member function (4/4). 130
7.29 Scala closure inside a local function. 131
7.30 Scala closure inside a local function (converted). 131
7.31 Detached closure inside a local function. 132
7.32 Detached closure inside a local function (1/4). 133
7.33 Detached closure inside a local function (2/4). 133
7.34 Detached closure inside a local function (3/4). 134
7.35 Detached closure inside a local function (4/4). 134
7.36 Scala closure inside a closure. 135
7.37 Scala closure inside a closure (converted). 135
7.38 Detached closure inside a closure. 136
7.39 Detached closure inside a closure (1/4). 136
7.40 Detached closure inside a closure (2/4). 137
7.41 Detached closure inside a closure (3/4). 137
7.42 Detached closure inside a closure (4/4). 137
7.43 Scala closure referring out of scope objects. 139
7.44 Scala closure referring out of scope objects (converted). . . 139
7.45 Detached closure referring out of scope objects. 140
7.46 Detached closure referring out of scope objects (1/6). . . . 140
7.47 Detached closure referring out of scope objects (2/6). . . . 141
7.48 Detached closure referring out of scope objects (3/6). . . . 141
7.49 Detached closure referring out of scope objects (4/6). . . . 142
7.50 Detached closure referring out of scope objects (5/6). . . . 142
7.51 Detached closure referring out of scope objects (6/6). . . . 142
7.52 Extending Scala with detached closures. 144
7.53 Scala closure inside a class (`Typer`). 148
7.54 Scala closure inside a class (`Uncurry`). 149
7.55 Scala closure inside a class (`ExplicitOuter`). 149
7.56 Scala closure inside a class (`LambdaLift`). 150
7.57 The marker object `detach` (Scala API). 153

Résumé

Scala est un langage de programmation généraliste développé à l'EPFL. Il combine les principaux concepts mis en oeuvre dans les langages orientés-objets et fonctionnels. Scala est un langage statiquement typé; en particulier il possède un système de types évolué et supporte l'inférence de type locale. D'autre part, il s'intègre facilement avec les plateformes Java et .NET: leur bibliothèques sont directement accessibles et le compilateur Scala génère du code pour les deux environnements d'exécution.

Le langage Scala offre plusieurs fonctionnalités qui font de lui un candidat idéal pour la programmation d'applications distribuées. En particulier, il supporte les fonctions de première classe qui sont utiles en relation avec les notions de portée distribuée et de mobilité du code. Dans ce contexte, l'absence de support des types à l'exécution représente un inconvénient important de l'environnement d'exécution Java comme plateforme cible.

Cette thèse se focalise sur la réalisation d'un concept nouveau combinant des notions essentielles de la programmation fonctionnelle et de la programmation distribuée et impliquant l'extension de la notion de portée lexicale au contexte distribué. En quelques mots, nous défendons l'idée selon laquelle la notion d'abstraction lambda permet de traiter de manière élégante la liaison dynamique des références locales dans un environnement d'exécution distribué.

Les principales idées exposées dans ce travail de recherche ont été mises en œuvre dans notre compilateur Scala. Cela nous a permis d'évaluer la qualité des techniques utilisées, en particulier leur impact sur la fiabilité et les performances des programmes distribués.

A ce jour, la plupart des travaux de recherche en relation avec le présent sujet se sont concentrés sur les langages de programmation fonctionnels, en particulier sur la famille des langages ML.

Mots-clés: langage de programmation, programmation distribuée, code mobile, évaluation distante, appel de méthode distante, liaison dynamique, portée distribuée, compilation, Scala.

Abstract

Scala is a general-purpose programming language developed at EPFL. It combines the most important concepts found in object-oriented and functional languages. Scala is a statically typed language; in particular it features an advanced type system and supports local type inference. Futhermore it integrates well with the Java and .NET platforms: their libraries are accessible without glue code and the Scala compiler generates code for both execution environments.

The Scala programming language has several features that make it desirable as a language for distributed application programming. In particular, it supports first-class functions which are useful in relation with the notions of distributed scope and code mobility. In that context, the missing support for run-time types is one important drawback of the Java run-time environment as a target platform.

This thesis focuses on the realisation of a new concept combining essential notions from the functional and distributed programming and implying the extension of the notion of lexical scoping to the distributed context. In short, we claim that the notion of lambda abstraction provides an elegant way for dealing with the dynamic rebinding of local references in a distributed execution environment.

The key ideas exposed in this research work have been implemented in our Scala compiler. This helped us to evaluate the used techniques, in particular their impact on the reliability and the performance of distributed programs.

So far, most research works related to the present subject have focused on functional programming languages, in particular on the ML language family.

Keywords: programming language, distributed programming, mobile code, remote evaluation, remote method invocation, dynamic binding, distributed scope, compilation, Scala.

Remerciements

Je remercie tout d'abord mon directeur de thèse, le Prof. Martin Odersky, pour m'avoir accueilli dans son groupe et m'avoir donné la possibilité de travailler sur ce captivant projet de recherche.

J'exprime en second lieu ma gratitute à mes anciens collègues Philippe Altherr, Michel Schinz et Matthias Zenger qui m'ont aidé à comprendre les méchanismes internes du compilateur Scala, et qui, de manière plus générale, m'ont fait profiter de leur grande expertise du langage de programmation Java. Je remercie également Vincent Cremet, Iulian Dragos et Philipp Haller pour leur aide précieuse durant la réalisation de ce projet.

De nombreuses autres personnes ont activement contribué au projet Scala, — qui a évolué au cours des ans de Scala 1 à Scala 2 —; travailler à leur côté m'a apporté beaucoup de satisfaction. Outre les personnes déjà citées je souhaite remercier ici Gilles Dubochet, Burak Emir, Nikolay Mihaylov, Sean McDirmid, Adriaan Moors, Tiark Rompf, Lukas Rytz, Lex Spoon et Geoffrey Washburn.

J'adresse ensuite mes pensées amicales à trois collègues avec qui j'ai successivement partagé mon bureau pendant ces six dernières années, à savoir Massimo Merro, Erik Stenman et Fabien Salvi; ensemble nous avons passé de bons moments à débattre de sujets liés au travail ou ayant trait avec la vie en générale.

Finalement, je remercie Phil Bagwell, Johannes Borgström, Sébastien Briais, Daniel Bünzli, Danielle Chamberlain, Rachele Fuzzati, Simon Kramer, Maxime Monod, Uwe Nestmann, Christine Röckl et Yvette Dubuis, notre secrétaire à la retraite, pour leurs encouragements sincères dans les bons comme dans les moins bons moments de mon travail de recherche.

Lausanne, avril 2008

Stéphane Micheloud

Acknowledgments

I first thank my supervisor Prof. Martin Odersky for accepting me in his group and for giving me the opportunity to work on this interesting research project.

Secondly, I express my gratitude to my past colleagues Philippe Altherr, Michel Schinz and Matthias Zenger who helped me to understand the internals of the Scala compiler, and more generally shared with me their great expertise of the Java programming language. I also thank Vincent Cremet, Iulian Dragos and Philipp Haller for their helpful inputs during the realization of this project.

Many people actually contributed to the Scala project — which evolved over the years from Scala 1 to Scala 2 — and working with all of them proved very enjoyable. Besides the persons already mentioned, I would therefore like to thank Gilles Dubochet, Burak Emir, Nikolay Mihaylov, Sean McDirmid, Adriaan Moors, Tiark Rompf, Lukas Rytz, Lex Spoon and Geoffrey Washburn.

I also address my friendly thoughts to three colleagues I successively shared my office with over the last six years, namely Massimo Merro, Erik Stenman and Fabien Salvi; together we spent pleasant moments debating both on work-related and general interest topics.

Finally, I thank Phil Bagwell, Johannes Borgström, Sébastien Briais, Daniel Bünzli, Danielle Chamberlain, Rachele Fuzzati, Simon Kramer, Maxime Monod, Uwe Nestmann, Christine Röckl and Yvette Dubuis, our retired secretary, for their friendly encouragements in the good and less good moments of my research work.

Lausanne, April 2008

Stéphane Micheloud

Chapter 1

Introduction

The best way to predict the future is to invent it.

Alan Kay[1]

Languages supporting mobile code differ from other languages or middleware for distributed systems (like CORBA) because they explicitly model the concept of separate computational environments and how code and computations move among these environments.

Some concepts shared with traditional languages acquire new dimensions in the case of languages supporting mobile code (e.g. scope rules and name resolution) and others are typical of those languages (e.g. dynamic binding and security).

Mobility has a strong impact on programming language features:

1. Programs are *location aware* [100]. Knowing where computation happens is necessary in order to write mobile programs effectively. The structure of the underlying computer network is made explicit to the programmer.

2. Mobility is *under program's control*. Code units may be shipped or fetched to/from remote nodes.

Code mobility places new demands on programming languages and run-time systems. We mention here two of them:

[1]Alan Kay was honored with the ACM Turing Award 2003 for pioneering many of the ideas found in OO languages; he also contributed to the development of Smalltalk.

1. *Code manipulation*. The language must provide means to identify and transmit code units and the run-time environment must be able to execute received code units within the existing address space of the recipient.

2. *Type safety*. Type safety[2] is the property that every operation the program performs is executed on values of the appropriate type. It is the responsibility of the compiler to enforce type safety. In order to ensure type safety a high-level language must provide a correct mapping of typing rules between the type checker of the compiler and the verifier tool (e.g. the bytecode verifier in the Java VM) of the run-time environment.

Mobile code languages (MCL)[3] [35, 43] facilitate the transmission of code between remote locations in a distributed system. A piece of code which is generated at one location can be transmitted for execution at another location which may exhibit characteristics different from its place of origin.

1.1 Motivation

In this work we focus our attention to language-enforced disciplines that guarantee abstraction safety — presupposing the existence of reliable communication. However, this work is fully compatible with the usage of cryptographic techniques to enforce the reliability and authenticity of the transmitted code.

The objectives of this work are two-fold: propose a new approach to dynamic binding in the context of mobile code and take advantage of the expressiveness of the Scala language [89] to extend it with programming abstractions for mobile code.

First we perform an analysis to identify the language abstractions important to our model. Concretely, we study and compare the different existing approaches found in the literature [28, 33, 56, 59, 96]. Second we design the new language abstraction and propose a formalization of our programming model. We focus in particular on typical concepts of languages supporting mobile code such as dynamic binding and type safety.

[2]More terms are defined in appendix "Glossary".
[3]More acronyms are listed in appendix "Abbreviations".

Finally we integrate the programming model into the language Scala and implement a few sample applications to validate our conceptual choices. The target platform is the Java VM [72].

Although our approach is targeted at Java, it is not heavily based on its features. Indeed we propose a more general translation that abstracts from the language features; we only assume that the target language provides support for weak mobility — code mobility without migration of execution state — and distributed objects.

We consider first two different approaches found in existing programming languages to address the problem of dynamic reference binding inherent to the concept of mobile code. The main technical issue is to find a meaning for *higher-order distributed computations*.

On one side, the language Obliq [27, 28] uses *network references* to remotely access objects at the source location. Obliq objects have state; they are local to a site and are never automatically copied over the network. Regarding the free variables of network-transmitted functions, Obliq takes the view that such variables are bound to their original locations and network sites, as prescribed by lexical scoping.

On the other side, Java adopts the opposite approach where the applet code is moved and bound to the target location.

In this work we propose a more general approach which retains the two approaches just described and extends the abstraction model with a solution based on the lambda calculus for the dynamic (re-)binding of local references.

A fundamental question about dynamic binding in the context of mobile code lies in the way *object references* are handled at run-time.

In this work we handle references in three different ways:

- Either they are routed back to the source location. In this case we need some *proxy mechanism* based on remote objects.

- Or they are copied together with the transmitted code. We use here *marshalling* based on serializable objects.

- Or they are (re-)established at the target location. We think that this case can be handled by a simpler solution as described in existing works [10, 56]. This observation constitutes the main motivation of this research work.

While the first two cases can be solved using well-known techniques, such as Java RMI [120] for remote objects and Java object serialization [119] for serializable objects, we propose for the last case a solution based on the lambda calculus.

When values or computations are marshalled from a running system and moved elsewhere, either by network communication or via a persistent store, some of their identifiers may need to be *dynamically rebound*. These may be both identifiers of language run-time library functions or identifiers from application libraries which exist in the new context.

There are several reasons motivating this approach. In particular, Scala supports closures, meaning that the facilities for remote invocation and distributed scope pioneered by Obliq [28] can be easily added without any change to the underlying language semantics.

Example

Consider the following scenario which is quite common in distributed environments: Several network locations are hosting some agenda information, e.g. agenda entries about people working in the same company. In order to arrange a project meeting we want to check the availability of all concerned persons.

Principle From a functional perspective it looks quite natural to consider that mobile code is just a function that we parameterize with the variables to be locally bound at the target location.

We illustrate that principle with two small pieces of pseudo-code:

```
"send_to_B"(
  a: Agenda => { ... a.doSomething() ...} )
val e = "receive_from_B"
```

Listing 1.1: Source location "A".

```
val a = new Agenda(...)
val f = "receive_from_A"
"send_to_A"(f(a))
```

Listing 1.2: Target location "B".

The client process at location "A" (Listing 1.1) sends a closure containing the parameterized code — in this example with parameter `a` — to some location "B" and waits for some result. The server process at location "B" (Listing 1.2) executes the transmitted code using the locally bound variable `a` and sends the result value back to the client. In the above

pseudo-code we rely on the two operations `send_to` and `receive_from` to deal with the type-safe transmission of higher-order functions.

Why has this not been done before? Note that the closure could also access other names from its environment that need to be rebound. For instance, it could use a `Math.max` function, in that case the reference to the `Math` module needs to be rebound. Such rebindings are possible in Scala, unlike in other programming languages, because:

- Modules are objects in Scala [88] and can thus appear as free variables in the transmitted code.
- The import clause in Scala [89] may also appear in a block, which is for example not possible in Java.

Code sample Let us illustrate the basic idea of our solution with a small example[4] based on the above scenario. We limit our example to processes running on a single machine and we assume the availability of typed channels to establish a type-safe communication between them. The source code excerpts presented below are coded in Scala.

In Listing 1.3 we define a minimal framework with a class `Agenda` managing a list of agenda entries and some library functions like `freeSlots` to operate on `Agenda` objects.

```
object agendas {
  type Day = ...
  type Entry = ...
  class Agenda(e: Entry*) {
    def get(day: Day): List[Entry] = ...
    ... (more methods)
  }
  def freeSlots(a1: Agenda, a2: Agenda, d: Day): List[Entry] = ...
  ... (more methods)
}
```

Listing 1.3: Agenda pseudo-code (library).

In Listing 1.4 the client process sends a closure marked with `detach` to the target location specified by the communication channel `chan`. The `detach` primitive is used to "detach" the closure from its lexical environment. In the closure body we consider the agendas of two people and look for free time slots available on the week day `day`. The code itself

[4]The same example will be discussed in more details in Section 5.6.

accesses three non-local variables: the application-specific object `agendas`, the non-local object `agendaBob` and the local variable `day`.

```
...
val agendaBob = new Agenda(..)
def client {
  val server = new Channel(host, port)
  var day = "Mon"
  server ! detach(
    (a: Agenda) => agendas.freeSlots(a, agendaBob, day)
  )
  val e = server.receive[List[Entry]]
  println("e = "+e)
}
```

Listing 1.4: Agenda pseudo-code (client).

The server process in Listing 1.5 simply waits for incoming requests, executes the received closure code with its actual parameter `agendaTom` and, finally, sends the evaluation result back to the client.

```
...
val agendaTom = new Agenda(..)
def server {
  val server = new ServerChannel(port)
  loop {
    val client = server.accept
    val f = client.receive[Agenda => List[Entry]]
    client ! f(agendaTom)
  }
  server.close()
}
```

Listing 1.5: Agenda pseudo-code (server).

Code Transformations For supporting the new programming abstraction we intend to perform code transformations only on source code containing occurences of the primitive `detach`. In Listing 1.6 we sketch how the transformed program code for the client may look like:

```
class Env(x1: Proxy1, x2: Proxy2, x3: RemoteObjectRef)
extends ... with Function[Agenda, List[Entry]] {
  def apply(a: Agenda): List[Entry] =
    x2.freeSlots(a, x1.agendaBob, (Day) x3.elem)
}
val agendaBob = new Agenda(...)
def client {
  val server = new Channel(host, port)
  val day = new ObjectRef("Mon")
  server.!({
    val x1 = bind(name1, new Proxy1Impl(this))
    val x2 = bind(name2, new Proxy2Impl(agendas))
    val x3 = bind(name3, new RemoteObjectRefImpl(day))
    new Env(x1, x2, x3)
  });
  val e = server.receive[List[Entry]]
  println("e = " + e)
}
```

Listing 1.6: Agenda pseudo-code (client converted).

Several transformations are actually applied to the original code:

- The synthetic class `Env` allows us to capture the free variables (in this case variables `agendas`, `agendaBob` and `day`) and to turn the function into an object;
- The type of the captured variables `agendas`, `agendaBob` and `day` is changed to remote reference types (declarations of traits `Proxy1`, `Proxy2` and `RemoteObjectRef` are omitted here for brevity);
- A method `apply` with parameter `a` is added to the class `Env` to allow the deferred evaluation of the closure body.

Implementing the above example in a distributed environment requires additional handling of object references. In particular we need to:

- Use remote instead of local references to access free variables.
- Provide some remote interface to classes containing functions which are called in the mobile code executed remotely at the target location.
- Rely on some global service to permit type-safe communication between separate computational environments (i.e. address spaces [30]).

1.2 Scope

The aim of this work is neither to propose a new process calculus nor to design a fully-fledged mobile code language (MCL).

We further recognize that considering all aspects of a distributed system is not realistic; we thus limit our study to some dynamic aspects of relocated code — such as dynamic binding and type safety — and deliberately keep aside considerations related to inter-process communication — such as concurrency and security —. The Scala Actors library and the Java security library already provide good support in those domains.

A practical goal of this work is to extend the Scala programming language with programming facilities for mobile code without compromising type safety. By programming facilities we mean either language or library extensions.

1.3 Background

1.3.1 The Scala Language

Scala [89, 90] is a general-purpose programming language developed by Prof. Martin Odersky and his team at EPFL since 2002. Scala incorporates several advanced concepts from recent research and has been used successfully in the development of large-scale programs.

Scala is a *multi-paradigm language* which combines functional and object-oriented elements [22, 101] and thus supports both programming styles. From the functional world, it takes the concepts of higher-order functions, algebraic data types and pattern matching; from the object-oriented world, it takes the concepts of classes, objects and virtual types.

Scala is a *statically typed language* featuring an advanced type system and supporting local type inference [95]. In particular, Scala is equipped with a rich set of language constructs providing powerful type abstractions:

Type parametrization Classes and methods support type parametrization (or parametric polymorphism) and the specified type parameters can have both lower and upper bounds, which can refer to the parameter itself (so-called F-bounded polymorphism).

Higher-kinded types *Type constructor polymorphism* [82, 83] — also known as higher-kinded types — allows to abstract over generic types; a type constructor is like a function on the level of types that takes

type arguments to yield a type. The Scala collection library rely heavily on type constructor polymorphism[5].

Type members A class can include type members as well as value members; like value members, type members can be either abstract or concrete. Abstract type members [34] — also known as *virtual types* — are bounded by a type, and this bound can be refined by subclasses.

Compound types A value can be declared to have a compound type consisting of a set of class types and an optional type refinement. When present the refinement constrains the type of the class members by specifying more precise types. A compound type thus expresses the fact that a value has simultaneously all the component types.

Structural types A value can be declared to have a structural type consisting of a (usually implicit) base type and a mandatory type refinement. The refinement constrains the type of the class members by specifying more precise types.

Existential types Existential types are similar to the Java wildcard types which provide type-safe casts for erased types at run-time. In practice, they are mainly useful for accessing Java wildcard types and raw types from Scala.

Self types A self type specifies the requirements on any concrete class when one or more traits are mixed into. Then it may be necessary to specify which of those other traits should be assumed. Technically, a self type is an assumed type for variable `this` whenever `this` appears within the class.

Singleton types Singleton types represent a basic form of dependent types and always refer to a particular value. A singleton type thus expresses the fact that a value is the only instance of its type.

Scala further supports powerful programming techniques such as implicit conversions [90, §21.1], implicit parameters [90, §21.5], default parameter values and named arguments for adapting existing code or writing new code in a more flexible and concise way.

[5]The refactoring of the collection library has been achieved in version 2.8 but Scala supports type constructor polymorphism since version 2.5.

NOTE

Since the project's beginning the Scala language has undergone one major evolution in March 2006; while software versions released before that date implement the Scala 1 specification actual versions of the language implementation follow the Scala 2 specification [89].
The Scala compiler and libraries (e.g. the collection library and the Actors library) are actively maintained — a new software distribution is released every 2-3 months — and are still evolving through smooth steps.

The Scala compiler is targeted both to the Java and the .NET platforms and can read respectively generate either Java class files or MSIL assembly files (separate compilation). The Scala compiler is itself written in the Scala programming language and is bootstrapped.

The design of Scala is mainly — but not only — influenced by the language Java (Java-like syntax, class libraries) and the Java infrastructure (run-time environment and libraries). Scala also integrates concepts pioniered by other languages like the uniform object model of Smalltalk, the systematic nesting of Beta and functional aspects of ML and Haskell.

In particular, Scala benefits from its interoperability with Java and its greater expressiveness to get adopted by many programmers from the Java community. Nevertheless, this pragmatic approach has both advantages and disadvantages: while the functionality of the numerous Java libraries is directly accessible to the Scala programmers the interaction with some Java features does complicate the tasks of the Scala implementors.

1.3.2 The Java Platform

In this project we adopt the Java platform as our global target architecture and thus avoid the implementation challenges related to the heterogeneity of the network locations.

Indeed, while Java as an object-oriented language doesn't offer language support for distributed programming, Java as a development platform provides library and run-time support for developing distributed applications. Its cross-platform capability together with mechanisms like dynamic class loading, object serialization and distributed objects facilitate the development of distributed systems.

Furthermore, the Java security framework was designed with the security aspects of network programming in mind and features run-time mechanisms at both the platform level and the application level.

1.4 Contributions

The main contributions of this work can be summarized as follows:

- We present a novel approach based on the notion of lambda abstraction for dealing with the dynamic rebinding of local references in a distributed execution environment.

  The lambda-calculus has had a tremendous influence on the design and analysis of programming languages; realistic languages are indeed too large and complex to study from scratch as a whole. That applies in particular to distributed programming languages which promote the cooperation between concurrent processes.

  In this work we defend the idea that the naked notion of lambda abstraction — with no domain-specific extensions — offers an effective way to build a new programming abstraction for mobile code.

- We introduce a new *programming abstraction* for mobile code which relies on two fundamental binding mechanisms: the dynamic rebinding mechanism provided by the lambda abstraction applies to variables captured in the lexical context and the generalized data binding mechanism provided by the concept of module objects allows the selective import of object members to any lexical context.

  Concretely, we define *detached closures* as lexical closures with remote references which can move in a distributed execution environment. This follows the functional way of dealing with higher-order functions and allows us to stay close to the concept of lambda abstraction with our new programming abstraction.

- We add *programming support* for detached closures to the language Scala. The new facility preserves language semantics and relies on three existing features: Scala higher-order functions, Java code mobility and Java distributed objects.

  We address the different challenges associated with the conversion of detached closures in a distributed settings: first, the function body is a parametrized piece of code that can move between different environments; second, the code preserves bindings to variables in their originating scope; and finally, the code performs operations on type-safe remote references.

  Concretely, we extend the compiler front-end with a new transformation phase and provide the needed run-time infrastructure as an

extension of the Scala standard library. After analyzing the code of the detached closure, the new phase generates remote interfaces and substitutes occurences of free objects with remote references to proxy objects. The run-time library extension provides a global naming service for remote references and remote interfaces for predefined types.

1.5 Outline

The structure of the thesis is as follows:

Chapter 1 introduces the key idea and the design goals of this work; it also defines its scope, presents some backgrounds and resumes its main contributions.

Chapter 2 gives an overview of multi-paradigm programming. In particular, it focuses on two concepts of the functional and distributed programming which are essential in this project for the realization of our new programming abstraction: lexical closures and distributed objects.

Chapter 3 handles important aspects of code mobililty. In particular, it introduces two forms of code mobility, discusses security issues related to mobile code and finally focus on the mechanisms featured by the Java platform to support code mobility.

Chapter 4 focus on previous research efforts to study the concept of dynamic rebinding in higher-order distributed computations. In particular, we observe that most research on mobile code has been carried out in the context of functional programming languages.

Chapter 5 gives several code examples to demonstrate the practical usage of our programming abstraction. In particular, the new abstraction relies only on existing language features and combines smoothly with other functionalities provided by the existing Scala libraries.

Chapter 6 presents the abstraction model for extending the Scala language with support for mobile code. In particular, it builds upon the idea that the notion of lambda abstraction provides an elegant way for dealing with the dynamic rebinding of local references in a distributed execution environment.

Chapter 7 presents the implementation of the compiler and library extensions for supporting mobile code. In particular, the addition of the new facility does not require any language change.

Finally, Chapter 8 concludes.

Chapter 2

Multi-paradigm Programming

The idea of a multi-paradigm language is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms.

Timothy Budd[1]

Timothy Budd [22] describes a programming paradigm as ".. *a way of conceptualizing what it means to perform computation, of structuring and organizing how tasks are to be carried out on a computer*" in his seminal book on multi-paradigm programming.

Examples of programming paradigms include the imperative paradigm (Turing machine), functional paradigm (lambda calculus), logic paradigm (predicate calculus) or object-oriented programming (object calculus).

Some languages tend to subscribe to a single programming paradigm like Smalltalk and Java, two representatives of the object-oriented paradigm. Other languages like OCaml, Leda, Oz and Scala integrate concepts borrowed from several programming paradigms. Since every paradigm comes with its own strengths and weaknesses, no single paradigm offers the simplest, most elegant and efficient solution to every problem. This quite naturally motivates the need to use a combination of paradigms in a manner that best fits the problem(s) at hand.

Multi-paradigm extensions to Java have been experimented by several researchers [23, 91]. Furthermore languages such as Groovy and Scala go one step further in adopting the multi-paradigm concept without sacrifying their interoperability with Java.

[1] Timothy Budd is the designer of Leda, a multi-paradigm language based on Pascal.

2.1 Functional Programming

Functional programming (FP) [62] is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. FP emphasizes the application of functions, in contrast with the imperative programming style that emphasizes changes in state.

In particular, *referential transparency* — the program behaves the same when an expression is replaced by its value — is an important principle of FP, allowing referentially transparent functions to be memoized or to be included in interprocedural dependence analysis.

FP supports functions as first-class objects, also called *function values*. Essentially, function values are objects that act as functions and can be operated upon as objects — allowing them to be stored in data structures, passed as arguments and returned as results.

A function value that captures free (non-local) variables is also called a *closure* (discussed later in Section 2.1.2). Many research works have focused on supporting closures in non-functional languages such as Ada [61], C++ [20, 127] and Java [45].

FP also supports *higher-order functions* (HOFs). A HOF can take functional arguments or can return function values.

These two constructs together allow for elegant ways to modularize programs, which is one of the biggest advantages of adopting the FP style. Curried functions [90, §9.3] is another powerful FP construct we'll keep aside as it is not directly relevant for this work.

2.1.1 Higher-Order Functions

Besides functional languages like Erlang, Haskell, ML or Scheme, which are well-known to make use of higher-order programming techniques, dynamic languages like Groovy, JavaScript, Python or Ruby also provide good support for higher-order programming.

As a citizen of both the functional and object-oriented worlds, Scala also features many FP concepts for supporting higher-order programming. HOFs are for instance used in simulation programs to execute actions at specified points in simulated time or to install triggers that associate actions with state changes.

Several proposals have been submitted by the Java community to provide support for closures in Java; the Java source code listed in Table 2.3 and Table 2.6 follows BGGA's proposal[2] [45].

[2] BGGA: Gilad Bracha, Neal Gafter, James Gosling and Peter von der Ahé.

Also, the possibility to compose HOFs is a powerful language abstraction of the FP paradigm. In particular, the function value returned by the function composition is a closure whose formal parameters are bound to actual arguments visible from the calling environment.

In the following we illustrate the usage of HOFs with the `map` function and give source code examples in several programming languages. Basically, the `map` function takes a function `f` and a sequence `s` as arguments and returns a list containing the results of applying `f` to the elements of `s`. Iterating over a sequence using the `map` function obliviates the need for an explicit counter — typical with the imperative programming style.

Table 2.1, Table 2.2 and Table 2.3 illustrate the usage of *anonymous function values* (or *function literals*) in several functional languages, dynamic languages and object-oriented languages, respectively.

In particular, we can make the following observations:

- In functional languages `map` is a first-class function taking a sequence as argument while in object-oriented languages `map` is a member function of some sequence type. The difference in the syntactical order becomes explicit then function composition comes into the play (an example is presented later in this section).
- The language Java — in opposite to C# and JavaFX — has not yet adopted closures as control abstraction; Java code samples presented later in this chapter are thus based on BGGA's proposal.
- Most languages featuring function values associate local type inference with a lightweight syntax to enforce their conciseness.

Clojure	`(map (fn [x] (* x 2)) [1 2 3 4 5])`
Erlang	`lists:map(fun(X) -> X*2 end, [1,2,3,4,5]).`
Haskell	`map (\x -> x*2) [1,2,3,4,5]`
OCaml	`List.map (function x -> x*2) [1;2;3;4;5];;`
SML	`map (fn (x) => x*2) [1,2,3,4,5];`
Scheme	`(map (lambda (x) (* x 2)) '(1 2 3 4 5))`

Table 2.1: `map` function in functional languages.

<table>
<tr><td>Groovy</td><td>[1,2,3,4,5].collect {x -> x*2}</td></tr>
<tr><td>JavaScript</td><td>var res =[1,2,3,4,5].map(function(x) {return x*2});
print("res="+res);</td></tr>
<tr><td>Perl</td><td>my @res = map {$_*2} 1..5;
print "res=", toStr(@res), "\n";</td></tr>
<tr><td>Python</td><td>map(lambda x: x*2, [1,2,3,4,5])</td></tr>
<tr><td>Ruby</td><td>[1,2,3,4,5].map {|x| x*2}</td></tr>
</table>

Table 2.2: `map` function in dynamic languages.

<table>
<tr><td>C# (3.0)</td><td>class hof_map1 {
    static void Main(string[] args) {
        var res =
            new List<int> {1,2,3,4,5}.Select(x => x*2);
        Console.Write("res=" + Utils.ToString(res));
}}</td></tr>
<tr><td>Java (BGGA)</td><td>// <A,B> List<B> map(List<A> xs, {A => B} f) {..}
public class hof_map1 {
  public static void main(String[] args) {
    List<Integer> res =
      map(asList(1,2,3,4,5), {int x => x*2});
    out.println("res="+res);
}}</td></tr>
<tr><td>JavaFX</td><td>// function map(xs: Integer[],
//   f: function(Integer): Integer
//   ): Integer[] {..}
var res = map([1..5], function(x: Integer) {x*2});
out.println("res={toString(res)}");</td></tr>
<tr><td>Scala [1]</td><td>List(1,2,3,4,5) map (_ * 2)</td></tr>
<tr><td>Smalltalk (Squeak)</td><td>|res|
res := #(1 2 3 4 5) collect: [:x|x*2].
Transcript show: 'res='; show: res; cr.</td></tr>
</table>

Table 2.3: `map` function in object-oriented languages.

[1]Scala intentionally appears here, near to Java, although it is both a functional and an object-oriented language.

Similarly, Table 2.4, Table 2.5 and Table 2.6 illustrate the usage of *named function values* in several functional languages, dynamic languages and object-oriented languages, respectively.

In particular, we can make the following observations:

- Most languages adopt a similar (if not identical) syntax for specifying anonymous and named function values while in languages such as Haskell, OCaml, Python and Ruby the function body is introduced by a different token (e.g. "->" versus "=" in Haskell).
- Languages such as Perl and Ruby require an unusual syntax for specifying function values as function arguments. For instance, the Ruby programmer must write `|x| twice(x)` instead of providing the partially applied function `twice` as argument (see Table 2.5).
- Again, most languages featuring function values associate local type inference with a lightweight syntax to enforce their conciseness.

Clojure	`(defn twice [x] (* x 2))` `(map twice [1 2 3 4 5])`
Erlang[1]	`Twice = fun(X) -> X*2 end.` `lists:map(Twice, [1,2,3,4,5]).`
Haskell	`let twice x = x*2` `map twice [1,2,3,4,5]`
OCaml	`let twice x = x*2;;` `List.map twice [1;2;3;4;5];;`
SML	`fun twice x = x*2;` `map twice [1,2,3,4,5];`
Scheme	`(define (twice x) (* x 2))` `(map twice '(1 2 3 4 5))`

Table 2.4: `map` function in functional languages.

[1]Variable names in Erlang must start with a capital letter.

Groovy	`twice = {x -> x*2}` `[1,2,3,4,5].collect twice`

JavaScript	`function twice(x) {return x*2};`
	`var res = [1,2,3,4,5].map(twice);`
	`print("res="+res);`

Perl[1]	`sub twice { my $x = shift; $x*2 };`
	`my @res = map {twice($_)} 1..5;`
	`print "res=", toStr(@res), "\n";`

Python	`def twice(x): return x*2` `map(twice, [1,2,3,4,5])`
Ruby	`def twice(x) x*2 end` `[1,2,3,4,5].map {|x| twice(x)}`

Table 2.5: `map` function in dynamic languages.

[1] `toStr` is a user-defined subroutine.

Language	Code

C# (3.0)	class hof_map2 {
	    static void Main(string[] args) {
	        Func<int, int> twice = x => x*2;
	        var res =
	            new List<int> {1,2,3,4,5}.Select(twice);
	        Console.Write("res=" + Utils.ToString(res));
	}}

Java (BGGA)[2]	public class hof_map2 {
	    public static void main(String[] args) {
	        {int => int} twice = {int x => x*2};
	        List<Integer> res =
	            map(asList(1,2,3,4,5), twice);
	        out.println("res="+res);
	}}

JavaFX[2]	function twice(x: Integer): Integer {x*2}
	var res = map([1..5], twice);
	out.println("res={toString(res)}");

Scala	def twice(x: Int) = x*2 List(1,2,3,4,5) map twice
Smalltalk (Squeak)	|twice res| twice := [:x|x*2]. res := #(1 2 3 4 5) collect: twice. Transcript show: ’res=’; show: res; cr.

Table 2.6: `map` function in object-oriented languages.

[2]Function `map` is user-defined (see Table 2.3).

HOFs can further be composed to build more complex programming abstractions; for example they provide an elegant way to express queries over some data collection.

In Listing 2.1 (C#), Listing 2.2 (Python), Listing 2.3 (Ruby), Listing 2.4 (Scala) and Listing 2.5 (Scheme) we look for the salary of the highest paid programmer in some software company. Each employee is described by his name, role and salary, and is stored in a list of employees.

The query itself is composed of three successive operations: a filter function retains employees who are programmers, a map function returns the salary of those programmers and, finally, a fold (or reduce) function picks up the highest salary.

```
class Employee {
    private string _name;
    private string _role;
    private int _salary;
    public Employee(string name, string role, int salary) {
        _name = name; _role = role; _salary = salary;
    }
    public string role() { return _role; }
    public int salary() { return _salary; }
}

class hof_comp1 {
    static void Main(string[] args) {
        var employees = new List<Employee> {
            new Employee("John", "programmer", 4500),
            new Employee("Tom", "programmer analyst", 6000),
            new Employee("Jessica", "programmer", 5000),
            new Employee("Alvin", "software architect", 7000),
            new Employee("Peter", "programmer", 5200)
        };
        int maxSalary = employees.
            Where(emp => "programmer" == emp.role()).
                Select(emp => emp.salary()).
                    Aggregate(0, (m, v) => (m > v) ? m : v);
        Console.WriteLine("Max. salary=" + maxSalary);
    }
}
```

Listing 2.1: Composition of HOFs in C#.

```
class Employee:
    def __init__(self, name, role, salary):
        self.name = name
        self.role = role
        self.salary = salary
    def __str__(self):
        return '%s(%s,%d)' % (self.name, self.role, self.salary)

employees = (\
    Employee('John', 'programmer', 4500),
    Employee('Tom', 'programmer analyst', 6000),
    Employee('Jessica', 'programmer', 5000),
    Employee('Alvin', 'software architect', 7000),
    Employee('Peter', 'programmer', 5200)
)
maxSalary =\
    reduce(lambda m, v: max(m, v),\
        map(lambda emp: emp.salary,\
            filter(lambda emp: 'programmer' == emp.role,\
                employees)), 0)

print 'Max. salary=%d\n' % maxSalary
```

Listing 2.2: Composition of HOFs in Python.

```
class Employee
  def initialize(name, role, salary)
    @name = name; @role = role; @salary = salary
  end
  attr_reader :name, :role, :salary
  def to_s
    @name+"("+@role+","+@salary.to_s+")"
  end
end

employees = [
  Employee.new("John", "programmer", 4500),
  Employee.new("Tom", "programmer analyst", 6000),
  Employee.new("Jessica", "programmer", 5000),
```

```
  Employee.new("Alvin", "software architect", 7000),
  Employee.new("Peter", "programmer", 5200)
]
maxSalary = employees.
  select {|emp| "programmer" == emp.role}.
    map {|emp| emp.salary}.
      inject {|m, v| m > v ? m : v}

print "Max. salary=", maxSalary, "\n"
```

Listing 2.3: Composition of HOFs in Ruby.

```
case class Employee(name: String, role: String, salary: Int)

val employes = List(
  Employee("John", "programmer", 4500),
  Employee("Tom", "programmer analyst", 6000),
  Employee("Jessica", "programmer", 5000),
  Employee("Alvin", "software architect", 7000),
  Employee("Peter", "programmer", 5200)
)
val maxSalary = employes.
  filter {emp => "programmer" == emp.role}.
    map {emp => emp.salary}.
     foldLeft(0) {(m, v) => if (m > v) m else v}

println("Max. salary="+maxSalary)
```

Listing 2.4: Composition of HOFs in Scala.

```
(define (get-salary emp) (cadr (cdr emp)))
(define (get-role emp) (cadr emp))

(define employees
  '(("John" "programmer" 4500)
    ("Tom" "programmer analyst" 6000)
    ("Jessica" "programmer" 5000)
    ("Alvin" "software architect" 7000)
    ("Peter" "programmer" 5200)
)  )
```

```
(define maxSalary
  (foldl
    0
    (lambda (x y) (max x y))
    (map
      (lambda (emp) (get-salary emp))
      (filter
        (lambda (emp) (equal? (get-role emp) "programmer"))
        employees
) ) ) )

(println "Max. salary=" maxSalary)
```

Listing 2.5: Composition of HOFs in Scheme.

Furthermore the expressiveness of HOFs also comes from the ability to abstract over functional behavior. Examples of such an abstraction are the concurrent primitives in the pi-calculus or the `fold` operation.

