Service Combinators for Farming Virtual Machines

Karthikeyan Bhargavan
Microsoft Research

Andrew D. Gordon
Microsoft Research

Iman Narasamdya
University of Manchester

December 2007

Technical Report
MSR–TR–2007–165

Microsoft Research
Roger Needham Building
7 J.J. Thomson Avenue
Cambridge, CB3 0FB
United Kingdom

Service Combinators for Farming Virtual Machines

Karthikeyan Bhargavan
Microsoft Research

Andrew D. Gordon
Microsoft Research

Iman Narasamdya
University of Manchester

Abstract

Management is one of the main expenses of running the server farms that implement enterprise services, and operator errors can be costly. Our goal is to develop type-safe programming mechanisms for combining and managing enterprise services, and we achieve this goal in the particular setting of farms of virtual machines. We assume each server is service-oriented, in the sense that the services it provides, and the external services it depends upon, are explicitly described in metadata. We describe the design, implementation, and formal semantics of a library of combinators whose types record and respect server metadata. We describe a series of programming examples run on our implementation, based on existing server code for order processing, a typical data centre workload.

Contents

1 Introduction **7**
- 1.1 Background: Farms and Roles . . . 7
- 1.2 Background: Managing Server Farms in Software . . . 7
- 1.3 Background: Service Orientation . . . 7
- 1.4 Service Combinators for Farming Virtual Machines . . . 8
- 1.5 Contributions . . . 10
- 1.6 Contents of the Paper . . . 11

2 Service Combinators by Example **11**
- 2.1 Generating a Typed Interface to Resources . . . 11
- 2.2 Example 1: Creating an Isolated VM Farm . . . 13
- 2.3 Example 2: Importing and Exporting Services . . . 14
- 2.4 Example 3: Par and Or Intermediaries . . . 14
- 2.5 Example 4: References, Updating References, and Events . . . 16
- 2.6 Example 5: Snapshots of VMs . . . 17

3 An Implementation of Service Combinators **18**
- 3.1 Background: Virtual Server Farms . . . 18
- 3.2 Baltic Architecture . . . 18
- 3.3 Basic Module: B−c.ml . . . 19
- 3.4 Environment Module: Em−c.ml . . . 20
- 3.5 Assembling the Baltic Manager . . . 20

4 The Partitioned λ-Calculus **20**

5 A Formal Semantics of Service Combinators **24**
- 5.1 Service Combinator Semantics: B.ml . . . 24
- 5.2 Environment Model: Em.ml . . . 27
- 5.3 Example . . . 29
- 5.4 Assembling the Formal Model of a Baltic Script . . . 29

6 Related Work and Conclusions **31**

A An Extended Example **34**

B Metadata for Services and Resources **35**

C Metadata Compiler and Baltic Scripts **38**
- C.1 Generating the Interface: Em.mli . . . 38
- C.2 Generating Em−c.ml . . . 39

D The Partitioned λ-Calculus **40**
D.1 Syntax 40
D.2 Operational Semantics of Expressions 43
D.3 Type System 46
D.4 Translational Semantics of Modules 53

1 Introduction

1.1 Background: Farms and Roles

Abstractly, a *server farm* is a collection of servers that runs one or more (parallel) programs, such as hosting a website or running compute jobs. Servers in a farm may have both local and remote dependencies. They may receive requests from remote clients, such as a web browser. They may also send requests to remote servers, to perform a credit card transaction, for example. Servers may also send local requests to other servers within the farm; for example, a front end web server may send a request to a database server.

We assume each server boots off a *disk image*, such as the contents of a local hard drive, or an image fetched over the network. Typically, each server plays a particular *role*, such as web server, mail server, application server, and so on. At any time there may co-exist multiple versions of each role, and multiple servers instantiating each version. We assume the disk images and hence the behaviours of the servers in any one version are essentially the same; the small differences relate to the identity of each server (for example, machine name, security identifiers, licensing data). The functionality embodied in disk images is often referred to as *business logic*, as it codifies the steps needed to enact various business processes—how to sell a book, for example.

1.2 Background: Managing Server Farms in Software

Conventionally, operations staff manage server farms using a mixture of command prompts, scripts, various graphical tools, and actual physical configuration. Management includes provision and interconnection of servers, as well as responding to events such as peaks and troughs in load, or failures of individual servers. Human error is a leading cause of service failures [Oppenheimer et al., 2003].

Manual management of server farms is expensive; low-level tools and the sheer complexity of the task make it prone to human error. Typed APIs such as our service combinators can help reduce the possibilities for human error.

Technologies such as network booting [Intel 1999] and virtualization [Meyer and Seawright, 1970] of commodity hardware [Wolf and Halter, 2005, Barham et al., 2003] allow hardware resources to be dynamically allocated to server roles. Moreover, to eliminate physical configuration completely, virtual machines (VMs) can even be rented on demand over the web [Bavier et al., 2004, Hoykhet et al., 2004, Ama, 2006, Garfinkel, 2007]. These technologies reduce the need for human intervention and transform server farm management into a programming problem. The programming problem is to write *operations logic* to codify the tasks performed by operations staff.

1.3 Background: Service Orientation

Let a *procedure* be some functionality exposed on a communication port via a protocol such as RPC-style request/response, and let a *service* be a set of procedures. Server roles are increasingly *service-oriented*, in the sense that each role is described as importing and exporting services. The role implements its exports, and has dependencies

on its imports. These imports and exports have explicit types that describe message contents and message patterns.

For example, an enterprise order processing application (drawn from published server code [Pallmann, 2005]) has an order entry role implementing a service (its export) consisting of a single procedure SubmitOrder, and conforming to the IOrderEntry interface. We give this and related interfaces below.

```
public interface IOrderEntry { string SubmitOrder(Order order); }
public interface IPayment { string AuthorizePayment(Payment payment); }
public interface IOrderProcessing { void SubmitOrder(Order order); }
```

A request message sent to the exported procedure is an invocation of the SubmitOrder method and must include a value of type Order. The response message includes the result, a string.

The code for SubmitOrder needs to consult a remote site to make an authorization decision. Hence, the order entry role has a dependency on the IPayment interface (its import). After authorization, SubmitOrder fulfills the order by calling an order processing procedure in the IOrderProcessing interface (its second import).

In the simplest configuration, the example code implementing each interface is installed on a separate disk image, and all three roles run locally on servers in the same farm. In practice, the payment service is likely to be hosted on a remote site.

Service-oriented designs often use SOAP [Box et al., 2000] messages for requests and responses, while service metadata, such as request and response types, is often described using the Web Services Description Language (WSDL) [Christensen et al., 2001]. SOAP and WSDL are platform-independent XML formats. There are many development tools and software platforms for producing service-oriented disk images, where the imports and exports are described with WSDL. In our example, the order processing code is in C#. It relies on .NET communication libraries and tools to exchange SOAP messages and to map between C# interfaces and WSDL metadata.

1.4 Service Combinators for Farming Virtual Machines

If the server farm is the computer, what is the program? Our proposal is that a program to run on a server farm consists of (1) a set of pre-existing disk images, (2) a set of URIs for the services exported and imported by the program, and (3) a script for assembling the VMs, interconnecting them, sometimes via intermediaries for tasks such as load balancing, and managing the resulting system. The disk images implement the business logic of the program, while the script implements the operations logic.

Program for a Server Farm = Disk Images + External URIs + Script

Our main goals are (1) to develop a typed combinator-based API for scripting operations logic, in particular, to treat service-oriented disk images as components to be interconnected using standard networking protocols; and (2) to develop a formal semantics to support reasoning about reachable configurations.

