Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Generic operations and capabilities in the JR concurrent programming language

Hiu Ning (Angela) Chan[a], Andrew J. Gallagher[a], Appu S. Goundan[a], Yi Lin William Au Yeung[a], Aaron W. Keen[b], Ronald A. Olsson[a,*]

[a] *Department of Computer Science, University of California, Davis, CA 95616, USA*
[b] *Computer Science Department, California Polytechnic State University, San Luis Obispo, CA 93407, USA*

A R T I C L E   I N F O

Article history:
Received 24 January 2008
Accepted 16 May 2008

Keywords:
Concurrency
JR operations
Java
Generics
Language design
Language implementation

A B S T R A C T

The JR concurrent programming language extends Java with additional concurrency mechanisms, which are built upon JR's operations and capabilities. JR operations generalize methods in how they can be invoked and serviced. JR capabilities act as reference to operations. Recent changes to the Java language and implementation, especially generics, necessitated corresponding changes to the JR language and implementation. This paper describes the new JR language features (known as JR2) of generic operations and generic capabilities. These new features posed some interesting implementation challenges. The paper describes our initial implementation (JR21) of generic operations and capabilities, which works in many, but not all, cases. It then describes the approach our improved implementation (JR24) uses to fully implement generic operations and capabilities. The paper also describes the benchmarks used to assess the compilation and execution time performances of JR21 and JR24. The JR24 implementation reduces compilation times, mainly due to reducing the number of files generated during JR program translation, without noticeably impacting execution times.

1. Introduction

The JR concurrent programming language [1,2] extends Java to provide a rich concurrency model, based on that of the SR concurrent programming language [3]. JR performs synchronization using an object-oriented approach, and provides dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, asynchronous communication, rendezvous, and dynamic process creation.

JR provides much of this added functionality via its *operation* abstraction, which can be considered a generalization of a method and which can be invoked and serviced in different ways. Along with operations, JR provides *capabilities*, which act as (first-class) references or pointers to operations and which are useful in concurrent programs to connect together the different participants in the computation.

During program compilation, JR source programs are translated into standard Java programs, which are then compiled using the standard Java compiler. During program execution, JR programs use the JR runtime support system, which helps provide JR's extended functionality. Both the JR language translator and the JR runtime support system are written in standard Java.

The original JR language, which we call JR1, was an extension of Java 1.x. Its implementation originated from the implementation of Java 1.2.

* Corresponding author. Tel.: +1 530 752 7004; fax: +1 530 752 4767.
E-mail address: olsson@cs.ucdavis.edu (R.A. Olsson).

1477-8424/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2008.05.002

Java 5.0 (or 1.5)[1] [4] made significant additions to Java 1.x. Java 5.0 contains several new language features including generics, enhanced for-loop, autoboxing/unboxing, typesafe enums, varargs, static import, and annotations [5]. Generics allows a type or method to operate on objects of various types while providing compile-time safety. It adds compile-time safety to Java's Collections Framework and eliminates the drudgery of casting [5]. The implementation of Java also changed for Java 5.0, including a redesign and restructuring of the translator and in how it handles RMI. (The latest version of Java is 6.0 (or 1.6). The differences between Java 5.0 and Java 6.0 are not significant for the focus of this paper.)

To accommodate these changes in Java, we made corresponding changes to the JR language and implementation [6]. We use JR2 to refer to this new version of JR. The most interesting and challenging of the changes is generics, specifically how we added generic operations and generic capabilities. The design aspects were straightforward, but the implementation aspects were not. The implementation of JR2 is based on the implementation of Java 5.0. That is, we re-implemented JR, merging our changes for JR1 (and adapting those changes to fit within the framework of the Java 5.0 implementation) and adding new code for the new JR2 features.

This paper describes our experience with adding generic operations and generic capabilities to JR. Our initial approach extended our implementation of JR1, the most recent version being JR 1.00061. This initial implementation of JR2 was designated JR 2.00001 [6]; we refer to it as JR21. Although this approach works in most cases, it does not work in some important cases involving capabilities for generic operations. We devised an alternate implementation, which works in all cases. This implementation of JR2 is designated JR 2.00004.[2]

This new approach involves a fundamental change in the code that the JR translator generates. In JR21, each operation is represented as its own class with the operation's arguments represented exactly as declared. In JR24, all operations are represented as one predefined class with the operations' arguments represented as a single array of Java Objects. This new approach has the benefit of significantly reducing the amount of code produced for each JR program; specifically, it reduces the number of files produced for JR operations and capabilities. Thus, it reduces compilation times yet produces execution times that are nearly the same.

The rest of this paper is organized as follows. Section 2 presents a brief overview of the JR language. Section 3 describes the new JR2 language features of generic operations and generic capabilities. Section 4 discusses the JR2 implementation in general; how JR21 implements generic operations and capabilities, which works in many, but not all, cases; and the approach the JR24 implementation uses to fully implement generic operations and capabilities. Section 5 compares the compilation and execution time performances of JR21 and JR24; as noted earlier, JR24 reduces compilation times without impacting execution times. Section 6 explores some related issues and presents some related work. Finally, Section 7 concludes.

2. Overview of the JR concurrent programming language

JR extends Java with a richer concurrency model [2,7], based on that of SR [3,8]. JR provides dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, dynamic process creation, rendezvous, asynchronous message passing, semaphores, and shared variables.

Of specific interest in this paper are JR's operations and capabilities, which are used to effect some of the above features.

An operation can be considered a generalization of a method. Like a method, it has a name and can take parameters and return a result. Like a method declaration, an operation declaration specifies the *signature* of the operation, i.e., the return type and the types of parameters. Unlike a method, an operation can be invoked and serviced in different ways, which yields flexibility in solving concurrent programming problems.

More specifically, an operation can be invoked in two ways: synchronously by means of a call statement (`call`) or asynchronously by means of a send statement (`send`). An operation can also be serviced in two ways: by a method or by input statements (`inni`). This yields the following four combinations (from [2]):

Invocation	*Service*	*Effect*
`call`	method	Procedure (method) call (possibly remote)
`call`	`inni`	Rendezvous
`send`	method	Dynamic process creation
`send`	`inni`	Asynchronous message passing

JR allows several abbreviations for common uses of operations: process declarations, op-method declarations, receive statements, and semaphores. JR also provides a few additional statements that involve operations and that are useful for concurrent programming: the reply, forward, and concurrent invocation statements.

[1] Both version numbers "1.5.0" and "5.0" can be used to identify this release. Version "5.0" is the product version, while "1.5.0" is the developer version.

[2] Version 2.00002 was the first version using this new approach. Versions 2.00003 and 2.00004 contain several bug fixes and other enhancements. Version 2.00006 is the latest release (January 2008); it corrects several bugs in JR24. None of the bug fixes has a significant effect on the performance results described in this paper. Version 2.00005 was an experimental version described in [6]. It was based on JR21 and predates 2.00003 and 2.00004; it was a first step toward JR24, but it was not a released version of JR.