In Listing 2.6 (C#), Listing 2.7 (Python), Listing 2.8 (Ruby), Listing 2.9 (Scala) and Listing 2.10 (Scheme) we first define recursively the addition and multiplication on number sequences using the two binary operators + and $*$ together with the corresponding neutral elements. In a second step we abstract over the functional behavior of those two operations to obtain the same functionality using HOFs.

```
class hof1 {
    static int Sum1(List<int> xs) {
        if (xs.Count == 0) return 0;
        else return xs.ElementAt(0) + Sum1(xs.Skip(1).ToList());
    }
    static int Mul1(List<int> xs) {
        if (xs.Count == 0) return 1;
        else return xs.ElementAt(0) * Mul1(xs.Skip(1).ToList());
    }
    static int Sum2(List<int> xs) {
        return xs.Aggregate(0, (x, y) => x + y);
    }
    static int Mul2(List<int> xs) {
        return xs.Aggregate(1, (x, y) => x * y);
    }
```

```
    static void Main(string[] args) {
        var xs = new List<int> {1,2,3,4,5};
        Console.WriteLine(Sum1(xs)); // 15
        Console.WriteLine(Mul1(xs)); // 120
        Console.WriteLine(Sum2(xs)); // 15
        Console.WriteLine(Mul2(xs)); // 120
    }
}
```

Listing 2.6: Abstracting over functional behavior in C#.

```
def sum1(xs):
  if xs == []: return 0
  else: return xs[0] + sum1(xs[1:])
def mul1(xs):
  if xs == []: return 1
  else: return xs[0] * sum1(xs[1:])

def sum2(xs): return reduce(lambda x, y: x+y, xs, 0)
def mul2(xs): return reduce(lambda x, y: x*y, xs, 1)

xs = [1,2,3,4,5]
sum1(xs) # prints 15
mul1(xs) # prints 120
sum2(xs) # prints 15
mul2(xs) # prints 120
```

Listing 2.7: Abstracting over functional behavior in Python.

```
def sum1(xs)
  if xs == [] then 0 else xs[0] + sum1(xs[1..-1]) end
end
def mul1(xs)
  if xs == [] then 1 else xs[0] * mul1(xs[1..-1]) end
end

def sum2(xs) xs.inject(0) {|x, y| x + y} end
def mul2(xs) xs.inject(1) {|x, y| x * y} end

xs = [1,2,3,4,5]
```

```
sum1(xs) # prints 15
mul1(xs) # prints 120
sum2(xs) # prints 15
mul2(xs) # prints 120
```

Listing 2.8: Abstracting over functional behavior in Ruby.

```
def sum1(xs: List[Int]): Int =
  if (xs.isEmpty) 0 else xs.head + sum1(xs.tail)
def mul1(xs: List[Int]): Int =
  if (xs.isEmpty) 1 else xs.head * mul1(xs.tail)

def sum2(xs: List[Int]): Int = xs.foldLeft(0)((x, y) => x+y)
def mul2(xs: List[Int]): Int = xs.foldLeft(1)((x, y) => x*y)

val xs = List(1,2,3,4,5)
println(sum1(xs)) // prints 15
println(mul1(xs)) // prints 120
println(sum2(xs)) // prints 15
println(mul2(xs)) // prints 120
```

Listing 2.9: Abstracting over functional behavior in Scala.

```
(define (sum1 xs) (if (null? xs) 0 (+ (car xs) (sum1 (cdr xs)))))
(define (mul1 xs) (if (null? xs) 1 (* (car xs) (mul1 (cdr xs)))))

(define (foldl z f xs)
        (if (null? xs) z (foldl (f z (car xs)) f (cdr xs))))
(define (sum2 xs) (foldl 0 + xs))
(define (mul2 xs) (foldl 1 * xs))

(define xs '(1 2 3 4 5))
(sum1 xs) ; prints 15
(mul1 xs) ; prints 120
(sum2 xs) ; prints 15
(mul2 xs) ; prints 120
```

Listing 2.10: Abstracting over functional behavior in Scheme.

2.1.2 Lexical Closures

The term *lexical closure* refers to a function value that captures free (non-local) variables while preserving lexical scoping, i.e. bindings to the surrounding scope are retained until the function is actually applied to its arguments (see also Section 6.1).

Every programming language must have some rules to determine the declaration to which each variable reference refers. These rules are typically called scoping rules. The portion of the program in which a declaration is valid is called the scope of the declaration.

Lexical scopes are nested: to find which declaration corresponds to a given use of a variable, we search outward from the use until we find a declaration of the variable. With *lexical scoping* (or *static scope*) a local declaration thus takes precedence over a previous declaration using the same name.

The concept has been around in programming languages since at least Algol-60 and in the following we use the terms *closures* and *lexical closures* interchangeably[3].

NOTE

Scheme has introduced lexical scoping and lexical closures to the Lisp world; early Lisp implementations were indeed purely dynamically scoped. Concretely, the name of a dynamically scoped variable is looked up in the call stack and resolved in the most recent stack frame.

Languages such as Common Lisp [105, §6] and Scala support both ways to declare variables; Scala adopts the lexical scoping rule and the Scala standard library provides support for dynamically scoped variables:

```
object Scope extends Application {
  val xs = List(1,2,3,4)
  val ys = xs.map((x: Int) => x :: xs)
  println("ys="+ys.mkString("List(\n  ", ",\n  ", ")"))

  def map[A, B](f: A => B,
              v: DynamicVariable[List[A]]): List[B] = {
    val xs = v.value
    if (xs.isEmpty) Nil
    else f(xs.head) :: v.withValue(xs.tail) { map(f, v) }
  }
  val xs2 = new DynamicVariable(List(1,2,3,4))
  val ys2 = map((x: Int) => x :: xs2.value, xs2)
  println("ys2="+ys2.mkString("List(\n  ", ",\n  ", ")"))
}
```

[3]Nowadays most languages support lexical closures.

(continued)

The printed values for the resulting lists `ys` (static scope) and `ys2` (dynamic scope) differ according to the adopted scoping rule:

```
ys=List(                          ys2=List(
  List(1, 1, 2, 3, 4),              List(1, 1, 2, 3, 4),
  List(2, 1, 2, 3, 4),              List(2, 2, 3, 4),
  List(3, 1, 2, 3, 4),              List(3, 3, 4),
  List(4, 1, 2, 3, 4))              List(4, 4))
```

Table 2.7 illustrates the usage of closures with a source code sample written in several functional languages; Table 2.8 and Table 2.9 present the same example for well-known dynamic and object-oriented languages.

In the presented code, the function `multiply` takes two parameters, a list `list` of integers and some integer value `n`, and returns a list of integers containing the results of the multiplication of each element of `list` with `n`. Variable `x` is bound to the lexical scope of the closure passed as argument to the `map` function while `n` is a free variable.

Language	Code
Clojure	`(defn multiply [n list] (map (fn [x] (* x n)) list))` `(multiply 2 [1 2 3 4 5])`

Erlang	`Multiply = fun(N, List) ->`
	`    lists:map(fun(X) -> X*N end, List) end.`
	`Multiply(2, [1,2,3,4,5]).`

Haskell	`let multiply n list = map (\x -> x*n) list` `multiply 2 [1,2,3,4,5]`

OCaml	`let multiply n list =`
	`  List.map (function (x) -> x*n) list;;`
	`multiply 2 [1; 2; 3; 4; 5];;`

SML	`fun multiply n list = map (fn x => x*n) list;` `multiply 2 [1,2,3,4,5];`

Scheme	`(define (multiply n list)`
	`        (map (lambda (x) (* x n)) list))`
	`(multiply 2 '(1 2 3 4 5))`

Table 2.7: Closures in functional languages.

In each example, the evaluation of function `multiply` with the arguments `2` and `[1,2,3,4,5]` simply returns the list `[2,4,6,8,10]`.

<table>
<tr><td>Groovy</td><td><pre>multiply = {n, list -> list.collect {x -> x*n}}
multiply(2, [1,2,3,4,5])</pre></td></tr>
<tr><td>JavaScript</td><td><pre>function multiply(n, list) {
  if (typeof list.map != "function")
    throw new TypeError();
  return list.map(function(x) {return x*n});
}
var res = multiply(2, [1,2,3,4,5]);
print("res="+res);</pre></td></tr>
<tr><td>Perl</td><td><pre>sub multiply {
    my ($n, @list) = @_; map {$_*$n} @list;
}
my @res = multiply(2, 1..5);
print "res=", toStr(@res), "\n";</pre></td></tr>
<tr><td>Python</td><td><pre>def multiply(n, list):
    return map(lambda x: x*n, list)
multiply(2, [1,2,3,4,5])</pre></td></tr>
<tr><td>Ruby</td><td><pre>def multiply(n, list) list.map {|x| x*n} end
multiply(2, [1,2,3,4,5])</pre></td></tr>
</table>

Table 2.8: Closures in dynamic languages.

NOTE

One can approximate the functionality of closures in Java using anonymous inner classes. There are two problems with this, though. First, anonymous inner classes are unnecessarily verbose. Second, Java doesn't really close over the surrounding scope, it copies it. To hide this implementation detail, any variable referenced inside the inner class must be declared final outside it.

The combination of anonymous functions and higher-order functions together is where the programmer gets the power from, the anonymous functions essentially specialize the higher-order functions. Any language that supports passing functions as parameters can support higher-order functions, but without anonymous functions, they won't get used too often.

C# (3.0)

```
class clo1 {
    static void Main(string[] args) {
        Func<int, List<int>, List<int>> multiply =
            (n, list) => list.Select(x => x*n).ToList();
        var res = multiply(2, new List<int>{1,2,3,4,5});
        Console.Write("res=" + Utils.ToString(res));
}}
```

Java (BGGA)[1]

```
public class clo1 {
    static List<Integer>
    multiply(int n, List<Integer> list) {
        return map(list, {int x => x*n});
    }
    public static void main(String[] args) {
        List<Integer> res =
            multiply(2, asList(1,2,3,4,5));
        out.println("res="+res);
        {int, List<Integer> => List<Integer>} multiply2 =
            {int n, List<Integer> list =>
                map(list, {int x => x*n})};
        List<Integer> res2 =
            multiply2.invoke(2, asList(1,2,3,4,5));
        out.println("res2="+res2);
}}
```

JavaFX

```
function multiply(n: Integer,
                  list: Integer[]): Integer[] {
  map(list, function(x: Integer): Integer {x*n});
}
var res = multiply(2, [1..5]);
out.println("res={toString(res)}");
```

Scala

```
def multiply(n: Int, list: List[Int]): List[Int] =
  list map (x => x*n)
multiply(2, List(1, 2, 3, 4, 5))
```

Smalltalk (Squeak)

```
|multiply res|
multiply := [:n :list | list collect: [:x|x*n]].
res := multiply value: 2 value: #(1 2 3 4 5).
Transcript show: 'res='; show: res; cr.
```

Table 2.9: Closures in object-oriented languages.

[1] `map`, `asList` and `out` are static imports.

2.2 Distributed Programming

Support for cooperative distributed applications is an important research topics involving developments in operating systems as well as in programming languages.

In the 1980's, a model has emerged for the support of cooperative distributed applications, that of a distributed shared universe organized as a set of objects. Distributed object-oriented systems such as Emerald [15, 16], Clouds [36] or Guide [53] belong to this family of systems. More recently, the growth of the Internet, which is now used daily for cooperation, logically leads to the deployment of cooperative distributed applications over the Internet.

Today, distributing applications on the Internet is often linked to the Web (essentially URLs) and Java. This is mainly because they provide a global naming scheme and machine independent code. A first step to provide distributed shared objects on the Internet was Java RMI [128] which supports remote method invocation between Java objects. Distributed programs can exchange remote object references using Java RMI or they can send copies of objects using the object serialization mechanism (as ONC RPC [37, 73]).

2.2.1 Historical Survey

One of the first few steps towards providing some abstraction at the programming level in a distributed system was made by Andrew Birell and Bruce Nelson in the 1980's. They proposed and implemented the RPC model, which allows programmers to invoke procedures on remote machines as if they were local [12, 84] (see Section 2.2.2). The transparency of remote calls was made possible by the use of client stubs. A stub is piece of code (automatically generated by a software tool) that is responsible with marshalling the request (along with the parameters) and sending it over the network. On the server side, a server stub do the reverse process, it invokes the local procedure and sends the results back to the caller.

This model proved to be successful and other companies created their own RPC modules. Sun Microsystems implemented their own RPC library (SunRPC, in 1985); this is still in use today as an underlying mechanism for the NFS file system. However, it became clear that — although useful — RPC was still a low-level tool for distributed programming and that a more abstract approach was needed.

While some languages — e.g. Emerald [14] and Linda — were designed to meet the requirements of a distributed programming language, none of these languages became popular and a new approach was proposed.

Instead of using a single language to write distributed applications, the CORBA specification (Section 2.2.4) defines a standard architecture for connecting existing objects, written in any language. An IDL (Interface Definition Language) is used in order to define wrappers around objects, so that they can interact in a standard way. A client application can access a CORBA object via a local IDL stub that talks to an IDL skeleton (on the server-side) via an ORB (Object Request Broker).

The initial release of Java in 1995 was another step in the efforts to provide programming abstractions in a distributed system. Although Java is not a true distributed programming language, it makes distributed programming easy by incorporating a lot of useful network-related packages in its core API. It also provides mechanisms for dynamic class loading [11, 104, 111] and for dynamic discovery of objects' capabilities (the "reflection" package). These features along with its platform independence and simplicity made it a successful language both in software industry and academic research.

In addition to its low level packages for socket programming, Java provides — since version 1.2 (1997) — a mechanism for remote method invocation (RMI). The RMI model (discussed later in Section 2.2.5) provides a higher-level of abstraction and gives Java the flavor of a distributed language. It is suited for client-server architectures and uses the same idea of stubs and skeletons as RPC and CORBA.

2.2.2 Remote Procedure Calls

RPC [47, 84] provides a communication paradigm for invoking subroutines across a network[4]. RPC implements a logical client-server communication system designed specifically for the support of network applications. The idea of RPC has been discussed in the literature since 1976 [125]; Nelson's doctoral dissertation [84] presents extensively the design possibilities for an RPC system.

The primary purpose of RPC is to make distributed computing easy. It was previously observed that the construction of communicating programs was a very painful task. Since procedure calls are a well-understood mechanism for transfer of control and data inside a program running on

[4] RPC works just as well on a single machine.

a single machine, it was proposed to extend the same mechanism to transfer data across a network. Note that RPC systems require *call-by-value* since addresses are context dependent and have no meaning in the remote environment.

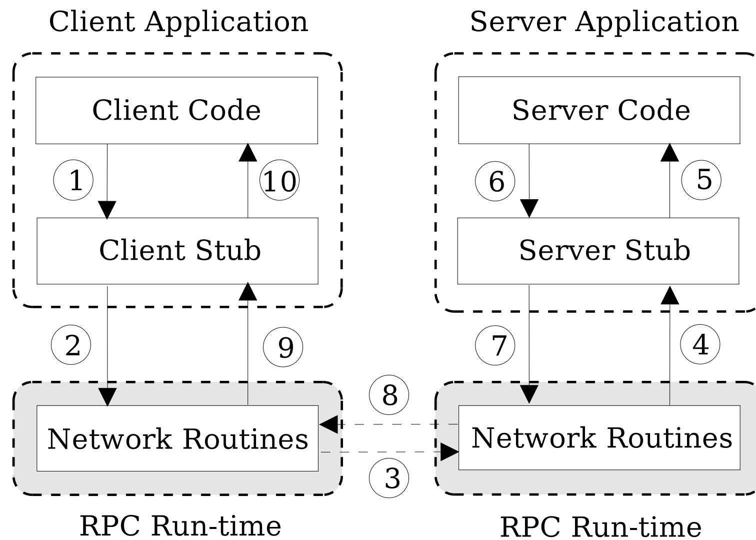


Figure 2.1: RPC architecture.

The flow of control in a RPC call is depicted in Figure 2.1. The control goes logically through two processes: the client process and the server process. First, the client process sends a call message that includes the procedure parameters to the server process. Then the client process waits for a reply message[5]. Next, the server process, which waits for the arrival of a call message, extracts the procedure parameters, computes the results, and sends a reply message. Finally, the client process receives the reply message, extracts the results of the procedure, and resumes execution.

RPC presumes the existence of a low-level transport protocol, such as TCP or UDP, for carrying the message data between communicating programs. RPC provides an authentication process that identifies the server and client to each other and includes a slot for the authentication parameters on every remote procedure call so that the caller can identify itself to the server.

The RPC Language (RPCL) [115] is similar to the C language and formally describes the procedures that operate on the XDR (External Data

[5] The server process provides a service via a request-reply protocol which requires that clients obey strict synchronization rules.

Representation) data types defining the service protocols. Both RPC and XDR type definitions are then compiled into C type definitions in the output header file.

Because most RPC implementations do not support asynchronous client-server interaction, the RPC architecture is not well-suited for applications involving distributed objects or, more generally, object-oriented programming.

> **NOTE**
> Appendix A presents implementations of a simple distributed application built upon RPC, CORBA, RMI and .NET Remoting respectively.

2.2.3 Remote Evaluation

In 1990 an alternative architecture called Remote Evaluation (REV) was proposed by Stamos and Gifford [113, 114]. In REV, instead of invoking a remote procedure as with RPC, the client sends its own procedure code to a server, requesting that the server executes it and returns the results (see Figure 2.2).

In RPC, data travels between the client and server, in both directions. In REV, both code and data go from the client to the server and data returns. Both mechanisms also have similarities: they generate stubs at compile-time and perform static type checking, they presuppose reliable communication and can recover from failures, and they transmit arguments and results between nodes. The main difference between REV and RPC arises from the transmission of executable code: first, REV gives the programmer fine-grained control over the location of processing in a distributed system; second, the server application can provide a more general service which can be used in different ways by the client applications.

A REV request specifies the relocated procedure, the arguments and the server that executes the REV request. Its syntax in CLU is:

```
rev_expression ::= at srv_expression eval invocation
```

The REV request first determines the server, then it evaluates the actual arguments in the procedure invocation and, finally, it sends the relocated procedure and its actual arguments to the server.

The validity of a REV request depends on:

- the transmissibility of the relocated procedure (checked at compile-time). A procedure `P` may be transmittetd to a server supporting

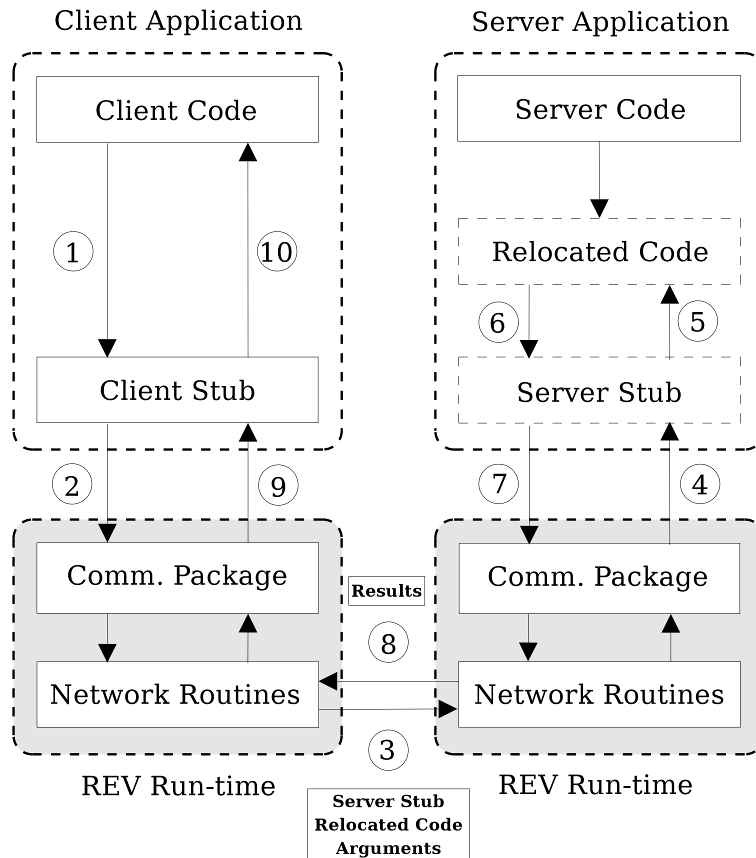


Figure 2.2: REV architecture.

service S if P is contained in service S or if P meets the following condition: P has no free variables and no own variables.

- the transmissibility of the arguments and the results. An object of type T may be transmitted to the server if it satisfies the following conditions: T is a transmissible type [58], both the client and the server implement type T and the argument is not a procedure.

The client code given in Listing 2.11 requests a file service to count the number of vowels occuring in a text file stored on some remote server. Concretely, the client process sends a REV request consisting of the procedure code to be executed remotely together with its arguments (vowels and file name).

```
LetterCount = proc (fname, chars: string) returns (array[int])
  % ...
end LetterCount
```

```
VowelCount = proc () returns (array[int])
  fname: string := ".."
  vowels: string := "aeiouAEIOU"
  service: FileService := FileService$Locate("..")
  answer: array[int] := at service eval LetterCount(fname, vowels)
  return(answer)
end VowelCount
```

Listing 2.11: REV request.

In order to experiment with the REV mechanism the authors have implemented a Lisp-based prototype where the procedure code is transmitted as a compressed form of list structure. They haven't developed any distributed application using REV and have limited their evaluation to a few direct comparisons between the RPC and REV mechanisms. In particular, they observed that REV overperforms RPC when the client-server communication cost is larger than the average execution time of server procedures.

REV Examples

PostScript PostScript is an interpreted language designed to describe and manipulate documents in a system-independent way; it is executed by a specialized execution engine which may be a virtual machine such as a GostView interpreter or a physical machine such a PostScript printer. Postscript documents can thus be transmitted over a network and displayed respectively printed out remotely.

SQL Similarly to PostScript SQL is an interpreted language designed to describe and manipulate data queries in a system-independent way; it is executed by a SQL interpreter which may interact with many different DBMS systems. A set of data is returned as a result of the evaluation of a transmitted SQL query. Again, the key idea is to bring ressources and data processing together in order to execute the task in a dedicated environment.

Mathematica Mathematica is a high-level computing environment supporting computer algebra, graphics and multi-paradigm programming. Furthermore it provides several toolkits like `Mathlink` to communicate with other programming environments, or `Parallel` to execute unrelated tasks in parallel. In particular, `Parallel` provides several commands to

send jobs to slave kernels — hosted locally or on remote computers — for remote evaluation and to retrieve results.

2.2.4 OMG CORBA

Released in 1991 CORBA [93] specifies a system which provides interoperability between objects in a heterogeneous, distributed environment and in a way transparent to the programmer. This open distributed infrastructure has been implemented by many software vendors with language bindings for Ada, C++, COBOL, Java and Smalltalk, among others.

The Object Request Broker (ORB) is the central component of CORBA and encompasses all of the communication infrastructure necessary to identify and locate objects, handle connection management and deliver data.

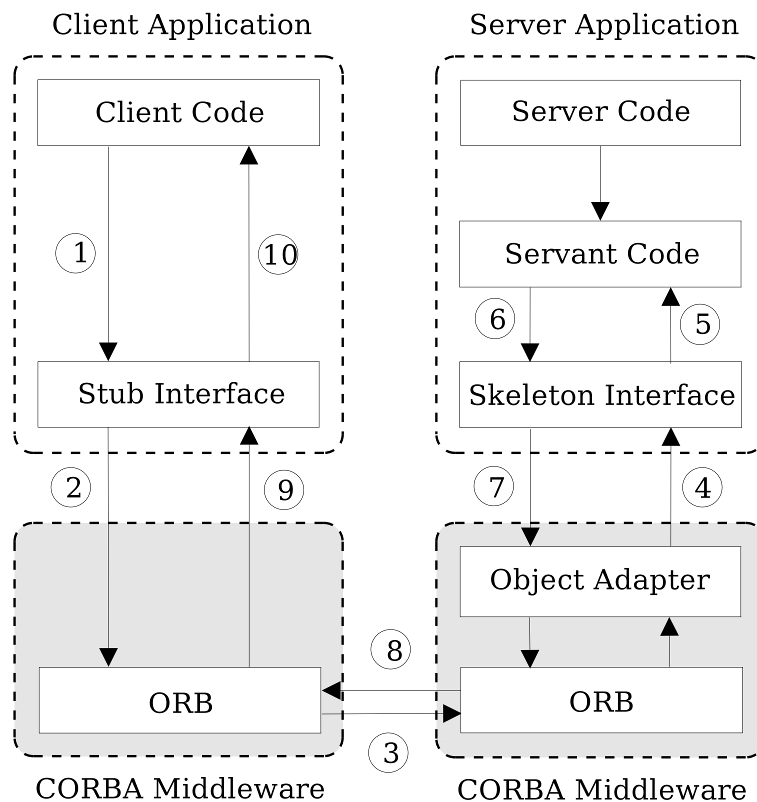


Figure 2.3: CORBA architecture.

The idea behind CORBA is essentially the same as the one behind RMI and the fundamental difference is that CORBA is programming-language independent while Java RMI is confined to the Java platform.

In the CORBA architecture (Figure 2.3) clients request services from objects through a well-defined interface. This interface is specified in OMG IDL (Interface Definition Language). A client accesses an object by issuing a request to the object. The request is an event, and it carries information including an operation, the object reference of the service provider, and actual parameters (if any).

NOTE

Appendix A presents implementations of a simple distributed application built upon RPC, CORBA, RMI and .NET Remoting respectively.

2.2.5 Java RMI

Remote Method Invocation (RMI) [120] was released in February 1997 as part of Sun's JDK 1.1 and is the object-oriented version of RPC.

Java RMI was designed to simplify the communication between two objects in different JVMs by allowing transparent calls to methods in remote virtual machines while preserving the object-oriented paradigm. The system combines aspects of the Modula-3 Network Objects system [13] and Spring's subcontract [54].

Once a remote object reference is obtained, it is possible to call methods of that remote object in the same way as methods of local objects. Since the remote object resides in a different virtual machine, a global naming service is needed to manage remote references; Java RMI implements a remote registry. When an RMI server wants to make its local methods available to remote objects, it registers those methods to the local registry. A remote object connects to the remote registry, which listens to a well-known socket, and obtains a remote reference.

The general Java RMI architecture is depicted in Figure 2.4. First the server creates a remote object and registers it to the local registry (1). The client then connects to the remote registry and obtains the remote reference (2). At this point, a stub of the remote object is transferred from the server VM to the client VM. When the client (3) invokes a method at a remote object, the method is actually invoked at the local stub. The stub marshals the parameters and sends a message (4) to the associated skeleton on the server side. The skeleton unmarshals the parameters and invokes the appropriate method (5). The remote object executes the method and passes the return value back to the skeleton (6), which marshals it and sends a message to the associated stub on the client side (7). Finally the stub unmarshals the return value and passes it to the client (8).

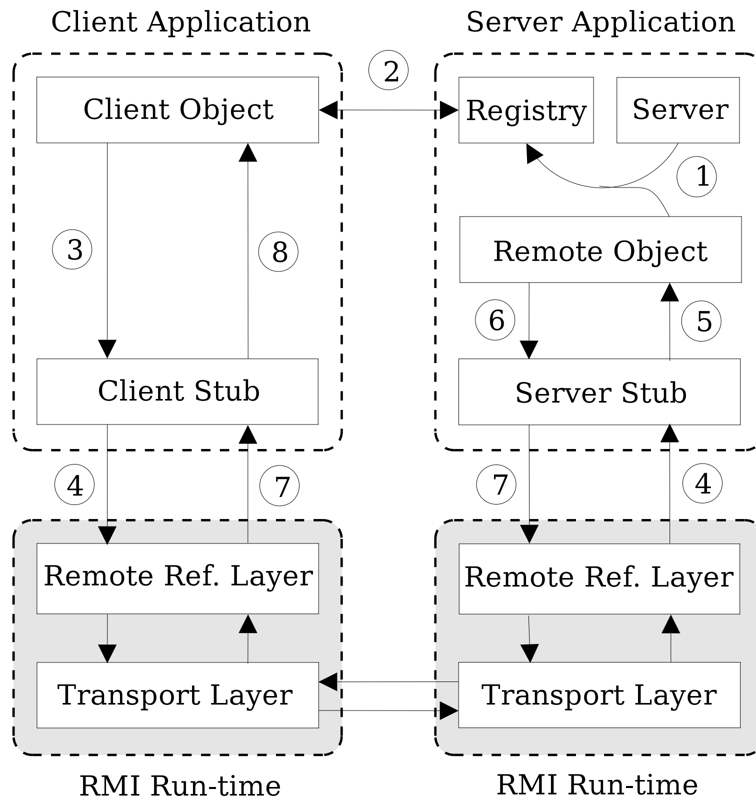


Figure 2.4: RMI architecture.

2.2.6 .NET Remoting

Targeted at the Microsoft's Win32 platforms the .NET Remoting [77, 87, 99] framework allows distributed objects to interact together across application domains in a way similar to Java RMI.

.NET Remoting provides an abstract approach to interprocess communication that separates the remote object from a specific client or server application domain and from a specific mechanism of communication. As a result, it is flexible and easily customizable. You can replace one communication protocol with another, or one serialization format with another without recompiling the client or the server.

NOTE

Released in 2002 .NET Remoting is based on Distributed COM (DCOM), an extension of Microsoft's COM, which allows the interaction between objects running in different machines. In particular, it provides support for parameter marshalling in method calls and distributed garbage collection.

COM allows a binary compatibility between the client and the object, written in arbitrary languages, through the use of interfaces in a similar manner as Java RMI. COM objects and interfaces are specified using Microsoft IDL. When the client is separated from the server, the data must be marshalled. As in Java RMI and CORBA, marshalling is accomplished by a proxy and a stub object that handle the cross–process communication. All COM objects are registered in a component database.

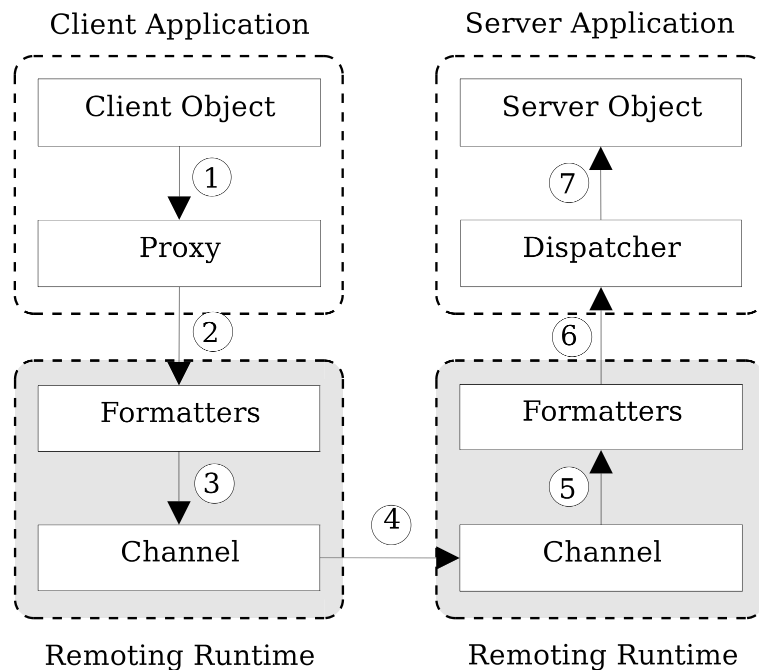


Figure 2.5: .NET Remoting architecture.

2.3 Distributed Programming in Java

Java features a library package for socket programming since its first release in 1995 and support for distributed programming appears one year later with the addition of a distributed object model [128], mainly an object-oriented version of RPC (see Section 2.2.2).

Its integration in the Java run-time environment is achieved through the extension of the exception handling and security mechanisms and the addition of a distributed garbage collector.

2.3.1 Distributed Objects

Java RMI provides language-level support for distributed objects.

Under the term *distributed object* (or *remote object* or *server object*) we mean an object which resides in a separate address space and methods of which can be called remotely (a remote call is issued in an address space separate to the address space where the target object resides).

By convention, the code issuing the call is referred to as the *client*. The set of methods which implements one of the server object's interfaces is sometimes designated as a *service* that this object provides. Similarly, the process in which the server object is located is referred to as a *server*.

An important goal of the client and server abstractions is to make it transparent how far the client and server spaces actually are — whether they reside on the same machine[6] or on different network nodes —.

The design goal for the Java RMI architecture was to create a *distributed object model* (JDOM) that integrates naturally into the Java programming language and the local object model (JOM). The RMI architecture is based on one important principle: the definition of behavior and the implementation of that behavior are separate concepts. Specifically, the definition of a remote service is coded using a Java interface and its implementation is coded in a class.

Similarly to Java objects, remote Java objects are passed as arguments of method invocations, and can be cast to any remote interface allowed by the JOM. Changes to the invocation semantics of the remote objects include passing non-remote arguments of remote methods by copy, passing remote arguments of remote methods by reference, specialized semantics of several methods of the `Object` class, and more complex exception handling required by remote objects.

[6]As a special case, client and server may share the same address space.

NOTE

While Java RMI does preserve the regular Java invocation syntax, parameters of remote operations are treated differently from those of local operations. Remote passing is done as follows:

- If an actual parameter has a primitive type it is passed by value.
- If an actual parameter has a reference type and its class implements the `Remote` interface, it is passed by reference (the local reference being replaced by a network reference).
- If a class does not implement `Remote` but rather implements the `Serializable` interface, the object is passed by value (using serialization).
- If a class implements neither `Remote` nor `Serializable` an exception is raised. Java arrays are serializable by default.

2.3.2 Distributed Exception Handling

All methods of a remote interface, an interface that extends `Remote`, must list `RemoteException` in their throws clause[7]. Additionally, they can throw any other Java exceptions and the client code can catch them.

`RemoteException` is the common superclass for many communication-related exceptions that may be thrown during the execution of a remote method call. They can be organized into several categories depending on the performed RMI operation; for brevity, we give just one example per category:

Naming exceptions A `NotBoundException` is thrown when attempting to look up a name that is not bound.

Export exceptions A `StubNotFoundException` is thrown if no valid stub class was found for a remote object to be exported.

Calling exceptions A `NoSuchObjectException` is thrown when attempting to invoke a method on an object that is no longer available.

Return exceptions An `UnmarshalException` can be thrown while unmarshalling the parameters or results of a remote method call.

2.3.3 Distributed Garbage Collection

Thanks to garbage collection Java programmers don't spend their time chasing memory management errors; the increase in code reliability and

[7] Java programmers must wrap remote method calls with `try/catch` blocks.

productivity largely compensate the run-time costs associated with that mechanism. Garbage collectors (GC) implemented on a single VM are typically designed to maximize the performance of applications at the expense of predictability.

As opposed to the classical techniques used in single address space, garbage collection in distributed environment is a more complex issue that has been the subject of much research (e.g. Ferreira and al. [42]). In particular, coordination becomes essential for the management of distributed system resources. In Java the main difficulty with distributed garbage collection is that object references can cross node boundaries.

Futhermore garbage collection in Java is not guaranteed to be timely[8]. Indeed, as garbage collection is potientially expensive, the JVM is free to do it only when necessary [118].

In contrast to the local garbage collector, the Java RMI distributed garbage collector's (DGC) behavior is well defined and reliable with respect to its timing. The DGC employs a lease mechanism for remote references. All remote references are leased to clients for a default period of time. The client VM must request a new lease before the period runs out. Otherwise the server considers the remote reference to be dead and releases the corresponding remote object to the local GC for potential collection.

The RMI system provides a reference counting distributed garbage collection algorithm based on Modula-3's network objects [13]. Thus, activity on nodes must not be suspended while collecting, as it is the case with the traditional mark-and-sweep scheme.

NOTE

The RMI system uses the broker approach which evaluates the number of live references: it keeps track of the live TCP/IP connections. Basically, each registration of a remote reference in the RMI Registry implies also one live connection. If the number of live connections reaches zero, the server object is handled as if it was a local object in the server VM and thus being a subject of the standard local garbage collection process.

If the number of live remote references to the remote object in the current JVM (before deserializing this one) had been zero, then the client-side DGC implementation will attempt to call the server-side DGC once, synchronous with the deserialization, in an effort to maintain referential integrity (if that attempt fails, retries will only be attempted in the background). Deserializing a remote stub behaves in this respect similarly to unmarshalling one as part of a remote call's arguments or return value.

[8] How long the garbage collector will run is another source of non-determinism.

Similarly to Java, the Emerald system [66] calls for two collectors: a node-local GC that can be run independently of other nodes and a DGC that requires the nodes to cooperate in collecting distributed garbage.

2.4 Discussion

The key characteristic of the mobile code paradigm is to give programmers control over the mobility of code across the network by providing appropriate language features. Therefore, a typical MCL is expected to facilitate the expression and execution of language constructs containing mobile code. The dynamism and flexibility offered by this form of computation, however, brings around a number of problems, the most challenging of which are relevant to type safety and security.

A functional computation has some characteristics which makes it promising to adopt a functional language as the core of a language supporting mobile code. One such characteristic is that functions are *first-class values*; they can be passed as arguments to other functions or they can be returned as results. Furthermore, the rebinding mechanism inherent to transmitted functions greatly facilitates their adaptation to a new execution environment.

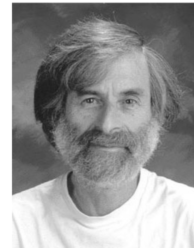
In a setting where the communication between different sites is supposed to be facilitated, functions become a natural candidate for modelling code mobility and thus enhance the dynamic behavior of a system [67]. Regarding type safety, functional languages are also attractive because they are often equipped with expressive and well understood type systems; this applies to Scala too.

Chapter 3

Code Mobility

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Leslie Lamport[1]



The term *mobility* means a change of location performed by the entities of a system and is associated with concerns like communication protocols and interaction with multiple environments. Starting from simple data, the mobility has evolved to abstraction mechanisms to move the execution control, the code and finally the execution environment.

Examples of the first two steps of that evolution are the mobility of files — for example with the FTP protocol — and the RPC concept (see Section 2.2.2) which had a great impact in computer science. Then came the idea to move code.

Code mobility characterizes the capability to dynamically change the bindings between a code fragment and the location in which it is executed. The main idea behind mobile code is that, by bringing the code close to the resources needed for a certain task, it is possible to perform the task in a more effective way. The ability to relocate code is a powerful concept that started a broad range of developments and studies [29, 35, 43].

[1]Leslie Lamport is best known for his seminal work in distributed systems and as the initial developer of LaTeX.

NOTE

Fuggetta and al. [43] identify and compare three main design paradigms exploiting code mobility: code on demand, remote evaluation and mobile agent.

Code on demand Code on demand (COD), is the download of executable content (*code fetching*) to a client environment as the result of a client request to a server. A well-known example of this approach is the download of Java applets or JavaScript code in a Web browser. Executable contents may also be embedded in electronic documents such as JavaScript-enabled PDF documents [1].

Remote evaluation A different form of code mobility is represented by the upload of code (*code shipping*) to a server. The uploaded code is executed by the server and, possibly, the results of the computation are sent back to the client. This form of mobility, also known as *remote evaluation* (REV) [2, 114], allows the client to perform a computation close to the resources located at the server's side so that network interaction can be reduced (see Section 2.2.3).

Mobile agent In the mobile agent (MA) paradigm mobile components can explicitly relocate themselves across the network, usually preserving their execution state (or part thereof) across migrations. Examples of systems supporting this type of mobility are Telescript [126] and IBM's aglets [70].

In middleware systems like Java RMI (and Jini), code migration takes place behind the scenes, where code mobility is exploited to increase the flexibility of service invocation. In other systems, the ability to trigger code migration is directly under the control of the programmer and can be explicitly relocated from one host to another.

Furthermore mobility of code has a direct impact on design decisions regarding computation state; in short, how should state be handled when code is moving ? We give here three choices commonly found in distributed systems:

- *No state*. The code transmission involves no state migration, e.g. Java applets;
- *Autonomous state*. The transmitted code contains no external reference and computation state is represented alone by the current execution state, e.g. IBM's aglets;
- *Global state*. The transmitted code may contain external references — this includes active network connections —, e.g. Obliq objects.

3.1 Forms of Code Mobility

Mobile code has many incarnations: Java applets, ActiveX controls[2], software agents, JavaScript scripts, Visual Basic scripts, e-mail attachments, push technology and so on. All of them are programs that are transmitted from one host and executed on another.

With today's Internet clients, such as Web browsers or mail readers, it is often no longer clear when a user is downloading data for perusal versus downloading a program for execution on his machine.

Several different approaches have emerged for providing some assurance against malicious behavior. Thus, Sun comes up with a technological solution (sandbox mechanism of Java) to constrain malicious behavior, while Microsoft provides a trust-based solution (digital signatures attached to any mobile piece of software).

3.1.1 Weak Mobility

In the context of this work we use the terms *code mobility*, *mobile code* and *weak mobility* interchangeably. A system supports weak mobility if it allows an executing unit in a node to be bound dynamically to code coming from a different node, but no execution state is transferred across the network.

In most environments code mobility rely on a platform-independent representation of the transmitted code; an extra execution environment infrastructure needs to be present at the target node (e.g. a Java VM for Java applets or a JavaScript engine for JavaScript scripts).

3.1.2 Strong Mobility

Similarly, the terms *computation mobility*, *mobile computation* and *strong mobility* are used interchangeably in the context of this work.