On the other hand, the tasks of writing business logic and of building disk images are outside our scope—there are many existing tools for these purposes. Also,

although the techniques of process calculi adapt well to formalizing the semantics of our combinators, our primary intention is not to develop another process calculus.

In this paper, we describe the design and implementation of *Baltic*, a type-safe API for expressing operations logic. Our API consists of a set of combinators for importing and exporting SOAP services described by WSDL metadata, for assembling services provided by the VMs via intermediaries, and for managing the resulting system. Concretely, the combinators are functions in the F# dialect of ML [Syme, 2005]. These combinators allow an ML script to control a small-scale server farm consisting of a set of SOAP intermediaries interconnecting a set of VMs managed by a Virtual Machine Monitor (VMM). Our particular VMM is Virtual Server [Armstrong, 2007], running on dual processor hardware suitable for test automation or for (modest) production workloads.

The intended scope of this paper is relatively small farms of servers, such as could be supported by a small number of VMMs. Our implementation is a research prototype, but if engineered for production, we believe it would usefully support small websites or test environments. We have not attempted to design a comprehensive set of intermediaries, although we can easily add more. Further research on scalability would be needed for our approach to apply to large-scale server farms used for parallel data processing (see Dean and Ghemawat [2004], for example). Still, even if our practical implementation is on a small scale, we demonstrate for the first time scripts that both (1) manipulate VMs and interconnect them with standard TCP/IP protocols, and (2) have a formal semantics suitable for typechecking and static analysis.

Next, we introduce some features of our model.

A Type of Remote Procedures. A value of a type (α,β) proc is the network address (a URI) of a SOAP procedure, hosted either on the physical server or on one of the managed VMs. The procedure expects SOAP requests and returns SOAP responses whose bodies correspond to the ML types α and β, respectively.

Service-Oriented Metadata for Resources. We refer to the disk images and the URIs of the exported and imported services, of a program running in a farm as its *resources*. We propose a *resource metadata* format, which includes WSDL descriptions of each export, each import, and of the services imported and exported by each disk image. (To the best of our knowledge, there is no prior service-oriented metadata format for disk images.) If *m* is the metadata for the resources of an application, we generate a typed management interface, named E*m*.mli, for the application. (The notation E*m*.mli denotes an interface that is a function of the metadata *m*.) This interface includes ML types corresponding to the WSDL request and response types for each service mentioned in the metadata. In our running example, we have the types:

```
type tOrderEntry = (Order,string) proc
type tPayment = (Payment,string) proc
```

The ML definitions of the Order and Payment types correspond to the types mentioned in the C# interfaces used to implement this service on this particular disk image. There is no direct dependency on the implementation language of the service; the ML

types are generated from the WSDL description, which itself can be generated from a wide range of implementation languages.

The E*m* interface also includes a function for booting a fresh VM from each disk image described by *m*. Given the imported procedures it returns the VM identifier, together with its exported procedures. In the case of our order entry role, we have:

val createOrderEntryRole : tPayment → tOrderProcessing → (vm × tOrderEntry)

The disk image is stored as an ordinary file, and a VMM can boot a VM off such a file. Our createOrderEntryRole function is a higher-level abstraction that knows the path to the disk image, boots a VM using the disk image as a fresh virtual disk, configures the VM with the address of the tPayment procedure, and eventually returns a tOrderEntry procedure.

A distinctive feature of our approach is that instead of presenting disk images as untyped files, we generate code, like createOrderEntryRole, that presents disk images as functions manipulating typed procedures. Hence, type checking catches interconnection errors that would otherwise cause failures at run time, either during initial configuration or later during reconfigurations.

Typed Access to External Procedures. We also need to refer to external URIs and to implement services at fixed URIs on the Baltic server. Instead of including these directly in Baltic scripts, we declare them together with their procedure types as part of the metadata used to generate the E*m* module.

For example, the E*m* module includes the following typed function to give access to a remote payment service called Payment1. The URI itself is declared in metadata.

val importPayment1 : unit → tPayment

Similarly, E*m* includes a function for exporting a service procedure on an externally addressable port on the Baltic server. The actual port is declared in metadata.

val exportOrderEntry : tOrderEntry → tOrderEntry

Since VMs are not directly attached to the external network, both these functions create intermediaries on the Baltic server that relay between the internal procedures and the external network.

1.5 Contributions

We advocate that operations logic be scripted with respect to explicit *resource metadata* that describes the types of the resources—disk images, and imported and exported services.

The main benefits of our approach, compared to the alternative, low-level scripting languages [Wolf and Halter, 2005], are the following.

- Baltic abstracts from networking details and automatically links together the procedures imported and exported by machines and intermediaries.
- Baltic catches construction errors by typechecking, rather than sometime during run time. For example, if server metadata stipulates a dependency on a service type, such as IPayment, Baltic never supplies a procedure with another type.

The main technical contributions of the paper are the following.

- The design and implementation of service combinators for compositional description of server farms and their operations logic.
- The idea that disk images should be seen as functions, with type signatures generated from service-oriented metadata, such as WSDL.
- A formal semantics of these combinators by an encoding in a concurrent λ-calculus, with partitions representing VMs. Via a type preservation result for our λ-calculus, we obtain type soundness for programs running against our API.

In future work, we intend to address some of the limitations in our present implementation. Our implementation does not consider security (we are essentially trusting the code on disk images), control of multiple VMMs for performance and fault tolerance, and persisting state (any transient changes to disk images are discarded). Intermediaries are limited to SOAP request/responses and do not maintain SOAP-level sessions. We only support SOAP services, not arbitrary webpages, nor database connections. On the basis of the formal semantics developed in this paper, in future work we intend to develop techniques for reasoning about operations logic expressed using our combinators.

1.6 Contents of the Paper

Section 2 introduces service combinators by example. Section 3 describes the overall architecture of our implementation. Section 4 presents the partitioned λ-calculus, a fragment of ML extended with a concurrency API. Section 5 describes a formal semantics of our service combinators in the partitioned λ-calculus. By typechecking, we establish our main theorem, that messages sent within the server farm are always of the correct form. Section 6 discusses related work and concludes.

Appendix A is an additional, extended example of a Baltic script. Appendix B describes our metadata format. Appendix C describes a tool, that given service-oriented metadata *m*, generates the typed interface $\mathsf{E}m$ to server roles and external procedures. Appendix D details the syntax, type system, and semantics of the partitioned λ-calculus.

2 Service Combinators by Example

We implement several variations of the enterprise order processing (EOP) application introduced in the previous section. Each example creates a configuration of VMs and SOAP intermediaries. The SOAP intermediaries are objects in a process, called the *Baltic Server*, running on the same physical machine as the VMM that hosts the VMs.

