Getting Operations Logic Right

Types, Service-Orientation, and Static Analysis

Karthikeyan Bhargavan Andrew D. Gordon
Microsoft Research

Abstract

The human operators of datacentres work from a manual, sometimes known as the run book, that lists how to perform operating procedures such as the provisioning, deployment, monitoring, and upgrading of servers. To improve failure and recovery rates, it is attractive to replace human intervention by software, known as operations logic, that automates such operating procedures. We advocate a declarative programming model for operations logic, and the use of static analysis to detect programming errors, such as the potential for misconfiguration.

1. Background: The Datacentre is the Computer

A datacentre is some housing (typically a room or a building) that physically contains a cluster of commodity servers, and provides them with power, cooling, and networking. Datacentres are the computers that run applications such as websites (search and mail in particular), financial services, computational science, and virtual worlds.

If the datacentre is the computer [Patterson, 2008], what is the program? We divide the code of a datacentre program into two parts: business logic and operations logic.

Datacentre Program = Business Logic + Operations Logic

The *business logic* is code that formalizes business tasks, that is, the core functionality of the application, such as how to manage an inbox, or how to sell a book. By analogy, the *operations logic* is code that formalizes operations tasks. The operations logic determines how to run and manage the application on a cluster of machines.

To illustrate the idea of business logic, consider a website implemented conventionally as three roles: web server, application server, and database server. A typical way to run such an application is to package up the code to run on each server role in a *disk image*, that is, a file containing the contents of the server's disk; the disk image holds all the code that runs on an individual server. The disk image for each role contains an operating system together with components specific to the role, such as web server, database server, and binaries and local configuration files specific to the application. Depending on load, there may be multiple servers running off the disk image for each role; these servers are known as *instances* of the role. According to our definition, the disk images for the three roles constitute the business logic of the application: this is the code that determines how to conduct the business implemented by the website.

Turning to operations, datacentres are run by human operators who work from a *run book*, a manual that lists how to perform various operating procedures. Here are some example tasks; for a comprehensive discussion of operations tasks see Anderson [2006].

- Provisioning: how many instances of each role where.
- Deployment: install disk images, start instances.
- Interconnection: connect the instances together.
- Monitoring: monitor instances for failure, overload.
- Evolution: respond to events, change deployment, versioning.

To improve failure and recovery rates, and to reduce need for actual operator involvement, many of these tasks would be better performed in software than manually. For example, events such as abrupt increases of load or machine failure should be handled swiftly and accurately. The empirical evidence [Oppenheimer et al., 2003, Nagaraja et al., 2004] is that operator errors in datacentre applications are common and costly. The scale and diversity of datacentre applications is another reason to automate operations tasks and hence reduce the need for human intervention.

This idea (sometimes known as *run book automation*) of encoding operations tasks as software, operations logic, is now a well-established trend. Various examples include include the declarative configurations of Anderson [1994], the self-healing systems of Burgess [1998], the automated deployment and configuration of SmartFrog [Goldsack et al., 2003], the manifesto for autonomic computing [Kephart and Chess, 2003], and the fault-tolerant programming model for large-scale clusters of MapReduce [Dean and Ghemawat, 2004]. More recently, Isard [2007] reports AutoPilot, a system to automate provisioning, deployment, and monitoring in large clusters.

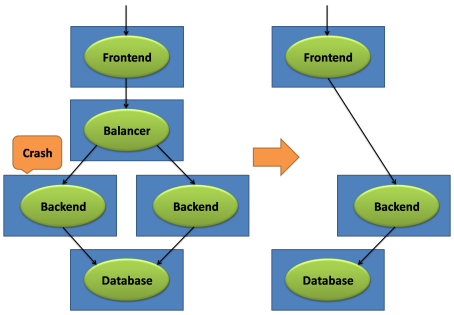
2. Problem: Getting Operations Logic Right

We claim that whereas development methods for business logic are well advanced (though not infallible), writing correct operations logic is more problematic (although much progress has been made). One reason is that the bulk of operations logic remains as low-level imperative code, written in scripting languages without the benefit of static typing.

For example, virtual machine monitors (VMMs) allow the provisioning, deployment, and monitoring of virtualized servers; the operations tasks associated with VMMs can be controlled entirely in software, but the programming interfaces exposed by today's VMMs are quite low-level. VMMs can easily be configured with scripts to be called in the event of machine failure or overload, but these scripts are hard to test, and are likely to contain bugs, particularly if they are called in response to relatively rare events.

To address the problem of getting operations logic right, our position is that operations logic would be better developed in high-level, statically typed programming languages, just as business logic is, than in low-level scripting languages, the common case at present.

Moreover, we advovate the static analysis of operations logic to help detect errors early. For example, techniques such as type-checking and model-checking are well suited to checking the behaviour of error recovery code.