In addition to weak mobility strong mobility requires interrupting the execution, moving the state of a run-time system (stacks, for instance) from one site to another, and then resuming execution.

For providing strong mobility to Java, it is necessary to change the compilation model (by using a preprocessor, by modifying the compiler, or by modifying the generated bytecode) or to modify the virtual machine. For example, Fünfrocken [44], Baumgartner [124] and

[2] Released in 1996 ActiveX combines OLE and COM, two prevalent technologies of Microsoft® Windows.

Sekiguchi [106] use a preprocessor, while Suri [121] uses a custom virtual machine in his Nomads system. In her research work Bouchenak [17, 18] compares application-level and platform-level approaches and describes her CTS[3] system, a JVM-based solution for serializing Java threads.

Systems such as Telescript [126], Agent Tcl [49] provide strong mobility by using a dedicated language interpreter to capture and resume the process's execution state.

Other systems such as X-Klaim [6, 7, 8] transform programs supporting strong mobility into code that rely only on weak mobility[4].

In his work Tarau [122] extends BinProlog with mobile threads and shows that part of the functionality of mobile computations can be emulated in terms of remote predicate calls combined with remote code fetching.

3.2 Security

Unlike traditional systems whose design relies on heavy address space protection mechanisms to ensure system reliability (e.g. in multi-user environments), mobile code is intended for quick, lightweight execution, which conflicts with the cost of such security mechanisms [21].

The security issues in mobile code systems can be divided into two categories: one is the protection of hosts from malicious and untrusted mobile code programs [75] and the other is the protection of mobile code programs from malicious and untrusted hosts or intermediaries. For example, a malicious mobile code could allocate memory (or create new threads) until the host runs out (denial of service).

NOTE

In its policy guidance for use of mobile code technologies the U.S. Department of Defense (DoD) [38] defines mobile code as a technology allowing for the creation of executable information which can be delivered to an information system and directly executed on any hardware/software architecture which has an appropriate host execution environment.

The DoD further divides mobile code technologies into three risk categories and restricts their application within DoD based on their potential to cause damage if used maliciously.

[3] Capture time-based thread serialization.

[4] Weak mobility is supported at run-time by the Java package `Klava`.

(continued)

Category 1 Category 1 mobile code exhibits a broad functionality, allowing *unmediated access* to workstation, host and remote system services and resources. It typically requires an all or none decision. Examples include ActiveX, WSH scripts, Unix shell scripts and DOS batch files.

Category 2 Category 2 mobile code has full functionality, allowing *mediated or controlled access* to workstation, host and remote system services and resources. It typically provides known fine-grained, periodic or continuous safeguards. Examples include Java, VBA, LotusScript and PostScript.

Category 3 Category 3 mobile code supports limited functionality, with *no capability for unmediated access* to workstation, host and remote sytem services and resources. It typically provides known fine-grained, periodic or continuous safeguards. Examples include JavaScript, VBScript, PDF and Flash.

Security is a global property, so a security model must take into account all components of a system supporting the execution of code.

Mobile code is typically executed within a system where multiple environments — such as the run-time environment, the operating system and the network — interact in a complex way with none of them ensuring a clear separation of the responsabilities.

For example, Java classes loaded from the local file system are more trusted than classes loaded through the network and may perform more dangerous operations. Here the integrity of the system depends on both the local operating system and the Java run-time environment.

Type Safety Fundamentally, the issue of safe execution of code comes down to a concern about access to system resources. For instance, Java does not allow direct access to the address space of the program and provides automatic memory management (garbage collection).

Any running program has to access system resources in order to perform its task. Dealing with executable content and ressources in distributed environments impose additional requirements on those environments.

The type safety of a programming language is an essential security factor to ensure the safe execution of mobile code. Type systems can guarantee that code cannot violate variable typing or that methods can always be found. These features are needed to ensure that various code units do not interfere with each other or with the system, and that programs do not crash.

NOTE

As part of their formal analysis of a distributed object-oriented language Ahern and Yoshida [5] introduce a core-calculus for a class-based Java-like programming language with basic primitives for distribution, including dynamic class downloading and serialization. In particular, they demonstrate that the integration of those thunk-passing primitives based on Java RMI and class downloading mechanisms preserve type safety.

The type system can further be used to determine the operations that processes want to perform at each locality, and to check whether they comply with the declared intentions and whether they have the necessary rights to perform the intended operations at the specific localities.

3.3 Code Mobility in Java

A key feature of Java is code mobility.

Java allows classes to be dynamically loaded from remote locations. Such mobility of code requires code portability and security enforcement. Code portability is provided by interpretation of the Java bytecode while security is mainly enforced by the type safety of the Java language (see Section 3.3.4).

Since 1996 Java code mobility has been widely used for the Web. Most Web browsers include a Java plugin and HTML pages can refer to Java programs called *Java applets*. While applets are a well-known example illustrating the dynamic loading of classes during code execution, runtime facilities such as object serialization (see Section 3.3.3) and Java RMI (see Section 2.2.5) also rely on that mechanism since data and code move separately in Java. Many services (e.g. servlets, EJB, etc.) featured by the enterprise edition of the Java platform also make an extensive use of those standard mechanisms.

3.3.1 Java Applet

Java applets are the most famous example of code mobility in Java. In the 1990's their interoperability with the emerging World Wide Web has largely contributed to the success of the Java platform.

A Java applet is a (small) network-downloadable piece of code that is intended not to be run on its own, but rather to be embedded inside another application. Unlike a local application, an applet is typically deemed as untrusted and runs under the standard applet security manager.

NOTE

Concretely, an *applet* is an embeddable window (inheriting from class `java.awt.Panel`) with a few extra methods that the applet context can use to initialize, start, and stop the applet. The *applet context* is an application that is responsible for loading and running applets. For example, the applet context could be a Web browser or an applet development environment.

Example In the following client-server (C/S) application the client application is a Java applet (Listing 3.3 and Listing 3.4) displaying a regularly updated message transmitted from the server application (Listing 3.2) using Java RMI.

```
public interface RemoteService extends Remote {
    String getMessage() throws RemoteException;
}
class RemoteServiceImpl implements RemoteService {
    public String getMessage() {
        return "Server date/time is "+getDateTime();
    }
}
```

Listing 3.1: Java applet using RMI (service).

The server application exports a service with just one operation (Listing 3.1): the method `getMessage` returns a formatted text message reporting the current server time. The method `init` initializes both variables `port` and `name` and the method `repl` starts a basic REPL thread.

```
public class Server {
    public static void main(String[] args) throws Exception {
        init(args);
        Registry registry = LocateRegistry.createRegistry(port);
        RemoteService service = new RemoteServiceImpl();
        UnicastRemoteObject.exportObject(service);
        registry.rebind(name, service);
        repl();
    }
}
```

Listing 3.2: Java applet using RMI (server).

On the client side the refresh thread `th` periodically updates the displayed message (Figure 3.1) as long as the containing HTML page is visible in the Web browser (using methods `start` and `stop`).

```
public class ClientApplet extends JApplet implements Runnable {
    private Registry registry;
    private String name;
    private RemoteService service = null;
    private JLabel label = new JLabel();
    private Thread th;

    public void init() {
        add(label);
        try {
            String host = getHost(this);
            int port = getPort(this);
            String path = getPath(this);
            registry = LocateRegistry.getRegistry(host, port);
            name = "//"+host+":"+port+"/"+path;
        } catch (Exception e) {
            label.setText(filterMessage(e));
        }
    }
    public void run() {
        while (Thread.currentThread() == th) {
            String msg = getMessage();
            if (msg != null) label.setText(msg);
            try { Thread.sleep(1000); }
            catch (InterruptedException e) { break; }
        }
    }
    public void start() { th = new Thread(this); th.start(); }
    public void stop() { th = null; }
    public String[][] getParameterInfo() { return pinfo; }

    private String getMessage() {
        String msg = null;
        try {
            checkConnection();
            msg = service.getMessage();
        } catch (Exception e) {
```

```
            msg = filterMessage(e);
            service = null;
        }
        return msg;
    }
    private void checkConnection() throws Exception {
        if (service == null)
            service = (RemoteService) registry.lookup(name);
    }
}
```

Listing 3.3: Java applet using RMI (client).

A Java applet is generally deployed as one or more Java archive files (JAR) which can be digitally signed to ensure their authenticity. The HTML tag `<applet>` serves as an anchor for embedding an applet into a Web page (Listing 3.4); the applet parameters are specified as attribute-value pairs (e.g. `("archive", "applet_client.jar")`) and are passed to the downloaded code in order to customize its behavior (see method `init` in Listing 3.3).

```
<html>
<head>
  <title>Date/time via an Applet using Java RMI</title>
</head>
<body>
  <h1>Date/time via an Applet using Java RMI</h1>
  <p>
    <applet code="examples.ClientApplet" height="50" width="340">
      <param name="archive" value="applet_client.jar"/>
      <param name="registry.host" value="127.0.0.1"/>
      <param name="registry.port" value="8888"/>
      <param name="registry.path" value="RMI_Example"/>
    </applet>
  </p>
</body>
</html>
```

Listing 3.4: Java applet using RMI (HTML page).

Figure 3.1: Java applet using RMI.

3.3.2 Dynamic Class Loading

Every mobile code system (MCS) requires the ability to load code from outside an execution environment into the system dynamically.

The Java platform provides unique support for dynamic class loading through the following feature [65, 71]:

- Classes are loaded on demand (*lazy loading*). The JVM dynamically loads classes and interfaces when they are either needed or explicitly demanded[5].
- Type safety is preserved without additional run-time checks (*type-safe linking*).
- Class loaders provide separate namespaces allowing classes of the same name to be treated as distinct types by the JVM.
- Class loaders are ordinary objects allowing user-definable class loading policies.

The JVM's main task is to execute Java bytecode. Bytecode is stored in Java class files which are loaded into the JVM via a *class loader*. Loading a class means locating a class file that contains the desired type, based on the type's name, and then creating the class from that file. Once a class is loaded (and linked) into a JVM, it becomes part of the program's execution. Finally, the JVM initializes the class by calling a special initialization method, which essentially corresponds to static initialization of the class.

[5]A class is unloaded when its class loader becomes unreachable.

NOTE

The basic JVM knows how to load bytecode only from the local file system. To load code from anywhere else, one has to subclass the abstract class `ClassLoader` (and to override method `defineClass`).

Alternatively the programmer may also use (or subclass) one of the following classes provided by the Java standard library: `java.net.URLClassLoader`, `java.rmi.RMIClassLoader` and `java.security.SecureClassLoader`.

Furthermore several class loaders are used internally by the Java run-time environment:

- `sun.applet.AppletClassLoader` is the class loader used to start a Java applet.
- `sun.misc.Launcher.AppClassLoader` is the default class loader used to start a Java application (may be overriden using the system property "`java.system.class.loader`").
- `sun.plugin.security.PluginClassLoader` implements support for RSA verification (deployment of RSA-signed applets) and dynamic trust management (import of signer certificates).
- `com.sun.jnlp.JNLPClassLoader` is the class loader used in the Java network launch protocol (e.g. Java Web Start).

Example In the following C/S application we use dynamic class loading to periodically update a service which consists of the single operation `getMessage` (Listing 3.5); the service implementation is updated at fixed intervals by the server application (Listing 3.6).

```
public interface RemoteService extends Remote {
    String getMessage() throws RemoteException;
}
```

Listing 3.5: Service class reloading (interface).

Concretely the server application loads a different class (line 6) and rebinds a new instance of the selected service implementation (line 21) to the exported service name. The class loader `cl` with type `URLClassLoader` loads the class corresponding to the specified service implementation (e.g. class `services.Service1`) from some user-defined location. The method `init` (line 7) initializes the static variables `port`, `name`, `cl` and `millis` while the method `repl` starts a basic REPL thread.

```
public class Server {
2   private static RemoteService updateService() throws Exception {
        Class c = cl.loadClass(nextClassName());
```

```
4           return (RemoteService) c.newInstance();
        }
6       public static void main(String[] args) throws Exception {
            init(args);
8           Registry registry = LocateRegistry.createRegistry(port);
            repl();
10          while (true) {
                RemoteService service = updateService();
12              UnicastRemoteObject.exportObject(service);
                registry.rebind(name, service);
14              Thread.sleep(millis);
            }
16      }
    }
```

Listing 3.6: Service class reloading (server).

The client application (Listing 3.7) resolves the service name, invokes the method `service.getMessage` and prints out the returned string value to the console. Thus, successive executions of the client application will output different results depending on the currently available service.

```
    public class Client {
2       public static void main(String[] args) throws Exception {
            init(args);
4           Registry registry = LocateRegistry.getRegistry(host, port);
            RemoteService service = (RemoteService) registry.lookup(name);
6           System.out.println(service.getMessage());
        }
8   }
```

Listing 3.7: Service class reloading (client).

When a JVM starts up, it loads a class using a *bootstrap class loader*. This first class must have a `public static void main(String[] args)` method. The JVM calls that method after initializing the class, and then starts executing code specified in the `main()` method.

Code in the `main()` method typically references classes not yet loaded into the JVM. When the JVM encounters such a reference, it asks its class loader to load, link, and initialize this class as well. The class loader uses a *codebase* to locate the requested class file. The codebase is the location

where the class loader searches for class files.

If there were only a bootstrap class loader, this process would not be very powerful, since all classes would have to be installed at a predetermined location. Fortunately, the JVM's class loader architecture (Figure 3.2) is modular with a hierarchical organization [104]. In addition to its built-in class loader, a JVM allows any number of user-defined class loaders. A Java programmer may thus partition the JVM so that the reduced visibility of a class makes it possible to have multiple, different definitions of the same class loaded.

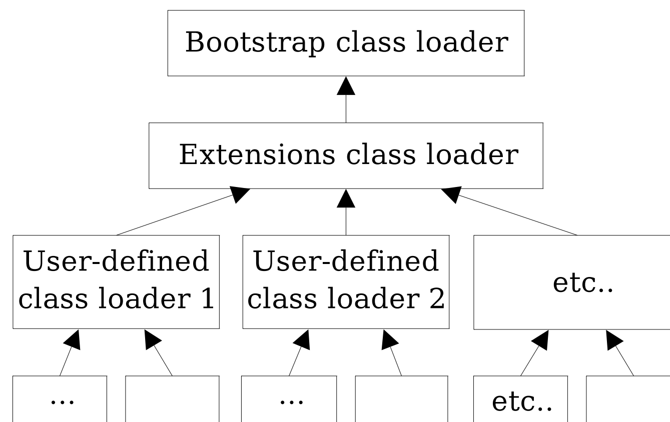


Figure 3.2: Class loader architecture.

NOTE

The bootstrap class loader considers the `CLASSPATH` environment variable to determine its codebase. By default, the Java core class library, supplied by the Java run-time environment (JRE), is also a part of that codebase.

Since Java does not provide any explicit class versioning mechanism, the programmer has to use custom class loaders with extended capabilities. In Java the versioning information is setup at the level of a Java package [117] and the corresponding `Package` object is made available by the `ClassLoader` instance that loaded the class(es)[6]. An application can check if the package is compatible with a particular version with the method `isCompatibleWith`.

Several researchers have studied the class loading mechanism to raise the assurance that the JVM as specified and implemented by Sun is safe;

[6]The versioning information is typically stored in the manifest file that is distributed with the classes.

for instance, Qian and al. [97] have formalized and proved the type safety of the main aspects of class loading in the Java VM.

3.3.3 Object Serialization

The JVM is able not only to load classes, but also to load objects (instances of classes) from a storage device or from the network via the Java object serialization mechanism [50, 119]. Object serialization stores an object (or more generally a graph of objects) in a binary stream in such a way that a Java program can reconstruct the object's state at a later time. The binary stream can then be saved on a persistent storage or sent over a wire[7].

NOTE

Along with instance data, the object serialization mechanism writes a special object to the stream to represent the serializable object's class. This object is of type `ObjectStreamClass`, and is essentially a descriptor for the `Class` object associated with the serialized object. It contains the class's name, its unique version number (`serialVersionUID`), and the class fields.

The serialization run-time calculates a default `serialVersionUID` value for serializable classes that do not explicitly declare it. The Java Object Serialization Specification [119] strongly recommends that all serializable classes explicitly declare `serialVersionUID` values since their computation is highly sensitive to class details and may vary between different Java compiler implementations.

Object serialization supports encryption, both by allowing classes to define their own methods for serialization and deserialization (inside which encryption can be used), and by adhering to the composable stream abstraction (the output of a serialization stream can be channelled into another filter stream which encrypts the data).

NOTE

The Java run-time restricts access to fields declared to be private, package protected, or protected. No such restriction can be made on an object once it has been serialized; the stream of bytes resulting from object serialization can be read and altered by any object that has access to that stream. Consequently, Java developers who declare a class to be `Serializable` must first give some thought to the possible consequences of that declaration.

[7] Only the object's state is saved; the object's class file and methods are not saved but must be accessible from the system in which the restoration occurs.

Codebase Annotation

Given a class's name from the descriptor found in the serialized stream, the JVM still needs to know where it should load the actual class from. Once it has that information, the JVM can create a class loader with the codebase pointing to that location, load the class, link it to the current execution environment, and initialize its class data.

While the JVM could use the data contained in the serialized object stream to create an instance of the class and initialize the object's instance data, the best way is to stamp the code location for the class onto the serialized object stream – in other words, to annotate the serialized object with the codebase URL. This method facilitates dynamic code mobility, because the JVM can decide at run-time where it should download the classes from. This is also the fundamental technique used by Java RMI.

The Java RMI run-time mechanism provides a special output stream which serializes the stub of a remote object (i.e. an object which implements the `java.rmi.Remote` interface) instead of the object itself and annotates the location of the objects'class to the serialized stream.

> **NOTE**
> The codebase is specified to the JVM using the Java RMI property `java.rmi.server.codebase`. The `codebase` property is a space-separated list of URLs. When deserializing an RMI stub annotated with the codebase, the RMI run-time (i.e. `java.rmi.RMICLassLoader`) will create a class loader for each codebase URL specified in this list.

3.3.4 Security

The Java security framework is organized into three layers, each one addressing differents needs:

- At the platform level, the Java security infrastructure provides several security mechanisms such as the bytecode verifier or the sandbox mechanism for Java applets. At run-time Java applications can download classes on demand; classes can thus be loaded from either the local file system (built-in classes) or from a network. The main security concern of the bytecode loader is to prevent built-in classes from being "spoofed" by other classes. This is accomplished by partitioning classes into separated namespaces (see Section 3.3.2).

- At the language level, the Java compiler delves into extensive static checking to detect as many errors as possible at the compilation

stage. In particular the generated bytecode is guaranteed that all references to objects, methods, and variables are of the appropriate type, that Java 's access control mechanism is not violated, and so on.

- At the application level, the Java security framework provides a broad set of security mechanisms available to applications for implementing a requested security policy. For example, Java RMI requires the installation of a security manager in order to use dynamic class loading; without it, servers could easily attack their clients by sending malicious code that masquerades as a remote stub.

Java's security primitives are largely based on where code originated from. Thus, security policy files grant permissions based on where code was loaded from, and location is specified using URLs. The Java class loaders have a a crutial responsability here; a vulnerability [8] in the `AppletClassLoader` would for example allow a remote user to connect to local sockets on the target system.

Executable code is categorized based on its URL of origin and the private keys are used to sign the code. The security policy maps a set of access permissions to code characterized by particular origin/signature information. Protection domains can be created on demand and are tied to code with particular `Codebase` and `SignedBy` properties.

NOTE

The class `java.security.CodeSource` extends the concept of a codebase to encapsulate not only the location (URL) but also the certificate chains that were used to verify signed code originating from that location.

3.4 Discussion

RPC-based technologies are connection-oriented, since first the connection must be established (by requesting a reference to a remote object via the name server) and then used (throughout the interaction with the remote object). Code that uses remote objects should thus be cluttered with many checks in order to deal with possible network communication failures.

Nevertheless, benefits of mobile code include fault-tolerance, service customization, code deployment and maintainance.

[8] SecurityTracker 1018428.

Chapter 4

State of the Art

You have to design distributed systems with the expectation of failure.

Ken Arnold[1]

While much research efforts have been devoted to the field of distributed programming most published studies focus on aspects related to inter-process communication while the present work primarily considers aspects related to the dynamic relocation of code fragments.

In this chapter we concentrate our analysis on studies and projects dealing specifically with the dynamic rebinding mechanism in a distributed environment. First, we present two language calculi and sketch relevant aspects of their formalization; second, we look at four distributed languages and evaluate their respective solution in relation with our approach presented in the introduction (see Section 1.1).

Most research works related to this project have focused on functional programming languages, in particular on the ML language family.

The key ideas presented in this thesis have been influenced by the lambda calculi proposed by Bierman and al. [9] and the language Obliq designed by Luca Cardelli [28] and share the same objectives which are to improve the design of programming languages for distributed computation.

[1]Ken Arnold is an inventor of Jini and a designer of JavaSpaces.

4.1 Calculi

4.1.1 λ_{marsh} Calculus

Bierman and al. [9, 10] identify and formalize core mechanisms for dynamic rebinding in distributed ML-like languages. Their study relies on language features like high-order functions for expressiveness, call-by-value reduction for simple evaluation orders and static typing for early error detection.

They formulate the key question as follows:

> *"When a value is moved between scopes, how can the user specify which identifiers should be rebound and which should be fixed ?"*

To formalize their answer Bierman and al. propose the λ_{marsh} calculus, a calculus based on a simply-typed call-by-value (CBV) lambda calculus [31] which contains primitives for packaging a value such that some of its identifiers are fixed to bindings in the current context while others will be rebound when unpacked in the new scope. They also sketch an extension of the λ_{marsh} calculus with support for network communication.

In order to evaluate their calculi Bierman and al. consider the concrete case of a network-transmitted function value which might contain identifiers for:

- ubiquitous standard library calls which should be rebound at the destination;
- application-specific location-dependent library calls which should also be rebound at the destination;
- application code which is not location-dependent but should be rebound rather than sent; and
- other let-bound application values, which should be sent with it.

The λ_{marsh} calculus provides high-level representations of marshalled values and primitives to manipulate them (see Figure 4.1). For simplicity, the λ_{marsh} calculus focuses on the case of rebinding application-specific libraries which the authors consider as the more interesting case.

The **mark** primitive allows the programmer to cleanly and fexibly notate which definitions should be fixed and which should be rebindable; definitions occuring within the named context **mark** M are copied into a package value. In that way a different context may be choosen for each **marshal** and **unmarshal** operation.

$$
\begin{array}{lll}
t & ::= & \ldots \\
 & & \mid \textbf{mark}\, M\, \textbf{in}\, t \\
 & & \mid \textbf{marshal}\, M\, t \\
 & & \mid \textbf{unmarshal}\, M\, t \\
 & & \mid \textbf{marshalled}\, \Gamma\, t \\
T & ::= & \ldots \\
 & & \mid Marsh\, T
\end{array}
$$

Figure 4.1: λ_{marsh} extended language syntax.

An expression **marshal** $M\, t$ first reduces expression t to a value v and then packages v with all the bindings within the nearest enclosing context **mark** M; these bindings are essentially static.

The typing rules in Figure 4.2 define the typing derivations dealing with the types $Marsh\, T$ and T.

$$
\frac{\Gamma \vdash t : T}{\Gamma \vdash \textbf{mark}\, M\, \textbf{in}\, t : T}\ \text{(T-Mark)}
\qquad
\frac{\Gamma \vdash t : Marsh\, T}{\Gamma \vdash \textbf{unmarshal}\, M\, t : T}\ \text{(T-Unmarsh)}
$$

$$
\frac{\Gamma \vdash t : T}{\Gamma \vdash \textbf{marshal}\, M\, t : Marsh\, T}\ \text{(T-Marsh)}
\qquad
\frac{\Gamma' \vdash v : Marsh\, T}{\Gamma \vdash \textbf{marshalled}\, \Gamma'\, v : Marsh\, T}\ \text{(T-Marshed)}
$$

Figure 4.2: λ_{marsh} typing rules.

Identifiers of v not bound within the mark are recorded in a type environment within the packaged value, which has form **marshalled** $\Gamma\, v$, and can be rebound.

The authors conclude their work by stating that they still need to provide a type system for λ_{marsh}; in particular, their would like to statically prevent all run-time errors for programs that make only simple use of the **marshal** and **unmarshal** primitives.

4.1.2 Calculus of Module Systems

In their lazy module calculus *CMS*^{l,v} Ancona and al. [4, 41] present a simple and powerful mechanism for dynamic rebinding of mixin modules. Mixin modules are mutually recursive modules allowing redefinition of components.

CMS^{l,v} is an extension[2] of *CMS*, a calculus of module systems previously developped by Ancona and Zucca. The *CMS*^{l,v} calculus supports redefinition of virtual components — a feature analogous to method overriding in object-oriented languages — together with the interleaving of execution and configuration phases.

The *CMS*^{l,v} syntax defines two basic expressions which are standard in module calculi [39]:

- A basic module $[\iota; o; \rho]$ consists of three mappings: a mapping ι from variables into input names, a mapping o from output names[3] into expressions and a mapping ρ from local variables into expressions.
- A basic configuration $[\iota; o; \rho \mid e]$ consists of a basic module and an expression e, also called program.

Thus, in module `M1 = module { virtual X=1; virtual Y=X+1; }` — written in some hypothetical module language — `X` and `Y` are virtual components of module `M1` whose semantics is given by the basic module $M_1 = [x : X, y : Y; X : 1, Y : x + 1;]$

Since *CMS*^{l,v} supports dynamic rebinding, it provides a natural formal basis for modeling marshalling and update.

In the following basic configuration

```
M2 = module {
  virtual X=1;
          Y=2; (*frozen*)
          Z=3; (*frozen*)
} with main marshal X+Y+X+Z rebind Y;  (*Y deferred*)
```

the main expression `X+Y+X+Z` depends on the three components `X`, `Y` and `Z`. It is however possible to specify a list of components which have to be rebound when the expression will be eventually unmarshalled; in the above example, a new definition of `Y` must be provided before `M2` can be evaluated.

[2] l and v stand respectively for "linking" and "virtual" in *CMS*^{l,v}.
[3] Input names are called virtual if they are output names as well, deferred otherwise.

In *CMS*^{l,v} the above marshalled expression is translated into an expression e_2 whose value results from the application of the operator `marshal` to the module configuration for `M2`:

$$e_2 = \texttt{marshal}([x : X, y : Y; X : 1, Z : z; z : 3 \mid x + y + x + z])$$

When executing the main expression, its actual context is represented by the module $[x : X, y : Y; X : 1, Y : y, Z : z; y : 2, z : 3]$, where component `Y` is removed since it must be rebound.

In the example below, the evaluation of the `unmarshal` expression `M3` depends on the update of components `Y`, `X` and `Z`:

```
M3 = unmarshal result(M2) bind Y:4, X:5, Z:6;
```

As before `M3` is translated into an expression e_3 whose value results from the application of the extraction operator $\uparrow$ to the module configuration below:

$$e_3 = (\texttt{unmarshal}(e_2) \ \backslash_X \ \backslash_Z + [; Y : 4, X : 5, Z : 6;])\uparrow$$

The sum operator $+$ combines the unmarshalled expression e_2 — from which `X` and `Z` are removed using the delete operator — with the module expression $[; Y : 4, X : 5, Z : 6;]$.

After evaluating $\texttt{unmarshal}(e_2)$, e_3 reduces to

$$e_3 = ([x : X, y : Y; X : 1, Z : z; z : 3] \mid x + y + x + z] \ \backslash_X \ \backslash_Z + [; Y : 4, X : 5, Z : 6;])\uparrow$$

Now, in the main expression $x + y + x + z$, the first occurence of x is bound to 1; then y is bound to 4 and the second occurrence of x is bound to 5. Finally, z is bound to 3 (the update of Z has no effect); expression e_3 thus evaluates to 13.

The *CMS*^{l,v} calculus allows an executing program to use virtual variables and thus to refer to components whose definition may change by applying module operators. It also provides a simple mechanism to express interaction of execution by combining module configurations with evaluation operators.

Finally, Ancona and al. [4] mention the integration of lazy module calculi with mobile code as future work; this mainly concerns the design of calculi for the dynamic reconfiguration where code to be used for reconfiguring the running program can migrate from a different process.

4.2 Languages

4.2.1 Obliq

Obliq [25, 26, 27, 28] is a lexically scoped, dynamically typed, prototype-based language, designed for distributed object-oriented computations. Computations in Obliq are network transparent — i.e. site independent — and the code mobility is managed explicitly at the language level.

To support network transparency, Obliq extends the static scope to the network: free variables in the transmitted code can refer to objects from the origin site. Obliq objects have state and are local to a site, *network references* to objects can be transmitted from site to site without restrictions. Object migration is explicit and can be encoded using cloning and redirection. Obliq computations can thus roam over the network, while maintaining network connections (Figure 4.4).

The distributed computation mechanism of Obliq is based on Modula-3 network objects [13]. Network objects are implemented as a library extension of Modula-3 enabling remote procedure call services. Objects are not mobile, but they can be passed by value or by reference.

NOTE

An object in Obliq is a collection of attributes (e.g. `{x => 3, y => 4}`); an attribute can be protected against modification, cloning or aliasing from outside the object using the `protected` keyword. Since there are no classes in Obliq the language provides special operations on objects:

- *Selection/invocation* E.g. `p.x` selects attribute `x` of object `p`.
- *Updating/overriding* E.g. `p.x <- 4` assigns the value `4` to attribute `x` of object `p`.
- *Cloning* Cloning an object creates a shallow copy: immediate values of attributes are copied while structured values introduce sharing.
- *Aliasing* Attributes can be redirected to attributes in other objects; all operations on alias objects are forwarded. An alias itself can point to another alias. This is the way alias-chains are constructed [27].

As synchronization mechanisms a mutex object is built into every Obliq object; it serializes the execution of field selection, method invocation, update, and cloning operations on its object.

At the source site an object `{m1 => a1, ...}` with name `"myObj"` can be initially shared through a name server `Namer`:

```
net_export("myObj", Namer, {m1 => a1, ...});
```

At the target site the network reference `o` is obtained from the server `Namer` using the name `"myObj"` and the method `m1` is invoked remotely.

```
let o = net_import("myObj", Namer);
o.m1(b);
```

- Obliq data is network-transparent: immutable data may be duplicated, but state is never automatically duplicated (cloning allows explicit state duplication).
- Obliq computations are network-transparent: their effect on free variables is the same no matter where they execute. However, procedures may receive different parameters at different sites.

Object migration in Obliq is implemented using the cloning and aliasing operations; it is presented by the author to be network-transparent. However, Nestmann, Merro and others have shown that the original scheme for object migration is not transparent [86] and suggest an amended semantics for Obliq [78] where object migration is proved to be transparent to clients.

Distributed lexical scoping in Obliq is the key mechanism for managing distributed computations. An object may become accessible over the network either by the mediation of a name server, or by simply being used as the argument or result of a remote method.

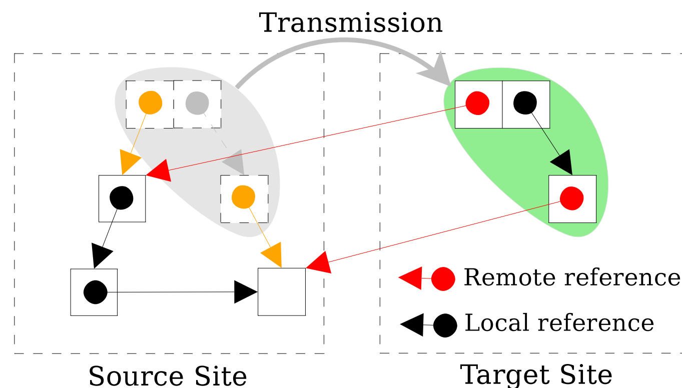


Figure 4.3: Value transmission in Obliq.

When computations are transmitted, lexically scoped free identifiers retain their bindings to the originating sites. Through these free identifiers, migrating computations can maintain connections to objects and locations residing at various sites.

In fact, Obliq extends a familiar language feature, *lexical scoping*, to a distributed context. The approach is analogous but more general than the extension of local procedure call to remote procedure call.

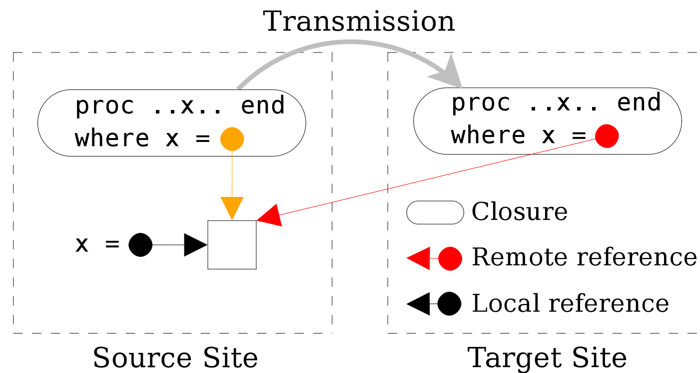


Figure 4.4: Closure transmission in Obliq.

The main technical issue is to find a meaning for *higher-order distributed computations*: what happens to the free identifiers of network-transmitted procedures ? Obliq takes the view that such identifiers are bound to their original locations and network sites, as prescribed by lexical scoping (Figure 4.4).

NOTE

Courtney has developed Phantom [33], an interpreted, strongly typed, higher-order language, which uses a class-based object model. The core language is based on the syntax and semantics of Modula-3 and the distribution model of Phantom uses the same basic model as Obliq [27].

Courtney's approach to security is based on strict lexical scoping in the context of distribution and higher-order functions. When an executing unit receives a function from a remote site, it can perform a single, static check to ensure that all free identifiers in the code for the function have a corresponding entry in the set of bindings received with the function.

The Obliq interpreter can easily be embedded in a Modula-3 application and its functionality can be extended either with new built-in operations that invoke Modula-3 code or with new Obliq library packages.

The Obliq implementation [25, §2.4] is split into several packages in order to generate minimal Obliq interpreters that can act as (relatively) small network servers: the `obliqrt-ip` package implements the Obliq runtime kernel, the `obliqparse-ip` package contains routines to parse and evaluate Obliq phrases from a reader (REPL), the `obliqprint-ip` package performs pretty-printing and, finally, the `obliq-ip` package brings together the packages that are needed to build a stand-alone Obliq interpreter.

4.2.2 ML3000

ML3000 [40] is a distributed type-based dialect of ML, which supports dynamic typing, user-definable marshalling, built-in objects (similar to Obliq objects, see Section 4.2.1) and structural type equivalence.

ML3000 relies on the mechanism of *Dynamics*[4] to generate run-time type informations and follows the fingerprint approach taken in Modula-3 network objects for their transmission.

Futhermore, Duggan and Przybylski provide mechanisms for user-defined marshalling for supporting type polymorphism in ML3000 such that instances of (un-)marshalling operations for a specific type can be registered by the programmer.

Rather when relying on ML polymorphic functions which are second-class, ML3000 provides built-in objects — which may contain polymorphic methods — as abstractions for network connections as is done in distributed object systems. Similarly to Obliq, an object is transmitted as a network reference that is unmarshalled at the target location as a proxy calling back to the original object.

The compiler front-end – including type-checker — was written from scratch and the ML3000 backend is based on the Moscow ML compiler with modifications to support RTTIs. The bytecode interpreter is itself based on the CAML Light interpreter with modifications to support RTTIs. While Obliq uses Modula-3 network objects as transport mechanism, ML3000 uses the ILU multi-language remote object system [32].

The ML3000's backend allows transmission of code segments as parts of closures; a separate string constant is generated for every code segment and is saved as part of the global data table in the compiler output. Marshalling of closures introduces the greatest complications; in order to marshall a closure, its environment slots must be marshalled. Every instruction which extend the environment is modified to provide the needed type information to be added to the global data table.

The main challenge concerns global values referenced in the code of closures: their type index in one address space becomes meaningless when the code with references on those values in transmitted to another address space.

In resume Duggan and Przybylski provide with ML3000 a distributed type-based extension of ML whose design approach is similar to the one of the Obliq language. Built-in objects serve as distribution abstractions and RTTIs are added to the transmitted code.

---

[4] The primitives `dynamic` and `typecase` are used for packing a value with its type respectively for inspecting a type in a dynamic.

4.2.3 MobileML

MobileML [56] is a ML-based programming language which features transparent migration (see Section 3.1.2) of mobile code and dynamic linking with distributed resources by means of *contexts* [55, 106, 107]. Hashimoto and al. introduce the notion of contexts to succintly describe the interactions between mobile code and environments at the target location. The semantic model for contexts is based on Plotkin's λ_v-calculus and *tuple spaces*.

A context is a program expression with holes in it. The basic operation for a context is to fill its hole with an expression by capturing free variables. When the code is filled, the free variables in the code are dynamically bound to the variables defined in the context.

For example, the agent expression `agent(e)` specifies the code region which migrates to the context "`//hal::2001/k`" using the instruction `go`.

```
let name = "HAL" in
agent(go "//hal::2001/k"; print_string("Hello, "^name));
```

Then, variable `name` is dynamically bound to value "`HAL`" in the program context "`//hal::2001/k`".

NOTE

In his implementation of Facile [68, 69], a dialect of MobileML, Knabe uses *proxy structures* (vs. *contexts* in MobileML) in order to specify the variables which become bound at a remote site. *Proxy structures* are generated from a remote signature which specifies names and types of the values provided at the remote site and are treated as if they were local structures.

In case specifications of the proxy structures are modified, it is necessary to get the new remote signatures and to recompile programs which reference them. MobileML does not need such recompilation since such consistency is dynamically checked.

In order to reduce the space cost of generating their transmissible representation Facile performs compile-time marshalling only for those functions which have been explicitly annotated by the programmer to be potentially transmissible.

The authors have implemented an experimental interpreter system as a first step of the full-flegded language system.

4.2.4 PLAN

PLAN [52, 59] is a domain-specific, strongly typed, simple functional language for programs which form the packets of an active network. It is based on a subset of ML with some primitives to support remote evaluation (see Section 2.2.3).

A PLAN application is composed of a series of PLAN packets which are injected into the active network through a port connected to the local PLAN interpreter. Mobility of PLAN packets and the remote evaluation of chunks are at the heart of the execution of PLAN programs. PLAN's expressiveness is limited to permit actives nodes to evaluate PLAN programs without requiring authentification[5]: all PLAN programs are statically typeable and are guaranteed to terminate[6].

A PLAN packet encapsulates a *chunk* and several control fields. A chunk consists of the PLAN code, an entry point function within that code, and the function's arguments. The code itself contains a series of definitions which bind names to functions, values and exceptions where the names of the services available at the source location form the initial bindings in the namespace. The arguments are evaluated locally in a call-by-value fashion and the actual evaluation of the function call is delayed until the packet arrives at its destination site.

Chunks [81] are first-class values whose execution can be initiated by the core service `eval`. The function call takes place in an environment where all top-level bindings are available; PLAN departs here from the discipline of static scoping adopted by others systems.

Nevertheless, the chunk abstraction provides an elegant way to express common network mechanisms: on one side chunks representing micro-protocols can be arbitrarily ordered and composed to create more complex protocols, on the other side chunks are carried in the packet themselves and thus permit asynchronous protocol adaptation.

Moore and al. [81] report two drawbacks with their implementation: the evaluation of PLAN programs is quite costly because it is based on bytecode interpretation; also the space cost of carrying the code in the packet needs to be reduced.

As future work, the authors see a promising approach in the addition of language-level *remote references*. Since all PLAN values (including chunks) are immutable, that would allow further space savings as some of a chunk's bindings could be savely cached.

[5] Authenticating every active packet results in unacceptable performance degradation.
[6] So far they only call node-resident services that terminate.

4.3 Discussion

The language calculi from Bierman and al. and Ancona and al. (Section 4.1) rely on different binding mechanisms to formalize the marshalling and the update of relocated expression values: the **mark** primitive in λ_{marsh} defines a named context with definitions to be rebound while $CMS^{l,v}$ relies on lazy modules to model marshalling and update.

λ_{marsh}, the language calculus from Bierman and al. (Section 4.1.1), relies on the same language features — high-order functions, call-by-value reduction and static typing — as in our work. However, while their calculus provides high-level representations of marshalled values and primitives to manipulate them, our approach, based on lambda abstraction, requires only minimal library support to delimit the context of the remote code evaluation. Futhermore, Bierman and al. do not describe the choosen representation for the marshalled values and their calculus is still missing a type system for static typing.