2.1 Generating a Typed Interface to Resources

The resources available to an application consist of the following: (1) disk images for each server role; (2) addresses of external procedures that the application can use; (3) addresses of procedures published by the application. We propose a metadata format

Basic Interface: B.mli

```
type vm
type vm_snapshot
type event = VM_Crash | VM_Shutdown |
             VM_Overload | VM_Underload
type (α,β) proc
type (α,β) procref
val call : (α,β) proc → α → β
val eOr : (α,β) proc → (α,β) proc → (α,β) proc
val ePar : (α,β) proc → (α,β) proc → (α,β) proc
val eRef : (α,β) proc → ((α,β) proc ×
(α,β) procref)
val eRefUpdate : (α,β) procref → (α,β) proc → unit
val eVM : vm → (event → unit) → unit
val eDelete: (α,β) proc → unit
val shutdownVM: vm → unit
val snapshotVM : vm → vm_snapshot
val restoreVM : vm_snapshot → unit
```

Environment Interface: Em.mli

```
type Payment
type Order
type tPayment=(Payment,string) proc
type tOrderEntry=(Order,string) proc
type tOrderProcessing=
                    (Order,unit) proc
val createOrderEntryRole:
  tPayment → tOrderProcessing →
                  (vm × tOrderEntry)
val createOrderProcessingRole :
      unit → (vm × tOrderProcessing)
val createPaymentRole :
      unit → (vm × tPayment)
val importPayment1: unit → tPayment
val importPayment2: unit → tPayment
val exportOrderEntry:
      tOrderEntry → tOrderEntry
```

such that metadata *m* describes these resources, together with their types, expressed in WSDL. We show below an excerpt from the metadata for our example application—our implementation uses an XML format, but in this paper we use ML syntax.

```
[VM {name = "OrderEntry"; disk = "OrderW2K3.vhd";
     inputs = [payment_ty; orderProc_ty]; outputs = [orderEntry_ty]}; ...
 Import {name = "Payment1"; binding = payment_ty;
         url = "http://creditagency1.com/CA/service.svc"}; ...
 Export {name = "OrderEntry"; binding = orderEntry_ty;
         url = "http://localhost:8080/OE/service.svc"}]
```

The first record specifies that the OrderEntry role is defined by the disk image in the file OrderW2K3.vhd. The role takes two services as input, described by payment_ty and orderProc_ty, which are the WSDL descriptions corresponding to the C# interfaces IPayment and IOrderProcessing, given earlier. The role exports a single binding described by orderEntry_ty, which corresponds to IOrderEntry.

The second and third records describe services imported and exported by the application; the records include the actual URIs as well as their WSDL descriptions.

The table above lists the Baltic API for this application. The Baltic API consists of a *basic interface*, B.mli, which is fixed, plus an *environment interface*, Em.mli, which gives access to the resources described by *m*. Given *m* we have a tool that compiles *m* to the interface Em.mli, and to a module Em−c.ml that contains the types and functions described in the interface. The types are ML representations of the request and response types in the WSDL descriptions of procedures. The functions provide typed access to the various resources. For compilation to succeed, the metadata must be *well-formed* in the sense that it satisfies certain syntactic constraints. Appendix C describes this tool in detail.

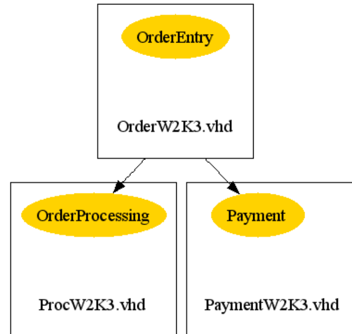

Figure 1: One VM depending on two others.

2.2 Example 1: Creating an Isolated VM Farm

Our first example is an isolated instance of the EOP case study. The three server roles are implemented by VMs that are isolated from the external environment, a configuration useful during development and testing.

The script below calls createPaymentRole and createOrderProcessingRole to boot VMs from the disk images of the order processing and payment roles. These calls return the procedures ePay and eProc exported by these roles. Since these roles import no procedures, the corresponding functions take no parameters. Finally, the third line boots a VM for the order entry role, using ePay and eProc as inputs.

```
let (vm1,ePay) = createPaymentRole ()
let (vm2,eProc) = createOrderProcessingRole ()
let (vm3,eEntry) = createOrderEntryRole ePay eProc
```

The following function, from B.mli, makes a call to a procedure. Given an (α,β) proc and a request of type α, it serializes the request into a SOAP message, sends it to the procedure, awaits and then deserializes the response, and returns the result as a value of type β. It is useful, for example, for running tests.

```
val call: (α,β) proc → α → β
```

For example, to test the order entry procedure eEntry, we invoke it using call:

```
let o:Order = makeOrder "Alice"
let result = call eEntry o
```

(The function makeOrder generates a default Order for user Alice.)

The Baltic state after running the script is shown in Figure 1. Each VM is a rectangle labelled with the name of the disk image. The ellipses within a VM show its exported procedures. The arrows from a VM show its imported procedures.

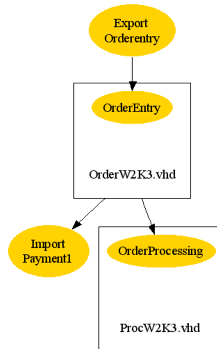

Figure 2: Importing and exporting procedures.

2.3 Example 2: Importing and Exporting Services

Our next example illustrates a simple deployment. We publish an internal procedure as a public service on the Baltic Server. Rather than use a local payment service to authorize orders, we rely on a remote service.

The external addresses of the payment service and the published service are specified in the metadata m, and are named Payment1 and OrderEntry. These addresses correspond to the typed functions importPayment1 and exportOrderEntry in Em.mli.

The script below calls the function importPayment1 to create a SOAP forwarder on the Baltic Server, returning the internal procedure ePay. Any request sent to ePay is forwarded to the external URI Payment1. The call to the function exportOrderEntry with parameter eEntry creates another forwarder on the Baltic Server, listening at the local URI OrderEntry defined in the metadata. Any request sent to this URI is forwarded to the internal procedure eEntry. The Baltic state after running the script below is in Figure 2.

```
let ePay = importPayment1 ()
let (vm1,eProc) = createOrderProcessingRole ()
let (vm2,eEntry) = createOrderEntryRole ePay eProc
let eo = exportOrderEntry eEntry
```

2.4 Example 3: Par and Or Intermediaries

Servers may sometimes be overloaded, say during office hours, leading to increased latency and reduced reliability. Suppose there are two sites hosting the payment authorization service, at Payment1 and Payment2, and that they are distributed geographically so that when one location is in office hours, the other is not. We can improve the reliability of the OrderEntry service by using both payment services in parallel.

The call ePar ePay ePay' returns a procedure exported by a freshly created *Par intermediary*. Any request sent to this intermediary is forwarded to both ePay and ePay'; the first response received is returned, while the second is discarded. (The

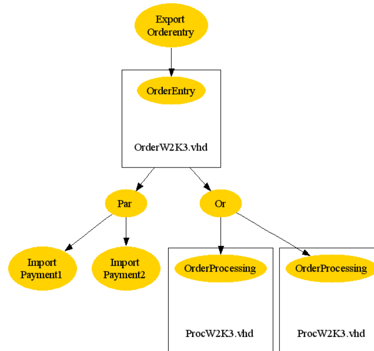

Figure 3: Creating Par and Or intermediaries.

ePar combinator is inspired by the service combinator $S_1 \mid S_2$ of Cardelli and Davies [1999].) The script below extends the previous example: it calls importPayment2 to create a second payment procedure ePay' that forwards requests to Payment2; it then uses ePar to parallelize access to ePay and ePay'.

Another use of parallelism is to "scale out" a role, by running multiple instances in parallel, together with some load balancing mechanism. The call eOr eProc eProc' returns a procedure exported by a freshly created *Or intermediary*, which acts as a load balancer. Any request sent to this intermediary is forwarded to either eProc or eProc', chosen according to some strategy. (For now, we use a random strategy, but we intend to enrich our API to allow explicit expression of the strategy.) The script below calls createOrderProcessingRole to create a second VM eProc' in the order processing role; it then calls eOr to situate a load balancer in front of eProc and eProc'.

```
let ePay' = importPayment2 ()
let epar = ePar ePay ePay'
let (vm1',eProc') = createOrderProcessingRole ()
let eor = eOr eProc eProc'
let (vm2,eEntry) = createOrderEntryRole epar eor
```

Figure 3 shows the Baltic state after running this script. The Par and Or intermediaries are directly hosted as objects on the Baltic Server, so they appear outside the VM boxes.