(a) An Evolving Three-tier Application

```
1 let startupThreeTier db backend frontend =
2   let (d,dbServ) = start db () in
3   let (b1,bServ1) = start backend dbServ in
4   let (b2,bServ2) = start backend dbServ in
5   let (b,balance) = start balancer [bServ1;
6                                     bServ2] in
7   let (f,fServ) = start frontend balance in
8   let _ = register b1 (function Crash →
9     reconfigure f bServ2; stop b1; stop b) in
10  let _ = register b2 (function Crash →
11    reconfigure f bServ1; stop b2; stop b) in
12  fServ
```

(b) An Example Operations Logic Program

```
1 type (α,β) Service
2 val call: (α,β) Service → α→ β
3 type (γ) Server
4 type (γ,'services) Setup
5 val start: (γ,'services) Setup → γ→
6                     (γ) Server × 'services
7 val stop: (γ) Server → unit
8 val reconfigure: (γ) Server → γ→ unit
9 type event = Crash
10 val register: (γ) Server →
11                     (event → unit) → unit
12 val balancer: ((α,β) Service list,
13                (α,β) Service) Setup
```

(c) A Typed Service Management API

3. Declarative Approaches to Operations Logic

Baltic: Service Combinators for Virtual Machines The Baltic system [Bhargavan et al., 2008] addresses the problem raised in the previous section. The operations logic of a Baltic application is written in the functional language F# (a dialect of ML) [Syme et al., 2007]. Baltic applications are composed from virtualized servers and intermediaries, such as load balancers.

The programming model is based on the observation that a server is a software component that imports and exports a set of communication endpoints. For example, a web server may import an endpoint exported by a database server, and export a communication endpoint implementing a URL. The capability to create a server instance from a particular disk image is represented in Baltic as a typed F# function: the function receives the imported endpoints as a set of typed values, boots a machine instance, and returns the exported endpoints as a set of typed values. If we let a *service* be a set of endpoints, we may say that the Baltic model is *service-oriented*. The function to boot a server transforms the service it imports to the service it exports. Operations logic in Baltic manipulates a typed call graph; nodes are role instances, labelled with exported services, while edges are potential calls from one role to an endpoint on another.

This high-level view of instance creation contrasts with the low-level view in conventional programming models for operations logic. Typical programming models for VMMs (for example, Virtual Server [Armstrong, 2007]) are lower-level in that they are *device-oriented* rather than being service-oriented: programs manipulate a graph whose nodes are virtual devices (disks, processors, network adapters) and whose edges are virtual wiring. For example, the Virtual Server function to create a role instance simply boots a machine from a disk image; no matter their imports and exports, all disk images are represented as untyped files. In Baltic, by contrast, there is a distinctly typed function to create each role.

Our initial implementation [Bhargavan et al., 2008] establishes the feasibility of this new approach. We have operations logic in F# that manages pre-existing disk images from a sample multi-tier web application running on a single VMM. There is a semantics based on a typed concurrent λ-calculus with partitions, and an implementation using Virtual Server. Type-checking the operations logic statically detects some errors, such as endpoint interconnection bugs. We can also symbolically simulate the operations logic, without the business logic, to find other errors.

Still, there is much to be done to validate this approach. One direction is to demonstrate its viability at the scale of a datacentre rather than a single VMM. Another is to perform more sophisticated static analysis than type-checking.

Example We illustrate the style of the Baltic approach through the example depicted at the top of this page. For the sake of brevity, we use simplified pseudo-code, rather than the actual code running in Baltic. Our aim is to write a program to set up and manage a simple three-tier application as shown in Figure (a); it consists of a frontend server that forwards incoming requests to a backend server that in turn may send a message to a database. The backend server can be a bottleneck; hence, it is replicated and a load balancer divides requests between two backend servers to avoid overloading either one. Once the application is up and running, if one of the backend servers were to crash, we would like to reconfigure the frontend to forward requests directly to the other backend, bypassing the redundant load balancer.

The business logic consists of three packages, db, backend, and frontend, that consist of software and configuration data that fully describe each of the three server roles.

The operations logic is written as the F# program in Figure (b) using functions in the service management API of Figure (c). The function startupThreeTier takes db, backend, and frontend as its arguments (line 1) and then starts up instances of each server. Line 2 calls start to provision a server d to act as a database, install the package db on this server, start it up, and return the database service address dbServ. (Here, db requires no configuration data hence the second argument to start is unit.) Line 3 provisions a backend server b1, installs the package backend on it, configures the server with the address of the database (dbServ), starts up b1 and returns the address of its backend service bServ1. Similarly, line 4 starts up a second backend server b2 and returns its service address bServ2. Lines 5–6 provision a load balancing server b that has the balancing software balancer installed on it. The balancer is configured with a list of available backend services (here just bServ1 and bServ2) and returns a service, balance, that divides requests equally amongst the available backends. Line 7 provisions a frontend server f, installs the package frontend on it, configures it with the balancer address balance, starts it up, and returns the service fServ that can be accessed by the external world. At this stage, the three-tier application is up and running in its initial configuration. The next four lines register event handlers that prescribe how the application should be reconfigured in the event of a server crash. Lines 8–9 register, via the function register, an event handler that constantly monitors the backend server b1; when it detects a Crash event, it reconfigures the frontend f to point to the other backend service bServ2 (bypassing the balancer); it then stops the crashed server b1 and the redundant balancer b. Lines 10–11 register a similar event handler for b2. Finally, the function startupThreeTier returns the external service fServ (line 12). Hence, this program performs all the standard management tasks of operations logic: it provisions and deploys servers for each role, interconnects them using application-specific configuration, monitors them to detect events, and evolves the application by reconfiguring the frontend.