$CMS^{l,v}$, the module calculus from Ancona and al. (Section 4.1.2), supports the redefinition of virtual components in dynamically reconfigurable modules. Unfortunately, the integration of its rebinding mechanisms with mobile code is only mentioned and left as future work.

The programming languages examined in the context of this work (Section 4.2) provide more concrete solutions to express higher-order distributed computations.

First, the prototype language Obliq (Section 4.2.1) supports network transparent computations through distributed lexical scoping and network references to local Obliq objects. However, since Obliq is a DSL embedded in a Modula-3 system, it represents a niche solution for dealing with mobile code in a distributed environment.

Second, the language ML3000 (Section 4.2.2) introduces the operation **dynamic** — a feature similar to **mark** in λ_{marsh} — for packing transmitted values with their type. ML3000 relies on built-in objects to abstract over network connections in a way similar to network references in Obliq.

Third, the ML-based language MobileML (Section 4.2.3) provide the primitive `agent` to deal with the transparent migration of mobile code (discussed in Section 3.1.2), a feature not retained in our solution.

Finally, the domain-specific language PLAN (Section 4.2.4) support the transmission of chuncks — immutable first-class function values — whose remote evaluation takes place in a global environment in contrast to the static scoping rule adopted by other systems such as Obliq. Furthermore, the authors mention as future work the addition of remote references in order to reduce the amount of transmitted data.

Chapter 5

Programming Examples

Programming is usually taught by examples.

Niklaus Wirth[1]

In this chapter we illustrate the usage of detached closures with several programming examples written in the language Scala. Following the client-server paradigm, the presented Scala applications consist of a client process and a server process interacting through different communication mediums.

The first example presents the marshalling of detached closures on a single machine; the local file system serves as communication medium.

The second and third examples introduce more elaborated use cases of distributed lexical scoping and rely on typed channels respectively on remote actors for client-server communication.

The fourth example is adapted from a compute server example originally written in Obliq.

Finally, the two last examples build upon more concrete scenarii and aim to demonstrate the strengths of code parametrization based on lambda abstraction in a distributed programming environment.

---

[1] Niklaus Wirth was honored with the ACM Turing Award 1984 for his work in the area of compiler construction; he is the designer of Pascal, Modula-2 and Oberon.

5.1 Marshalling Example

We start with a client-server console application illustrating the marshalling of detached closures on a single machine. On one side, the client application serializes a detached closure and writes it to some local file; on the other side, the server application reads the serialized data back and executes the parametrized code applied to the interactively provided arguments.

For simplicity, we use the un-/marshalling operations `load` and `dump` (Listing 5.1) of the Scala standard library; those operations rely on the serialization mechanism of Java (see Section 3.3.3) and save both the argument and its associated run-time type information represented as a Scala type manifest and provided automatically by an implicit parameter.

```
object Marshal {
  def dump[A](o: A)(implicit m: Manifest[A]): Array[Byte] = ...
  def load[A](a: Array[Byte])(implicit m: Manifest[A]): A = ...
  //...
}
```

Listing 5.1: Type-safe marshalling in Scala.

The server application (Listing 5.2) implements a simple REPL which waits for user input (interaction details are hidden in trait `ServerConsole`). The user can either load and execute (lines 8-9 and lines 15-16) a client-marshalled function using the command "`f <args>`" or he/she can exit the interactive server shell with "`quit`". The auxiliary function `signatureOf` (line 6) is called by the server process to determine the operation to be performed and to read the expected arguments.

```
object Server extends ServerConsole with ManifestParser {
2  private var bytes: Array[Byte] = null
  def mainBody(args: Array[String]) {
4    val data = readData()
    if (data != null) bytes = data
6    if (bytes != null) signatureOf(bytes) match {
      case "Int=>Int" =>
8        val x = args(0).toInt
        val f = load[Int => Int](bytes)
10        println("f("+x+")="+f(x))
        _notify()
```

```
12        case "(String,Int)=>String" =>
            val x = args(0)
14          val y = args(1).toInt
            val f = load[(String, Int) => String](bytes)
16          println("f(\""+x+"\", "+y+")="+f(x, y))
            _notify()
18        case m =>
            println("error: unknown manifest "+m)
20  } } }
```

Listing 5.2: Marshalling application (server).

```
    object Client {
2     var y = 1
      def main(args: Array[String]) {
4       var z = 2
        println("y="+y+", z="+z)
6       writeData(dump(detach((x: Int) => x + y + z)))
        _wait()
8       println("y="+y+", z="+z)
        y += 10
10      _wait()
        println("y="+y+", z="+z)
12      z += 20
        _wait()
14      println("y="+y+", z="+z)
        writeData(dump(
16        detach((s: String, x: Int) => {z += 1; s + (x + y + z)})
        ))
18      _wait()
        println("y="+y+", z="+z)
20  } }
```

Listing 5.3: Marshalling application (client).

The client application (Listing 5.3) first transforms the detached function literal

```
detach((x: Int) => x + y + z)
```

into a byte array, writes it to some file (line 6) and waits for the server

process to load and execute the marshalled code. Second, the client prints out the actual values for the two captured variables y and z (line 8). Then, both variables y and z are updated such that further executions of the closure code in the server thread return different results.

Later on (line 15) we choose to update the code to be executed in the server process and perform another marshalling operation with the detached function literal

```
detach((s: String, x: Int) => z += 1; s + (x + y + z))
```

as argument, where s and x are bound variables and y and z are free variables.

Both the output in the client console and the user interaction with the server process are given in Figure 5.1; program execution terminates when the user enters the command "quit" in the server console.

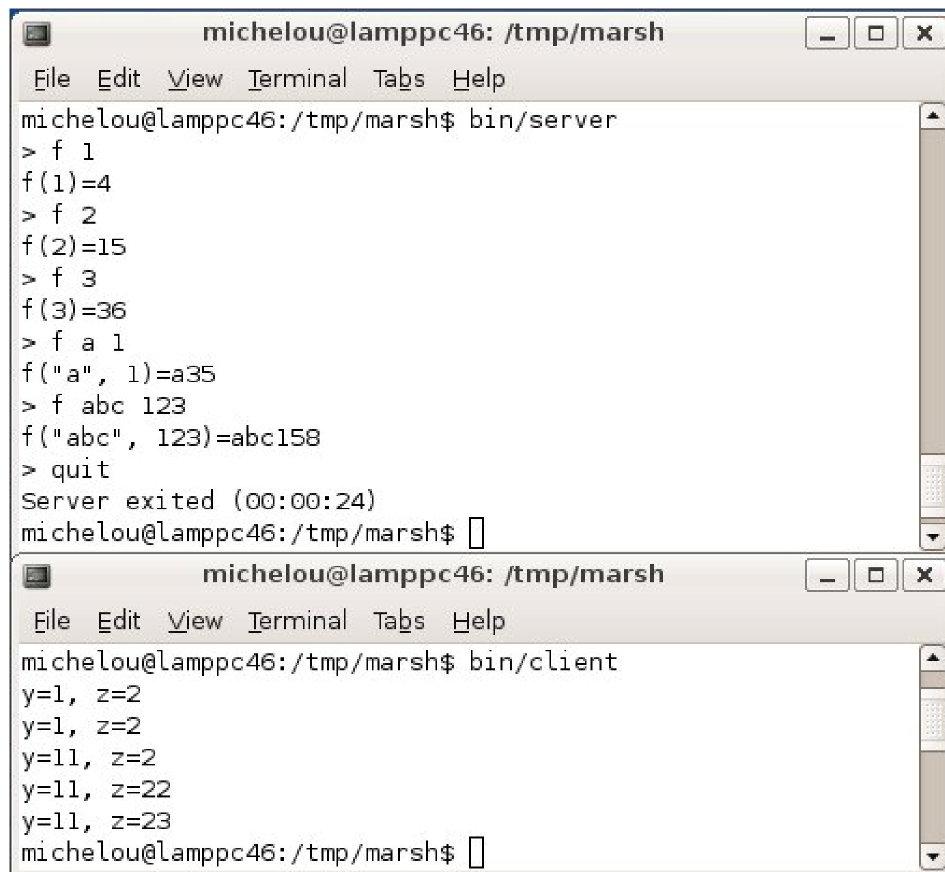


Figure 5.1: Marshalling application (consoles).

5.2 Basic Example using Typed Channels

In the following distributed client-server (C/S) application the client process (Listing 5.7) and the server process (Listing 5.6) communicate through a typed channel.

A typed channel provides a type-safe abstraction for network communication and is implemented by the two classes `Channel` (Listing 5.4) and `ServerChannel` (Listing 5.5) of the library package `scala.remoting`[2]. The class `Channel` defines basic generic send/receive operations.

```
class Channel(socket: Socket) {
  def this(host: String, port: Int) =
    this(new Socket(host, port))
  val host = socket.getInetAddress.getHostAddress
  val port = socket.getLocalPort
  def receiveInt  = receive[Int]
  def receiveLong = receive[Long]
  //.. more primitive types
  def receive[T](implicit expected: Manifest[T]): T = { /*..*/ }
  def ?[T](implicit m: Manifest[T]): T = receive[T](m)
  def send[T](x: T)(implicit m: Manifest[T]) { /*..*/ }
  def ![T](x: T)(implicit m: Manifest[T]) { send(x)(m) }
  def close() { socket.close() }
}
```

Listing 5.4: The `Channel` class (Scala API).

The class `ServerChannel` represents the second endpoint of the communication channel and accepts incoming client requests on a given service port. The Scala programmer can define a subclass of `AbstractServerChannel` in order to customize receive operations for user-defined messages.

```
class ServerChannel(p: Int) extends AbstractServerChannel[Channel] {
  def newChannel(s: Socket) = new Channel(s)
}
abstract class AbstractServerChannel[T <: Channel](_port: Int) {
  def this() = this(0) // any free port
  val host = serverSocket.getInetAddress.getHostAddress
  val port = serverSocket.getLocalPort
  //..
```

[2]The current implementation is based on Java sockets and Scala manifests.

```
  protected def newChannel(s: Socket): T
  def accept: T = { /*..*/ newChannel(serverSocket.accept) }
  def close() { serverSocket.close() }
}
```

Listing 5.5: The `ServerChannel` class (Scala API).

On the server side (Listing 5.6) we first create a server channel for accepting incoming requests and then enter the main loop (as for the previous example interaction details are hidden in trait `ServerConsole`). Once communication is established with a client, we receive the detached closure and evaluate it (line 4); the resulting integer value is then sent back to the client (line 12).

```
object Server extends ServerConsole {
2  private def computation(f: Int => Int): Int = {
    //some time-consuming task
4    f(2)
  }
6  def main(args: Array[String]) {
    val server = new ServerChannel(args(0).toInt)
8    loop {
      val client = server.accept
10      val f = client.receive[Int => Int]
      val result = computation(f)
12      client ! result
    }
14    server.close()
  }
16 }
```

Listing 5.6: Basic C/S application (server).

On the client side (Listing 5.7) we first create a channel to communicate with the server, we send a detached closure over it and, finally, we wait for the evaluation result and print the integer value `11111`[3] to the console. The closure code contains several variable references and method calls declared in different contexts:

- Uses of variables include the formal parameter `x` of the anonymous function, the instance variables `yInstVal` and `yInstVar` declared in

[3] `x+yInstVal+yInstVar+zLocVal+zLocVar = 2+10+99+1000+(9998+2) = 1111`.

object `Client` (lines 5-6) and the local variables `zLocVal` and `zLocVar` declared in a try/catch block (lines 13-14);

- Function calls include the library methods `println`, `System.out.println` and `Debug.info`, method **this**`.trace` of the `Client` instance, method `Foo.trace` of the top-level object `Foo` and method `Bar.trace` of the field object `Bar`.

```
object Foo {
  def trace(s: String) { info("[Foo.trace] "+s)}
}
object Client {
  val yInstVal: Int = 10
  var yInstVar: Int = 99
  object Bar {
    def trace(s: String) { info("[Bar.trace] "+s) }
  }
  def main(args: Array[String]) {
    init(args)
    val server = new Channel(host, port)
    val zLocVal: Int = 1000
    var zLocVar: Int = 9998
    server ! detach(
      (x: Int) => {
        println("yInstVal = "+yInstVal)
        this.trace("yInstVar = "+yInstVar)
        Bar.trace("zLocVal = "+zLocVal)
        Foo.trace("zLocVar = "+zLocVar)
        zLocVar += 2
        System.out.println("zLocVal = "+zLocVal)
        Debug.info("zLocVar = "+zLocVar)
        x + yInstVal + yInstVar + zLocVal + zLocVar
      })
    val result = server.receiveInt
    println("result received: " + result)
  }
  private def trace(s: String) { info("[Client.trace] "+s) }
}
```

Listing 5.7: Basic C/S application (client).

Building RMI-based applications typically consists of three main phases

which include the development phase (build process and configuration setup), the installation phase (deployment and customization) and the execution phase (test and production environments).

- In this example the distribution files generated during the build process include three Java archive files (client, server and deployment files), one preconfigured Java security policy file and two shell scripts to launch the client and server applications.
- These files are then installed according to the chosen configuration settings (targeted to one, two or three different machines). Software requirements for the client and server applications are a recent Java run-time environment (version 1.5 or newer) and a modified Scala distribution (version 2.6 or newer with the `detach`-extension[4]).
- The two launch scripts (Listing 5.8 and Listing 5.9) can then be executed (in that order!). The server application runs forever and waits for incoming client requests.

```
#!/bin/sh
${JAVACMD:=java} \
-Xbootclasspath/a:$SCALA_HOME/lib/scala-library.jar \
-Dscala.remoting.logLevel=${LOGLEVEL:=warning} \
-Djava.security.manager \
-Djava.security.policy=$GENDIR/java.policy \
-Djava.rmi.server.codebase=$DEPLOYDIR/basic_deploy.jar \
-Djava.rmi.server.useCodebaseOnly=true \
-jar $GENDIR/basic_server.jar ${PORT:=8889}
```

Listing 5.8: Basic C/S application (server script).

```
#!/bin/sh
${JAVACMD:=java} \
-Xbootclasspath/a:$SCALA_HOME/lib/scala-library.jar \
-Dscala.remoting.logLevel=${LOGLEVEL:=warning} \
-Djava.security.manager \
-Djava.security.policy=$GENDIR/java.policy \
-jar $GENDIR/basic_client.jar 127.0.0.1 ${PORT:=8889}
```

Listing 5.9: Basic C/S application (client script).

[4]Implementation details are presented later in Section 7.2.

A Java policy file (Listing 5.10) must be specified in both launch scripts (Listing 5.8 and Listing 5.9); it defines two groups of permission rules (or grant entries) : the first group of rules applies to class files loaded from any location and grants access permissions to Java sockets and library-defined properties; the second group of rules applies to the application class files and grants access permissions to RMI specific ressources.

```
grant {
    permission java.net.SocketPermission "*:80", "connect,accept,listen";
    permission java.net.SocketPermission "*:1024-", "connect,accept,listen";
    permission java.util.PropertyPermission "scala.remoting.logLevel", "read";
    permission java.util.PropertyPermission "scala.remoting.port", "read";
};
grant codeBase "@PROJECT_LIB_BASE@" {
    permission java.lang.RuntimePermission "getClassLoader";
    permission java.util.PropertyPermission "java.rmi.server.codebase", "read";
    permission java.util.PropertyPermission "java.rmi.server.hostname", "read";
    permission java.util.PropertyPermission "sun.rmi.dgc.server.gcInterval",
        "read, write";
};
```

Listing 5.10: Basic C/S application (policy file).

Both the client and server applications can be executed with differents levels [5] of logging information.

The two console outputs in Figure 5.2 correspond to the normal execution (`LOGLEVEL=warning`) of the shell scripts presented in Listing 5.8 and Listing 5.9. The instructions `println` on line 17 and `System.out.println` on line 22 (Listing 5.7) write to the server console while the instruction `println` on line 27 writes to the client console.

In Figure 5.3 we add some logging information to trace the execution of the client and server applications by setting the environment variable `LOGLEVEL` to value `info`.

For instance, the three instructions **`this`**`.trace`, `Foo.trace` and `Bar.trace` (lines 18-20, Listing 5.7) write to the client console since they are application-specific, while the instructions `println`, `System.out.println` and `Debug.info` (lines 17, 22 and 23) write to the server console since those three methods are defined in the Scala standard library.

Unlike the above logging information which is generated by instructions appearing in the user code, the three notifications starting with

[5] Class `Debug` in package `scala.runtime.remoting` currently defines five logging levels `ERROR`, `WARNING`, `VERBOSE`, `INFO` and `SILENT`.

`unreferenced` in the client console mean that the specified remote objects are no more referenced (their binding can thus be removed from the remote registry).

```
michelou@lamppc46: /tmp/basic
File Edit View Terminal Tabs Help
michelou@lamppc46:/tmp/basic$ bin/server
> yInstVal = 10
zLocVal = 1000
quit
Server exited (00:00:17)
michelou@lamppc46:/tmp/basic$
```

```
michelou@lamppc46: /tmp/basic
File Edit View Terminal Tabs Help
michelou@lamppc46:/tmp/basic$ bin/client
result received: 11111
michelou@lamppc46:/tmp/basic$
```

Figure 5.2: Basic C/S application (consoles).

```
michelou@lamppc46: /tmp/basic
File Edit View Terminal Tabs Help
michelou@lamppc46:/tmp/basic$ env LOGLEVEL=info bin/server
> yInstVal = 10
zLocVal = 1000
(info): zLocVar = 10000
quit
Server exited (00:00:26)
michelou@lamppc46:/tmp/basic$
```

```
michelou@lamppc46: /tmp/basic
File Edit View Terminal Tabs Help
michelou@lamppc46:/tmp/basic$ env LOGLEVEL=info bin/client
(info): [Client.trace] yInstVar = 99
(info): [Bar.trace] zLocVal = 1000
(info): [Foo.trace] zLocVar = 9998
result received: 11111
(info): unreferenced: Client/proxy$3$3bd7bf94:122abd9d1f9:-8000
(info): unreferenced: Client/proxy$5$3bd7bf94:122abd9d1f9:-8000
(info): unreferenced: Client/proxy$4$3bd7bf94:122abd9d1f9:-8000
michelou@lamppc46:/tmp/basic$
```

Figure 5.3: Basic C/S application (consoles, option `info`).

Finally, in Figure 5.4 we trace the execution of the server application and print out additional logging information about operations performed in typed channels by setting `LOGLEVEL` to value `info,lib` before running the server script.

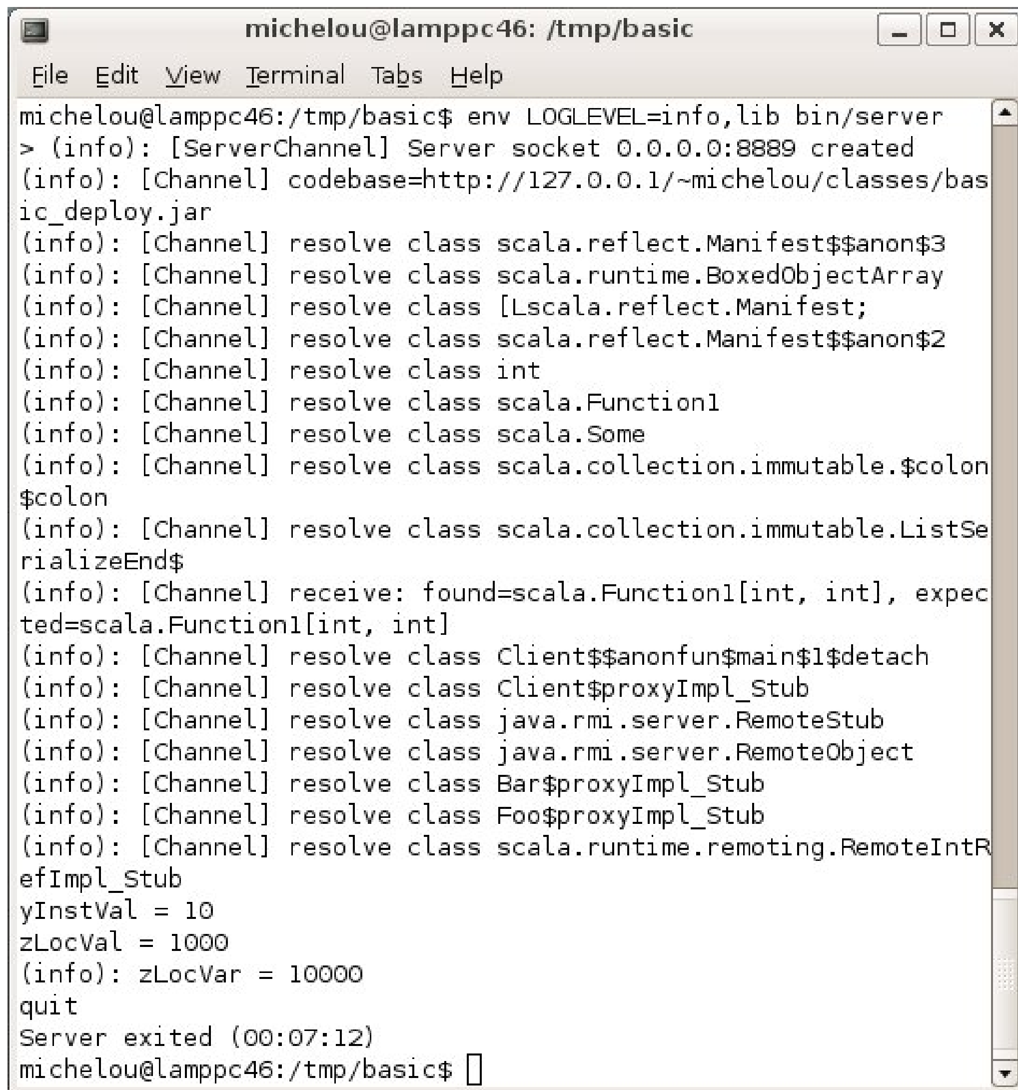


Figure 5.4: Basic C/S application (server console, option `info,lib`).

Similarly, Figure 5.5 presents the displayed logging information when setting `LOGLEVEL` to value `info,lib` before running the client script.

Figure 5.5: Basic C/S application (client console, option `info,lib`).

5.3 Basic Example using Remote Actors

The example presented in Section 5.2 relied on typed channels for client-server communication. The same C/S application can easily be adapted to make use of the Scala Actors library for remote communication (and synchronization).

The server application (Figure 5.11) consists of a remote actor responding on some user-defined port (line 9) and registered with the name `'Server` (line 10). The remote actor reacts on incoming messages of type `Int => Int` (line 13), performs some computation and sends the integer value resulting from the code evaluation back to the sender (line 15).

```
object Server extends ServerConsole {
  private def computation(f: Int => Int): Int = {
    //some time-consuming task
    f(2)
  }
  def main(args: Array[String]) {
    actor {
      classLoader = serverClassLoader
      alive(args(0).toInt)
      register('Server, self)
      loopWhile(isRunning) {
        react {
          case f: (Int => Int) =>
            val result = computation(f)
            sender ! result
        }
      }
    }
  }
}
```

Listing 5.11: C/S application with Scala actors (server).

The client application (Listing 5.12) only differs from the original source code (Listing 5.7) in the manner it connects to the server (line 13) and it receives the result of the remote evaluation (line 26).

```
object Foo {
  def trace(msg: String) { info("[Foo.trace] "+msg)}
```

```
}
object Client {
  val yInstVal: Int = 10
  var yInstVar: Int = 99
  object Bar {
    def trace(msg: String) { info("[Bar.trace] "+msg) }
  }
  def main(args: Array[String]) {
    init(args)
    actor {
      val server = select(Node(host, port), 'Server)
      val zLocVal: Int = 1000
      var zLocVar: Int = 9998
      server ! detach(
        (x: Int) => {
          println("yInstVal = "+yInstVal)
          this.trace("yInstVar = "+yInstVar)
          Bar.trace("zLocVal = "+zLocVal)
          Foo.trace("zLocVar = "+zLocVar)
          zLocVar += 2
          System.out.println("zLocVal = "+zLocVal)
          Debug.info("zLocVar = "+zLocVar)
          x + yInstVal + yInstVar + zLocVal + zLocVar
        })
      react {
        case result: Int =>
          println("result received: " + result)
          Predef.exit(0)
      }
    }
  }
  private def trace(msg: String) { info("[Client.trace] "+msg) }
}
```

Listing 5.12: C/S application with Scala actors (client).

5.4 Compute Server Example

The following example — a very simple compute server — is adapted from Cardelli's paper [28] on distributed mobile computation in Obliq (see Section 4.2.1). The Obliq programmer uses the two primitives `net_export` and `net_import` to export respectively import services.

In Listing 5.13 the string parameter *service* specifies the registration name for the exported object and the string parameter *host* provides the IP address or name of the machine running the desired name server.

Server

```
var q = proc() end;
net_export(service, host,
  { rexec => meth(s, p) q := p; p() end }
```

Client

```
let cs = net_import(service, host);
var x = 3;
cs.rexec(proc() x := x+1 end)
x;   (* is now 4 *)
```

```
q;   (* is now proc() x := x+1 end *)
q();
```

```
x;   (* is now 5 *)
```

Listing 5.13: A compute server in Obliq.

The same example written in Scala consists of a service interface and implementation[6] (Listing 5.14), a server application (Listing 5.15) and a client application (Listing 5.16).

```
@remote
trait ComputeService extends java.rmi.Remote {
  def rexec(p: () => Unit)
}
class ComputeServer extends ComputeService {
  private var qBlk: () => Unit = () => ()
  private var qSet: Boolean = false
  def rexec(p: () => Unit) = { qBlk = p; qSet = true; p() }
```

[6]Method `q` is implemented using at-most-once semantics.

```
  def q() { if (qSet) { qBlk(); qSet = false } }
}
```

Listing 5.14: Compute application (service).

```
object Server extends ServerConsole {
  def main(args: Array[String]) {
    init(args)
    val cs = new ComputeServer
    netExport("computeServer", port, cs)
    println("Server is running")
    loop {
      Thread.sleep(millis)
      cs.q()
    }
  }
```

Listing 5.15: Compute application (server).

On the client side (Listing 5.16) the detached closure is sent for remote evaluation to the server (line 7). In particular, method `println` of the library object `scala.Console` is invoked in the target environment and the printed message will be display in the server console. Thus, the Scala programmer must pass that object as an explicit argument in order to invoke the method `println` in the source environment.

```
object Client {
  def main(args: Array[String]) {
    init(args)
    val cs: ComputeService =
      netImport("computeServer", host, port)
    var x: Int = 3
    cs rexec detach(() => { println("rexec: x="+x); x+= 1 })
    println("x="+x) // is now 4
    Thread.sleep(3000)
    println("x="+x) // is now 5
    exit(0)
  }
}
```

Listing 5.16: Compute application (client).

5.5 Bank Account Example

In the following example we consider the processing of a simple bank order: a client wants to withdraw US $100 from his/her bank account and, if necessary, to credit it with fresh money when it has reached some balance limit.

For simplicity the class `Account` (Listing 5.17) defines just the three operations `deposit`, `withdraw` and `saldo`.

```
class Account {
  private var amount = 0
  def deposit(n: Int): Boolean = synchronized {
    if (0 <= n) { amount += n; true }
    else false
  }
  def withdraw(n: Int): Boolean = synchronized {
    if (0 <= n && n <= amount) { amount -= n; true }
    else false
  }
  def saldo: Int = synchronized { amount }
}
```

Listing 5.17: Bank account application (Account).

The server application (Listing 5.18) manages bank accounts and waits for client requests to process the transmitted sequence of operations and apply them to the client's account identified by an account number and an authentication code.

```
object Server extends ServerConsole {
  private val accounts = Map(
    123456 -> ("secret", new Account),
    654321 -> ("1a9z-A", new Account)
  )
  def main(args: Array[String]) {
    val server = new ServerChannel(args(0).toInt)
    loop {
      val client = server.accept
      val accnr  = client.receiveInt
      val accpwd = client.receiveString
      accounts get accnr match {
```

```
        case Some((pwd, account)) if accpwd == pwd =>
14        client ! true
          val f = client.receive[Account => Unit]
16        f(account)
          client ! account.saldo
18      case _ =>
          client ! false
20    }
    }
22  server.close()
  }
24 }
```

Listing 5.18: Bank account application (server).

The client application (Listing 5.19) first provides an account number (line 5) and an authentication code (line 6) to access the client's account; once access is granted it sends a detached closure (line 8) describing the operations to be executed remotely by the server.

```
object Client extends ClientHelper {
2  def main(args: Array[String]) {
    init(args)
4    val server = new Channel(host, port)
    server ! accnr
6    server ! accpwd
    if (server.receiveBoolean) {
8      server ! detach(
        (a: Account) => {
10        import a._
          withdraw(100)
12        if (saldo < limit) deposit(limit)
          info("saldo="+saldo)       // server console
14        this.trace("saldo="+saldo) // client console
        })
16      val result = server.receiveInt
      println("received result "+result)
18    }
    else
20      println("Authentication error")
    server.close()
```

```
22   }
     private def trace(msg: String) {
24     info("[Client.trace] "+msg)
     }
26 }
```

Listing 5.19: Bank account application (client).

In particular the clause `import a._` on line 10 gives access to the operations `deposit`, `withdraw` and `saldo` defined for the formal parameter `a` of type `Account` and rebound to the actual parameter `account` in the server code (line 13).

5.6 Calendar Example

In this section we consider in more details the code sample presented in the introduction (see Section 1.1). As a reminder the client application looks for the free time slots available on Monday between Bob's agenda and Tom's agenda, two agendas hosted respectively on the client node and the server node.

The object `agendas` (Listing 5.20) defines a minimal class `Agenda` with the two methods `get` and `free` and the operation `freeSlots`. For the sake of brevity the implementation details of `agendas`[7] are hidden in an auxiliary object.

```
object agendas {
  type Day = ‘some Day‘
  type Time = ‘some Time‘
  val Time = ‘some Time‘
  type Entry = ‘some Entry‘
  val Entry = ‘some Entry‘
  type Slots = ‘some Slots‘
  class Agenda(e: Entry*) extends ‘some Agenda‘(e: _*) {
    def get(day: Day): List[Entry] = ‘some get‘(day)
    def free(day: Day): List[Entry] = ‘some free‘(day)
  }
  def freeSlots(a1: Agenda, a2: Agenda, day: Day): Slots =
    ‘some freeSlots‘(a1, a2, day)
  implicit def pair2time(p: (Int, Int)) = Time(p._1, p._2)
  implicit def int2time(h: Int) = new Time(h)
}
```

Listing 5.20: Calendar application (agendas).

The server application (Listing 5.21) waits for incoming client requets, executes the transmitted closure code (line 13) and sends the evaluation result — an array of agenda entries — back to the client.

```
object Server extends ServerConsole {
2   val agendaTom = new Agenda(
      Entry("Mon", "Faculty meeting", (15, 20)),
4     Entry("Tue", "Scala meeting", 14),
      Entry("Tue", "PL lunch", (12, 30)),
```

[7] Source code in Figure 5.20 is a valid Scala program.

```
6     Entry("Wed", "ACIDE meeting", 17)
    )
8   def main(args: Array[String]) {
      val server = new ServerChannel(8888)
10    loop {
        val client = server.accept
12      val f = client.receive[Agenda => Slots]
        val result = f(agendaTom)
14      client ! result
      }
16    server.close()
    }
18 }
```

Listing 5.21: Calendar application (server).

On the client side (Listing 5.22) we first send the request to the server, then we wait for the evaluation result of the remotely executed code and, finally, we print out the list of free time slots.

```
  object Client extends ClientHelper {
2   val agendaBob = new Agenda(
      Entry("Mon", "Java Course", (8, 15), 45),
4     Entry("Mon", "French Course", 15, 90),
      Entry("Tue", "Scala meeting", 14),
6     Entry("Tue", "PL lunch", (12, 30)),
      Entry("Tue", "Compiler Course", (14, 15), 45)
8   )
    def main(args: Array[String]) {
10    init(args)
      val server = new Channel(host, port)
12    var day = "Mon"
      server ! detach(
14      (a: Agenda) => freeSlots(a, agendaBob, day)
      )
16    val result = server.receive[Slots]
```

Listing 5.22: Calendar application (client).

The free time slots for Monday are listed below; an appointment between Bob and Tom can now be fixed given those time intervals.

```
result =
   Mon 00:00-08:15: <free>
   Mon 09:00-15:00: <free>
   Mon 16:30-24:00: <free>
```

5.7 Discussion

The examples presented in this Chapter all show how straightforward the new programming abstraction interacts with standard Scala code. And since we are concerned with distributed programming we also have to configure the individual applications for their respective execution environments.

Chapter 6

Programming Abstractions

In a distributed language a closure may escape not only the scope of its free identifiers, but even their address space.

Luca Cardelli[1]

Hilfinger [60] provides a short history of major abstraction mechanisms in programming languages, with an emphasis on procedure and data abstraction. This history does not mention control abstraction, although the mechanisms for control abstraction are present in early functional languages such as Lisp.

Control abstraction has been used in several sequential languages to support data abstraction. For example, CLU iterators (or generators) are a limited form of control abstraction that allows the user of an ADT to operate on the elements of the type without knowing its underlying representation. Both abstractions are fundamental tools for modular programming.

Data abstraction allows the programmer to separate the interface and the (possibly several) implementation(s) of an ADT, and to control visibility over the implementation details. Within the context of a single running application, the subtleties of resource usage and abstraction safety are much better understood as for distributed applications. Indeed, when code (or data) is transmitted between programs running on different machines, the question about what kinds of guarantees a language infrastructure should offer arises inevitably (see Section 3.2).

[1]Luca Cardelli is well-known for his research in type theory and operational semantics; he also implemented the first ML compiler in 1980.

Control abstraction allows the programmer to abstract the common parts of a repetitive code pattern while relying on the caller to provide the parts that differ. Thus, similarly to data abstraction, which hides the implementations of an ADT from the user of the type, control abstraction hides the exact sequencing of operations from the user of the control construct. In particular, it increases the expressiveness of a language by providing a means to extend the set of available statements. For instance, while Java is pretty good at data abstraction, it is less flexible with respect to control abstraction since it doesn't support closures (see Section 2.1.2).

Our primary goal when designing our programming abstraction for mobile code (see Section 1.2) is to provide distribution capabilities at the language level. Since Scala [89] provides powerful language abstractions which can be smoothly combined and statically type-checked, we follow the same design philosophy when extending the language features with our new programming abstraction.

On one side, Scala is a functional language; it naturally supports the notion of lambda abstraction with its unification of objects and functions (see Section 2.1). On the other side, Scala provides a generalized data binding mechanism; everything is an object and import clauses can open arbitrary objects (see Section 6.2.2).

Both language features are presented in the following two sections. Then we introduce and formalize our programming model. Finally, we discuss several aspects of our model in relation with other research works (presented in Chapter 4).

6.1 Closures as Control Abstraction

Closures (introduced in Section 2.1.2) are a powerful control abstraction which allows to suspend the evaluation of a function body with bindings to its lexical scope.

Previous work on CML [67] and other languages shows that first-class functions and channels increase the expressive power of a language where concurrently executing processes can communicate by exchanging values.

The expressiveness of closures is particularly well emphasized in the implementation of several Scala core libraries, e.g. message handling in the Scala Actors library, event handling in the Scala Swing library or support for embedded DSL in the Scala parsing library. The present work takes a similar approach to extend the notion of lexical scope in a distributed programming environment.

6.1.1 Lexical Closures

A lexical closure is an anonymous function that may escape the scope of its free identifiers. The corresponding variable bindings therefore need to be captured when defining closures in order to be able to later access their actual values. Here, it is important to understand that it's the binding, not the value of the variable, that is captured. Thus, a closure can not only access the value of the variables it closes over but can also assign new values that will persist between calls to the closure.

Furthermore, closure conversion breaks the stack-based storage allocation scheme for function invocations and instead requires heap-based allocation. Formally, a lexical closure is a closed lambda term where every variable occurence is bound.

Closure conversion becomes slightly more complicated when the language provides some binding mechanism to access variables declared outside the lexical context such as static variables or variables declared in Scala objects.

To illustrate that case let us consider the two Scala code samples in Listing 6.1: the source code on the left side contains an anonymous function `(k: Int) => {..}` with type `Int => Int` which simply performs the sum of integer variables bound to different contexts; the transformed code on the right side shows its intermediate representation as generated by the Scala compiler front-end.

```
object x {
  var i = 0
  object y { var i = 1 }
}
class O { // outer
  var i = 2
  class I { // inner
    var j = 3
    object z { var i = 4 }
    (k: Int) =>
      k + x.i + x.y.i +
      i + j + z.i
  }
}
```

```
//...
class O$I$$anonfun$1 extends Object
with Function1 with ScalaObject {
  def this($outer: O$I) /*..*/
  //...
  def apply(k: Int): Int =
    k.+
    (x.i()).+
    (x$y.i()).+
    (this.$outer.O$I$$$outer().i()).+
    (this.$outer.j()).+
    (this.$outer.z().i());
  val $outer: O$I = _;
};
```

Listing 6.1: Scala closure and free variables.

Concretely, we can see from the above example that variables used in the function body are accessed in three different ways:

- Variable `k`, a formal parameter of method `apply`, is bound and is directly accessible in the code;
- Variables `x.i` and `x.y.i` are free, and, since they are declared in singleton objects[2], they are not explicitely captured during the closure conversion;
- The remaining three free variables are made accessible through an outer field `$outer` (see Section 7.2.1) with the type `O$I` of the enclosing instance.

In particular, the access to free variables occurs through some synthetic getter/setter methods belonging to their respective declaration scopes.

Furthermore, with the declaration `class O` (see left side in Listing 6.1) the operand `i` in the body of the anonymous function is transformed into `this.$outer.O$I$$$outer().i()` (third case); if we change the above declaration to `object O` the operand `i` is then transformed into `O.this.i()` (second case).

When transforming the code of anonymous functions in a distributed environment we must find a mechanism to preserve bindings to variables in their originating scope.

6.1.2 Detached Closures

We define a *detached closure* as a lexical closure which can be moved outside of an execution environment — e.g. outside the Java run-time environment — and brought back to another environment while keeping variable bindings to their originating scopes. The local environment of a detached closure as prescribed by lexical scoping is thus extended to a distributed environment and it may escape the address space of its free identifiers.

A concern when moving objects containing references is deciding how much to move. An object is typically part of a graph of references, and one could move a single object, several levels of objects, or the entire graph. The simplest approach — moving the specified object alone — may be inappropriate. Depending on how the object is implemented, invocations of the moved object may require remote references that would have been avoided if other related objects had been moved as well.

[2]Either a global object or an object declared in a singleton object.

Lexical closures in Scala are serializable objects and the serialization of their arguments is under the programmer's responsability. In this work we adopt the same policy for detached closures and extend the handling of free variables to a distributed scope. Since free objects may be stateful objects we choose not to move them together with the detached closure and introduce remote proxies to remotely access their members.

6.2 Generalized Data Binding Mechanism

Lexical binding does not provide any mechanism for the selective import of bindings to different lexical contexts. Many programming languages feature a top-level language construct in order to support that concept of modularity, e.g. uses clauses in Pascal and Ada, include directives in C++, using clauses in C# and import clauses in Java and Modula-2[3].

Compared to Java and other object-oriented languages Scala— as well as Python — features a generalized data binding mechanism that provides a flexible name resolution mechanism and a uniform access to object name spaces:

- Classes in Scala contain no static members; Scala objects act as dynamic modules[4] [64] and make Java static members useless.
- Import clauses in Scala may occur in any scope in the source code; member renaming can be used to solve ambiguities between imports appearing at the same scope level.

6.2.1 Module Objects

The concept of code modularity first appeared in Pascal-like languages; examples of language features include the Turbo Pascal `unit` construct, the Modula-2 `module` declaration and the Ada `package` statement. Later on, the notion of *namespace* was introduced to allow the logical grouping of unique identifiers; thus, a namespace in C++ is defined with a namespace block while in Java the idea of a namespace is embodied in Java packages.

The language Scala unifies the concepts of objects, modules and packages.