The intermediary created by eOr switches between two procedures. More generally, the derived combinator orList creates a series of intermediaries to switch between a list of procedures; it relies on a fold operator to create a series of binary intermediaries.

```
let orList : (α,β) proc list → (α,β) proc = List.fold1_left eOr
```

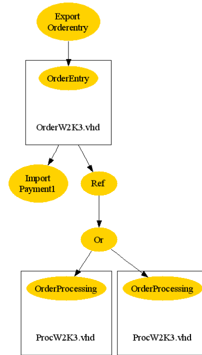

Figure 4: Creating Ref intermediaries.

2.5 Example 4: References, Updating References, and Events

The communication topologies of our previous examples are fixed. Our next example introduces the idea of changing the topology in response to an event.

The combinator eRef e returns a procedure exported by a freshly created *Ref intermediary*, together with an identifier r for the intermediary. Any request sent to this intermediary is forwarded to e. Moreover, the intermediary r is mutable; a call to the combinator eRefUpdate r e' updates r to forward subsequent requests to e'.

A VMM, such as Virtual Server, can detect various events during the execution of a VM, such as changes of VM state, the absence of a "heartbeat" (likely indicating a crash), and so on. Baltic provides a simple event handling mechanism, to allow a script to take action when an event is detected by the underlying VMM. The Baltic function eVM vm h associates the handler function h with the machine named vm. The handler function is of type event → unit where event is a datatype describing the event. (Our present implementation only handles a few kinds of events, but is extensible.)

To illustrate these operators, consider the two VM instances of the order processing role, vm1 and vm1', in the previous example. If one of these VMs crashes, we should reconfigure our application to avoid sending messages to the crashed VM. The code in the following script creates a Ref intermediary that forwards messages to the eor procedure. If either VM crashes, an event handler updates the Ref intermediary to forward messages to the procedure exported by the other VM.

```
let (eref,r) = eRef eor
let (vm2,eEntry) = createOrderEntryRole epar eref
let h other ev = match ev with VM_Crash → eRefUpdate r other | _ → ()
let _ = eVM vm1 (h eProc')
let _ = eVM vm1' (h eProc)
```

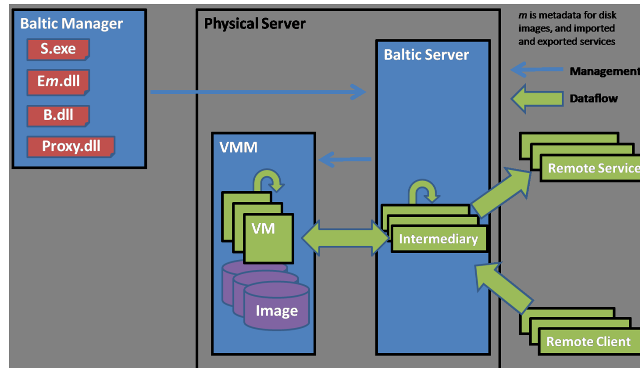

Figure 5: Baltic architecture

2.6 Example 5: Snapshots of VMs

When a VM has been booted from a disk image, the current state of the running VM consists of the memory image plus the current state of the virtual disk. Some VMMs, including Virtual Server, allow this state to be stored in disk files; typically, the memory image is directly stored in one file, while the current state of the virtual disk is efficiently represented by a *differencing disk*, which records the blocks that have changed since the machine started. We refer to this file system representation of a VM state as a *snapshot*. A snapshot can be saved, perhaps multiple times, and subsequently restored.

Baltic includes a simple facility for saving and restoring snapshots. If vm is the name of a running VM, snapshotVM vm creates a snapshot, and returns a value ss of type vm_snapshot that points to the saved files. Conversely, the call restoreVM ss discards the current state of vm, and replaces it by restoring the snapshot. (These combinators do not allow two snapshots of the same VM to run at once, a restriction imposed by the underlying VMM. However, the create*n*Role functions in E*m*.mli can be called repeatedly to create multiple instances of any one role.)

As a variation of the previous example, we record a snapshot of vm1 and vm1' just after booting and modify the event handler to restore the snapshot if the machine subsequently crashes. Snapshots allow faster recovery then rebooting.

```
let svm1 = snapshotVM vm1
let svm1' = snapshotVM vm1'
let h ss ev = match ev with VM_Crash → restoreVM ss | _ → ()
let _ = eVM vm1 (h svm1)
let _ = eVM vm1' (h svm1')
```

3 An Implementation of Service Combinators

3.1 Background: Virtual Server Farms

A *virtual server farm* consists of one or more physical servers that use operating system virtualization to host *virtual machines* (VMs). Each physical server runs a *Virtual Machine Monitor* (VMM) to manage its VMs. Common examples of VMMs are VMware ESX Server and Microsoft Virtual Server. The rest of this section focuses on Virtual Server; the functionality of other VMMs is similar.

Each VM boots off a disk image stored as a *Virtual Hard Disk* (VHD) file. VHD files contain an entire file system and can be mounted as secondary disk drives on physical machines. A common usage is to prepare a pristine *root* VHD file with all the required software and then create multiple versions of it for use in different VMs. To avoid copying and to save disk space, such versions are usually stored as *differencing* VHD files that only record the changes made from the root VHD.

A VM may be connected to a *Virtual Network* (VN) through a *Virtual Network Adapter* (VNIC), enabling communication between VMs. Each VM provides a set of procedures, exposed as SOAP web services in our implementation. Such services accept SOAP-formatted request messages from a virtual network, perform some local processing, and return response messages to the sender. Some virtual networks may be bridged through to the physical network using the host physical server's network adapter, enabling machines on the physical network to communicate with VMs. For security, however, virtual networks are often isolated from the physical network. In such scenarios, VMs can communicate only with other VMs or with the host server through its *Loopback Adapter*; all messages to or from the physical network must be forwarded through carefully managed SOAP intermediaries on the physical server.

When using Virtual Server, some management functions are scriptable through a COM API. The state manipulated by this API forms a graph. Each node represents a virtual device such as a VM, VHD, VNIC, or VN; each edge represents a connection between devices, such as a VNIC being attached to a VN, or a VM being booted of a VHD. The API provides functions to create, read, write, and delete these nodes and edges. However, the Virtual Server API treats VMs as opaque; it provides no direct way to compose functionalities provided by different VMs, to reconfigure VMs after they have been started, or to track dependencies between VMs. This is the functionality that our service combinators seek to provide.

3.2 Baltic Architecture

Figure 5 depicts the Baltic architecture for managing a single physical server. A *remote client* is a consumer of a service located at an address on the physical server. A *remote service* is a service called by computations running on the physical server.

The *Baltic Server* is a process running on the physical server. It implements the procedures exported by the physical server, as well as the procedures associated with SOAP intermediaries. The VMM also runs on the physical server, under control of the Baltic Server. (Our VMM is Virtual Server; other VMMs implement management APIs similar to that of Virtual Server, and hence could also support the Baltic API.)

The VM disk images and other files, such as VM snapshots, used by the VMM are held on disks mounted on the physical server. The Baltic Server mediates all access between VMs and remote procedures, and exports functions for managing the VMM.