A Typed Service Management API The general forms of the functions start, stop, register, reconfigure, and balance used in our example are shown in the typed interface of Figure (c).

The type (α,β) Service represents the address of a service that takes requests of type α and returns responses of type β. The function call makes a remote procedure call to such a service. For instance, suppose the frontend service in our example has type FrontendService = (int,bool) Service; then one may call it with an integer request to get a boolean response.

The type (γ) Server represents the name of a server providing services that have configuration data of type γ. The type (γ,'services) Setup represents a package containing software for services of type 'services that have configuration data of type γ. For instance, the packages db, backend, and frontend are all of this type; frontend has type (BackendService,FrontendService) Setup, meaning that it can install a service of type FrontendService that needs to be configured with the address of a service of type BackendService. The function start takes a package of type (γ,'services) Setup, configuration data of type γ and returns the name of a new server of type (γ) Server that provides services of type 'services. The function stop stops a server; reconfigure installs new configuration data.

The type event records events relevant to operations logic; for simplicity, we only record server Crash events. The function register takes a server and an event handler of type event $\rightarrow$ unit and starts a process that monitors the server and invokes the handler when it detects an event.

Service management APIs typically include preprogrammed packages for common server tasks. Here, the package balancer takes a list of services, each of type (α,β) Service, as its configuration data and installs a load balancing service of the same type.

The service management API of Figure (c) may be implemented and interpreted in several ways. In the Baltic system, packages are implemented as disk images and servers are virtual machines within a VMM. In Autopilot, packages are self-installing software manifests and servers are generic computers running within a datacentre. For each interpretation, we can define a formal semantics for programs written against the API, by defining the functions of the API in terms of expressions in a partitioned lambda-calculus [Bhargavan et al., 2008]. Such a semantics can be used to reason about operations logic programs, such as the one in Figure (b), and to symbolically simulate and test such programs before deployment.

Our service management API is typed, and so are our programs. Type-checking catches common interconnection errors statically. For instance, in line 9 of Figure (b), if we mistyped bServ2 as dbServ, hence reconfiguring the frontend with a database service in place of a backend service, the type-checker would catch this error. In an untyped setting, such a misconfiguration would be detected only after deployment, when a frontend becomes unresponsive or returns fault messages after the application evolves in response to a (possibly rare) server crash. Conversely, some errors are difficult for type-checking to detect. For instance, the event handlers in Figure (b) do not attempt to restart failed servers or to notify human operators about degraded services; so the operations logic survives one failure but not two. Type-checking does not find such behavioural problems, but other static analyses such as model-checking should be applicable.

To summarize, type-checking operations logic is a good start. Still, there remains plenty of scope for applying stronger static analyses to provide better guarantees about the programs running on datacentres.

Acknowledgments

We thank Iman Narasamdya for his work on the Baltic implementation. Galen Hunt drew our attention to the term “operations logic”. Ioannis Baltopoulos commented on a draft of this paper.

References

P. Anderson. *System Configuration*, volume 14 of *Short Topics in System Administration*. SAGE, 2006.

P. Anderson. Towards a high-level machine configuration system. In *Proceedings of the 8th Large Installations Systems Administration (LISA) Conference*, pages 19–26, Berkeley, CA, 1994.

B. Armstrong. *Professional Microsoft Virtual Server 2005*. Wiley, 2007.

K. Bhargavan, A. D. Gordon, and I. Narasamdya. Service combinators for farming virtual machines. In *COORDINATION'08*, 2008. To appear. Extended version available as Microsoft Research Technical Report MSR–TR–2007–165.

M. Burgess. Computer immunology. In *LISA '98: Proceedings of the 12th USENIX conference on System administration*, pages 283–298. USENIX Association, 1998.

J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, pages 137–150, 2004.

P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog: Configuration and automatic ignition of distributed applications, 2003. Presented at 2003 HP Openview University Association conference. Available at `http://www.hpl.hp.com/research/smartfrog/`.

M. Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, 2007.

J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan. 2003.

K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, pages 61–76, Berkeley, CA, USA, 2004. USENIX Association.

D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *4th Usenix Symposium on Internet Technologies and Systems (USITS'03)*, 2003.

D. A. Patterson. Technical perspective: the data center is the computer. *Commun. ACM*, 51(1):105–105, 2008. ISSN 0001-0782.

D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.