[3] The language Modula-2 supports both selective import and export clauses.
[4] Object members may also be imported in Scala.

6.2.2 Import Clauses

Scala packages and their members can be imported using import clauses [90, §13.2]. Imported members can then be accessed by a simple name like `Random`, as opposed to requiring a qualified name like `scala.util.Random`.

Import clauses in Scala are more general than Java's: they can appear anywhere, not just at the beginning of a compilation unit; they can import packages themselves, not just their non-package members[5]; finally, they can also rename or hide members.

Example 1 The source code in Listing 6.2 illustrates the shadowing effect of import clauses as prescribed by the scoping rules of Scala:

- the first import clause (line 1) appears at the top-level scope and imports class `Queue` from package `collection.immutable`,
- the second import clause (line 6) imports the mutable library class `Queue` into the scope of object `import1`,
- the third import clause (line 11) appears inside an expression and reintroduces the (shadowed) immutable library class `Queue` in the local scope and, finally,
- the last import clause (line 16) uses renaming to access the immutable library class `Queue`.

```
import collection.immutable.Queue
object import1 extends Application {
  val p = new Queue enqueue 1
  println("p="+p.dequeue)

  import collection.mutable.Queue
  val q = new Queue[Int]; q enqueue 1
  println("q="+q.dequeue)

  val p2 = {
    import collection.immutable.Queue
    new Queue enqueue 1
  }
  println("p2="+p2.dequeue)
```

[5] This is only natural if you think of nested packages being contained in their surrounding package.

```
16  import collection.immutable.{Queue => ImQueue}
    val p3 = new ImQueue enqueue 1
18  println("p3="+p3.dequeue)
}
```

Listing 6.2: Import clauses at any scope level.

Example 2 The source code in Listing 6.3 presents several use cases of import clauses[6] opening Scala objects:

- the clause `import List._` imports the library function `range` from object `List`[7],
- the clause `import a._` imports class `A` from package `a`,
- the clause `import b._` imports class `B` from object `b`,
- the clause `import c._` imports class `D` from variable `c`,
- the clause `import m._` (indirectly) imports the library function `max` from field `c.m` and, finally,
- the clause `import d._` imports the method `init` from variable `d`.

```
package a { case class A(x: Int) }
object b { case class B(x: Int) }
class C { case class D(x: Int); val m = Math }
trait D { type T; val init: T }

class E(c: C, d: D) {
  import List._; println(range(0, 2))
  import a._; println(A(1))
  import b._; println(B(2))
  import c._; println(D(3))
  import m._; println(max(3, 4))
  import d._; println(init)
}
```

Listing 6.3: Import clauses opening any object.

[6]Objects `List` and `Math` are imported implicitly as members of the package `scala`.
[7]All public object/package members are actually imported when specifying `_`.

6.3 Programming Model

Our programming model is based on the following postulate: the notion of lambda abstraction is more than just a founding concept of functional programming, it provides an effective way to model dynamic aspects of relocated code in a distributed settings.

First, our model adopts a similar approach as the Obliq's answer (see Section 4.2.1) to the technical challenge of finding a meaning for *higher-order distributed computations*.

- it handles free variables as local objects and uses network references to access them remotely.
- it extends the execution context of detached closures with remote bindings referring to the surrounding scope of their originating code location.

Second, our model borrows characteristics from both the REV paradigm (Section 2.2.3) and the RPC paradigm (Section 2.2.2):

- following the REV paradigm the client benefits from a more general service since it provides the code to be remotely executed; unlike REV, code arguments are provided by the server.
- following the RPC paradigm the server can remotely access object references located on the client node.

Finally our programming model for mobile code directly benefits from the expressiveness of the language Scala and promotes the strengths of lambda abstraction in a distributed settings, with no need for a specially designed new language. In particular, it can be realized without any change to the underlying semantics.

6.3.1 Detach Calculus

In this section we first formalize the meaning of the `detach` primitive in the restricted setting of a classical lambda calculus extended with locations. Then, in section 6.3.2, we formalize our approach in a more complete, object-oriented setting.

Syntax We define the syntax of our calculus inductively as follows, where the meta-variable x denotes a variable, the meta-variable M (or N) denotes a term and the meta-variable A (or B) denotes a location.

$$M, N \ ::= \ x \mid M\,N \mid \lambda x.\,M \mid [M]_A \mid \textbf{detach}(\lambda x.\,M)$$

The first three kinds of terms are borrowed from the classical lambda calculus: a term can be a variable x, an application $M\,N$ or a lambda abstraction $\lambda x.\,M$ (also called function literal in the following).

Our detach calculus contains in addition the primitive $[M]_A$, which means that M must be evaluated at location A, and **detach**$(\lambda x.\,M)$, which represents a function literal with a different behaviour, w.r.t. location, from the classical $\lambda x.\,M$ construct.

While both terms $\lambda x.\,M$ and **detach**$(\lambda x.\,M)$ are applied to some argument, meaning that the formal argument x is replaced with the actual argument, the location where the computation goes on is different:

- With the classical construct $\lambda x.\,M$, the computation continues at the location that directly enclosed $\lambda x.\,M$.
- With the new construct **detach**$(\lambda x.\,M)$ the computation continues at the location that called **detach**$(\lambda x.\,M)$.

Thus, in the former case the function literal is implicitly attached to its enclosing location, while in the later case it is explicitly "detached" and is moved to the caller location when applied.

Semantics We describe the semantics of our calculus with the equivalence relation $\equiv$ and the reduction relation $\rightarrow$.

The equivalence relation $\equiv$ reflects some trivial properties of locations; the two interesting rules are $[M\,N]_A \equiv [M]_A\,[N]_A$ and $[[M]_A]_A \equiv [M]_A$.

The reduction relation $\rightarrow$ is mostly similar to the one of the classical lambda calculus. There are two additional rules that deal with *distributed lambda redexes*, i.e. with situations where a function literal and its argument are located at different places.

$$[[\lambda x.\,M]_A\,N]_B \rightarrow [M\{x\backslash[N]_B\}]_A \qquad \text{(R-ATT)}$$

$$[[\textbf{detach}(\lambda x.\,M)]_A\,N]_B \rightarrow [M\{x\backslash[N]_B\}]_B \qquad \text{(R-DET)}$$

In the first rule (R-Att) the function literal $\lambda x.\,M$ is by default *attached* to its enclosing location A. So, the evaluation continues in the location A even though the top-level enclosing location was B (notation: an expression like $M\{x\backslash N\}$ represents the substitution of N for x in M).

In the second rule (R-Det), the function literal **detach**$(\lambda x.\,M)$ is *detached* from its enclosing location A. So, the evaluation continues in the location B where the call is invoked. In that case the body M is *moved* from location A to location B, which is the essence of mobile code.

Example Suppose we have the two locations $\mathbf{L}$ (local) and $\mathbf{R}$ (remote) and let M be the following term:

$$M \;=\; \lambda z.\, [\lambda y.\, \mathbf{detach}(\lambda x.\, x + y + z)]_{\mathbf{R}}$$

In the following we trace the evaluation of the term $[M\,1\,2\,3]_{\mathbf{L}}$, which starts at location $\mathbf{L}$.

$$\begin{array}{rcll}
[M\,1\,2\,3]_{\mathbf{L}} & = & [\lambda z.\, [\lambda y.\, \mathbf{detach}(\lambda x.\, x + y + z)]_{\mathbf{R}}\, 1\,2\,3]_{\mathbf{L}} & \text{(by substitution)} \\
& \rightarrow & [[\lambda y.\, \mathbf{detach}(\lambda x.\, x + y + 1)]_{\mathbf{R}}\, 2\,3]_{\mathbf{L}} & \text{(by } \lambda\text{-reduction)} \\
& \rightarrow & [[\mathbf{detach}(\lambda x.\, x + 2 + 1)]_{\mathbf{R}}\, 3]_{\mathbf{L}} & \text{(using R-Att)} \\
& \rightarrow & [3 + 2 + 1]_{\mathbf{L}} & \text{(using R-Det)}
\end{array}$$

What is typical of the primitive **detach** in this example is that the evaluation of the term $x + y + z$, which is initially located at $\mathbf{R}$, finally gets executed at location $\mathbf{L}$ (after arguments have been passed) because it has been explicitly detached from its enclosing location. In a concrete implementation this means that this piece of code is at some point moved from location $\mathbf{R}$ to location $\mathbf{L}$.

6.3.2 Formalization

This section presents a formal description of our type system. To facilitate the apprehension of the key idea, we limit our formalization to a core subset of Java [48] inspired by FJ [63]. Thus, the presented core language support only top-level classes and no top-level objects.

We add the declaration of abstract methods and the expression `detach`. The presented approach, however, extends to the whole of Java and other object-oriented languages like Scala.

The core language syntax is shown in Figure 6.1. The syntax of programs, classes and expressions is standard.

First we extend the expression rule t in the above language syntax with a lightweight syntax for function literals as shown in Figure 6.2. Our choice corresponds to the syntax of function literals in Scala: the character symbol $\Rightarrow$ introduces the function body whose type determines the return type of the function.

Second we introduce the primitive **detach** to specify that the function literal requires a special handling. In particular, the applied transformation is expected to be transparent in respect to the type of $(\overline{x : T}) \Rightarrow t$.

$$
\begin{array}{lll}
P & ::= \overline{cdef}\; t & C, D, .. \in Classes \\
cdef & ::= \textbf{class}\, C\, \textbf{extends}\, D\, \{\, \overline{fdef}\; \overline{mdef}\, \} & x, y, .. \in Variables \\
fdef & ::= f : T & f, g, .. \in Fields \\
mdef & ::= \textbf{def}\, m(\overline{x : T}) : T & m, n, .. \in Methods \\
 & \quad \mid \textbf{def}\, m(\overline{x : T}) : T = t & \\
t & ::= x & (variable) \\
 & \quad \mid t.f & (selection) \\
 & \quad \mid t.f = t_1 & (assignment) \\
 & \quad \mid t.m(\overline{t}) & (invocation) \\
 & \quad \mid \textbf{new}\, C(\overline{t}) & (instantiation) \\
T & ::= C & (class)
\end{array}
$$

Figure 6.1: Core language syntax.

$$
\begin{array}{lll}
t & ::= \ldots & (same\, as\, before) \\
 & \quad \mid (\overline{x : T}) \Rightarrow t & (function\, literal) \\
 & \quad \mid \textbf{detach}((\overline{x : T}) \Rightarrow t) & (detached\, function)
\end{array}
$$

Figure 6.2: Extended language syntax.

Well-Formedness For type checking programs we assume a fixed, well-formed class table CT that defines the nominal subtyping relation $<:$ and the structural subtyping relation $<:_{struct}$.

In addition to user-defined classes CT also contains a set FT of synthetic classes where each class has a unique name and defines an abstract method apply$(\overline{x : T}) : T$ such that $\overline{T}$ and T form a unique combination of user-defined and synthetic classes taken from the class table[8].

The exact set of classes for FT actually depends on the type-checked program; we stipulate that the set FT should be large enough to include all $ftype(\overline{T}, T)$ appearing in a typing derivation of that program.

Furthermore we provide the following auxiliary functions for retrieving type information from the class table:

- $fields(C) = \overline{f : D}$ where $\overline{f : D}$ are all fields in C and its parents;
- $mtype(C, m) = \overline{T} \rightarrow T$ where $\overline{T}$ are the types of the formal parameters of method m and T its result type;
- $mbody(C, m) = (\overline{x}, t)$ where $\textbf{def}\, m(\overline{x : T}) : T = t$ is defined in the most direct superclass of C that defines m;

[8] In practice, those classes are defined by the generic functions `Function1[T1, R]`, `Function2[T1, T2, R]`, etc.

- $override(C, m)$ = true if $\forall D$ with $C <: D$ $mtype(D, m)$ is either undefined or $mtype(C, m) = mtype(D, m)$.
- and $ftype(\overline{T}, T) = F$ where $F \in FT$ and $mtype(F, \text{apply}) = \overline{T} \to T$.

A method m is well-formed in a class C if its body is well-typed in an environment Γ that maps m's formal parameters to their declared types (WF-Method1), and **this** to C (WF-Method2). A class C is well-formed if all its method definitions are well-formed (WF-Class).

$$\frac{\Gamma = \overline{x : T} \qquad override(C, m)}{C \vdash \mathbf{def}\, m(\overline{x : T}) : T} \qquad \text{(WF-Method1)}$$

$$\frac{\Gamma = \overline{x : T}, \mathbf{this} : C \qquad override(C, m) \qquad \Gamma \vdash t : T_0 \qquad T_0 <: T}{C \vdash \mathbf{def}\, m(\overline{x : T}) : T \,=\, t} \qquad \text{(WF-Method2)}$$

$$\frac{C \vdash \overline{mdef}}{\vdash \mathbf{class}\, C\, \mathbf{extends}\, D \,\{\, \overline{fdef}\ \overline{mdef}\, \}} \qquad \text{(WF-Class)}$$

Figure 6.3: Well-formedness.

We also provide an auxiliary function to operate on terms:

- $type(t) = T$ where $T = C$ if $t = x$ with $x : C$ or $T = D$ if $t = t.f$ with $f : D$, and so on;

Typing Rules Figure 6.4 shows the typing rules for expressions in the core language. The typing judgement has the form $\Gamma \vdash t : C$; Γ maps (free) variables to types. The rules for typing variables (T-Var) and field selections (T-Sel) are standard. In the typing rule for assignments (T-Ass) we choose to use type `C` in the judgment to support chaining of variable updates. Finally, the typing rules for typing method invocations (T-Call) and instantiations (T-New) are also standard.

The additional typing rules given in Figure 6.5 are introduced together with the extended language; they rely on the three transformations *close*, *proxy* and *detach* which are introduced later.

While the closure conversion of a normal function literal results in the partial application of the function body to its explicitly constructed environment, the transformation of a detached function requires additional formalization work.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash t : C \qquad f : D \in fields(C)}{\Gamma \vdash t.f : D} \quad \text{(T-SEL)}$$

$$\frac{\Gamma \vdash t : C \qquad f : D \in fields(C) \qquad \Gamma \vdash t_1 : E \qquad E <: D}{\Gamma \vdash t.f = t_1 : C} \quad \text{(T-ASS)}$$

$$\frac{\Gamma \vdash t : C \qquad mtype(C, m) = \overline{T} \rightarrow T \qquad \forall i \in \{1..|\overline{T}|\} \Gamma \vdash t_i : C_i \qquad \overline{C} <: \overline{T}}{\Gamma \vdash t.m(\overline{t}) : T} \quad \text{(T-CALL)}$$

$$\frac{fields(E) = \overline{f : D} \qquad \forall i \in \{1..|\overline{D}|\} \Gamma \vdash t_i : C_i \qquad \overline{C} <: \overline{D}}{\Gamma \vdash \textbf{new}\, E(\overline{t}) : E} \quad \text{(T-NEW)}$$

Figure 6.4: Typing rules.

$$\frac{\Gamma, \overline{x_1 : T_1} \vdash t : T \qquad close(\overline{x_1 : T_1}, \overline{x : T}, t) = E_{fresh} \qquad \Gamma \vdash \textbf{new}\, E_{fresh}(\overline{x_1}) : E_{fresh}}{\Gamma \vdash (\overline{x : T}) \Rightarrow t : E_{fresh}} \quad \text{(T-CLO)}$$

$$\frac{\Gamma, \overline{x_1 : T_1} \vdash t : T \qquad detach(\overline{x_1 : T_1}, \overline{x : T}, t) = E_{fresh} \qquad proxy(\overline{x_1 : T_1}) = \overline{t_1} \qquad \Gamma \vdash \textbf{new}\, E_{fresh}(\overline{t_1}) : E_{fresh}}{\Gamma \vdash \text{detach}((\overline{x : T}) \Rightarrow t) : E_{fresh}} \quad \text{(T-DET)}$$

Figure 6.5: Extended typing rules.

Thus, class E_{fresh} in rule (T-CLO) is a freshly created class with unique name which captures its environment at the point of its instantiation and defines a method **apply** with the closed expression t as body.

Similarly, class E_{fresh} in rule (T-DET) gives access to variables declared in the originating environment of the detached closure by the means of the freshly created proxy objects $\overline{t_1}$. Thus, a free variable $x_1 : T_1$ originally

used in t is accessed remotely through its associated proxy t_1.

$$\frac{\begin{array}{l}\Gamma, \overline{x_1 : T_1} \vdash t : T \qquad CT' = CT \otimes E_{fresh} \\ CT \vdash \textbf{class}\, E_{fresh}\, \textbf{extends}\, ftype(\overline{T}, T)\, \{ \\ \quad \overline{x_1 : T_1}; \\ \quad \textbf{def}\, \text{apply}(\overline{x : T}) : T = t \\ \} \end{array}}{CT \vdash close(\overline{x_1 : T_1}, \overline{x : T}, t) = E_{fresh}, CT'} \quad \text{(close)}$$

Figure 6.6: Transformation rule (close).

The $close$ transformation (Figure 6.6) adds to class table CT a new class definition E_{fresh} synthetized from the tuple $(\overline{x_1 : T_1}, \overline{x : T}, t)$ where $\overline{x_1 : T_1}$ represent the free variables captured by expression t.

$$\frac{\begin{array}{l}\Gamma, \overline{x_1 : T_1}, \vdash t : T \qquad CT' = CT \otimes \overline{P_{intf}} \otimes \overline{P_{impl}} \otimes E_{fresh} \\ \overline{P_{intf}} <:_{struct} \overline{T_1} \qquad t_1 = t[x_1/x_2] \\ CT \vdash \textbf{class}\, E_{fresh}\, \textbf{extends}\, ftype(\overline{T}, T)\, \{ \\ \quad \overline{x_2 : P_{intf}}; \\ \quad \textbf{def}\, \text{apply}(\overline{x : T}) : T = t_1 \\ \} \\ \overline{\textbf{class}\, P_{intf}\, \{} \\ \quad \textbf{def}\, m_i(\overline{x : T}) : T \\ \} \\ \overline{\textbf{class}\, P_{impl}\, \textbf{extends}\, P_{intf}\, \{} \\ \quad x_2 : type(x_1) \\ \quad \textbf{def}\, m_i(\overline{x : T}) : T = x_2.m_i(\overline{x}) \\ \} \end{array}}{CT \vdash detach(\overline{x_1 : T_1}, \overline{x : T}, t) = E_{fresh}, CT'} \quad \text{(detach)}$$

Figure 6.7: Transformation rule (detach).

$$\frac{\begin{array}{c}\Gamma, \overline{x : T} \vdash \qquad \overline{P_{intf}} <:_{struct} \overline{T_1} \\ \forall i \in \{1..|\overline{T}|\}\overline{x} \vdash t_i = \textbf{new}\, P_{impl}(x_i) : P_{impl}\end{array}}{\Gamma \vdash proxy(\overline{x : T}) = \overline{t : P_{impl}}} \quad \text{(proxy)}$$

Figure 6.8: Transformation rule (proxy).

Example In the following source code, written in the extended language (see Figures 6.1 and 6.2), we assume the existence of the root class `Object`, the class `Int` which represents integer values (with a method `+`, etc.) and the class `Function_Int_Int` which defines a type for functions from integer values to integer values (corresponding to $\mathit{ftype}(\{\texttt{Int}\}, \texttt{Int}) \in FT$, with a method `apply(x: Int): Int`).

```
class A extends Object {
  z: Int
  def m(y: Int): Function_Int_Int =
    detach((x: Int) => x.+(y).+(this.z))
}
new A(1).m(2).apply(3)
```

Listing 6.4: Core language example.

The transformation rule in Figure 6.7 is applied when encountering the primitive `detach` in the above source code. Here, the transformed code additionaly relies on the predefined class `RemoteObject`, a base type for remote objects, while the two classes `IntRef_proxy`, a remote type for integer references (with method `elem`, etc.), and `IntRef_proxyImpl` are omitted for brevity.

```
class A_proxy extends RemoteObject {
  def z(): Int
  def z_=(z0: Int): A_proxy
}
class A_proxyImpl extends A_proxy {
  a: A
  def z(): Int = a.z
  def z_=(z0: Int): A_proxy = a.z = z0
}
class A_detach extends Function_Int_Int {
  a: A_proxy
  y: IntRef_proxy
  def apply(x: Int): Int = x.+(y.elem()).+(a.z())
}
class A extends Object {
  z: Int
  def m(y: Int): Function_Int_Int = new A_detach(
    new A_proxyImpl(this), new IntRef_proxyImpl(y)
```

```
  )
}
new A(1).m(2).apply(3)
```

Listing 6.5: Core language example (transformed).

6.4 Discussion

Lexical closures are most interesting in programming languages that can treat functions as first-class values since storing a closure for later use implies an extension of the lifetime of the closed-over lexical scope beyond what one would normally expect.

The programming model presented in this Chapter applies the same concept to the distributed programming environment and extends the static scope of a detached closure to the network. It looks familiar from a user's perspective and its implementation mainly involves high-level code transformations which hide the boilerplate of hand-written code.

Chapter 7

Implementation

Comments lie. Code doesn't.

Ron Jeffries[1]

In the following we present our implementation of the programming model discussed in the previous chapter. We rely on the Java run-time environment to provide support for code mobility and security enforcement, but our solution could easily be ported to other programming environments such as .NET platform.

In the first section we focus on the transformations performed during closure conversion and discuss several code examples of function literals transformed into lexical closures respectively into detached closures. Chapter 5 gives several programming examples of detached closures transmitted via different communication mediums such as files and typed channels.

Then we describe in details the compiler and library extensions implemented as additional code transformations, run-time support and programming interface.

Finally we discuss several run-time constraints imposed by the target platform.

[1]Ron Jeffries is one of the three founders of the Extreme Programming (XP) software development methodology.

7.1 Closure Conversion

Closure conversion serves the powerful concept of control abstraction (see Section 2.1.2) and many aspects of closure conversion have been studied in past research works.

For example, Glew [46] presents a formal closure conversion translation for a second-order object language and proves it correct. Minamide and al. [79, 80] study the typing properties of *closure conversion* for simply-typed and polymorphic lambda calculi and validate their research by implementing type-preserving transformations in two ML compilers.

In the following we first look at closure conversion as performed by the Scala compiler developed at EPFL and then present the additional transformations involved in the conversion of *detached closures* as introduced in the previous chapter. An important property of closure conversion is that the representation of the surrounding environment is private to the closure; that property is also preserved by the transformation of detached closures.

7.1.1 Lexical Closures

In Listing 7.1 we define a function literal `(x: Int) => {..}` in some class `A`; its single parameter `x` is called a bound variable and the other variables — in this case `y` and `z` — referring to its surrounding context are called free (non-local) variables.

Mutable variables like `z` are declared using the **`var`** keyword and the Scala compiler handles them differently whether they appear as class members or as local variables.

```
class A {
  val y = 1
  var z = 2
  ((x: Int) => {z += 1; x + y + z})
}
```

Listing 7.1: Scala closure inside a class.

The Scala compiler analyzes the source code of Listing 7.1 and applies several transformations to the internal representation of the program; Listing 7.2 presents the transformed code after the `LambdaLift` phase (explained later in Section 7.2.1).

```
class A extends java.lang.Object with ScalaObject {
  //..
  def y(): Int = A.this.y;
  def z(): Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  {
    (new $anonfun$1(A.this): Function1)
  };
  class $anonfun$1
  extends java.lang.Object with Function1 with ScalaObject {
    def this($outer: A): $anonfun$1 = //..
    def apply(x: Int): Int = {
      $anonfun$1.this.$outer.z_=(
        $anonfun$1.this.$outer.z().+(1));
      x.+
      ($anonfun$1.this.$outer.y()).+
      ($anonfun$1.this.$outer.z());
    };
    //..
  }
}
```

Listing 7.2: Scala closure inside a class (converted).

The generated closure definition looks like a specialized class definition:

```
  class $anonfun$1
  extends java.lang.Object with Function1 with ScalaObject {
    def this($outer: A): $anonfun$1 = /*..*/
    def apply(x: Int): Int = /*..*/
  };
```

it defines a constructor for capturing the closure environment and a method `apply` for the deferred evaluation of the closure body. Actually it doesn't capture free variables defined in the enclosing class individually, but refers to the outer field `outer` — an instance of that class — for accessing them through their respective getter/setter methods.

Then the closure instantiation

```
new $anonfun$1(A.this)
```

invokes the class constructor with argument `A.this` and so captures the free identifiers declared in the surrounding scope (`class A`).

7.1.2 Detached Closures

The conversion of *detached closures* is implemented by an additional transformation phase in the Scala compiler front-end. The `Detach` phase[2] performs several code transformations when encountering the marker object `detach` in the Scala source code.

The marker object `detach` is declared in package `scala.remoting` of the Scala standard library and defines dummy `apply` methods to be handled by the compiler (similar to method `Predef.classOf`):

```
package scala.remoting
object detach {
  def apply[R](f: Function0[R]): Function0[R] = f
  def apply[T0, R](f: Function1[T0, R]): Function1[T0, R] = f
  def apply[T0,T1,R](f: Function2[T0,T1,R]): Function2[T0,T1,R] = f
  // ..
}
```

and can be applied to function literals with different arities[3]:

```
class A {
  val y = 1

  detach(() => y) // Function0[Int]
  detach((x: Int) => Math.abs(x)+y) // Function1[Int,Int]
  detach((x: Int, c: Char) => x+y+c) // Function2[Int,Char,Int]

  def f(x: Int): Int = x + y
  detach(f _) // Function1[Int,Int]

  val g = (x: Int) => x + y
  //detach(g) // error: detach inapplicable for value g
}
```

[2]The phase is currently enabled using the compiler option `-Ydetach`.

[3] `detach` accepts all the function types represented by the predefined Scala traits `Function0`, `Function1`, etc..

Given some detached closure defined in the enclosing class `A` the `Detach` phase performs the following transformations:

1. it generates in the top-level scope a remote proxy interface `A$proxy` (see Listing 7.5) and the corresponding proxy implementation `A$proxyImpl` (see Listing 7.6) for forwarding calls to the accessor methods[4] of the enclosing instance `A.this`;
2. it generates the remote proxy interfaces `B`$_i$`$proxy` and the corresponding proxy implementations `B`$_i$`$proxyImpl` to remotely access the objects `b`$_i$ (of type `B`$_i$) captured by the closure *and* not declared in the enclosing class `A`;
3. it transforms the local (λ-lifted) closure definition `$anonfun$j` into a detached closure definition `$anonfun$j$detach` (see Listing 7.7) which remotely access the objects `A.this` and `b`$_i$ through the remote proxy interfaces `A$proxy`, `B`$_i$`$proxy`;
4. and, finally, it creates remote references for the accessed objects `A.this` and `b`$_i$ and pass them as arguments to the closure instantiation `new $anonfun$j(..)` (see Listing 7.4).

Here, one implementation decision needs to be explained further: objects `b`$_i$ declared outside the enclosing class `A` are all accessed remotely using the remote proxy interfaces `B`$_i$`$proxy` (point 2 below). We could move serializable objects together with the detached closure – a serializable object — depending on if they are stateful or not. In the present implementation we choose not to distinguish further the case and simply handle all referenced objects in the same manner.

Rather than discussing the internals of the transformation phase itself we examine several interesting examples with both the Scala source code and the corresponding transformed code. Each example emphasizes a particular aspect of closure conversion; we presents first the code transformation of a lexical closure and then focus on the code transformation of a detached closure.

[4]Scala member variables are accessed using setter/getter methods.

Example 1: Inside a class

In the source code of Listing 7.3 we slightly modify the Scala code presented in Listing 7.1 by adding the marker object `detach` around the function literal `(x: Int) => {..}`:

```
class A {
  val y = 1
  var z = 2
  detach((x: Int) => {z += 1; x + y + z})
}
```

Listing 7.3: Detached Scala closure inside a class.

After the `Detach` phase the transformed code looks as follows:

- the remote proxy object `proxy$1` is passed as argument to the closure instantiation (Figure 7.4, line 12);
- the type of `proxy$1` is defined as the remote proxy type `A$Proxy` (Figure 7.5) together with its implementation `A$ProxyImpl` (Figure 7.6);
- the detached closure is defined as the serializable class `$anonfun$1$detach` containing the method `apply` with a transformed body (Figure 7.7).

```
class A extends java.lang.Object with ScalaObject {
  //..
  def y(): Int = A.this.y;
  def z(): Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  {
    val proxy$1: A$proxy =
      RemoteRef.bind(
        "A/proxy$1",
        new A$proxyImpl("A/proxy$1", A.this)).
      $asInstanceOf[A$proxy]();
    (new $anonfun$1$detach(proxy$1): Function1)
  };
};
```

Listing 7.4: Detached Scala closure inside a class (1/4).

The call to method `RemoteRef.bind` binds the synthetic name `"A/proxy$1"` to the remote proxy object `proxy$1` of type `A$proxyImpl` (Figure 7.6).

```
@remote
trait A$proxy
extends java.lang.Object with java.rmi.Remote with ScalaObject {
  def y(): Int;
  def z(): Int;
  def z_=(x$1: Int): Unit
};
```

Listing 7.5: Detached Scala closure inside a class (2/4).

The class `A$proxyImpl` (Listing 7.6) implements the behavior defined by the remote interface `A$proxy` (Listing 7.5) and merely forwards method calls to class `A`.

```
class A$proxyImpl
extends java.rmi.server.UnicastRemoteObject with A$proxy
with ScalaObject with java.rmi.server.Unreferenced {
  //..
  def this(x$1: String, x$2: A): A$proxyImpl = /*..*/
  def unreferenced(): Unit = {
    Debug.info("unreferenced: ".+(A$proxyImpl.this.x$1));
    RemoteRef.unbind(A$proxyImpl.this.x$1);
  };
  def y(): Int = A$proxyImpl.this.x$2.y();
  def z(): Int = A$proxyImpl.this.x$2.z();
  def z_=(x$1: Int): Unit = A$proxyImpl.this.x$2.z_=(x$1)
};
```

Listing 7.6: Detached Scala closure inside a class (3/4).

```
@SerialVersionUID(261399656060227L) @serializable
class $anonfun$1$detach
extends java.lang.Object with Function1 with ScalaObject {
  //..
  def this($outer: A$proxy): $anonfun$1$detach = /*..*/
  def apply(x: Int): Int = {
    $anonfun$1$detach.this.$outer.z_=(
      $anonfun$1$detach.this.$outer.z().+(1));
    x.+
    ($anonfun$1$detach.this.$outer.y()).+
```

```
    ($anonfun$1$detach.this.$outer.z());
  };
};
```

Listing 7.7: Detached Scala closure inside a class (4/4).

Figure 7.1 represents the two transformations graphically: the diagram above the dashed line corresponds to the transformed closure with its outer class `A` (see Listing 7.2) and the diagram below the dashed line describes the interaction between the transformed closure marked with `detach` and the same class `A`.

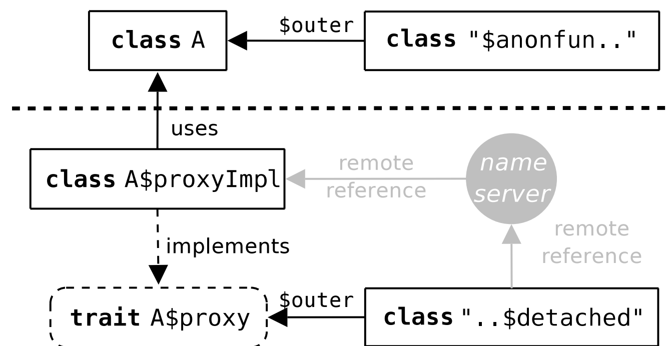


Figure 7.1: Detached closure and outer class.

Example 2: Inside an object

Source code in Listing 7.8 is similar to that of Listing 7.1: it defines the function literal `(x: Int) => {..}` in an object `A` instead of a class `A`. In this case the closure definition accesses the free variables directly (see Listing 7.9, lines 13-14) using the variable `A.this` instead of the outer field `outer` as there exists exactly one instance of the enclosing class `A`.

```
object A {
  val y = 1
  var z = 2
  ((x: Int) => {z += 1; x + y + z})
}
```

Listing 7.8: Scala closure inside an object.

```
class A extends java.lang.Object with ScalaObject {
2   //..
    def y(): Int = A.this.y;
4   def z(): Int = A.this.z;
    def z_=(x$1: Int): Unit = A.this.z = x$1;
6   {
      (new $anonfun$1(): Function1)
8   };
    class $anonfun$1
10  extends java.lang.Object with Function1 with ScalaObject {
      def this(): $anonfun$1 = /*..*/
12    def apply(x: Int): Int = {
        A.this.z_=(A.this.z().+(1));
14      x.+(A.this.y()).+(A.this.z());
      };
16    //..
    };
18 };
```

Listing 7.9: Scala closure inside an object (converted).

While lexical closures declared inside a class (Listing 7.2) or inside an object (Listing 7.9) are transformed slightly differently in respect to the `this` reference, detached closures declared in a class or in an object undergo the same transformation (compare Listing 7.7 and Listing 7.14).

```
import scala.remoting.detach
object A {
  val y = 1
  var z = 2
  detach((x: Int) => {z += 1; x + y + z})
}
```

Listing 7.10: Detached closure inside an object.

```
class A extends java.lang.Object with ScalaObject {
  //...
  def y(): Int = A.this.y;
  def z(): Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  {
    val uid$1: String = new java.rmi.server.UID().toString();
    val proxy$1: A$proxy = scala.runtime.RemoteRef.bind(
      "A/proxy1".+(uid$1),
      new A$proxyImpl("A/proxy1".+(uid$1), A.this)
    ).$asInstanceOf[A$proxy]();
    (new $anonfun$1$detach(proxy$1): Function1)
  };
};
```

Listing 7.11: Detached closure inside an object (1/4).

```
@remote
trait A$proxy extends java.lang.Object
with java.rmi.Remote with ScalaObject {
  def y(): Int;
  def z_=(x$1: Int): Unit;
  def z(): Int
};
```

Listing 7.12: Detached closure inside an object (2/4).

The class `A$proxyImpl` (Listing 7.13) implements the behavior defined by the remote interface `A$proxy` (Listing 7.12) and merely forwards method calls to object `A`.

```
class A$proxyImpl extends java.rmi.server.UnicastRemoteObject
with A$proxy with java.rmi.server.Unreferenced with ScalaObject {
  //...
  def this(x$1: String, x$2: A): A$proxyImpl = /*..*/
  def unreferenced(): Unit = {
    scala.remoting.Debug.info("unreferenced: ".+(A$proxyImpl.this.x$1));
    scala.runtime.RemoteRef.unbind(A$proxyImpl.this.x$1)
  };
  def y(): Int = A$proxyImpl.this.x$2.y();
  def z_=(x$1: Int): Unit = A$proxyImpl.this.x$2.z_=(x$1);
  def z(): Int = A$proxyImpl.this.x$2.z()
};
```

Listing 7.13: Detached closure inside an object (3/4).

```
@serializable @SerialVersionUID(156475164860L)
class $anonfun$1$detach extends java.lang.Object
with Function1 with ScalaObject {
  def this($outer: A$proxy): $anonfun$1$detach = /*..*/
  def apply(x: Int): Int = {
    $anonfun$1$detach.this.$outer.z_=(
      $anonfun$1$detach.this.$outer.z().+(1));
    x.+
    ($anonfun$1$detach.this.$outer.y()).+
    ($anonfun$1$detach.this.$outer.z())
  };
  //...
};
```

Listing 7.14: Detached closure inside an object (4/4).

Example 3: Inside an inner class

In Listing 7.15 we introduce two more nesting levels[5] in the surrounding context and now define the function literal `(u: Int) => {..}` in some inner class `C` declared locally to another class `B`, itself declared in the top-level class `A`.

```
class A {
  val r = 0
  class B {
    val y = 1
    var z = 2
    class C(x: Int) {
      val v = y
      var w = 4
      var b = new B
      ((u: Int) => {w+=1; z+=1; u + v + w + x + y + z + b.y + r})
    }
  }
}
```

Listing 7.15: Scala closure inside an inner class.

The transformed code for class `C` (Listing 7.16) declares two new synthetic fields: the member variable `$A$B$C$$x` (line 12) gets initialized with the constructor argument `x` of class `C` and the method `$A$B$C$$$outer()` (line 22) gives access to the members of class `B`.

```
class A extends java.lang.Object with ScalaObject {
2  //...
  def r(): Int = A.this.r;
4  class B extends java.lang.Object with ScalaObject {
    //...
6    def this($outer: A): A#B = //..
    def y(): Int = B.this.y;
8    def z(): Int = B.this.z;
    def z_=(x$1: Int): Unit = B.this.z = x$1;
10    class C extends java.lang.Object with ScalaObject {
      //...
```

[5]This program configuration is not covered by the language formalization of Section 6.3.2.

```
12        final val A$B$C$$x: Int = _;
          def this($outer: A#B, x: Int): A#B#C = //..
14        def v(): Int = C.this.v;
          def w(): Int = C.this.w;
16        def w_=(x$1: Int): Unit = C.this.w = x$1;
          def b(): A#B = C.this.b;
18        def b_=(x$1: A#B): Unit = C.this.b = x$1;
          {
20          (new $anonfun$1(C.this): Function1)
          };
22        def A$B$C$$$outer(): A#B = C.this.$outer;
        };
24      def A$B$$$outer(): A = B.this.$outer
      }
26  };
    class $anonfun$1 extends java.lang.Object
28  with Function1 with ScalaObject {
      def this($outer: A#B#C): $anonfun$1 = //..
30    def apply(u: Int): Int = {
        $anonfun$1.this.$outer.w_=(
32        $anonfun$1.this.$outer.w().+(1));
        $anonfun$1.this.$outer.A$B$C$$$outer().z_=(
34        $anonfun$1.this.$outer.A$B$C$$$outer().z().+(1));
        u.+
36      ($anonfun$1.this.$outer.v()).+
        ($anonfun$1.this.$outer.w()).+
38      ($anonfun$1.this.$outer.A$B$C$$x).+
        ($anonfun$1.this.$outer.A$B$C$$$outer().y()).+
40      ($anonfun$1.this.$outer.A$B$C$$$outer().z()).+
        ($anonfun$1.this.$outer.b().y()).+
42      ($anonfun$1.this.$outer.A$B$C$$$outer().A$B$$$outer().r())
      };
44    //...
      def A$B$C$$anonfun$$$outer(): A#B#C =
46      $anonfun$1.this.$outer;
    };
```

Listing 7.16: Scala closure inside an inner class (converted).

We now add the primitive `detach` to the source code of Listing 7.15 to inform the Scala compiler it should handle the function literal `(u: Int) => {..}`

as a detached closure. The modified source code is given in Listing 7.17.

```
import scala.remoting._
class A {
  val r = 0
  class B {
    val y = 1
    var z = 2
    class C(x: Int) {
      val v = y
      var w = 4
      var b = new B
      detach((u: Int) => {w+=1; z+=1; u + v + w + x + y + z + b.y + r})
    }
  }
}
```

Listing 7.17: Detached closure inside an inner class.