The *Baltic Manager* is an executable compiled from a Baltic script; it manages the Baltic Server (and hence the VMM) using remote procedure calls, and hence may run either on the physical server, or elsewhere.

The VMM hosts a virtual network to which each VM is attached. The virtual network is isolated from the external network. Hence, VMs can use TCP/IP over the virtual network to call procedures on other VMs directly. VMs may also use TCP/IP to call procedures hosted by intermediaries the Baltic Server, but cannot directly call remote services. Remote clients can also call procedures hosted by the Baltic Server, but not those hosted in VMs. Intermediaries hosted by the Baltic Server can call all three kinds of procedure: other intermediaries, remote services, or services exported by a VM. As the diagram illustrates, Baltic scripts create and interconnect VMs and intermediaries, but are not involved in the actual SOAP dataflow.

Baltic consists of the Baltic API, the Baltic client, and the Baltic service. The Baltic API consists of types and functions. We provide dual *symbolic* and *concrete* implementations of the API. The former implementation relies on the implementation of the interface E*m*.mli that simulates what the Baltic service performs. For example, the name of a procedure is usually the address of the procedure, while the symbolic implementation names procedures with symbolic names. The Baltic client is a proxy that communicates with the Baltic service in the concrete implementation. Both the service and the client are implemented using the .NET framework of the WCF.

3.3 Basic Module: B−c.ml

The module B−c.ml is our concrete implementation of the basic interface B.mli. It implements the types in B.mli as follows. A value of type vm is a VM identifier, as defined by the VMM. A value of type vm_snapshot is a group of files implementing a VM snapshot, together with a VM identifier. A value of type (α,β) proc is a SOAP address, consisting of a URI and a SOAP action, and located either on a VM or the physical server. The API generates a value of this type only when there is a web service of the appropriate type listening at the address. A value of type (α,β) procref is the SOAP address of a mutable intermediary on the Baltic Server.

The functions in B−c.ml are implemented as remote procedure calls, via proxy code (proxy.dll), to the Baltic Server. They create and manipulate intermediaries on the physical server as described in Section 2. For example, eOr takes two procedures e1 and e2 as arguments and proceeds as follows: (1) using proxy.dll, eOr calls a function on the Baltic server; (2) the Baltic server starts a new intermediary at a fresh address e on the physical server; the intermediary listens for requests on e and forwards them either to e1 or to e2 (based on a coin-toss); (3) eOr returns the procedure address e.

The implementation of ePar e1 e2 is similar, except that the intermediary forwards requests to both e1 and e2. The function eRef generates an intermediary that forwards all its requests to an address stored in a mutable variable; eRefUpdate updates this variable with a new address.

The Virtual Server API triggers events when the state of a VM changes; the Baltic server listens for these events and looks for registered event handlers. A typical event captured by the server is VM_Crash, meaning that a VM has stopped responding. The function eVM registers an event handler for a particular VM at the Baltic server; when this event is triggered, the server makes a call back to the Baltic script, which then executes the handler code.

The functions snapshotVM and restoreVM are implemented using save and restore functions in the Virtual Server API, plus some basic file management.

3.4 Environment Module: E*m*−c.ml

This module is our automatically-generated concrete implementation of the environment interface, E*m*.mli, which enables access to the resources described in the metadata *m*. Each function in E*m*−c.ml is implemented using remote procedure calls, via proxy.dll, to the Baltic Server to create new VMs and intermediaries. For example, the function createOrderEntryRole takes two procedures as arguments; it then calls a function on the Baltic server that boots a new VM from the disk image file `"OrderW2K3.vhd"`, configures the VM with the addresses of the two input procedures, and returns the address of the new order entry procedure exported by the VM. Similarly, for the imported and exported procedures in *m*, E*m*−c.ml defines functions that start new intermediaries on the Baltic server to forward requests to the procedures. These intermediaries enable controlled communication between internal VMs and remote sites.

3.5 Assembling the Baltic Manager

Figure 6 shows how Generator is used together with conventional compilation to build a Baltic Manager executable (as in Figure 5) from an *m*-script S.ml. The service combinator implementation B−c.ml is compiled to B.dll; the environment implementaion E*m*−c.ml is compiled to E*m*.dll; and the two DLLs (along with proxy.dll) are linked to the Baltic script S.ml to create the Baltic manager S.exe.

4 The Partitioned λ-Calculus

As a basis for the semantics of Baltic in Section 5, this section defines a concurrent λ-calculus, with partitions for representing virtual machines. All source programs in our calculus are written within ML syntax; to allow our semantics to describe configurations arising at run time, we enhance this syntax with notations in the style of a process calculus, as explained below. The full syntax and semantics are in Appendix D.

A Functional Core. We begin with a λ-calculus that has algebraic types and pattern-matching. This is a core language that does not support the definition of polymorphic functions, although we allow some primitive operators to be polymorphic. Functions are written $\mu y.\lambda x.A$ where y is the optional function name used for recursive calls, and

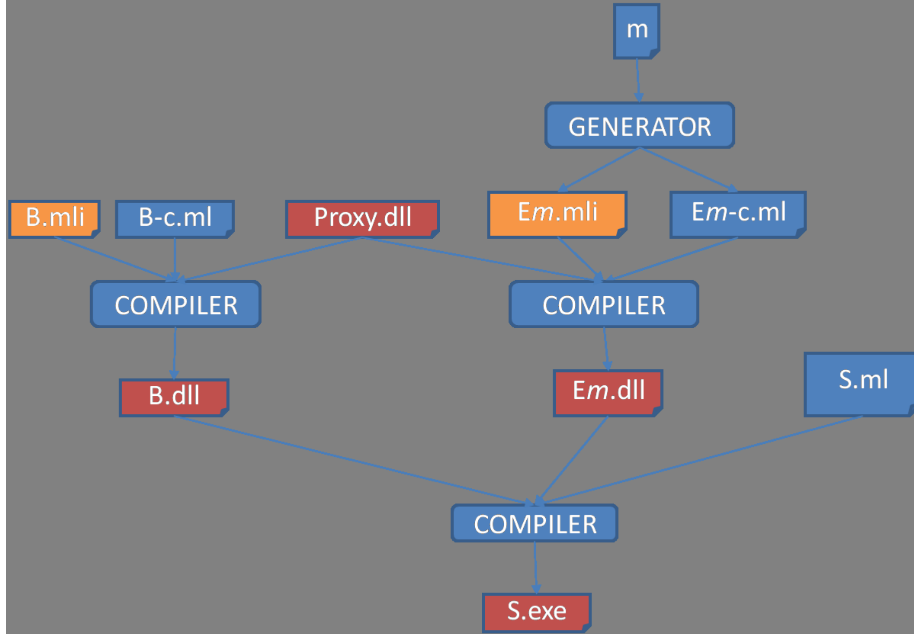

Figure 6: Building a Baltic Manager

x is the argument (both y and x are bound in A). Algebraic type definitions are written :

$$\textbf{type}\ (\alpha_1,\ldots,\alpha_n)F = (\mid f_i\ \textbf{of}\ T_{i1}\times\cdots\times T_{iw_i})^{i\in 1..m}$$

where $\alpha_1,\ldots,\alpha_n$ are type variables (which may occur in types on the right hand side); for any substitution $\sigma = \{U_1/\alpha_1\}\ldots\{U_n/\alpha_n\}$, each value constructor f_i takes arguments of types $T_{i1}\sigma,\ldots,T_{iw_i}\sigma$ and returns a value of type $(U_1,\ldots,U_n)F$. The language has standard constructs for applying functions and constructors, and a **match** construct for pattern-matching terms. It has a **let** construct for sequential composition (not for polymorphism), with an optional type annotation; in **let** $x : T = A$ **in** B, the annotation requires that the expression A must be of type T.

A program is a *module*, D, defined as a sequence of type and value definitions, with an *interface*, I or E, that exposes some types, abstracts other types, and declares types for accessible values.

Concurrency Operators. We extend our core λ-calculus with operators for communication, concurrency, and partitions. We introduce three new types: `name`, to represent fresh names, `chan`, to represent communication channels, and `part`, to represent partitioned computations. We define a set of pervasive operators p, each with a *type scheme* $T \to T'$ as follows:

Pervasive Operators: $p : T \rightarrow T'$

ref: $\alpha \rightarrow (\alpha)$ref	create fresh ref
get: (α)ref$\rightarrow \alpha$	get contents of ref
set: $(\alpha \times (\alpha)$ref$) \rightarrow$unit	set contents of ref
chan: string$\rightarrow (\alpha)$chan	create fresh channel
send: $((\alpha)$chan $\times \alpha) \rightarrow$unit	send message on channel
recv: (α)chan$\rightarrow \alpha$	receive message off channel
fork: (unit $\rightarrow$ unit)$\rightarrow$unit	start thread in parallel
name: string$\rightarrow$name	create fresh name
box: (unit $\rightarrow$ unit)$\rightarrow$part	create new partition
paste: (name $\times$ part)$\rightarrow$unit	paste named partition
cut: name$\rightarrow$part	cut named partition

The first three functions are ML's operators for manipulating references. The next four are standard operators for communication and concurrency as in various functional languages with concurrency [Holmström, 1983, Reppy, 1991]. The function chan generates a new communication channel; the string parameter is for debugging purposes—it may be used as part of the run time identity of the channel. A channel of type (α)chan may be used to communicate messages of type α; the functions send and recv enable asynchronous communication over such channels. The higher-order function fork takes a function f as parameter and forks off a new thread in parallel that executes f ().

The final four functions are novel; they represent the starting and stopping of *partitions*. Formally, a partition, $[A]$, is simply a paused expression A treated as a value; the expression A represents a snapshot of a component such as a virtual machine. The expression $a[A]$, where a is a name, represents a running, active partition. Code within A can communicate with code inside or outside the partition on shared communication channels. The name a is significant only for starting and stopping the partition. The function name creates a new partition name. The function box takes a function f and packages it as a partition [f ()] of type part; the expression within a partition remains dormant until it is started. A call paste$(a,[A])$ starts a running partition $a[A]$. Conversely, if there is such a partition, a call cut a stops it, and returns $[A]$ as a value. (We enhance our ML syntax with the the notations $[A]$ and $a[A]$ to express our formal semantics, but these notations cannot occur in source programs.)

A program in the partitioned λ-calculus may refer to these pervasive operators; in addition it may refer to types in the *pervasive environment*, E_p, which includes function types, tuples, and the types string, list, part, name and chan.

Semantics. Appendix D gives the dynamic semantics in the style of a process calculus [Milner, 1999], in terms of the following relations.

$A \equiv A'$	structural equivalence
$A \rightarrow A'$	reduction
$[\![D]\!](E,A) = (E',A')$	module semantics

The reduction relation defines rules for all the language constructs with special rules for three of the pervasive operators, recv, cut, and paste. The other pervasives can be derived in terms of the following constructs inspired by process calculi. The expression $M\langle N\rangle$ is a message N on channel M. The expression $A \upharpoonright B$ is the (asymmetric) parallel composition [Havelund and Larsen, 1993, Gordon and Hankin, 1998] of A and B; $A \upharpoonright B$ runs A and B in parallel, and returns the result of B, irrespective of whether A returns a result. The expression $(\nu a)A$ represents a fresh name a with scope A.

The semantics of a module is by translation to an expression: given a final environment E and expression A, $[\![D]\!](E,A)$ returns an environment E' that contains the type declarations in D, and an expression A' that represents the computation in D.

Intuitively, we require that messages sent and received by our programs are well-formed. Formally, we define the following syntactic notion of type safety in terms of type annotations on **let** expressions: every value assumed by a variable declared with algebraic type F is obtained from a constructor f_i of the type F. Constructor safety is enforced by typing, but is stated independently of the formal type system. We write $\varnothing$ for the empty environment, and $\mathscr{E}$ for an evaluation context.

Constructor Safety for Expressions and Modules:

Let μ be the type definition $\mu = (\textbf{type}\ (\alpha_1,\ldots,\alpha_n)F = (\mid f_i\ \textbf{of}\ T_{i1} \times \cdots \times T_{iw_i})^{i\in 1..m})$. An expression A is *constructor safe* for μ iff, whenever $A = \mathscr{E}[\textbf{let}\ x : T\ =\ M\ \textbf{in}\ B]$ where $T = (T_1,\ldots,T_n)F$, for some $\mathscr{E}$, x, $T_1,\ldots,T_n$, M, and B, then M takes the form $f_i(N_1,\ldots,N_{w_i})$ for some $i \in 1..m$.

A module D is *constructor safe* if and only if, if $[\![D]\!](\varnothing,()) = (E,A)$, for some E, A, and if $A \rightarrow^* A'$, then for all type definitions $\mu \in E$, A' is constructor safe for μ.

For example, consider the following module implementing a simple payment authorization procedure that listens for requests on channel c and only authorizes requests from Alice.

```
type Payment = P of string
let c = chan "network"
let AuthorizePayment () =
  let req:Payment = recv c in
    match req with
      P(s) → if s = "Alice" then "Approved" else "Rejected"
```

Constructor safety for this module guarantees that the **match** expression will never fail, since the type annotation on **let** ensures that req must take the form P(s).

Type system. To check constructor safety, we define a type system for the partitioned λ-calculus in terms of the following standard typing judgments.

Main Judgments of the Type System:

$E \vdash \diamond$	environment E is good
$E \vdash T$	type T is good
$E \vdash f : T_1,\ldots,T_n \rightarrow T$	term constructor f has an instance
$E \vdash A : T$	expression A returns type T

$E \vdash D \rightsquigarrow I$	module D implements interface I

The judgment $E \vdash A : T$ means that A is well-typed, and in particular that all type annotations on **let** constructs in A are satisfied. The rules of our type system are consistent with the usual ML typing rules, assuming the standard manoeuvre of transforming polymorphic definitions into multiple monomorphic duplicates. The only additional rules come from uses of our new pervasive operators; to allow expressions of the partitioned λ-calculus to be written in ML, and typechecked using the standard typechecker, we define an ML interface Pi.mliencoding the type schemes of these operators.

For our example above, the type system infers that c has type (Payment)chan, and checks that any req sent on c has type Payment.

Our first result is that the operational semantics preserves typing. Our second result is that a well-typed whole program D (that is, a module that is well-typed in the empty environment) is constructor safe.

Theorem 1 *If $A \equiv A'$ or $A \to A'$ then $E \vdash A : T$ implies $E \vdash A' : T$.*

Theorem 2 *If $\varnothing \vdash D \rightsquigarrow \varnothing$ then D is constructor safe.*

The proofs of these theorems are in Appendix D.