The `Detach` phase (Section 7.2.2) transforms the closure instantiation appearing in the original block expression (lines 18-20 in Listing 7.16) into the registration of the remote proxy `proxy$1` followed by the instantiation of the generated detached closure.

```
class A extends java.lang.Object with ScalaObject {
2   //...
  def r(): Int = A.this.r;
4   class B extends java.lang.Object with ScalaObject {
    //...
6     def this($outer: A): A#B = /*..*/
    def y(): Int = B.this.y;
8     def z(): Int = B.this.z;
    def z_=(x$1: Int): Unit = B.this.z = x$1;
10    class C extends java.lang.Object with ScalaObject {
      //...
12      def this($outer: A#B, x: Int): A#B#C = /*..*/
      def v(): Int = C.this.v;
14      def w(): Int = C.this.w;
      def w_=(x$1: Int): Unit = C.this.w = x$1;
16      def b(): A#B = C.this.b;
      def b_=(x$1: A#B): Unit = C.this.b = x$1;
```

```
18        {
            val uid$1: java.lang.String =
20            new java.rmi.server.UID().toString();
            val proxy$1: C$proxy = scala.runtime.RemoteRef.bind(
22            "A/B/C/proxy$1$".+(uid$1),
              new C$proxyImpl("A/B/C/proxy$1$".+(uid$1), C.this)
24          ).$asInstanceOf[C$proxy]();
            (new $anonfun$1$detach(proxy$1): Function1)
26        };
          def A$B$C$$$outer(): A#B = C.this.$outer;
28      };
        def A$B$$$outer(): A = B.this.$outer
30    }
   };
```

Listing 7.18: Detached closure inside an inner class (1/4).

```
@remote
trait C$proxy extends java.lang.Object
with java.rmi.Remote with ScalaObject {
  def x(): Int;
  def v(): Int;
  def w(): Int;
  def w_=(x$1: Int): Unit;
  def A$B$y(): Int
  def A$B$z(): Int;
  def A$B$z_=(x$1: Int): Unit;
  def A$r(): Int;
};
```

Listing 7.19: Detached closure inside an inner class (2/4).

```
class C$proxyImpl extends java.rmi.server.UnicastRemoteObject
with C$proxy with java.rmi.server.Unreferenced with ScalaObject {
  //...
  def this(x$1: String, x$2: A#B#C): C$proxyImpl = /*..*/
  def unreferenced(): Unit = {
    scala.remoting.Debug.info(
      "unreferenced: ".+(C$proxyImpl.this.x$1));
    scala.runtime.RemoteRef.unbind(C$proxyImpl.this.x$1)
  };
```

```
  def x(): Int =
    C$proxyImpl.this.x$2.A$B$C$$$outer().A$B$$$outer().A$B$C$$x;
  def v(): Int =
    C$proxyImpl.this.x$2.A$B$C$$$outer().A$B$$$outer().v()
  def w(): Int =
    C$proxyImpl.this.x$2.A$B$C$$$outer().A$B$$$outer().w();
  def w_=(x$1: Int): Unit =
    C$proxyImpl.this.x$2.A$B$C$$$outer().A$B$$$outer().w_=(x$1);
  def A$B$y(): Int = C$proxyImpl.this.x$2.A$B$C$$$outer().y();
  def A$B$z(): Int = C$proxyImpl.this.x$2.A$B$C$$$outer().z();
  def A$B$z_=(x$1: Int): Unit =
    C$proxyImpl.this.x$2.A$B$C$$$outer().z_=(x$1);
```

Listing 7.20: Detached closure inside an inner class (3/4).

```
@serializable @SerialVersionUID(262141039231930L)
class $anonfun$1$detach
extends java.lang.Object with Function1 with ScalaObject {
  def this($outer: A#B#C$proxy): $anonfun$1$detach = { /*..*/ };
  def apply(u: Int): Int = {
    $anonfun$1$detach.this.$outer.w_=(
      $anonfun$1$detach.this.$outer.w().+(1));
    $anonfun$1$detach.this.$outer.A$B$z_=(
      $anonfun$1$detach.this.$outer.A$B$z().+(1));
    u.+
    ($anonfun$1$detach.this.$outer.v()).+
    ($anonfun$1$detach.this.$outer.w()).+
    ($anonfun$1$detach.this.$outer.A$B$C$$x).+
    ($anonfun$1$detach.this.$outer.A$B$y()).+
    ($anonfun$1$detach.this.$outer.A$B$z()).+
    ($anonfun$1$detach.this.$outer.b().y()).+
    ($anonfun$1$detach.this.$outer.A$r())
  };
  //...
  def A$B$C$$anonfun$$$outer(): C$proxy =
    $anonfun$1$detach.this.$outer;
};
```

Listing 7.21: Detached closure inside an inner class (4/4).

Example 4: Inside a member function

In the source code of Listing 7.22 we consider a function value defined (and applied) in a method `f`. The free variables captured by the lexical closure `(u: Int) => {..}` are handled differently whenever they are declared locally or are members of class `A`.

```
class A {
  val y = 1
  var z = 2
  def f(x: Int): Int = {
    val v = 3
    var w = 4
    ((u: Int) => {w += 1; z += 1; u + v + w + x + y + z})(2)
  }
}
```

Listing 7.22: Scala closure inside a member function.

```
class A extends java.lang.Object with ScalaObject {
  //...
  def y(): Int = A.this.y;
  def z(): Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  def f(x$1: Int): Int = {
    val v$1: Int = 3;
    var w$1: scala.runtime.IntRef = new scala.runtime.IntRef(4);
    scala.Int.unbox({
      (new $anonfun$f$1(A.this, x$1, v$1, w$1): Function1)
    }.apply(scala.Int.box(2)));
  };
  class $anonfun$f$1
  extends java.lang.Object with Function1 with ScalaObject {
    def this($outer: A, x$1: Int, v$1: Int,
             w$1: scala.runtime.IntRef)): $anonfun$f$1 = { /*..*/ }
    final def apply(u: Int): Int = {
      $anonfun$f$1.this.w$1.elem =
        $anonfun$f$1.this.w$1.elem.+(1);
      $anonfun$f$1.this.$outer.z_=(
        $anonfun$f$1.this.$outer.z().+(1));
      u.+
```

```
      ($anonfun$f$1.this.v$1).+
      ($anonfun$f$1.this.w$1.elem).+
      ($anonfun$f$1.this.x$1).+
      ($anonfun$f$1.this.$outer.y()).+
      ($anonfun$f$1.this.$outer.z());
    };
  //...
};
```

Listing 7.23: Scala closure inside a member function (converted).

In Listing 7.24 we make one small addition to the source code presented in Listing 7.22: we mark the function literal `(u: Int) => {..}` with the `detach` primitive.

```
import scala.remoting.detach
class A {
  val y = 1
  var z = 2
  def f(x: Int): Int = {
    val v = 3
    var w = 4
    detach((u: Int) => {w+=1; z+=1; u + v + w + x + y + z})(2)
  }
}
```

Listing 7.24: Detached closure inside a member function.

The transformed code is presented below in Listing 7.25 (closure instantiation), in Listing 7.26 (proxy interface), in Listing 7.27 (proxy implementation) and in Listing 7.28 (closure definition).

```
class A extends java.lang.Object with ScalaObject {
  def y(): Int = A.this.y;
  def z(): Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  def f(x$1: Int): Int = {
    val v$1: Int = 3;
    var w$1: scala.runtime.IntRef = new scala.runtime.IntRef(4);
    scala.Int.unbox({
      val proxy$1: A$proxy =
        scala.runtime.RemoteRef.bind(
```

```
          "A/proxy$1",
          new A$proxyImpl("A/proxy$1", A.this)
        ).$asInstanceOf[A$proxy]();
      val proxy$2: scala.runtime.remoting.RemoteIntRef =
        scala.runtime.RemoteRef.bind(
          "A/proxy$2",
          new scala.runtime.remoting.RemoteIntRefImpl("A/proxy$2", w$1)
        ).$asInstanceOf[scala.runtime.remoting.RemoteIntRef]();
      (new $anonfun$f$1$detach(proxy$1, x$1, v$1, proxy$2): Function1)
    }.apply(scala.Int.box(2)));
  //..
};
```

Listing 7.25: Detached closure inside a member function (1/4).

```
@remote
trait A$proxy extends java.lang.Object
with java.rmi.Remote with ScalaObject {
  def y(): Int;
  def z(): Int;
  def z_=(x$1: Int): Unit
};
```

Listing 7.26: Detached closure inside a member function (2/4).

```
class A$proxyImpl
extends java.rmi.server.UnicastRemoteObject with A$proxy
with java.rmi.server.Unreferenced with ScalaObject {
  //...
  def this(x$2: String, x$3: A): A$proxyImpl = //..
  def unreferenced(): Unit = {
    Debug.info("unreferenced: ".+(A$proxyImpl.this.x$2));
    RemoteRef.unbind(A$proxyImpl.this.x$2);
  };
  def y(): Int = A$proxyImpl.this.x$3.y();
  def z(): Int = A$proxyImpl.this.x$3.z();
  def z_=(x$1: Int): Unit = A$proxyImpl.this.x$3.z_=(x$1)
};
```

Listing 7.27: Detached closure inside a member function (3/4).

```
@serializable @SerialVersionUID(8307678459700123073L)
class $anonfun$f$1$detach
extends java.lang.Object with Function1 with ScalaObject {
  def this($outer: A$proxy, x$1: Int, v$1: Int,
           w$1: scala.runtime.remoting.RemoteIntRef):
    $anonfun$f$1$detach = //..
  def apply(u: Int): Int = {
    $anonfun$f$1$detach.this.w$1.elem_=(
      $anonfun$f$1$detach.this.w$1.elem().+(1));
    $anonfun$f$1$detach.this.$outer.z_=(
      $anonfun$f$1$detach.this.$outer.z().+(1));
    u.+
    ($anonfun$f$1$detach.this.v$1.+
    ($anonfun$f$1$detach.this.w$1.elem()).+
    ($anonfun$f$1$detach.this.x$1).+
    ($anonfun$f$1$detach.this.$outer.y()).+
    ($anonfun$f$1$detach.this.$outer.z());
  };
  //..
};
```

Listing 7.28: Detached closure inside a member function (4/4).

Example 5: Inside a local function

In Listing 7.29 we consider a function value defined inside the local function `g`. Here, the Scala compiler front-end transforms `g` into the lifted method `g$1` (Figure 7.30, line 32) and adds a reference parameter (line 33) with type `IntRef` for the local mutable variable `w`.

```
class C {
  val y = 1
  var z = 2
  def f(x: Int): Int = {
    val v = 3
    var w = 4
    def g(u: Int): Int = {
      ((r: Int) => {w+=1; z+=1; r + u + v + w + x + y + z})(x)
    }
    g(4)
  }
}
```

Listing 7.29: Scala closure inside a local function.

```
class C extends java.lang.Object with ScalaObject {
2  //...
  def y(): Int = C.this.y;
4  def z(): Int = C.this.z;
  def z_=(x$1: Int): Unit = C.this.z = x$1;
6  def f(x$1: Int): Int = {
    val v$1: Int = 3;
8    var w$1: scala.runtime.IntRef = new scala.runtime.IntRef(4);
    C.this.g$1(4, x$1, v$1, w$1)
10  };
  class $anonfun$g$1$1 extends java.lang.Object
12  with Function1 with ScalaObject {
    def this($outer: C, x$1: Int,
14             v$1: Int, w$1: scala.runtime.IntRef,
             u$1: Int): $anonfun$g$1$1 = { /*..*/ };
16    def apply(r: Int): Int = {
      $anonfun$g$1$1.this.w$1.elem =
18        $anonfun$g$1$1.this.w$1.elem.+(1);
      $anonfun$g$1$1.this.$outer.z_=(
```

```
20          $anonfun$g$1$1.this.$outer.z().+(1));
          r.+
22        ($anonfun$g$1$1.this.u$1).+
          ($anonfun$g$1$1.this.v$1).+
24        ($anonfun$g$1$1.this.w$1.elem).+
          ($anonfun$g$1$1.this.x$1).+
26        ($anonfun$g$1$1.this.$outer.y()).+
          ($anonfun$g$1$1.this.$outer.z())
28      };
        def C$$anonfun$$$outer(): C = $anonfun$g$1$1.this.$outer;
30      //...
      };
32    def g$1(u$1: Int, x$1: Int,
              v$1: Int, w$1: scala.runtime.IntRef): Int =
34      scala.Int.unbox({
          (new $anonfun$g$1$1(C.this, x$1, v$1, w$1, u$1): Function1)
36      }.apply(scala.Int.box(x$1)))
    }
```

Listing 7.30: Scala closure inside a local function (converted).

Again, given the source code in Listing 7.29, we mark the function literal `(r: Int) => {..}` with the `detach` primitive (Listing 7.31) and present the corresponding transformed code.

```
import scala.remoting.detach
class A {
  val y = 1
  var z = 2
  def f(x: Int): Int = {
    val v = 3
    var w = 4
    def g(u: Int): Int = {
      detach((r: Int) => {w+=1; z+=1; r + u + v + w + x + y + z})(x)
    }
    g(4)
  }
}
```

Listing 7.31: Detached closure inside a local function.

```
class A extends java.lang.Object with ScalaObject {
  //..
  def y(): Int = A.this.y;
  def z(): Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  def f(x$1: Int): Int = {
    val v$1: Int = 3;
    var w$1: scala.runtime.IntRef = new scala.runtime.IntRef(4);
    A.this.g$1(4, x$1, v$1, w$1)
  };
  def g$1(u$1: Int, x$1: Int, v$1: Int,
              w$1: scala.runtime.IntRef): Int =
    scala.Int.unbox({
      val proxy$1: A$proxy =
        scala.runtime.RemoteRef.bind(
          "A/proxy$1",
          new A$proxyImpl("A/proxy$1", A.this)
        ).$asInstanceOf[A$proxy]();
      val proxy$2: scala.runtime.remoting.RemoteIntRef =
        scala.runtime.RemoteRef.bind(
          "A/proxy$2",
          new scala.runtime.remoting.RemoteIntRefImpl("A/proxy$2", w$1)
        ).$asInstanceOf[scala.runtime.remoting.RemoteIntRef]();
      (new $anonfun$g$1$1$detach(
        proxy$1, x$1, v$1, proxy$2, u$1): Function1)
    }).apply(scala.Int.box(x$1)))
  }
};
```

Listing 7.32: Detached closure inside a local function (1/4).

```
@remote
trait A$proxy extends java.lang.Object
with java.rmi.Remote with ScalaObject {
  def y(): Int;
  def z(): Int;
  def z_=(x$1: Int): Unit
};
```

Listing 7.33: Detached closure inside a local function (2/4).

```
class A$proxyImpl extends java.rmi.server.UnicastRemoteObject
with A$proxy with java.rmi.server.Unreferenced with ScalaObject {
  //...
  def this(x$2: String, x$3: A): A$proxyImpl = { /*..*/ };
  def unreferenced(): Unit = {
    Debug.info("unreferenced: ".+(A$proxyImpl.this.x$2));
    RemoteRef.unbind(A$proxyImpl.this.x$2);
  };
  def y(): Int = A$proxyImpl.this.x$3.y();
  def z(): Int = A$proxyImpl.this.x$3.z();
  def z_=(x$1: Int): Unit = A$proxyImpl.this.x$3.z_=(x$1)
};
```

Listing 7.34: Detached closure inside a local function (3/4).

```
@serializable @SerialVersionUID(8559650693438568463L)
class $anonfun$g$1$1$detach
extends java.lang.Object with Function1 with ScalaObject {
  def this($outer: A$proxy, x$1: Int,
           v$1: Int, w$1: scala.runtime.remoting.RemoteIntRef,
           u$1: Int): $anonfun$g$1$1$detach = { /*..*/ };
  def apply(r: Int): Int = {
    $anonfun$g$1$1$detach.this.w$1.elem_=(
      $anonfun$g$1$1$detach.this.w$1.elem().+(1));
    $anonfun$g$1$1$detach.this.$outer.z_=(
      $anonfun$g$1$1$detach.this.$outer.z().+(1));
    r.+
    ($anonfun$g$1$1$detach.this.u$1).+
    ($anonfun$g$1$1$detach.this.v$1).+
    ($anonfun$g$1$1$detach.this.w$1.elem()).+
    ($anonfun$g$1$1$detach.this.x$1).+
    ($anonfun$g$1$1$detach.this.$outer.y()).+
    ($anonfun$g$1$1$detach.this.$outer.z());
  };
  //..
};
```

Listing 7.35: Detached closure inside a local function (4/4).

Example 6: Inside a closure

In the source code of Listing 7.36 the function value is defined (and applied) inside another closure. Since lexical closures in Scala are handled like normal classes by the Scala compiler this example appears to be a special case of Example 3.

```
class A {
  val y = 1
  var z = 2
  ((w: Int) =>
    w + ((x: Int) => {z += 1; w + x + y + z})(2)
  )
}
```

Listing 7.36: Scala closure inside a closure.

```
class A extends java.lang.Object with ScalaObject {
  //...
  def y(): Int = A.this.y;
  def z(): Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  {
    (new $anonfun$1(A.this): Function1)
  };
  class $anonfun$1
  extends java.lang.Object with Function1 with ScalaObject {
    def this($outer: A): $anonfun$1 = { /*..*/ };
    def apply(w: Int): Int = w.+(scala.Int.unbox({
      (new $anonfun$apply$1($anonfun$1.this): Function1)
    }.apply(scala.Int.box(2))));
    def A$$anonfun$$$outer(): A = $anonfun$1.this.$outer;
    //...
    class $anonfun$apply$1
    extends java.lang.Object with Function1 with ScalaObject {
      def this($outer: $anonfun$1, w$1: Int):
              $anonfun$apply$1 = { /*..*/ };
      def apply(x: Int): Int = {
        $anonfun$apply$1.this.$outer.A$$anonfun$$$outer().z_=(
          $anonfun$apply$1.this.$outer.A$$anonfun$$$outer().z().+(1));
        $anonfun$apply$1.this.w$1.+
```

```
        (x).+
        ($anonfun$apply$1.this.$outer.A$$anonfun$$$outer().y()).+
        ($anonfun$apply$1.this.$outer.A$$anonfun$$$outer().z());
      };
      def A$$anonfun$$anonfun$$$outer(): $anonfun$1 =
        $anonfun$apply$1.this.$outer;
      //...
    }
  }
}
```

Listing 7.37: Scala closure inside a closure (converted).

```
import scala.remoting.detach
class A {
  val y = 1
  var z = 2
  ((w: Int) =>
    w + detach((x: Int) => {z += 1; w + x + y + z})(2)
  )
}
```

Listing 7.38: Detached closure inside a closure.

```
class A extends java.lang.Object with ScalaObject {
  //..
  def y(): Int = A.this.y;
  def z(): Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  {
    (new $anonfun$1(A.this): Function1)
  };
  class $anonfun$1 extends java.lang.Object
  with Function1 with ScalaObject {
    def this($outer: A): $anonfun$1 = { /*..*/ }
    def apply(w: Int): Int = w.+(scala.Int.unbox({
      val uid$1: String = new java.rmi.server.UID().toString();
      val proxy$1: $anonfun$1$proxy = scala.runtime.RemoteRef.bind(
        "A/$anonfun$1/proxy$1$".+(uid$1),
        new $anonfun$1$proxyImpl(
```

```
          "A/$anonfun$1/proxy$1$".+(uid$1), $anonfun$1.this)
      ).$asInstanceOf[$anonfun$1$proxy]();
      (new $anonfun$apply$1$detach(proxy$1): Function1)
    }.apply(scala.Int.box(2))));
    //..
    def A$$anonfun$$$outer(): A = $anonfun$1.this.$outer;
  }
};
```

Listing 7.39: Detached closure inside a closure (1/4).

```
@remote
trait $anonfun$1$proxy extends java.lang.Object
with java.rmi.Remote with ScalaObject {
  def $outer(): A
};
```

Listing 7.40: Detached closure inside a closure (2/4).

```
class $anonfun$1$proxyImpl extends java.rmi.server.UnicastRemoteObject
with $anonfun$1$proxy with java.rmi.server.Unreferenced with ScalaObject {
  //..
  def this(x$1: String, x$2: $anonfun$1) /*..*/
  def unreferenced(): Unit = {
    scala.remoting.Debug.info(
      "unreferenced: ".+($anonfun$1$proxyImpl.this.x$1));
    scala.runtime.RemoteRef.unbind($anonfun$1$proxyImpl.this.x$1)
  };
  def $outer(): A =
    $anonfun$1$proxyImpl.this.x$2.A$$anonfun$$$outer().A$$anonfun$$$outer()
};
```

Listing 7.41: Detached closure inside a closure (3/4).

```
@serializable @SerialVersionUID(9085527522896121L)
class $anonfun$apply$1$detach extends java.lang.Object
with Function1 with ScalaObject {
  def this($outer: $anonfun$1$proxy, w$1: Int) /*..*/
  def apply(x: Int): Int = {
```

```
    $anonfun$apply$1$detach.this.$outer.A$$anonfun$$$outer().z_=(
      $anonfun$apply$1$detach.this.$outer.A$$anonfun$$$outer().z().+(1));
    $anonfun$apply$1$detach.this.w$1.+
    (x).+
    ($anonfun$apply$1$detach.this.$outer.A$$anonfun$$$outer().y()).+
    ($anonfun$apply$1$detach.this.$outer.A$$anonfun$$$outer().z())
  };
  //..
  def A$$anonfun$$anonfun$$$outer(): $anonfun$1$proxy =
    $anonfun$apply$1$detach.this.$outer;
};
```

Listing 7.42: Detached closure inside a closure (4/4).

Example 7: Referring out of scope objects

In the source code of Listing 7.43 the function literal `(x: Int) => {..}` accesses the two objects `B` and `Math` directly.

```
object B {
  var z = 0
}
class A {
  object C {
    var w = 0
  }
  val y = 1
  ((x: Int) => {B.z += 1; x + y + Math.max(B.z, C.w)})
}
```

Listing 7.43: Scala closure referring out of scope objects.

```
class B extends java.lang.Object with ScalaObject {
  //...
  def z(): Int = B.this.z;
  def z_=(x$1: Int): Unit = B.this.z = x$1
};
class A extends java.lang.Object with ScalaObject {
  //...
  class C extends java.lang.Object with ScalaObject {
    //...
    def this($outer: A): A#C = { /*..*/ };
    def w(): Int = C.this.w;
    def w_=(x$1: Int): Unit = C.this.w = x$1;
    def A$C$$$outer(): A = C.this.$outer
  };
  def C(): A#C = { /*..*/ };
  def y(): Int = A.this.y;
  {
    (new $anonfun$1(A.this): Function1)
  };
  class $anonfun$1 extends java.lang.Object
  with Function1 with ScalaObject {
    //...
    def this($outer: A): $anonfun$1 = { /*..*/ };
```

```
    def apply(x: Int): Int = {
      B.z_=(B.z().+(1));
      x.+
      ($anonfun$1.this.$outer.y()).+
      (scala.Math.max(B.z(), $anonfun$1.this.$outer.C().w()))
    };
    def A$$anonfun$$$outer(): A = $anonfun$1.this.$outer;
  }
};
```

Listing 7.44: Scala closure referring out of scope objects (converted).

As above the detached function `(x: Int) => {..}` in Listing 7.45 accesses the two objects `B` and `Math` directly. However, object `Math` is now handled as an ubiquitous reference (Listing 7.51) since it is part of the Scala standard library, while object `B` is accessed through a remote proxy (Listing 7.47).

```
import scala.remoting._
object B {
  var z = 0
}
class A {
  object C {
    var w = 0
  }
  val y = 1
  detach((x: Int) => {B.z += 1; x + y + Math.max(B.z, C.w)})
}
```

Listing 7.45: Detached closure referring out of scope objects.

```
class B extends java.lang.Object with ScalaObject {
  //...
  def z(): Int = B.this.z;
  def z_=(x$1: Int): Unit = B.this.z = x$1;
};
class A extends java.lang.Object with ScalaObject {
  //...
  class C extends java.lang.Object with ScalaObject {
    //...
```

```
    def this($outer: A): A#C = /*..*/
    def w(): Int = C.this.w;
    def w_=(x$1: Int): Unit = C.this.w = x$1;
    def A$C$$$outer(): A = C.this.$outer
  };
  def C(): A#C = { /*..*/ };
  def y(): Int = A.this.y;
  {
    val uid$1: java.lang.String =
      new java.rmi.server.UID().toString();
    val proxy$2: B$proxy = scala.runtime.RemoteRef.bind(
      "A/proxy$2$".+(uid$1),
      new B$proxyImpl("A/proxy$2$".+(uid$1), B)
    ).$asInstanceOf[B$proxy]();
    val proxy$3: A$proxy = scala.runtime.RemoteRef.bind(
      "A/proxy$3$".+(uid$1),
      new A$proxyImpl("A/proxy$3$".+(uid$1), A.this)
    ).$asInstanceOf[A$proxy]();
    (new A$$anonfun$1$detach(proxy$2, proxy$3): Function1)
  };
};
```

Listing 7.46: Detached closure referring out of scope objects (1/6).

```
@remote
trait B$proxy extends java.lang.Object
with java.rmi.Remote with ScalaObject {
  def z(): Int;
  def z_=(x$1: Int): Unit
};
```

Listing 7.47: Detached closure referring out of scope objects (2/6).

```
class B$proxyImpl extends java.rmi.server.UnicastRemoteObject
with B$proxy with java.rmi.server.Unreferenced with ScalaObject {
  //...
  def this(x$1: java.lang.String, x$2: object B): B$proxyImpl = /*..*/
  def unreferenced(): Unit = {
    scala.remoting.Debug.info("unreferenced: ".+(B$proxyImpl.this.x$1));
    scala.runtime.RemoteRef.unbind(B$proxyImpl.this.x$1)
  };
```

```
  def z_=(x$1: Int): Unit = B$proxyImpl.this.x$2.z_=(x$1);
  def z(): Int = B$proxyImpl.this.x$2.z();
};
```

Listing 7.48: Detached closure referring out of scope objects (3/6).

```
@remote
trait A$proxy extends java.lang.Object
with java.rmi.Remote with ScalaObject {
  def C(): A#C;
  def y(): Int
};
```

Listing 7.49: Detached closure referring out of scope objects (4/6).

```
class A$proxyImpl extends java.rmi.server.UnicastRemoteObject
with A$proxy with java.rmi.server.Unreferenced with ScalaObject {
  //...
  def this(x$3: java.lang.String, x$4: A): A$proxyImpl = /*..*/
  def unreferenced(): Unit = {
    scala.remoting.Debug.info(
      "unreferenced: ".+(A$proxyImpl.this.x$3));
    scala.runtime.RemoteRef.unbind(A$proxyImpl.this.x$3)
  };
  def C(): A#C = A$proxyImpl.this.x$4.C();
  def y(): Int = A$proxyImpl.this.x$4.y();
};
```

Listing 7.50: Detached closure referring out of scope objects (5/6).

```
@serializable @SerialVersionUID(261399656060227L)
class A$$anonfun$1$detach extends java.lang.Object
with Function1 with ScalaObject {
  //...
  def this(proxy$1: B$proxy, $outer: A$proxy) = /*..*/
  def apply(x: Int): Int = {
    A$$anonfun1detach.this.proxy$1.z_=(
      A$$anonfun$1$detach.this.proxy$1.z().+(1));
    x.+
    (A$$anonfun1detach.this.$outer.y()).+
    (scala.Math.max(A$$anonfun$1$detach.this.proxy$1.z(),
```

```
                    A$$anonfun$1$detach.this.$outer.C().w()))
  };
};
```

Listing 7.51: Detached closure referring out of scope objects (6/6).

Figure 7.2 represents the two transformations graphically: the diagram above the dashed line corresponds to the transformed closure with its outer class `A` (see Listing 7.44) and the diagram below the dashed line describes the interaction between the transformed closure marked with `detach` and the same class `A`.

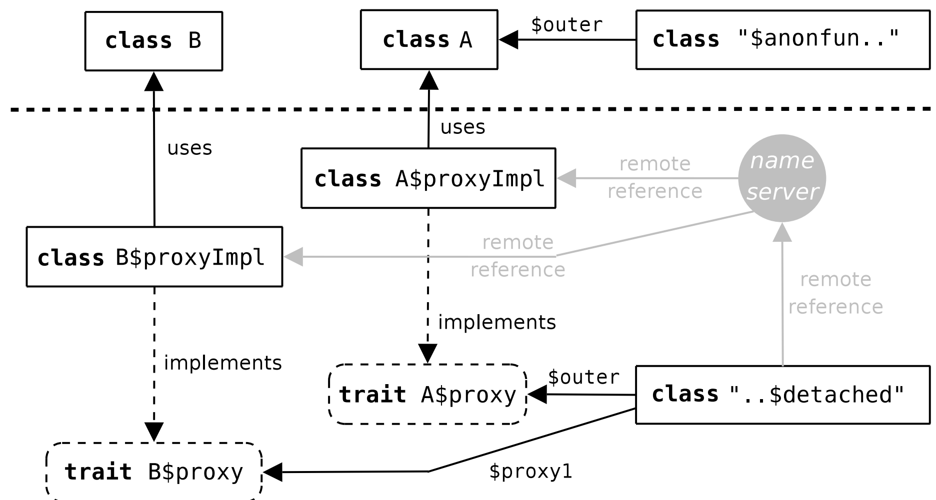


Figure 7.2: Detached closure and free variables.

7.2 Extending Scala

The programming support for mobile code is implemented as an extension (see Listing 7.52) of the Scala compiler and the Scala standard library developed at EPFL.

First, the Scala compiler front-end is augmented with the new transformation phase `Detach`; second, the Scala standard library is extended with the package `scala.runtime.remoting` which provides the required run-time support for the Java VM and the package `scala.remoting` as a simple programming interface (API).

```
1220 compiler/scala/tools/nsc/transform/Detach.scala
 231 library/scala/remoting/Channel.scala
  27 library/scala/remoting/Debug.scala
  49 library/scala/remoting/detach.scala
  68 library/scala/remoting/ServerChannel.scala
 182 library/scala/runtime/RemoteRef.scala
  85 library/scala/runtime/remoting/Debug.scala
 192 library/scala/runtime/remoting/RegistryDelegate.scala
  50 library/scala/runtime/remoting/RemoteBooleanRef.scala
  50 library/scala/runtime/remoting/RemoteByteRef.scala
  50 library/scala/runtime/remoting/RemoteCharRef.scala
  49 library/scala/runtime/remoting/RemoteDoubleRef.scala
  49 library/scala/runtime/remoting/RemoteFloatRef.scala
  50 library/scala/runtime/remoting/RemoteIntRef.scala
  50 library/scala/runtime/remoting/RemoteLongRef.scala
  50 library/scala/runtime/remoting/RemoteObjectRef.scala
  49 library/scala/runtime/remoting/RemoteShortRef.scala
2501 total
```

Listing 7.52: Extending Scala with detached closures.

7.2.1 Scala Compiler

The overall structure of the Scala compiler developed at EPFL follows the traditional compiler structure consisting of a front-end and a back-end, each one working on its own internal representation.

Over 80% of the compiler complexity is concentrated in the front-end (with around 60% for the type-checker alone), testifying of the high degree of expressiveness featured by the language Scala.

Both the front-end and the back-end are organized as a graph of successive phases (Figure 7.3):

- The front-end phases analyze the source files, reporting errors if the input files don't make up a valid Scala program, and generate a fully attributed abstract syntax tree (AST) . Then, the AST is successively simplified by several transformation phases [3] — including high-level optimizations such as the elimination of tail-recursive calls — until it looks very similar to Java code.
- The back-end phases transform the AST into a portable intermediate code, perform several code optimizations — such as code inlining, tail call elimination and dead code elimination — and finally generate either Java class files or .NET assemblies.

NOTE

> The rewriting of the Scala 2 compiler coincides with the reordering of several transformation phases [3]. In the new compiler, the `Erasure` phase comes early and in particular now precedes the `LambdaLift` phase. On one side, that change has the advantage that later phases do not have to handle the most complex Scala types; on the other side, erasing types early may prevent a later phase from performing some optimizations relying on type informations.

Transformation Phases

Transformation phases (see right part in Figure 7.3) perform either tree transformations or both tree and type transformations.

Tree transformers are applied to the AST of the compiled program while type transformers are applied to the type of all identifiers referenced during compilation, i.e. defined in the AST or loaded from files compiled separately.

Concretely, transformation phases analyze their input tree, looking for the concept(s) they should translate away or substitute with extended trees, and return a possibly transformed tree. For example, the `UnCurry` phase transforms curried functions into multi-argument functions.

NOTE

> The `Typer` phase has two important functions in relation with the transformation phases: first, it ensures that those (later) phases perform their tasks on a valid Scala program, and, second, it checks that the successive transformations applied to the program preserve type correctness.

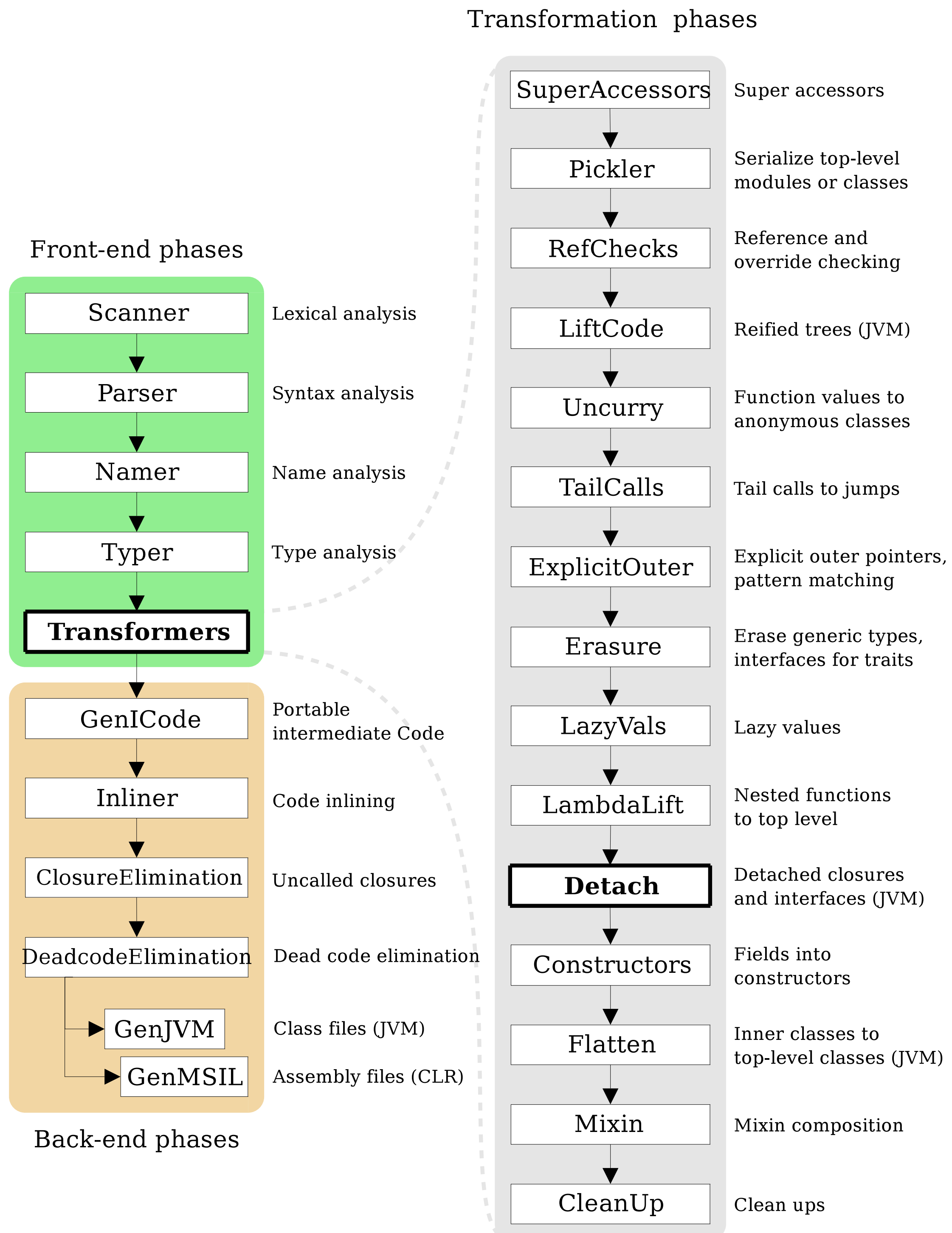


Figure 7.3: Scala compiler phases.

The addition of a new transformation phase for the conversion of detached closures is described in detail in Section 7.2.2; we present here several transformation phases that interact in some ways with the phase to be added.

`ExplicitOuter` Phase The `ExplicitOuter` phase comes after the `UnCurry` phase and before the `Erasure` phase (see Figure 7.3).

Nested classes in Scala have full access to the members of their enclosing class(es). The task of the `ExplicitOuter` phase is to lift nested classes to the top-level scope by augmenting them with an explicit reference to their enclosing object. This reference is then used to access all the members of enclosing classes.

The `ExplicitOuter` phase performs several transformations on types, e.g. it adds an outer parameter to the formal parameters of a constructor in a inner non-trait class or it adds an outer accessor `outer$C` to every inner class with fully qualified name `C` that is not an interface.

Also, the `ExplicitOuter` phase performs several transformations on terms, e.g. a class which is not an interface and is not static gets an outer accessor, a reference `C.this` where `C` refers to an outer class is replaced by a selection `this.$outer$$C`.

`Erasure` Phase The `Erasure` phase performs a partial erasure of the types appearing in a Scala program. Partial erasure maps Scala types into JVM types respectively CLR types [103]. Thus, a Scala program compiled to JVM bytecode must be annotated with valid Java types. Erased types are Java types that define a type discipline on the program that is equivalent to, although less precise than that of the program typed with Scala types [89, §3.6].

Generics in Java 5 [48] are a well-known example of partial erasure mapping, a translation initially implemented in GJ [19], an extension of Java with generic types and methods. In that case, the erasure consists of removing type parameters and replacing type variables by the erasure of their bounds.

The `Erasure` phase performs several transformations on types, e.g. it replaces a type-bound structure by the erasure of its upper bound, it erases type references `scala.Any` or `scala.AnyVal` to `java.lang.Object`, it replaces a non-empty type intersection by the erasure of its first parent.

`LambdaLift` Phase Classes and functions in Scala can be defined inside many different contexts e.g. blocks, classes, objects or functions. Classes

and objects can additionally appear inside a package — either a user-specified package or the predefined empty package —; we call them top-level classes respectively top-level objects.

The task of the `LambdaLift` phase is to lift classes either to the top-level scope or to the scope of their enclosing class; functions are transformed into members of their enclosing class (e.g. private methods). Lifted classes and functions get additional parameters for the free variables appearing in their (initial) bodies. Mutable variables are turned into reference cells.

Transformation phases invoked after `LambdaLift` (which itself comes after `ExplicitOuter`) thus see a Scala program as a set of top-level classes.

Transformation Example

Given the code example from Listing 7.1, the `Typer` phase introduces accessor functions for the fields `y` and `z` — e.g. a getter method `def z: Int` and a setter method `def z_=(x$1: Int)` for field `z` — and updates accordingly their references in the function body. Listing 7.53 presents the transformed code ready to be processed by the later transformation phases.

```
class A extends java.lang.Object with ScalaObject {
  private[this] val y: Int = 1;
  def y: Int = A.this.y;
  private[this] var z: Int = 2;
  def z: Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  ((x: Int) => {
    A.this.z_=(A.this.z.+(1));
    x.+(A.this.y).+(A.this.z))
  }
  //..
}
```

Listing 7.53: Scala closure inside a class (`Typer`).

In the following we present the closure transformations performed by the three phases `Uncurry`, `ExplicitOuter` and `LambdaLift`.

The `Uncurry` phase translates the function value to an anonymous class with arity 1 (unary function) and body for the method `apply` (Listing 7.54).

```
class A extends java.lang.Object with ScalaObject {
  //..
  def y(): Int = A.this.y;
  def z(): Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  {
    class $anonfun
    extends java.lang.Object with (Int) => Int with ScalaObject {
      def this(): $anonfun = //..
      final def apply(x: Int): Int = {
        A.this.z_=(A.this.z().+(1));
        x.+(A.this.y()).+(A.this.z())
      }
    };
    (new $anonfun(): (Int) => Int)
  }
}
```

Listing 7.54: Scala closure inside a class (`Uncurry`).

The `ExplicitOuter` phase replaces the reference to the enclosing class with the outer field `$outer` and updates the constructor parameter list and the body of the method `apply` accordingly (Listing 7.55).

```
class A extends java.lang.Object with ScalaObject {
  //..
  def y(): Int = A.this.y;
  def z(): Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  {
    class $anonfun
    extends java.lang.Object with (Int) => Int with ScalaObject {
      def this($outer: A.this.type): $anonfun = //..
      final def apply(x: Int): Int = {
        $anonfun.this.$outer.z_=($anonfun.this.$outer.z().+(1));
        x.+($anonfun.this.$outer.y()).+($anonfun.this.$outer.z())
      };
      private[this] val $outer: A.this.type = _;
      def A$$anonfun$$$outer(): A = $anonfun.this.$outer
    };
```

```
    (new $anonfun(A.this): (Int) => Int)
  }
}
```

Listing 7.55: Scala closure inside a class (`ExplicitOuter`).

Then the `LambdaLift` phase moves (and renames) the anonymous class `$anonfun` to the scope of its enclosing class[6] (Listing 7.56).

```
class A extends java.lang.Object with ScalaObject {
  //..
  def y(): Int = A.this.y;
  def z(): Int = A.this.z;
  def z_=(x$1: Int): Unit = A.this.z = x$1;
  {
    (new $anonfun$1(A.this): Function1)
  };
  class $anonfun$1
  extends java.lang.Object with Function1 with ScalaObject {
    def this($outer: A): $anonfun$1 = //..
    def apply(x: Int): Int = {
      $anonfun$1.this.$outer.z_=(
        $anonfun$1.this.$outer.z().+(1));
      x.+
      ($anonfun$1.this.$outer.y()).+
      ($anonfun$1.this.$outer.z());
    };
    //..
  }
}
```

Listing 7.56: Scala closure inside a class (`LambdaLift`).

7.2.2 Detach Phase

The `Detach` phase follows `LambdaLift` (see Figure 7.3) in the transformation process applied to the internal tree representation and looks for applications of the marker object `detach` to function literals.

[6]The `Flatten` phase moves `$anonfun$1` to the top-level scope when targeting the Java VM.

Concretely, the `Detach` transformation phase proceeds in three successive steps when processing a compilation unit:

- first it traverses the AST to gather the relevant symbol and tree informations (e.g. captured `this` instance);
- then it adds new proxy symbols to the top-level scope and generates the corresponding trees (added later to the appropriate statement block by the method `transformStats`);
- finally, it transforms both the class definition and the class instantiation of detached closures.

Tree traversal Informations collected during the tree traversal include:

- a `HashMap[Symbol, SymSet]` with function calls occuring in detach closures,
- a `HashMap[Symbol, SymSet]` with module objects captured by detached closures,
- a `HashMap[Symbol, ClassDef]` with class definitions containing a tree node `Apply` for the marker object `detach`;
- a `HashMap[Symbol, Symbol]` with `this` instances captured by detached closures;
- a `HashMap[Tree, Apply]` with expressions containing a tree node `Apply` for object `detach` and instantiations of detached closures.

Proxy generation Proxy class definitions for the objects referenced in the body of the detached closures are added to the top-level scope. Proxy interfaces extend the RMI interface `Remote` and proxy implementations extend the RMI class `UnicastRemoteObject`.

Closure transformation The operations performed during the tree transformation include:

- the transformation of the closure definition representing the function value marked with the object `detach`,
- the transformation of the original closure instantiation into a block expression where remotely accessed references are dynamically bound before the detached closure is actually instantiated.

7.2.3 Scala Run-time

The Scala standard library is extended with the package `scala.runtime` which provides the RMI-based run-time support for the transformation phase `Detach` described in Section 7.2.2.

While Scala programmers are not supposed to access those functionalities it is nevertheless possible to write Scala source code by hand which behaves the same as the generated one.

- The object `RemoteRef` in package `scala.runtime` provides several facilities to setup the name server and bind/unbind remote objects to/from the name server.

- The run-time classes `RemoteByteRef`, `RemoteCharRef`, etc.. defined in package `scala.runtime.remoting` are dual to `ByteRef`, `CharRef`, etc.. from package `scala.runtime`; they implement remote references[7] for primitive types such as `Byte`, `Char`, etc..

NOTE

> In order to enforce some level of security, the standard RMI registry implementation only allows processes on the same host to register objects in the registry. So, by design, if a process tries to bind an object to a remote registry an exception will be thrown.
>
> The simplest technical solution to the remote registration problem is to have a registry delegate. A registry delegate is an object that serves as a proxy for the real registry. The delegate itself usually appears in the registry under a well known name. It implements the `Registry` interface and simply delegates all method calls to the appropriate methods of the real registry. The delegate is allowed to perform bind and unbind operations because it is running on the same host as the registry.
>
> The class `RegistryDelegate` in package `scala.runtime.remoting` implements the registry delegate concept.

7.2.4 Scala API

The marker object `detach` in package `scala.remoting` is a compiler hint[8] used to control the generation of the detached closures and their associated remote proxies. The overloaded methods `apply` defined in object `detach` (see Listing 7.57) implement the identity function by default.

[7] By inheriting from the RMI class `UnicastRemoteObject`.

[8] The conversion is currently enabled with the compiler option `-Ydetach`.

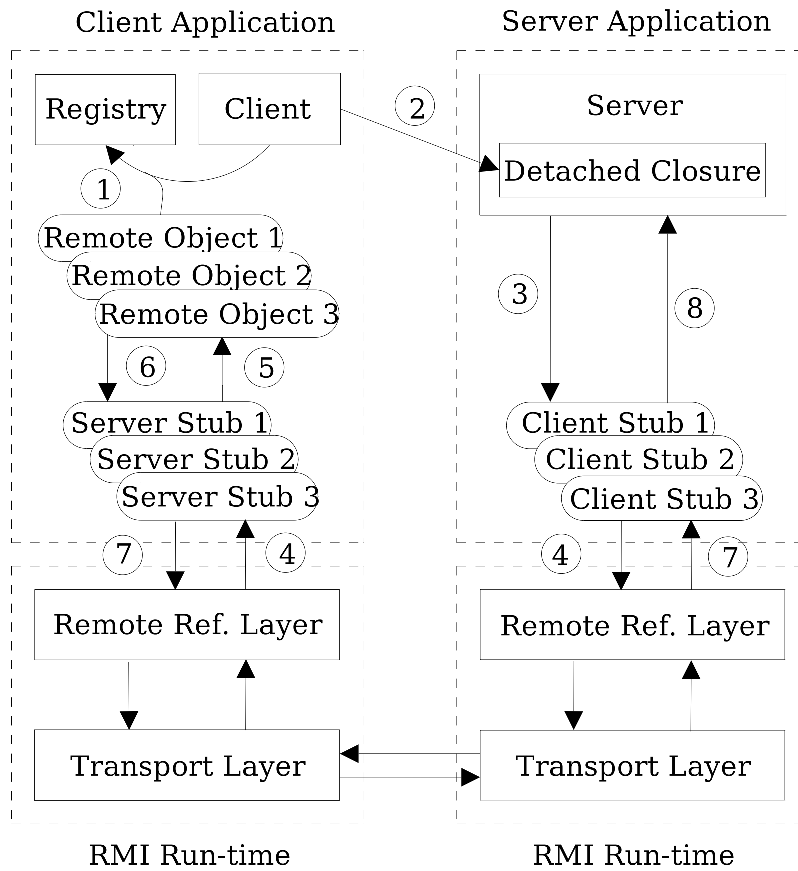


Figure 7.4: Detached Scala closure architecture.

```
package scala.remoting
object detach {
  def apply[R](f: Function0[R]): Function0[R] = f
  def apply[T0, R](f: Function1[T0, R]): Function1[T0, R] = f
  //... (function arities 2 up to 22)
}
```

Listing 7.57: The marker object `detach` (Scala API).

7.3 Java Platform

In this work we limit ourselves to *weak mobility* (no migration of execution state, see Section 3.3.1) as we intend to use an unmodified Java VM [72] as computational environment.

Implementing *strong mobility* (migration of code and execution state, see Section 3.1.2) in Java is indeed difficult. In order to move computations transparently the call stack needs to be preserved across migration. But the Java security policy forbids Java bytecode itself to manipulate the stack. Different approaches have been proposed for realizing transparent migration in Java: extending a Java VM [110], transforming source code [108], or transforming bytecode [102].

While garbage collection, dynamic class loading and security enforcement are under the responsability of the Java run-time environment type safety is ensured both by the static type checker and the bytecode verifier.

Distributed Garbage Collection In order to ensure that unreachable remote objects are unexported and garbage collected in a timely fashion we reduce the maximum interval that the RMI run-time allows between garbage collection of the local heap to one second (default is 60 seconds) using the Sun-specific property `sun.rmi.dgc.server.gcInterval`[9].

NOTE

In order to operate in a timely fashion, the DGC attempts to run local asynchronous garbage collection periodically ([116]). As a result, full GC cycles run at fixed intervals regardless of any attempts the local GC makes to avoid them.

Maintaining up-to-date awareness of all unreachable objects in a given JVM conflicts with the goals of a generational GC. To resolve this conflict, the RMI run-time provides the two Sun-specific system properties `sun.rmi.dgc.server.gcInterval` and `sun.rmi.dgc.client.gcInterval` to decrease the frequency with which RMI requests client and server-side asynchronous GC. The DGC is responsible for maintaining leases. On the server side, when all the outstanding leases to a particular server object expire, the DGC makes sure the RMI run-time does not retain any references to the server, thereby enabling the server to the garbage collected.

During the process, if the server implements the `Unreferenced` interface, the server can find out if there are no longer any open leases, `Unreferenced` consists of exactly one method: **`public void`** `unreferenced()`. This can be useful when the server needs to immediately release resources instead of waiting for garbage collection to occur.

[9] `sun.rmi.*` properties are not part of the public Java API and are subject to changes in future implementations.

Dynamic Class Loading We use the class `RMIClassLoader` of the Java RMI library to dynamically load classes referenced by the detached closures. `RMIClassLoader` requires both a codebase and a security policy file to be provided at execution time (see Listing 5.10).

Security In this work we rely on the Java VM to handle security issues such as access control (security policies [75]), user authentication (e.g. digital certificates), data integrity (e.g. code signing [51]) and data confidentiality (through encryption).

Type Safety The Scala language (see Section 1.3.1) features an advanced type system with powerful language abstractions that are statically checked. For the type-safe execution of Scala programs we rely on the Java VM whose bytecode verifier checks well-typedness at the bytecode level[92, 129].

7.4 Discussion

In this section we discuss both implementation considerations and possible improvements of the realized software extension.

They are two points worth mentioning regarding the implementation of our solution: on the plus side, the compiler redesign initiated with the language upgrade from Scala 1 to Scala 2 in March 2006 has notably facilitated the addition of a new front-end phase; on the negative side, the support for RTTI's proposed by Schinz [103] — a required feature in a distributed environment (e.g. [109]) — never became part of the main software distribution, mainly due to performance weakness.

Finally, possible ways of improvement include the performance of Java RMI and the secure transmission of the mobile code.

Performance

The performance of Java RMI depends on three main factors: object serialization, socket communication and the RMI framework itself (protocol and configuration).

Object Serialization Though Java serialization [57] provides great flexibility, its overhead can be quite high. This is because object serialization uses Java's reflection mechanism to determine what data to serialize and

how to serialize it. It is possible to convert objects to byte arrays more efficiently by writing out objects manually with Java's externalization mechanism.

The cost of using the marker interface `java.io.Serializable` is that reflection runs slower than straight method calls. For instance, Greanier[50] observes half-reduced average times when using the hand-written methods `write-/readObject` instead of the default mechanism.

Similarly, Mathew and al. [76] report a performance improvement of 20% or better for (de-)serializing objects using externalization on various Java VM. However, implementing the `java.io.Externalizable` interface requires more coding since read and write methods have to be hand-coded in the child classes. Therefore, future additions to classes using externalization will require more debugging than equivalent additions made using serialization.

Finally, systems in which objects can be serialized cannot guarantee that the environment to which an object initially belongs will be remotely similar to the one in which it was frozen. This can be a problem if the object assumes (as almost all code does) that its environment doesn't suddenly undergo drastic changes. This kind of problem can easily lead to type safety problems and unforseable security risks. Our solution does not address that technical issue, but the Java security framework provides means to do it.

Socket Communication The setup of a new TCP connection requires a three-way handshake between the connected endpoints before data transfer can proceed. On networks with long latency and for short transactions this handshake wastes valuable time [98].

RMI Framework Campadello [24] analyzes the performance of Java RMI for communications over a slow wireless link. For example, in a simple RMI call, the actual invocation takes up only 5% of the total transmitted data while 69% was related to the DGC protocol. He also reports an important performance issue due to the high number of round-trips involved in ping packets and TCP handshaking. Furthermore, Campadello notes that the time spent for object serialization during a RMI call typically amounts for one third of the total time overhead.

The RMI framework provided by Sun has to be considered as a reference implementation which satisfies the programmer's needs in many common cases but still keeps space for performance improvements. Several optimizations and implementations have been proposed [74, 85, 94,

130, 123] to increase the efficiency of Java RMI.

Security

Cryptographic techniques combined with socket communication can be used to ensure the reliability and authenticity of the transmitted code. Regarding the safe execution of relocated code the security model provided by the Java run-time is an intrinsic part of Java'architecture.

Socket Communication The `SSLServerSocket` extends `ServerSocket` and provides secure server sockets using protocols such as the SSL or TLS protocols.

RMI Framework The Java VM can for instance be configured to generate cryptographically strong random numbers for object identifiers using the system property `java.rmi.server.randomIDs`.

Chapter 8

Conclusion

Every program has (at least) two purposes: the one for which it was written, and another for which it wasn't.

Alan Perlis[1]

With the growing number of network-connected computers, the concept of distributing services among multiple computers has become increasingly possible and desirable. This concept has been widely implemented in modern operating systems and is raising interest in modern programming languages too.

In this context mobile code makes it possible to define higher-order communication protocols and enables the implementation of expressive distributed computations.

However the presence of run-time features the programmer is only aware of when something fails makes the behavior of a distributed computation much more complex and less understandable than a sequential one. Futhermore, while distribution and mobility of code increase the generality of a programming environment, they almost invariantly imply noticeable performance penalty and some additional security issues.

[1] Alan Perlis was honored with the ACM Turing Award 1966 for his influence in the area of compiler construction; he also contributed to the development of Algol.

8.1 Achievements

We have proposed a new approach to dynamic binding in the context of mobile code which is based on the concept of lambda abstraction and have extended the language Scala with a programming abstraction for mobile code.

Inspired by the language Obliq we have revisited the concept of distributed scope with a new programming abstraction for mobile code which builds upon the concept of lexical closures and benefits from the generalized data binding mechanism of the language Scala.

This thesis has been elaborated in the context of the Scala project at EPFL; both the Scala language and libraries result from a collaborative effort to provide a powerful development environment with advanced programming features. With its uniform programming model and its great expressiveness Scala is an attractive testbed for exploring new ideas in a variety of research fields, including concurrent and distributed programming.

The main achievements of this work are the following:

- We presented a novel approach based on the notion of lambda abstraction for dealing with the dynamic rebinding of local references in a distributed execution environment.

  In programming languages supporting lexical closures, the notion of lambda abstraction provides an intuitive mechanism to model the type-safe evaluation of mobile code.

- We proposed a programming model which preserves the language semantics and relies only on higher-order functions and distributed objects.

  Within our model we introduced the notion of detached closures, the first approach of combining lexical closures with remote references in a distributed execution environment.

- We added programming support for detached closures to Scala by extending the compiler front-end with a new transformation phase and by providing a Java RMI based run-time infrastructure as a lightweight extension of the Scala standard library.

8.2 Open Issues

Currently Scala's support for type manifests is quite basic; indeed class `Manifest` in the package `scala.reflect` of the Scala standard library provides only few functionalities to the programmer and limited type information at execution time (e.g. the subtyping test is done on the erasure of the type). Nevertheless a more elaborated implementation of the Scala manifests with better compiler support is planned for the next future.

More generally, run-time type information would be very useful to extend the Java idea of load-time bytecode verification; for instance, the .NET platform — another target platform of the language Scala — provides such a support in its common language run-time environment.

8.3 Future Work

We see several interesting directions of future work in relation with this project and the language Scala; we sketch here four of them.

- Our programming support was implemented on the Java platform; we think of an implementation based on .NET Remoting as a natural follow-up project since Scala is targeted both at the Java platform and the .NET platform.
- The Scala compiler was extended with a new transformation phase to support our programming model; the generality of our solution and its lightweight integration in the compiler front-end make it a possible candidate to be implemented as a compiler plugin [112].
- The compiler and library extensions were implemented independently from the official Scala distribution; it would require only minimal integration work to make them available in an official software release.
- The Scala Actors library provides the same programming model for local and remote Actors; we think that our programming abstraction for mobile code would benefit from the Actor's support for asynchronous communication in a distributed settings.

Abbreviations

ADT	Abstract Data Type
API	Application Programming Interface
AST	Abstract Syntax Tree
AWT	Abstract Windowing Toolkit
C/S	Client/Server
CAB	Cabinet Archive
CBN	Call-By-Name
CBV	Call-By-Value
CCS	Calculus of Communicating Systems
CLR	Common Language Runtime
CMS	Calculus of Module Systems
COD	Code On Demand
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DGC	Distributed Garbage Collector
DoD	Departement of Defense
DOS	Disk Operating System
DSL	Domain-specific Language
Facile	Functional And Concurrent Integrated Language
FP	Functional Programming
FTP	File Transfer Protocol
GC	Garbage Collector
GJ	Generic Java
HM	Hindley-Milner
HOF	Higher-Order Function
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol

ICFP	International Conference on Functional Programming
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
ILU	Inter-Language Unification
JAR	Java Archive
JDK	Java Development Kit
JDOM	Java Distributed Object Model
JNLP	Java Network Launch Protocol
JOM	Java Object Model
JRE	Java Runtime Environment
JSSE	Java Secure Socket Extension
JVM	Java Virtual Machine
Klaim	Kernel Language for Agent Interaction and Mobility
LNCS	Lecture Notes in Computer Science
LPC	Local Procedure Call
LTI	Local Type Inference
MA	Mobile Agent
MCL	Mobile Code Language
MCS	Mobile Code System
MOS	Mobile Object System
MSIL	Microsoft Intermediate Language
MT	Multi-Threaded
NFS	Network File System
OLE	Object Linking and Embedding
OMG	Object Management Group
OOP	Object-Oriented Programming
ORB	Object Request Broker
PDF	Portable Document Format
PLAN	Packet Language for Active Networks
POA	Portable Object Adapter
PoPL	Principles of Programming Languages
REPL	Read-Eval-Print Loop
REV	Remote Evaluation
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RPCL	Remote Procedure Call Language
RTTI	Run-Time Type Information

SQL	Structured Query Language
SSL	Secure Socket Layer
TCL	Tool Command Language
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TOPLAS	Transactions on Programming Languages and Systems
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VBA	Visual Basic for Applications
VM	Virtual Machine
WSH	Windows Scripting Host
XDR	External Data Representation (RFC 4506)
XML	Extensible Markup Language
XP	Extreme Programming

Glossary

abstraction safety *Abstraction safety* is the property that values of an abstract data type can only be constructed and inspected by the code of its declaration, thus preserving type invariants for all values.

aglet An *aglet* is a Java object that can move from one network host to another, taking along its program code as well as its data. While executing on one host an aglet can thus halt execution, dispatch itself to a remote host, and resume execution there. See *mobile computation*.

applet A Java applet is network-downloadable Java program that is embedded in another application using HTML code.

at-most-once semantics The *at-most-once semantics* is adopted by most distributed systems following the client-server paradigm. Under the at-most-once semantics, either the server executes a remote request (e.g. RPC, REV, RMI, etc.) exactly once and the client receives the results, or the server don't receive the request and the client is so informed. This ensures that partial or multiple evaluations of a request never occur.

bound variable A *bound variable* is a variable referred to in a function that is either a local variable or an argument of that function. See *free variable*.

class linking In Java *class linking* [48, §12.3] is the process of taking a binary form of a class or interface type and combining it into the run-time state of the JVM, so that it can be executed. A class or interface type is always loaded before it is linked. See *class loading*.

class loading In Java *class loading* [48, §12.2] refers to the process of finding the binary form of a class or interface type with a particular name, perhaps by computing it on the fly, but more typically by retrieving a binary representation previously computed from source

code by a compiler, and constructing, from that binary form, a `Class` object to represent the class or interface.

Clojure *Clojure* is both a Lisp-based programming language and a JVM-hosted dynamic environment. While Clojure can be embedded in a Java application or used as a scripting language, its primary programming interface is the REPL.

closure conversion *Closure conversion* is a program transformation used by compilers to separate code from data in higher-order functions. Functions with free variables are replaced by code abstracted on an extra environment parameter. *Free variables* in the body of the function are replaced by references to the environment. The abstracted code is partially applied to an explicitly constructed environment providing the bindings for these variables. This partial application of the code to its environment is in fact suspended until the function is actually applied to its argument; the suspended application, called a *closure*, is a data structure consisting of a piece of pure code and a representation of its environment.

CLU *CLU* is an ALGOL-based programming language created at MIT by Barbara Liskov and her students between 1974 and 1975. Some key features of CLU are iterators, multiple assignments, exception handling and automatic memory management.

code mobility *Code mobility* is the capability to dynamically reconfigure the binding between the software components of a distributed application and their physical location in the computer network. See *mobile code*.

computation mobility See *mobile computation*.

continuation A continuation is a *closure* that represents the current state of execution. Calling a continuation results in the computation resuming from where the continuation was captured.

curried function Function currying is a technique named from its author, H.B. Curry, wherein a function with multiple arguments can be logically represented as a HOF with only a single argument. While an ordinary function with `n` arguments binds all of its arguments at once, a *curried function* binds the first argument and returns another function of `n-1` arguments.

distributed computation A *distributed computation* is a single task performed by more than one network component in a distributed system. See *distributed system*.

distributed system A *distributed system* is an application that executes a collection of protocols to coordinate the actions of multiple processes on a network, such that all components cooperate together to perform a single or small set of related tasks.

domain-specific language A *domain-specific language* (DSL) is a limited form of computer language designed for solving a specific class of problems. The added abstraction layer makes user code more robust and readable and hides the boilerplate code of the underlying implementation. Implementing an external DSL differs from internal DSLs in that the parsing process operates on pure text input which is not constrained by any particular language.

dynamic binding *Dynamic binding* is the concept by which the value of some variable is not fixed statically (at the time it is introduced), but is determined dynamically from the current scope, each time the variable is used.

Concretely *dynamic binding* is a run-time lookup operation which extracts values corresponding to some *names* from some *environments*. See *environment*.

dynamic rebinding *Dynamic rebinding* is the ability of changing the definitions of names at execution time.

Emerald *Emerald* [16, 66] is a distributed, object-oriented programming language that was developed at the University of Washington starting in 1984. The goal of Emerald is to simplify the construction of distributed applications. This goal is reflected at every level of the system: its object structure, the programming language design, the compiler implementation, and the run-time support.

enclave An *enclave* [38] is an information system environment that is end-to-end under the control of a single authority and has an uniform security policy. See *security policy*.

environment An *environment* `e` is a finite set of bindings, where a binding is just a pair of a variable `v` and a term `t` which is usually written as `t/v`.

Erlang *Erlang* is a dynamically typed, single assignment language which uses pattern matching for variable binding and function selection, which has inherent support for lightweight concurrent and distributed processes, and has error detection and recovery mechanisms. It was developed at the Ericsson Computer Science Laboratory to satisfy a requirement for a language suitable for building large soft real-time control systems, particularly telecommunication systems.

explicitly typed A programming language is *explicitly typed* (opposite: *implicitly typed*) if its syntax enforces explicit type declarations for each declaration. Explicit typing is normally considered a specialisation of static typing. See *statically typed*.

Typed language such as Pascal or Java require that types are declared explicitly for all functions and variables in a program. Other typed languages such as Haskell, ML or Scala will infer the type of an expression depending on its structure and context. See *type inference*.

first-class function A *first-class function* (or *function object*) can be stored in data structures, nested, passed as arguments or returned as a result of others functions. See *higher-order function*.

free variable A *free* (or *non-local*) *variable* in an expression `e` is a variable that is used inside `e` but not defined inside `e`. The set of free variables of an expression `e` is defined by induction on the construction of `e`. See *bound variable*.

For instance, in the Scala function literal `(x: Int) => x + y`, both variables `x` and `y` are used, but only `y` is free, because it is is not defined in `e`.

function value In Scala, a *function value* is a function object that can be invoked like any other functions. The class of a function value extends one of the traits `scala.Function0`, `scala.Function1`, etc.. and is typically written using the *function literal* syntax. A function value is "invoked" when its `apply` method is called. A function value that captures free variables is also called a *closure*. See *first-class function*.

higher-order function A *higher-order function* (HOF) is a function which takes one or more functions as arguments and/or returns a function. In the untyped lambda calculus, all functions are higher-order. See *first-class function*.

Klaim *Klaim* [6, 7] is a core programming language for describing mobile agents and their interaction through multiple distributed tuple spaces. Klaim's primitives were heavily influenced by the process algebra and the Linda programming language.

lambda calculus Introduced by Alonzo Church and Stephen Cole Kleene in the 1930's, the lambda calculus (short λ-calculus) is a formal system with a minimal notation to capture the computational aspects of functions.

Informally, it consists of a syntax of terms and a set of rewrite rules for transforming terms.

lambda abstraction A *lambda abstraction* is a value that abstracts over another value. It is commonly called a function.

late binding See *dynamic binding*.

lexical closure A *lexical closure* (or *block closure*) is a execution-delayed function body (or block of code) with its associated context information consisting of bindings to non-local identifiers. Closures allow a programmer to refactor common code segments into a shared utility, with the difference between the use sites being abstracted into a local function or closure. Thus, a Scala closure represents at runtime a function value with bindings to its free variables. See *closure conversion*.

local type inference Type inference is *local* when the type of an expression depends only on the types of its subexpressions, and not on the context in which it occurs. See *type inference*.

malicious mobile code *Malicious mobile code* [38] is a mobile software module designed, employed, distributed, or activated with the intention of compromising the performance or security of information systems and computers, increasing access to those systems, providing the unauthorized disclosure of information, corrupting information, denying service, or stealing resources. See *mobile code*.

marker interface A *marker interface* (or *tagging interface*) is just a way to take advantage of the type identification that an interface provides, without declaring any behavior. Examples of *marker interfaces* in Java are `Cloneable`, `Serializable`, `Remote` and `EventListener` (other examples are also present in the `CORBA` and `security` packages).

A major problem with marker interfaces is that an interface defines a contract for implementing classes, and that contract is inherited by all subclasses. A better solution is to support metadata directly, such as attributes in the .NET framework or class annotations in Java 5 and Scala.

memoized function A *memoized function* "remembers" the results corresponding to some set of specific inputs. Subsequent calls with remembered inputs return the remembered result rather than recalculating it. A function can only be memoized if it is *referentially transparent*. See *referential transparency*.

mobile code *Mobile code* [38] is a software module obtained from remote systems outside the enclave boundary, transferred across a network, and then downloaded and executed on a local system without explicit installation or execution by the recipient.

MCS supporting weak mobility include Sun's NFS, Sun's Java applets and MCS supporting strong mobility include Agent TCL, Telescript, Odyssey, and IBM's Java aglets.

mobile computation *Mobile computation* (or *strong mobility*) requires interrupting the execution, moving the state of a run-time system (stacks, for instance) from one site to another, and then resuming execution.

mobile computing *Mobile computing* is about both physical and logical computing entities that move: physical entities are computers that change locations and logical entities are instances of a running user application or a mobile agent.

Mobile computing is characterized by several constraints: mobile elements are resource-poor relative to static elements, they rely on a finite energy source, their connectivity is highly variable in performance and reliability and their mobility is inherently hazardous.

node In the context of distributed computing a *node* (or *site*, *location*) consists of a computational environment and a node address which uniquely identifies that node throughout the network. Conceptually a network connects a possibly variable number of nodes.

normal form A λ-expression is in *normal form* if it cannot be further reduced using β- or η-reduction. β-reduction is the process whereby expressions are simplified as a result of values being substituted into

function arguments. Thus, the beta reduction of `((λ x. e) e')` is simply `e[x/e']`.

object serialization A serializable object can be written to a stream of bytes and saved to a persitent storage or transmitted over the network. Later one, the same object can be read back from the byte stream. See *serializable type*.

operational semantics The evaluation of an expression is defined in terms of the evaluation of its subexpressions. Each sentence of the form `e -> e'` defines one step in the evaluation so that `e'` is the result of the first step of evaluation of `e`. An evaluation rule may be an axiom in the form of a single sentence or an inference rule where the sentences above the bar represent the hypotheses and the sentences below represent the conclusion. Expressions which cannot be further evaluated are called canonical expressions.

PLAN *PLAN* is a resource-bounded functional programming language that uses a form of remote procedure call to realize active network packet programming. Developed at the University of Pennsylvania (1997-2001) it is part of the SwitchWare project, a research project on active networks.

pass by reference *Pass by reference* means that when an argument is passed to a function `f`, `f` receives the memory address of the original value, not a copy of the value. Therefore, if `f` modifies the parameter, the original value in the calling code is changed.

pass by value *Pass by value* means that when an argument is passed to a function `f`, `f` receives a copy of the original value. Therefore, if `f` modifies the parameter, only the copy is changed and the original value remains unchanged.

polymorphism *Polymorphism* in Java refers to the ability to define *interfaces* (or types) and classes separately. An interface is a definition of the signatures of the methods of a class, which is independent from any implementation. An interface can therefore be implemented by several classes. It is possible to declare a variable whose type is an interface and which can thus reference objects from different classes that implement the same interface.

primary constructor In Scala the *primary constructor* does not have an explicit definition; it is defined implicitly by the class parameters

and body. Auxiliary constructors (also named *secondary constructors*) can be added to the class; they are called `this` and must reference a previously defined constructor. It possibly invokes a superclass constructor (when `supers` is not empty) and initializes fields for any value parameters not passed to the superclass constructor.

reference type In Scala a *reference type* is a subclass of `scala.AnyRef`. Instances of reference types always reside on the JVM's heap at runtime.

referential transparency An expression is said to be *referentially transparent* if it can be replaced with its value without changing the program (in other words, yielding a program that has the same effects and output on the same input). For example, assignments are not transparent while arithmetic operations are referentially transparent as their result depends only on their input regardless of the context in which they are used.

scope The *scope* of an identifier is the range of program instructions over which the identifier is known. Name resolution rules determine which value is bound to each identifier in any point of a given program.

security policy In Java the *security policy* is specified through a list of access control rules which define the security behavior of the run-time environment. Concretely, the security policy associates permissions to code locations; those permissions define which system resources can be accessed and which signatures must be checked.

serializable type A Java type is serializable if it satisfies one of the following conditions:

- The type is primitive, such as `char`, `byte`, etc..
- The type is a primitive wrapper such as `Character`, `Byte`, etc..
- The type is part of the Java standard library and is either a throwable class, an AWT component, or one of classes `Date`, `String`, `Vector`, etc..
- The type is an enumeration (since Java 5).
- The type is an array of serializable types.
- The type is a serializable user-defined class.

statically typed A programming language is *statically typed* (opposite: *dynamically typed*) if types are fixed at compile-time (i.e. type analysis happens at compile-time). The absence of compile-time errors then guarantees not only the absence of (run-time) type errors but also that run-time type checks are not needed.

Typed languages like Java do not fulfill this ideal because they contain type casts and other constructs that (in general) cannot be fully checked statically. In particular, as Java uses a covariant rule for subtyping arrays, array stores require run-time type checks in addition to the normal array bounds checks. This means that the Java bytecode verifier is not the only part of the JVM that must be correct to ensure type safety; the interpreter must also perform run-time checks.

strong mobility See *mobile computation*.

strongly typed A programming language is *strongly typed* (opposite: *weakly typed*) if types are always enforced. Java (arrays excepted), Python, Scala and Smalltalk are strongly typed.

stub A *stub* in Java RMI is a class that does the work of formatting and transmitting method arguments to the RMI server and returning results to the RMI client. Stub classes are either generated with a RMI compiler or are built dynamically at run-time using dynamic proxies. Dynamic class loading allows client virtual machines to find stubs at run-time, without any special coding in the client.

trait In Scala a *trait* behaves similarly to a class, except that its constructor can't take parameters, a class can derive from only one superclass, but can be mixed in with several traits and, finally, the meaning of `super` in a trait is not defined until the programmer mix it into a class.

transparent migration See *mobile computation*.

tuple space A *tuple space* is a repository of tuples that can be accessed concurrently in parallel/distributed system. Producers post their data as tuples in the space, and consumers retrieve data from the space that match some pattern.

The language Linda introduced tuple spaces to support the concept of global object coordination; meanwhile tuple spaces have been implemented in many programming languages (e.g. JavaSpaces from Sun is a distributed system based on Jini services and tuple spaces).

type inference *Type inference* (or *type reconstruction*) is usually implemented as a unification algorithm which attempts to reconstruct the (omitted) static type of a declaration before being type checked by a compiler.

Scala uses a local, flow-based type inference scheme [90, §16.9]; while Scala's scheme has some limitations compared with the more global Hindley-Milner (HM) scheme (e.g. Haskell and ML), it deals much better with object-oriented subtyping than the HM scheme does.

type safety *Type safety* is the property that every operation the program performs is executed on values of the appropriate type. It is the responsibility of the compiler to enforce type safety. A compiler is equipped with a type system to perform a static analysis of the program and to approximate its run-time behavior before it is executed.

In a more rigorous setting, *type safety* is proved about a formal language — which is unambiguous and complete — by proving progress and preservation. See *type system*.

type system A *type system* guarantees that the reduction relation does not get stuck on ground terms of closed types (progress property) and preserves types (subject reduction property).

weak mobility See *mobile code*.

Index

abstraction
  abstraction safety, 95
  C/S abstraction, 40
  chunk abstraction, 71
  control abstraction, 95, 96
  data abstraction, 95
  lambda abstraction, xix, 11, 72, 96, 102, 160
  programming abstraction, 2, 11
  type abstraction, 8
access control, 155
ActiveX, 47, 49
Ada, 14, 36, 99
address
  address space, 40, 48, 98
  IP address, 87
ADT, 95
aglet, 46, 172
Ahern, Alexander, 50
Algol-60, 26
allocation
  heap-based allocation, 97
  stack-based allocation, 97
Ancona, Davide, 64
annotation
  class annotation, 172
AnyVal, 147
API, 31, 144
applet, 46, 51
  applet context, 51
architecture
  class loader architecture, 57
  client-server architecture, 31
AST, 145

Beta, 10
Bierman, Gavin, 61
binding
  dynamic binding, 3, 4, 70
  generalized data binding, 96
  top-level binding, 71
BinProlog, 48
Birell, Andrew, 30
Budd, Timothy, 13
bytecode, 50, 54, 60, 71, 147
  bytecode verifier, 155

C++, 14, 36
calculus
  λ_v-calculus, 70
  λ_{marsh} calculus, 62
  lambda calculus, 3, 13, 61, 103, 112
  module calculus, 64
  object calculus, 13
  predicate calculus, 13
  process calculus, 8
call
  remote call, 40
  tail-recursive call, 145
call-by-value, 32, 71
CAML, 69
Cardelli, Luca, 61, 87
channel

typed channel, 5
chunk, 71
Church, Alonzo, 171
class
abstract class, 55
anonymous class, 28, 148, 150
class file, 54
class loader, 54
class table, 105
class versioning, 57
inner class, 28, 122
lifted class, 148
nested class, 147
synthetic class, 7, 105
top-level class, 148
clause
`throws` clause, 41
import clause, 96, 99, 100
Clojure, 168
closure, 14, 15, 115
closure conversion, 112, 115
detached closure, 114, 115, 151
lexical closure, 11, 97, 160
ML closure, 69
Scala closure, 4
CLU, 33, 95, 168
CML, 96
COBOL, 36
code
code deployment, 60
code fetching, 46, 48
code inlining, 145
code migration, 46
code portability, 50
code shipping, 46
code signing, 155
code upload, 46
mobile code, 2, 7, 8, 47, 48
signed code, 60
codebase, 56, 57, 59, 155
compiler
compiler front-end, 11, 97, 114, 160
compiler implementation, 58
ML compiler, 112
Scala compiler, 10, 112, 144
computation
higher-order distributed computation, 3
console
server console, 88
context, 70
applet context, 51
surrounding context, 112, 122
CORBA, 31, 36
Courtney, Antony, 68
Curry, H.B., 168

data
data encryption, 155
data integrity, 155
DCOM, 39
DoD, 48
DSL, 72, 96
Duggan, Dominic, 69

Emerald, 30, 31
encryption, 58
environment
calling environment, 15
computational environment, 7, 154
distributed environment, 98
execution environment, xix, 11
multi-user environment, 48
EPFL, 8, 112, 144
Erlang, 14, 170
evaluation
deferred evaluation, 7, 113
remote evaluation, 33, 36, 46, 71, 72, 88
exception

NoSuchObjectException, 41
NotBoundException, 41
RemoteException, 41
StubNotFoundException, 41
UnmarshalException, 41
exception handling, 40
externalization, 156

Facile, 70
FJ, 104
Flash, 49
function
auxiliary function, 105
accessor function, 148
anonymous function, 97
curried function, 14, 145
first-class function, 15, 43, 96
function composition, 15
generic function, 105
higher-order function, 8, 11, 14, 62, 68, 160
identity function, 152
memoized function, 14, 172
unary function, 148

garbage collection, 42, 49, 154
distributed garbage collection, 39
Gifford, David, 33
GostView, 35
Greanier, Todd, 156
Groovy, 14

Haskell, 10, 14, 17, 170
HOF, 168
HTML, 50, 52, 167

IDL, 31, 37, 39
IDL skeleton, 31
IDL stub, 31
initialization
static initialization, 54
interface
Externalizable interface, 156
Remote interface, 41, 59, 151
Serializable interface, 41
marker interface, 156, 171
proxy interface, 115
remote interface, 41
interoperability, 13
interpreter
Obliq interpreter, 68
PLAN interpreter, 71
SQL interpreter, 35

JAR, 53
Java, 31
applet, 3, 46, 50, 51, 154
array, 41
bootstrap class loader, 56
CLASSPATH, 57
DGC, 42, 156
DGC lease value, 42
Java Web Start, 55
JOM, 40
plugin, 50
reflection, 155
sandbox, 47
VM, 42, 54
JavaScript, 14, 46, 47, 49
JavaSpaces, 175
Jini, 175
judgment, 106

Knabe, Frederick, 70

language
domain-specific language, 72
functional language, 43
language expressiveness, 96
library
Scala Actors library, 85
Java standard library, 55
library support, 72

RPC library, 30
Scala Actors library, 8, 10, 96
Scala collection library, 9, 10
Scala parsing library, 96
Scala standard library, 12, 26, 81, 114, 152, 160
Scala Swing library, 96
Linda, 31, 171, 175
Liskov, Barbara, 168
Lisp, 26, 95
location
source location, 71
target location, 7
LotusScript, 49

marshalling, 3
parameter marshalling, 39
Mathlink, 35
MCL, 2, 8, 43
mechanism
access control mechanism, 60
class loading mechanism, 57
class versioning mechanism, 57
data binding mechanism, 99
externalization mechanism, 156
name resolution mechanism, 99
rebinding mechanism, 43
message
call message, 32
reply message, 32
method
getter method, 148
abstract method, 104, 105
getter method, 98, 113
remote method, 67
setter method, 98
synthetic method, 122
Microsoft, 47
migration
state migration, 46
transparent migration, 70, 154
ML, xix, 10, 14, 61, 69–71, 112
mobile agent, 46
MobileML, 70
mobility
code mobility, 11, 47
computation mobility, 47
strong mobility, 47, 154
weak mobility, 3, 47, 154
model
abstraction model, 3
distributed object model, 40
local object model, 40
programming model, 160
semantic model, 70
uniform object model, 10
Modula-3, 37, 42, 68
module
dynamic module, 99
mixin module, 64
reconfigurable module, 72
Moore, Jonathan, 71
MSIL, 10

name
synthetic name, 116
namespace, 54, 59, 71, 99
Nelson, Bruce, 30, 31
network
active network, 71
network reference, 102
NFS, 30

object
CORBA object, 31
distributed object, 11, 40, 160
exported object, 87
marker object, 150, 152
network object, 42, 66
object migration, 66

- object reference, 7, 37
- object serialization, 58, 155
- Obliq object, 46, 72
- remote object, 3, 40
- serializable object, 3
- servant object, 204
- server object, 40
- top-level object, 79, 148

Obliq, 4, 68, 69, 72, 102
OCaml, 13, 17
Odersky, Martin, 8
OLE, 47
ORB, 31, 36

package, 57, 68, 100, 148
- empty package, 148

paradigm
- COD paradigm, 46
- imperative paradigm, 13
- mobile agent paradigm, 46
- object-oriented paradigm, 13
- REV paradigm, 46, 102
- RPC paradigm, 102

Parallel, 35
parameter
- formal parameter, 98
- implicit parameter, 9, 74

Pascal, 99, 170
PDF, 49
Perl, 17
permission
- access permission, 60

Phantom, 68
phase
- `Detach` phase, 150
- `Erasure` phase, 147
- `ExplicitOuter` phase, 147
- `LambdaLift` phase, 148
- `UnCurry` phase, 145
- transformation phase, 144

PLAN, 71, 173
polymorphism
- F-bounded polymorphism, 8
- parametric polymorphism, 8
- type constructor polymorphism, 8

PostScript, 35, 49
preprocessor, 47
procedure
- relocated procedure, 33

programming
- socket programming, 40
- higher-order programming, 14
- imperative programming, 14
- modular programming, 95
- multi-paradigm programming, 35
- socket programming, 31

property
- `Codebase` property, 60
- RMI property, 59
- system property, 154, 157

proxy
- dynamic proxy, 175
- proxy structure, 70

Python, 14, 17

Qian, Zhenyu, 58

reference
- network reference, 41, 102
- reference cell, 148

referential transparency, 14
registry
- local registry, 37
- remote registry, 37
- RMI registry, 42

REPL, 51, 68, 168
REV, 33, 46
RMI, 3, 37, 40, 50, 59, 160
- model, 31
- RMI property, 59

- RMI stub, 59
- RPC, 30, 31, 33, 37, 67
  - RPCL, 32
  - XDR, 32
- RTTI, 69
- Ruby, 14, 17

- Scala, 2, 5, 14
  - back-end, 145
  - closure, 4
  - function literal, 112, 150
  - partial erasure, 147
  - scalac, 112
  - separate compilation, 145
- Scheme, 14, 26
- scheme
  - mark-and-sweep, 42
  - naming scheme, 30
- scope
  - lexical scope, 68
  - top-level scope, 151
  - distributed scope, xix, 4
  - lexical scope, 96
  - local scope, 100
  - surrounding scope, 28, 102, 114
  - top-level scope, 100, 115
- scoping
  - distributed scoping, 67
  - lexical scoping, xix, 3, 26, 67
  - static scoping, 71
- security, 43
  - digital signature, 47
  - security manager, 50, 60
  - security policy, 60, 80, 154, 155
- semantics
  - at-most-once semantics, 167
- serialization, 41, 58, 155
  - object serialization, 30
  - serialization stream, 58
- server
  - name server, 87, 152
- service, 40
  - denial of service, 48
- Smalltalk, 10, 36
- socket, 37, 60, 81, 155
  - `SSLServerSocket`, 157
  - `ServerSocket`, 157
  - socket programming, 31
- SQL, 35
  - SQL query, 35
- SSL, 157
- stack, 154
  - call stack, 154
  - stack frame, 26
- Stamos, James, 33
- static
  - static initialization, 54
  - static member, 99
- stub, 60
  - client stub, 30
  - server stub, 30
- subtyping
  - nominal subtyping, 105
  - structural subtyping, 105
- Sun, 30, 47, 57

- Tarau, Paul, 48
- TCP, 32, 156
  - TCP/IP, 42
- Telescript, 46, 48, 172
- term
  - lambda term, 97
- TLS, 157
- tree
  - abstract syntax tree, 145
  - tree transformation, 145
  - tree traversal, 151
- tuple space, 70, 175
- type
  - compound type, 9
  - dependent type, 9
  - erased type, 147

- existential type, 9
- higher-kinded type, 8
- local type inference, 8
- primitive type, 152
- run-time type, xix
- self type, 9
- singleton type, 9
- structural type, 9
- type checker, 154
- type constructor, 8
- type inference, xix, 15, 17
- type manifest, 74
- type member, 9
- type parametrization, 8
- type refinement, 9
- type safety, 2, 43, 50, 54, 58
- type system, 8, 49, 104
- type transformation, 145
- virtual type, 8
- wildcard type, 9

typing
- static typing, 72
- typing rule, 63, 106

UDP, 32
UnicastRemoteObject, 151
URL, 30, 59, 60

value
- first-class value, 71
- function value, 14, 15, 148, 151

variable
- bound variable, 4, 112
- environment variable, 57
- free variable, 7, 14, 26, 34, 70, 112, 119, 148
- local variable, 6, 112
- mutable variable, 112, 148
- static variable, 55
- type variable, 147
- virtual variable, 65

VBA, 49
VBScript, 49

WSH, 49

X-Klaim, 48

Yoshida, Nobuko, 50

Bibliography

[1] Adobe Systems, Inc. Abode JavaScript Object Specification. www.adobe.com, 2005. → §3.