5 A Formal Semantics of Service Combinators

In this section, we sketch the semantics of a Baltic script S.ml in terms of the partitioned λ-calculus. We model the behaviour of intermediaries and VMs as message-passing processes, while abstracting the details of their internal implementation. We model the Baltic API as functions that create and manage the processes representing intermediaries and VMs. We model a Baltic script directly as an expression in the partitioned λ-calculus, which makes calls to the functions in the Baltic API.

We define the semantics of the types and functions in the fixed part of the Baltic API B.mli by writing an abstract symbolic implementation B.ml in the partitioned λ-calculus. Then, for each resource-specific Em.mli, we write a symbolic implementation Em.ml, utilizing B.mli. Hence, the full program, consisting of B.ml, Em.ml, and S.ml, constitutes our symbolic model of the Baltic script. We use this symbolic model to state our safety theorem. Moreover, it is an executable specification; it can be compiled using an ML compiler and run symbolically for debugging.

5.1 Service Combinator Semantics: B.ml

For use in the symbolic model, we introduce three more functions in B.mli; the concrete implementation B−c.ml does not include these functions and neither our scripts nor Em−c.ml use them.

Basic API (continued): B.mli

```
val epFun : (α → β) → (α,β) proc
val epFail : unit → (α,β) proc
val vm : event chan → vm
val vmName : vm → name
```

The module B.ml implements this full interface B.mli.

Procedures. Semantically, an (α,β) proc is a function that transforms inputs of type α into outputs of type β:

```
type (α,β) proc = Fun of (α → β)
let epFun f = Fun f
let epFail () = Fun (fun x → failwith "always␣fails")
let call (Fun f) x = f x
```

The function epFun constructs a procedure given a function f; this function is only ever used in our symbolic model; in real applications a procedure is constructed by compiling code and installing it as a web service. The function call represents a service call: it calls a remote procedure.

Using this definition of procedures, we can now define the semantics of each service combinator. To define ePar, we first define an auxiliary function par that constructs the parallel composition of two functions:

```
let par f1 f2 x =
  let c = chan "resp" in
    fork (fun () → send c (f1 x));
    fork (fun () → send c (f2 x));
    recv c
let ePar (ep1) (ep2) =
  epFun (fun x → par (call ep1) (call ep2) x)
```

The function par takes two functions f1 and f2 and an argument x; it runs both functions in parallel on x and returns the first result. The combinator ePar extends par to procedures; given two procedures ep1 and ep2, it creates a new procedure that forwards all its calls to both ep1 and ep2, returning the first result.

To define eOr, we use another auxiliary function pick that non-deterministically picks one of its two arguments:

```
let pick x1 x2 = par (fun () → x1) (fun () → x2) ()
let eOr ep1 ep2 =
  epFun (fun x → call (pick ep1 ep2) x)
```

The combinator eOr takes two procedures, ep1 and ep2, and creates a new procedure that forwards each of its calls to either ep1 or ep2.

The type (α,β) procref represent mutable procedure references; we define it as an ML reference that points to a procedure:

```
type (α,β) procref = Ref of (α,β) proc ref
let eRef ep =
```

```
  let epr = ref ep in
    (epFun (fun x → call (!epr) x),Ref epr)
let eRefUpdate (Ref epr) (ep') =
  epr := ep'
```

The combinator eRef takes a procedure ep and creates a reference epr pointing to ep; it then creates a procedure that forwards all its calls to the current procedure pointed to by epr, and returns both the new procedure and the procedure reference Ref epr. The function eRefUpdate updates epr to point to a new procedure ep'.

VMs. The type vm represents a handle to a running VM; it is used to monitor events at the VM, or to snapshot it. We define it to be a pair consisting of a name (representing the VM name) and a channel on which the VM can send events:

```
type event = VM_Crash | VM_Shutdown |
             VM_Overload | VM_Underload
type vm = VM of name × event chan
let vm e = VM (name "vm", e)
let vmName (VM(nm,e)) = nm
let shutdownVM (VM(nm,c)) =
  let p = cut nm in
    send c (VM_Shutdown)
```

For simplicity, the only event we monitor is VM_Crash, it is triggered when the virtual machine manager detects that a VM has become fatally unresponsive. We define two functions that are used only in our symbolic model: the function vm constructs a new vm name and pairs it with a given event channel, while vmName extracts the name of a vm.

The combinator eVM attaches an event handler to a VM; it relies on an auxiliary function eventHandler, defined as follows:

```
let rec eventHandler (c:event chan) (h:event→ unit) =
  let e = recv c in
    fork (fun () → h e);
    eventHandler c h
let eVM (VM(n,c)) h = fork (fun () → eventHandler c h)
```

The recursive function eventHandler takes a channel c and a handler function; it repeatedly waits to receive an event on c, and when it does, it forks a thread that calls the handler. The combinator eVM takes a vm and a handler and starts an eventHandler listening on the VM's event channel.

The type vm_snapshot represents a dormant snapshot of a VM that can be restored; in our semantics, we represent dormant processes by partitions (part); hence we define a vm_snapshot in terms of a vm (the name of the VM) and a part:

```
type vm_snapshot = Snap of vm × part
let snapshotVM (VM (n,e)) =
  let p = cut n in
    paste n p;
    (Snap (VM (n,e),p))
```

```
let restoreVM (Snap (VM (n,e),p)) =
  let _ = cut n in
    paste n p
```

The combinator snapshotVM takes a vm with name n, it stops the partition with name n (using cut) to get the current state p of the partition, it restarts n with p (using paste), and returns a snapshot containing vm and p. The function restoreVM takes a snapshot of a VM n containing a stored partition p, stops the current partition with name n, replaces it with p, and restarts it. The cut operation within restoreVM acts as a lock preventing concurrent invocations from releasing multiple snapshots.

5.2 Environment Model: E*m*.ml

The module E*m*.ml implements the interface E*m*.mli and represents a formal symbolic model of the disk images and external procedures in an application with metadata *m*. First, as in E*m*−c.ml, the type definitions in S_{ty} are embedded in E*m*.ml. Then each function in E*m*.mli is defined in terms of the combinators in B.mli.

VM Roles. Recall that for every VM role *n*, E*m*.mli declares a function create*n*Role; in our model, this function starts a new partition running processes that implement the services within the VM. The model is free to choose any implementation for these processes; for instance, we model the Payment role as follows:

```
let callChan c m =
  let rr = chan "response" in
    send c (m,rr);
    recv rr
let createPaymentRole () : (vm× tPayment) =
  let payChan = chan "authorizePayment" in
  let part = box (fun () →
    let pay (P p:Payment) =
      if p.id = "Alice" then "Approved!" else "Rejected!" in
    let rec servChan c f : unit =
      let (m,r) = recv c in
        send r (f m);
        servChan c f in
      servChan payChan pay) in
  let n = vm (chan "evPay") in
    paste (vmName n) part;
    let ep = epFun (callChan payChan) in
      (n,ep)
```

This code relies on two dual auxiliary functions servChan and callChan: servChan takes a channel *c* and a function *f* and implements a server that listens on *c* for an input pair (m,r) and returns the result f(m) on the channel r; callChan takes a channel *c* and a message *m*, it creates a new response channel rr, sends the pair (m,rr) on the channel and waits for the result on rr. The function createPaymentRole first creates a channel payChan for its payment service to listen on. It then creates a partition part representing a single process that will run within the VM: when part is started, it runs

a service listening on payChan; the service executes a simple payment function pay that approves any request from Alice and rejects all other requests. Having created the partition, createPaymentRole then generates a new name n for the VM to be run, and starts the partition part at n; it then creates a procedure ep for the payment service and returns n and ep.