[2] G.T. Almes and al. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, 11(1):43–59, January 1985. → §3.

[3] Ph. Altherr. *A Typed Intermediate Language and Algorithms for Compiling Scala by Successive Rewritings*. PhD thesis, EPFL, Thesis No 3509, March 2006. → §7.2.1, §7.2.1.

[4] D. Ancona, S. Fagorzi and E. Zucca. Mixin Modules for Dynamic Rebinding. In *Proceedings of TGC 2005, Edinburgh, UK*, pages 279–298, Berlin, 2005. Springer Verlag. → §4.1.2.

[5] A. Ahern and N. Yoshida. Formal Analysis of a Distributed Object-Oriented Language and Runtime. Technical report, Imperial College, London, 2005. → §3.2.

[6] L. Bettini. Design and Implementation of a Programming Language for Mobile Code. Master's thesis, University of Firenze, April 1998. → §3.1.2, §8.3.

[7] L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming and Their Implementations*. PhD thesis, University of Siena, February 2003. → §3.1.2, §8.3.

[8] L. Bettini and R. De Nicola. Translating Strong Mobility into Weak Mobility. In *Mobile Agents, LNCS 2240*, pages 182–197. Springer, 2001. → §3.1.2.

[9] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, K. Wansbrough. Dynamic Rebinding for Marshalling and Update, with Destruct-time

λ. In *Proceedings of ICFP'03*, pages 99–110. ACM Press, August 2003. → §4, §4.1.1.

[10] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, K. Wansbrough. Dynamic Rebinding for Marshalling and Update, with Destruct-time λ. Technical report, University of Cambridge, ISSN 1476-2986, February 2004. → §1.1, §4.1.1.

[11] C. Binildas. Internals of Java Class Loading. *O'Reilly ONJava.com*, January 2005. → §2.2.1.

[12] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984. → §2.2.1, §2.2.5.

[13] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. Technical Report 115, DEC Systems Research Center, December 1995. → §2.2.5, §2.3.3, §4.2.1.

[14] A. Black, N. Hutchinson. E. Jul and H. Levy. Object Structure in the Emerald System. In *Proceedings of OOSPLA'86*, pages 78–86, New York, NY, September 1986. ACM.

[15] A. Black, N. Hutchinson. E. Jul and H. Levy. The Development of the Emerald Programming Language. In *Proceedings of the 3rd ACM SIGPLAN Conference on Historic Programming Languages*, pages 11–51, New York, 2007. ACM. → §2.2.

[16] A. Black, N. Hutchinson, E. Jul, H. Levy, L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987. → §2.2, §8.3.

[17] S. Bouchenak. *Mobilité et persistance des applications dans l'environnement Java*. PhD thesis, Institut National Polytechnique de Grenoble, October 2001. → §3.1.2.

[18] S. Bouchenak and al. Experiences implementing efficient Java thread serialization, mobility and persistance. *Software Practice and Experience*, (34):355–393, January 2004. → §3.1.2.

[19] G. Bracha, M. Odersky, D. Stoutamire and Ph. Wadler. Making the future safe for the past: Adding Genericity to the Java™ Programming Language. In *Proceedings of OOPSLA'98*, volume 35, pages 183–200, Vancouver, BC, October 1998. → §7.2.1.

[20] T. M. Breuel. Lexical Closures in C++. In *Proceedings of the USENIX 1988*, pages 294–304, October 1988. → §2.1.

[21] L. Brown. Mobile Code Security. *AUUG96, Melbourne*, September 1996. → §3.2.

[22] T. A. Budd. *Multiparadigm Programming in Leda*. Addison Wesley, Massachusetts, 1995. ISBN-10 0-201-82080-3 → §1.3.1, §2.

[23] T. A. Budd. Multiparadigm Extensions to Java. Oregon State University, November 2000. → §2.

[24] S. Campadello. *Middleware Infrastructure for Distributed Mobile Applications*. PhD thesis, University of Helsinki, March 2003. → §7.4.

[25] L Cardelli. Obliq - A lightweight language for network objects. Digital System Research Center, Palo Alto, CA, November 1993. → §4.2.1.

[26] L. Cardelli. Obliq, a Language with distributed scope. Technical Report RR-122, Digital System Research Center, June 1994. → §4.2.1.

[27] L. Cardelli. A Language with Distributed Scope. In *Conference Record of POPL'95: 2nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA*, pages 286–297, New York, NY, 1995. → §1.1, §4.2.1.

[28] L. Cardelli. Distributed Mobile Computation in Obliq. Digital Systems Research Center, April 1997. → §1.1, §4, §4.2.1, §5.4.

[29] A. Carzaniga, G.P. Picco and G. Vigna. Is Code Still Moving Around? Looking Back at a Decade of Code Mobility. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 9–20, Minneapolis, Minnesota, USA, May 2007. → §3.

[30] H. Cejtin, S. Jagannathan and R. Kelsey. Higher-Order Distributed Objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, September 1995. → §1.1.

[31] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940. → §4.1.1.

[32] Xerox Corporation. ILU Reference Manual, Version 2.0b1, 1999. → §4.2.2.

[33] A. Courtney. Phantom: An Interpreted Language for Distributed Programming. In *USENIX Conference on Object-Oriented Technologies*, pages 83–102, June 1995. → §1.1, §4.2.1.

[34] V. Cremet. *Foundations for Scala: Semantics and Proof of Virtual Types*. PhD thesis, EPFL, Thesis N*o* 3556, May 2006. → §1.3.1.

[35] G. Cugola, C. Ghezzi, G.P. Picco and G. Vigna. Analyzing Mobile Code Languages. *LNCS Vol. 1222*, April 1997. → §3.

[36] P. Dasgupta and al. The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems*, 3(1):11–46, 1989. → §2.2.

[37] DEC. Programming with ONC RPC, March 1996. → §2.2.

[38] DoD. Policy Guidance for use of Mobile Code Technologies in DoD Information Systems. U.S. Department of Defense, November 2002. → §3.2, §8.3.

[39] D. Dreyer, K. Crary and R. Harper. A Type System for Higher-ordered Modules. In *Proceedings of POPL'03: The 30rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New Orleans, USA, January 2003. → §4.1.2.

[40] D. Duggan. A Type-Based Implementation of a Language with Distributed Scope. In 2nd *Intl. Workshop, MOS'96, Linz, Austria, July 8-9, 1996*, volume 1222, pages 277–292, Berlin, Germany, 1997. Springer. → §4.2.2.

[41] S. Fagorzi. *Module Calculi for Dynamic Reconfiguration*. PhD thesis, Università di Genova, 2005. → §4.1.2.

[42] P. Ferreira and M. Shapiro. Larchant - Persistence by Reachability in Distributed Shared Memory through Garbage Collection. In *Proceedings of the 16th Intl. Conference on Distributed Computing Systems*, Hong Kong, 1996. IEEE CS. → §2.3.3.

[43] A. Fuggetta, G.P. Picco and G. Vigna. Unterstanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998. → §3.

[44] S. Fünfrocken. Transparent Migration of Java-based Mobile Agents: Capturing and Reestablishing the State of Java Programs. In *Proceedings of the 2nd Intl. Workshop on Mobile Agents (MA'98)*, volume

1477, pages 26–37, Stuttgart, Germany, September 1998. Springer Verlag. → §3.1.2.

[45] N. Gafter and al. Closures for the Java Programming Language, Version 0.5. www.javac.info, March 2008. → §2.1.1.

[46] N. Glew. Object Closure Conversion. Technical Report TR99-1763, Cornell University, Ithaca, New-York, August 1999. → §7.1.

[47] P. Gomes. On remote procedure call. In *CASCON'92: Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative research*, pages 215–267. IBM Press, 1992. → §2.2.2.

[48] J. Gosling and al. *The Java Language Specification*. Prentice Hall, 3rd edition, 2005. ISBN 0-321-24678-0, → §6.3.2, §7.2.1, §8.3.

[49] R.S. Gray. Agent Tcl: a Flexible and Secure Mobile-Agent System. Technical Report PCS-TR98-327, Dartmouth College, Hanover, New Hampshire, January 1998. → §3.1.2.

[50] T. Greanier. Flatten your objects: Discover the secrets of the Java Serialization API. *JavaWorld*, July 2000. → §7.4.

[51] D. Griscom. Code Signing for Java Applets, April 2003. → §7.3.

[52] C.A. Gunter and al. PLAN — A Packet Language for Active Networks. University of Pennsylvania, 1997. → §4.2.4.

[53] D. Hagimont, P. Y. Chevalier, J. Mossiere and X. Rousset de Pina. Object Migration in the Guide System. In *ECOOP'95 Workshop on Mobility and Replication*, Aarhus, Denmark, 1995. → §2.2.

[54] G. Hamilton, M.L. Powell and J.G. Mitchell. Subcontract: a Flexible Base for Distributed Programming. In *SOSP'93: Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79, New York, NY, 1993. ACM. → §2.2.5.

[55] M. Hashimoto and A. Ohori. A Typed Context Calculus. *Theoretical Computer Science*, 266(1-2):249–272, 2001. → §4.2.3.

[56] M. Hashimoto and A. Yonezawa. MobileML: A Programming Language for Mobile Computation. In *Coordination Models and Languages, LNCS 1906*, pages 198–215, 2000. → §1.1, §4.2.3.

[57] B. Haumacher and M. Philippsen. More efficient object serialization. *In Parallel and Distributed Processing, LNCS 1586*, 1586:718–732, April 1999. → §7.4.

[58] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982. → §2.2.3.

[59] M. Hicks, P. Kakkar and al. PLAN, A Packet Language for Active Networks. In *Proceedings of ICFP'98*, pages 86–93. ACM Press, 1998. → §1.1, §4.2.4.

[60] P.N. Hilfinger. *Abstraction Mechanisms and Language Design*. MIT Press, Cambridge, MA, USA, 1983. → §6.

[61] F. A. Hosch. Generic Instantiations as Closures. *Ada Letters*, X(1):122–130, 1990. → §2.1.

[62] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Comput. Surv.*, 21(3):359–411, 1989. → §2.1.

[63] A. Igarashi, B.C. Pierce and Ph. Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, September 2001. → §6.3.2.

[64] S. Jagannathan. Dynamic Modules in Higher-Order Languages. In *Proceedings of the ICCL'94*, 1994. → §6.2.

[65] T. Jensen, D. Le Métayer and T. Thorn. Security and Dynamic Class Loading in Java: A Formalisation. In *IEEE Intl. Conference on Computer Languages*, pages 4–15, Chicago, Illinois, 1998. → §3.3.2.

[66] E. Jul, H. Levy, N. Hutchinson, A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988. → §2.2.1, §2.3.3, §8.3.

[67] Z. D. Kirli. *Mobile Computation with Functions*. PhD thesis, University of Edinburgh, 2001. → §2.4, §6.1.

[68] F. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, Pittsburgh, December 1995. → §4.2.3.

[69] F. Knabe. An Overview of Mobile Agent Programming. *5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Language, LNCS 1192*, pages 100–115, June 1996. → §4.2.3.

[70] D. Lange, M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998. ISBN-10 0-201-32582-9 → §3.

[71] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of OOPSLA'98*, volume 35, pages 36–44, Vancouver, BC, October 1998. → §3.3.2.

[72] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, April 1999. → §1.1, §7.3.

[73] Z. Lisiecki. Linux Remote Procedure Call Programmierung. `www.linuxhaven.de`, January 2000. → §2.2.

[74] J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. *ACM SIGPLAN Notices*, 34(8):173–182, August 1999. → §7.4.

[75] Q. H. Mahmoud. Security Policy: A Design Pattern for Mobile Java Code. In *Proceedings of the PLoP'02 Conference*, August 2000. → §7.3.

[76] A. Mathew, M. Roulo. Accelerate your RMI Programming. *JavaWorld*, September 2001. → §7.4.

[77] S. McLean, J. Naftel, K. Williams. *Microsoft .NET Remoting*. Microsoft Press, 1st edition, September 2002. ISBN-13 978-0-735617-78-0, → §2.2.6.

[78] M. Merro, J. Kleist and U. Nestmann. Mobile Objects as Mobile Processes. *Information and Computation*, 177(2):195–241, 2002. → §4.2.1.

[79] Y. Minamide, G. Morrisett and R. Harper. Typed Closure Conversion. Technical Report CMU-CS-95-171, July 1995. → §7.1.

[80] Y. Minamide, J.G. Morrisett and R. Harper. Typed Closure Conversion. In *Symposium on Principles of Programming Languages*, pages 271–283, 1996. → §7.1.

[81] J.T. Moore, M. Hicks, and S. Nettles. Chunks in PLAN: Language Support for Programs as Packet. In *Proceedings of the 37th Intl. Conference on Communication, Control and Computing*, 1999. → §4.2.4.

[82] A. Moors. *Type Constructor Polymorphism for Scala: Theory and Practice*. PhD thesis, KU Leuven, Leuven, May 2009, ISBN 978-94-6018-065-1. → §1.3.1.

[83] A. Moors, F. Piessens and M. Odersky. Towards Equal Rights for Higher-kinded Types. In *MPOOL 2007: 6th Intl. Workshop on Multiparadigm Programming with Object-Oriented Languages*, July 2007. → §1.3.1.

[84] B.J. Nelson. Remote Procedure Call. Technical Report CSL-81-9, Xerox Palo Alto Research Center, 1981. → §2.2.1, §2.2.2.

[85] Ch. Nester, M. Philippsen and B. Haumacher. A More Efficient RMI for Java. In *Java Grande*, pages 152–159, 1999. → §7.4.

[86] U. Nestmann, H. Huttel, J. Kleist and M. Merro. Aliasing Models for Mobile Objects. *Information and Computation*, 175(1):3–33, 2002. → §4.2.1.

[87] P. Obermeyer and J. Hawkins. Microsoft .NET Remoting: A Technical Overview. Microsoft MSDN, 2002. → §2.2.6.

[88] M. Odersky. Types for Objects and Modules. In *Proceedings of Dagstuhl Seminar on Domain-Specific Program Generation*, March 2003. → §1.1.

[89] M. Odersky and al. The Scala Language Specification. Technical report, Programming Methods Laboratory, EPFL, 2004. www.scala-lang.org → §1.1, §1.3.1, §6, §7.2.1.

[90] M. Odersky, A. Spoon and B. Venners. *Programming in Scala*. Artima Inc., 1st edition, November 2008. ISBN-13 978-0-9815316-0-1, → §1.3.1, §2.1.

[91] M. Odersky, Ph. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of POPL'97: The 24rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, 1997. → §2.

[92] D. von Oheimb. *Analyzing Java in Isabelle/HOL (§4: Type Safety)*. PhD thesis, University of Munich, November 2000. → §7.3.

[93] OMG. Common Object Request Broker Architecture, Version 3.1. www.omg.org, January 2008. → §2.2.4.

[94] M. Philippsen, B. Haumacher and Ch. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000. → §7.4.

[95] B. Pierce and D. Turner. Local Type Inference. In *ACM SIGPLAN–SIGACT Symposium on PoPL*, volume 22(1), pages 1–44, San Diego, California, January 2000. → §1.3.1.

[96] S. Prasad, A. Giacalone and P. Mishra. Operational and Algebraic Semantics for Facile: A Symmetric Integration of Concurrent and Functional Programming. In M. S. Paterson, editor, *Automata, Languages and Programming: Proc. of the 17th Intl. Colloquium*, pages 765–780. Springer, New York, 1990. → §1.1.

[97] Z. Qian, A. Goldberg and A. Coglio. A formal specification of Java^{TM} class loading. *ACM SIGPLAN Notices*, 35(10):325–336, 2000. → §3.3.2.

[98] H.M. Qusay. Advanced Socket Programming. *Sun Developer Network*, December 2001. → §7.4.

[99] I. Rammer, M. Szpuszta. *Advanced .NET Remoting*. APress, 2nd edition, February 2005. ISBN 1-590-59417-7, → §2.2.6.

[100] R. Ranganathan, A. Acharya, S. Sharma and J. Saltz. Network-aware Mobile Programs. In *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 91–104, Anaheim, CA, January 1997. → §1.

[101] P. Van Roy and S. Haridi. *Concepts, Techniques and Models of Computer Programming*. MIT Press, London, 2004. → §1.3.1.

[102] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *ASA/MA*, pages 16–28, 2000. → §7.3.

[103] M. Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, EPFL, Thesis N*o* 3302, September 2005. → §7.2.1, §7.4.

[104] A. Schäfer. Inside Class Loaders. *O'Reilly ONJava.com*, November 2003. → §2.2.1, §3.3.2.

[105] P. Seibel. *Practical Common Lisp*. APress, 1st edition, April 2005. ISBN 1-590-59239-5, → §2.1.2.

[106] T. Sekiguchi. *A Study on Mobile Language Systems*. PhD thesis, University of Tokyo, April 1999. → §3.1.2.

[107] T. Sekiguchi and A. Yonezawa. A Calculus for Code Mobility. In *Proceedings of FMOODS'97*, pages 21–36. Chapman & Hall, 1997. → §4.2.3.

[108] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In *Proceedings of the 3rd Intl. Conference on Coordination Models and Languages*, 1999. → §7.3.

[109] P. Sewell and al. Acute: High-level Programming Language Design for Distributed Computation. In *Proceedings of ICFP'05*, 2005. → §7.4.

[110] K. Shudo. Thread Migration on Java Environment. Master's thesis, University of Waseda, 1997. → §7.3.

[111] D. Sosnoski. Java programming dynamics, Part 1: Java classes and class loading. *IBM DeveloperWorks*, April 2005. → §2.2.1.

[112] A. Spoon. Writing Scala Compiler Plugins. www.lexspoon.org, June 2008. → §8.3.

[113] J.W. Stamos and D.K. Gifford. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering*, 16(7), July 1990. → §2.2.3.

[114] J.W. Stamos and D.K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–564, 1990. → §3.

[115] Sun Microsystems, Inc. ONC+ Developer's Guide. docs.sun.com, 1994. → §2.2.2.

[116] Sun Microsystems, Inc. Support Readiness Document, Java™ 2 SE, Version 1.4, Java RMI, March 2002. → §7.3.

[117] Sun Microsystems, Inc. Java™ Product Versioning Specification. java.sun.com, 2003. → §3.3.2.

[118] Sun Microsystems, Inc. Tuning Garbage Collection with the 5.0 Java™ Virtual Machine. java.sun.com, 2003. → §2.3.3.

[119] Sun Microsystems, Inc. Java Object Serialization Specification, Revision 1.5.0. java.sun.com, 2004. → §1.1, §3.3.3.

[120] Sun Microsystems, Inc. Java Remote Method Invocation. java.sun.com, 2004. → §1.1, §2.2.5.

[121] N. Suri and al. Strong Mobility and Fined-grained Resource Control in Nomads. In *Proceedings of the* 4th *Intl. Symposing on Mobile Agents*, pages 2–15, London, UK, September 2000. Springer Verlag. → §3.1.2.

[122] P. Tarau, V. Dahl and K. De Bosschere. A Logic Programming Infrastructure for Remote Execution, Mobile Code and Agents. In *Proceedings of ICFP'97*, pages 106–111, Leuven, Belgium, June 1997. ACM Press. → §3.1.2.

[123] R. Veldema and M. Philippsen. Compiler Optimized Remote Method Invocation. University of Erlangen-Nuremberg, 2003. → §7.4.

[124] X. Wang, J. Hallstrom and G. Baumgartner. Reliability Through Strong Mobility. *Proceedings of ECOOP 2001,* 7th *Workshop on Mobile Object Systems*, June 2001. → §3.1.2.

[125] J.E. White. High-level Framework for Network-based Resource Sharing (RFC 707). Stanford Research Institute, Menlo Park, CA, 1975. → §2.2.2.

[126] J.E. White. Telescript Technology: the Foundation for the Electronic Marketplace. General Magic, Inc., Mountain View, CA, 1994. → §3, §3.1.2.

[127] J. Willcock and al. Lambda expressions and closures in C++. Technical Report No N1968-06-0038, ATT Research, February 2006. → §2.1.

[128] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. In *Proceedings of the USENIX 1996*, Toronto, CA, June 1996. → §2.3.

[129] H. Xu. Java Security Model and Bytecode Verification. Technical Report EECS-12-99, University of Illinois, Chicago, December 1999. → §7.3.

[130] K. Yeung and P. Kelly. Optimising Java RMI Programs by Communication Restructuring. Middleware 2003, 2003. → §7.4.

A propos de l'Auteur

Né en 1964, Stéphane Micheloud grandit — et vit aujourd'hui encore — à Sion, le chef-lieu du Valais, une région magnifique située au coeur des Alpes suisses.

De 1979 à 1984 il étudie en section scientifique au Lycée-Collège des Creusets (LCC) à Sion où il découvre la rigueur des preuves mathématiques dans la classe du Prof. Dr. Jean-Claude Pont et fait ses premières armes avec le langage Pascal sous la conduite du Prof. Paul Epiney. Ces deux expériences influencent fortement son choix d'entreprendre une carrière professionnelle dans l'informatique.

De 1984 à 1990 il étudie l'informatique à l'ETH Zürich où il apprend les techniques de la construction de compilateurs dans la classe du Prof. Dr. Niklaus Wirth et approfondit ses connaissances de l'architecture des ordinateurs sous la conduite du Prof. Dr. Marco Annaratone. C'est à cette époque — pour la petite histoire — qu'il fait la rencontre des assistants Martin Odersky et Clemens Szyperski qui poursuivent aujourd'hui des carrières très réussies. En 1989 Stéphane Micheloud effectue son stage en entreprise chez Contraves AG dans le groupe du Dr. Urs Ammann, chef de projet de CZ-Ada, un système Ada temps-réel propriétaire. Il reçoit son diplôme d'ingénieur en informatique en 1990 après avoir achevé avec succès son travail de diplôme intitulé "*Oberon Compiler Back-End für Inmos Transputer*" sous la supervision du Prof. N. Wirth.

De 1991 à 1997 Stéphane Micheloud travaille comme ingénieur de développement dans la société Glance AG à Steinmaur (absorbée depuis 2007 par le groupe adesso Schweiz AG) et participe au développement de plusieurs compilateurs, en particulier M2CC (Modula-2) et POCC (Portal), des outils de traduction de code en ANSI C. Il travaille également comme collaborateur externe auprès de la société Landis&Gyr AG à Zug et participe au développement d'un nouveau système de commande du bâtiment au sein du projet ATLAS. Aux côtés des Dr. Hans Bärfuss et Dr. Hans-Ruedi Aschmann il s'initie à la gestion de projets informatiques et de la relation client. Il découvre alors le langage Java qu'il adopte rapide-

ment après plusieurs années d'expériences en programmation Modula-2 et C/C++.

De 1996 à 1999 Stéphane Micheloud entame à temps partiel des études postgrades en informatique et télécommunication auprès de NDIT (*eduswiss* depuis 1999, aujourd'hui affilié au Swiss Virtual Campus). Il suit différents cours avancés dirigés par des spécialistes tels que le Prof. Dr. Antoine Delay, le Prof. Dr. Jacques Savoy, Dr. Erich Gamma et Dr. Igor Metz.

De 1997 à 1999 il travaille comme ingénieur senior et plus tard comme chef de groupe au sein du département IT de la société Alcatel Schweiz AG à Zürich.

En 1999 Stéphane Micheloud est nommé Professeur HES en informatique à la Haute Ecole Valaisanne (HEVs) à Sierre. Outre ses tâches d'enseignement — notamment des cours sur la programmation orientée-objet et sur les systèmes d'exploitation —, il supervise des projets d'étudiants, intervient comme examinateur et expert de projets de diplôme, et participe à plusieurs projets CTI en tant que membre du groupe de compétences ISnet. Il participe également au programme de formation pédagogique organisé par la HES-SO avec le soutien de l'EPFL.

De 2002 à 2008 il est engagé comme assistant de recherche et comme administrateur système à temps partiel dans le groupe du Prof. Dr. Martin Odersky à l'EPF Lausanne; à ce titre il intervient comme assistant dans plusieurs cours de second cycle — principalement des cours de programmation avancée et sur la construction de compilateurs — et participe à plusieurs projets de recherche — MICS, PEPITO et Scala —. En particulier il contribue activement au développement du langage de programmation Scala. Il a notamment l'opportunité to partager ses idées et ses expériences avec d'autres chercheurs tels que Massimo Merro, Uwe Nestmann, Lex Spoon, Erik Stenman et Geoffrey Washburn, et avec d'éminents visiteurs tels que Gilad Bracha, Erik Ernst, Benjamin Pierce, Don Syme, Jan Vitek et Philip Wadler.

En avril 2004 il entame en tant qu'étudiant-doctorant son travail de recherche qui embrasse les domaines de la conception des langages et de la programmation distribuée. Le présent rapport décrit les motivations à la base de ce travail et résume les principales contributions et les expériences acquises durant les cinq dernières années.

Ses intérêts de recherche incluent les langages de programmation — notamment la conception des langages, les techniques de compilation et les environnements d'exécution —, la programmation distribuée — en particulier la sécurité et la mobilité du code —, et la programmation Internet — notamment les applications Web —.

About the Author

Born in 1964 Stéphane Micheloud grows up — and still lives — in Sion, the administrative centre of Valais, a beautiful region in the middle of the Swiss Alps.

From 1979 to 1984 he studies engineering sciences at the Lycée-Collège des Creusets (LCC) in Sion where he learns about the rigorousness of mathematical proofs in the class of Prof. Dr. Jean-Claude Pont and do his first arms with the language Pascal together with Prof. Paul Epiney. Those two experiences strongly influence his choice to embrace a career in the domain of computer science.

From 1984 to 1990 he studies computer science at ETH Zürich where he learns about the techniques of compiler construction in the class of Prof. Dr. Niklaus Wirth and deepens his knowledge of advanced computer architecture together with Prof. Dr. Marco Annaratone. At that time — to relate a small anecdote — he also meets the teaching assistants Martin Odersky and Clemens Szyperski who are now making very successful careers. In 1989 Stéphane Micheloud achieves his internship at Contraves AG in the group of Dr. Urs Ammann, project leader of CZ-Ada, a proprietary real-time Ada system. He receives his master degree in computer science in 1990 after having successfully achieved his diploma work entitled "*Oberon Compiler Back-End für Inmos Transputer*" under the supervision of Prof. N. Wirth.

From 1991 to 1997 he is employed as software engineer in the software engineering company Glance AG in Steinmaur (integrated since 2007 to the group adesso Schweiz AG) and participates to the development of several compiler tools, in particular M2CC (Modula-2) and POCC (Portal), two code translation tools to ANSI C. He also works as external contractor in the company Landis&Gyr AG in Zug and participates to the development of a new building control system in the project ATLAS. Together with Dr. Hans Bärfuss and Dr. Hans-Ruedi Aschmann he makes his first experiences with the management of software projects and customer relationship. At that time he discovers and quickly adopts the

language Java after several years of Modula-2 and C/C++ programming experiences.

From 1996 to 1999 Stéphane Micheloud pursues part-time postgraduate studies in computer sciences and telecommunication at NDIT (*eduswiss* since 1999, currently affiliated to the Swiss Virtual Campus). He attends a variety of advanced courses directed by domain specialists such as Prof. Dr. Antoine Delay, Prof. Dr. Jacques Savoy, Dr. Erich Gamma and Dr. Igor Metz.

From 1997 to 1999 he is employed as senior engineer and later as team leader in the IT department of the company Alcatel Schweiz AG in Zürich.

In 1999 Stéphane Micheloud is nominated Professor HES in computer science at the Haute Ecole Valaisanne (HEVs) in Sierre. Besides his teaching activities — i.e. bachelor courses on object-oriented programming and principles of operating systems —, he supervises student projects, acts as examinator and project expert, and participates to several CTI projects as member of the ISnet competence group. He also participates to the pedagogic program organised by the HES-SO and supported by the EPFL.

From 2002 to 2008 he is engaged as research assistant and as part-time system administrator in the group of Prof. Dr. Martin Odersky at EPF Lausanne; as such he acts as teaching assistant in several undergraduate courses — i.e. courses on advanced programming and compiler construction — and participates to several research projects — i.e. MICS, PEPITO and Scala —. In particular he contributes actively to the development of the Scala programming language. He also gets the opportunity to share ideas and experiences with other researchers such as Massimo Merro, Uwe Nestmann, Lex Spoon, Erik Stenman and Geoffrey Washburn, and with eminent visitors such as Gilad Bracha, Erik Ernst, Benjamin Pierce, Don Syme, Jan Vitek and Philip Wadler.

In April 2004 he starts his thesis work whose research area covers programming language design and distributed programming. The present report exposes the key ideas that motivated this work and summarizes its main contributions and the experiences gathered during the last five years.

His research interests encompass programming languages — i.e. language design, compiler techniques and runtime environments —, distributed programming — i.e. security and mobility of code —, and internet programming — i.e. web-based applications —.

Appendix A

Distributed Application Samples

In this appendix we provide code examples of a very simple distributed application built upon four RPC-like software frameworks and targeted at different run-time environments, namely:

- A Sun RPC application running natively on the Linux platform.
- A Java CORBA application running on the Java platform.
- A Java RMI application running on the Java platform.
- A .NET Remoting application running natively on the Windows platform.

Sun RPC

The following code has been adapted (mostly simplified) from the output generated by the RPCGEN tool, a generator tool for the RPC language (RPCL).

Given some interface definition (Listing 8.1), it generates several C source files — mainly a header file (Listing 8.2) with its (empty) implementation file (Listing 8.3) and (optionally) the corresponding server/-client template files (Listing 8.4 and Listing 8.5) — which include the needed glue code for accessing the RPC run-time library.

```
program HELLO {
    version VERSION_1 {
        string sayHello() = 1;
    } = 1;
} = 0x22222220;
```

Listing 8.1: RPC (RPC language).

```
#define HELLO 0x22222220
#define VERSION_1 1

#define sayHello 1
extern  char ** sayhello_1_svc(void *, struct svc_req *);
```

Listing 8.2: RPC (interface).

```
char ** sayhello_1_svc(void *argp, struct svc_req *rqstp) {
    static char *result = "Hello !";
    return &result;
}
```

Listing 8.3: RPC (implementation).

```
// static void hello_1_call(..) { .. }
static void hello_1(struct svc_req *rqstp, register SVCXPRT *transp) {
    switch (rqstp->rq_proc) {
        case NULLPROC:
            svc_sendreply(transp, (xdrproc_t)xdr_void, (char *)NULL);
            break;
        case sayHello:
            hello_1_call(rqstp, transp);
            break;
        default:
            svcerr_noproc(transp);
            break;
    }
}
int main(int argc, char **argv) {
    pmap_unset(HELLO, VERSION_1);
    register SVCXPRT *transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    svc_register(transp, HELLO, VERSION_1, hello_1, IPPROTO_TCP);
    svc_run();
}
```

Listing 8.4: RPC (server).

```
/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

char* sayHello_1(CLIENT *clnt) {
    static char *result = NULL;
    if (clnt_call(clnt, sayHello,
        (xdrproc_t) xdr_void, (caddr_t) NULL,
        (xdrproc_t) xdr_wrapstring, (caddr_t) &result,
        TIMEOUT) != RPC_SUCCESS) {
        clnt_perror(clnt, "call failed");
    }
    return result;
}
int main(int argc, char *argv[]) {
    CLIENT *clnt = clnt_create("localhost", HELLO, VERSION_1, "tcp");
    printf("%s\n", sayHello_1(clnt));
    clnt_destroy(clnt);
    return 0;
}
```

Listing 8.5: RPC (client).

CORBA IDL

The CORBA specification defines a standard architecture for connecting distributed objects written in any language; in the following we present a simple code example written in Java.

```
module idldemo
{
  interface Hello
  {
    string sayHello();
  };
};
```

Listing 8.6: Java IDL (interface).

Given the above file (Listing 8.6) containing IDL definitions, the IDL-to-Java compiler generates several auxiliary Java source files — e.g. the abstract class `HelloPOA` (inherited by the servant object, see Listing 8.7), the interface `Hello` and the helper class `HelloHelper` (used to bind resp. resolve the object reference, see Listing 8.8 and Listing 8.9) — which are then passed to the Java compiler, together with the user-defined server and client code.

```
class HelloImpl extends HelloPOA {
    public String sayHello() {
        return "\nHello world !!\n";
    }
}
```

Listing 8.7: Java IDL (implementation).

```
public class Server {
2    public static void main(String[] args) {
        try {
4            ORB orb = ORB.init(args, null);
            POA poa = POAHelper.narrow(
6                orb.resolve_initial_references("RootPOA"));
            poa.the_POAManager().activate();
8            Hello href = HelloHelper.narrow(
                poa.servant_to_reference(new HelloImpl()));
10            NamingContextExt ncRef = NamingContextExtHelper.narrow(
                orb.resolve_initial_references("NameService"));
12            NameComponent path[] = ncRef.to_name("Hello");
            ncRef.rebind(path, href);
14            System.out.println("Server started");
            orb.run();
16        }
        catch (Exception e) {
18            System.out.println("Server error: " + e.getMessage());
            e.printStackTrace();
20        }
    }
```

```
22 }
```

Listing 8.8: Java IDL (server).

The server application (Listing 8.8) proceeds in three steps to export the remote service(s):

- it first initializes the ORB (line 4) and activates the associated POA (line 7),
- it then gets the object reference for the servant (line 8) and binds it to the (root) naming context (line 13),
- and, finally, it waits for invocations from clients (line 15).

```
   public class Client {
 2     public static void main(String[] args) {
           try {
 4             ORB orb = ORB.init(args, null);
               NamingContextExt ncRef = NamingContextExtHelper.narrow(
 6                 orb.resolve_initial_references("NameService"));
               Hello helloImpl = HelloHelper.narrow(
 8                 ncRef.resolve_str("Hello"));
               System.out.println(helloImpl.sayHello());
10         }
           catch (Exception e) {
12             System.out.println("Client error: " + e.getMessage());
               e.printStackTrace();
14         }
       }
16 }
```

Listing 8.9: Java IDL (client).

The client application (Listing 8.9) proceeds in three steps to invoke the remote service(s):

- it first initializes the ORB (line 4),
- it then resolves the object reference in the (root) naming context (line 7),
- and, finally, it invokes the exported method `sayHello` of the server object (line 9).

Java RMI

```
public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Listing 8.10: Java RMI (interface).

```
public class HelloImpl extends UnicastRemoteObject implements Hello {
    public HelloImpl() throws RemoteException { super(); }
    public String sayHello() { return "Hello !"; }
}
```

Listing 8.11: Java RMI (implementation).

```
public class Server {
2   public static void main(String[] args) throws Exception {
        try { LocateRegistry.createRegistry(8888); }
4       catch (Exception e) { LocateRegistry.getRegistry(8888); }

6       Naming.rebind("//localhost:8888/hello", new HelloImpl());
        System.in.read();
8   }
}
```

Listing 8.12: Java RMI (server).

```
public class Client {
2   public static void main(String[] args) throws Exception {
        Hello obj = (Hello) Naming.lookup("//localhost:8888/hello");
4
        System.out.println(obj.sayHello());
6   }
}
```

Listing 8.13: Java RMI (client).

.NET Remoting

```
public interface IHello {
    String sayHello();
}
```

Listing 8.14: .NET Remoting (interface).

```
public class Hello: MarshalByRefObject, IHello {
    public String sayHello() { return "Hello !"; }
}
```

Listing 8.15: .NET Remoting (implementation).

```
public class Server {
    public static void Main(string[] args) {
        ChannelServices.RegisterChannel(
            new TcpChannel(8888), false/*ensureSecurity*/);
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(Hello),
            "hello",
            WellKnownObjectMode.SingleCall);
        Console.ReadLine();
    }
}
```

Listing 8.16: .NET Remoting (server).

```
public class Client {
    public static void Main(string[] args) {
        IHello obj = (IHello) Activator.GetObject(
            typeof(IHello),
            "tcp://localhost:8888/hello");
        Console.WriteLine(obj.sayHello());
    }
}
```

Listing 8.17: .NET Remoting (client).