External References: X*m*.mli. To model imported and exported procedures, we define an interface X*m*.mli representing channels accessible to the external environment. For each imported procedure, we add a channel in this interface and assume that there is a remote server listening on this channel. For each exported procedure, we also add a channel so that remote clients may send messages on this channel. Hence, for our example, the interface is as follows:

```
val Payment1: (Payment × string chan) chan
val Payment2: (Payment × string chan) chan
val OrderEntry: (Order × string chan) chan
```

E*m*.ml defines these channels by calling chan:

```
let Payment1 = chan "Payment1"
let Payment2 = chan "Payment2"
let OrderEntry = chan "Orderentry"
```

Exported Procedures. Recall that for every exported procedure *n*, E*m*.mli declares a function export*n*. In our example, the exportOrderEntry function has the signature:

```
val exportOrderEntry: tOrderEntry → tOrderEntry
```

It is defined as follows:

```
let exportOrderEntry (ep:tOrderEntry) =
  fork (fun () → servChan OrderEntry (fun x → call ep (O x)));
  epFun (fun (O x) → callChan OrderEntry x)
```

The function exportOrderEntry takes a procedure ep as input and forks a process that forwards all requests sent to the channel OrderEntry (defined in X*m*.mli above) to ep; it then returns a new procedure that forwards all calls to the OrderEntry channel.

Imported Procedures. Similarly, for every imported procedure *n*, E*m*.mli declares a function import*n*. For example, the importPayment1 function has the signature:

```
val importPayment1: unit → tPayment
```

It is defined as follows:

```
let importPayment1 () : tPayment =
  epFun (fun (P x) → callChan Payment1 x)
```

It creates and returns a new procedure that forwards all its calls to the Payment1 channel (defined in X*m*.mli above).

5.3 Example

The semantics of the first example script in Section 2 is a pair (E,A), where E consists of the type definitions in B.ml and E*m*.ml, and A reduces in a few steps to an expression A' of the following form:

```
A′= (ν vm1)(ν vm2)(ν vm3)
    (ν payChan)(ν procChan)(ν orderChan)
      (vm1[servChan payChan paymentFunction] ↱
       vm2[servChan procChan processingFunction] ↱
       vm3[servChan orderChan (orderEntryFunction payChan procChan)] ↱
       let result = callChan orderChan o in ())
```

The expression A' is the parallel composition of three running partitions and a top-level expression. Each partition contains an expression representing a procedure implemented as a function listening on a global channel. The code relies on an auxiliary function servChan that takes as arguments a channel c and a function f; it awaits an input (and a response channel) on the channel c; it then invokes the function f on the input, and returns the output on the response channel. Hence, the partitions vm1, vm2 and vm3 implement the functions paymentFunction, processingFunction, and orderEntryFunction, which listen on the global channels payChan, procChan, and orderChan respectively. These functions may have arbitrary implementations that may spawn local processes and communicate with them on local channels. The function orderEntryFunction depends on the payment and order entry procedures; hence, it is parameterized by the channels payChan and procChan. Finally, A has a top-level expression that represents a call to the order entry procedure with an order value o; this expression relies on another auxiliary function callChan, which is a dual to servChan: it sends an input (and a fresh response channel) on a given channel and waits for the output.

```
let rec servChan c f : unit =
    let (m,r) = recv c in
        send r (f m);
        servChan c f
```

```
let callChan c m =
    let r = chan "response" in
        send c (m,r);
        recv r
```

The function servChan takes a channel c and a function f and implements a server that listens on c for an input pair (m,r) and returns the result f(m) on the channel r; callChan takes a channel c and a message m, it creates a new response channel rr, sends the pair (m,rr) on the channel and waits for the result on rr. (These two functions are the same as in Section 5.2.)

5.4 Assembling the Formal Model of a Baltic Script

We say that a module S.ml is an *m-script* if it is well-typed against B.mli, the fixed part of the Baltic API, and E*m*.mli, which provides access to the roles and external procedures specified in *m*. Formally, S.ml is an *m-script* if and only if:

$$\text{B.mli}, \text{E}m\text{.mli} \vdash \text{S.ml} \rightsquigarrow \varnothing$$

An *m*-script is the actual code runnable in our implementation. Typechecking establishes that S.ml is an *m*-script.

We say that a module E.ml is an *m-environment* if it is well-typed against B.mli and implements the types and functions in Em.mli; that is, it is an abstraction within the partitioned λ-calculus of the disk images and procedures specified in m, assuming that all the disk images respect the types of the procedures they import and export.

Formally, a module E.ml is an *m-environment* if and only if:

$$\mathsf{B.mli} \vdash \mathsf{E.ml} \leadsto \mathsf{X}m\mathsf{.mli}, \mathsf{E}m\mathsf{.mli}$$

E.ml defines the semantics of the imported and exported procedures in m in terms of some external channels defined in Xm.mli. For instance, the module Em.ml is an m-environment.

We say that a module O.ml is an *m-opponent* if it represents the world external to the Baltic server; that is, it is an abstraction within the partitioned λ-calculus of any remote clients and servers that respect the types of procedures specified in m, but that cannot directly use the interfaces B.mli and Em.mli. Formally, a module O.ml is an *m-opponent* if and only if:

$$\mathsf{X}m\mathsf{.mli} \vdash \mathsf{O.ml} \leadsto \varnothing$$

Theorem 3 (Constructor Safety for *m*-Scripts) *Suppose that m is well-formed metadata, and suppose that* S.ml *is any m-script. For all* E.ml *and* O.ml*, if* E.ml *is an m-environment, and if* O.ml *is an m-opponent, then module* $D =$ B.ml E.ml S.ml O.ml *is constructor safe.*

Proof We know $\varnothing \vdash \mathsf{B.ml} \leadsto \mathsf{B.mli}$ by typechecking. By the rules for composing modules in Appendix D, we can derive $\varnothing \vdash D \leadsto \varnothing$. By Theorem 2, D is constructor safe. □

Many interconnection errors, where servers or intermediaries are connected to the wrong procedures, lead to requests or responses of unexpected types. These errors may arise at initial configuration, or during subsequent reconnections after handling events. Our safety property guarantees, by static type-checking, that such errors cannot arise. Catching bugs early by typing is particularly useful in this application area, as the boot time for the VMs in even the simplest script is several minutes.

Trusting the disk images and the remote clients and servers to respect types may be reasonable in some situations, but it is questionable in general; we intend to develop techniques for Baltic to preserve type safety in the presence of untrustworthy parties.

Moreover, having obtained a complete executable specification (200 lines) for Baltic within the partitioned λ-calculus, we have a solid foundation for checking more sophisticated behavioural properties of scripts (such as the extended example in Appendix A). Still, while we can rely on our semantics to verify or to find bugs in the source code of Baltic scripts, we instead trust that the implementation of the Baltic API conforms to our semantics. We have not attempted to establish a formal relationship between our semantics and the implementation of our API on top of Virtual Server.

6 Related Work and Conclusions

Related Systems. Baltic is partly inspired by the service combinators for web computing of Cardelli and Davies [1999]; these allow the construction of concurrent client-side programs that mimic the manual use of a web browser to download documents from the web. In contrast, our combinators construct server-side programs that mimic the manual use of a VMM management console. Cardelli and Davies' combinators deal with factors such as varying download rates and failures; analogously, our combinators can set up handlers to respond to machine failures and other events.

Edinburgh LCFG [Anderson, 1994] is a well-developed system for managing the configuration of large numbers of Unix-like machines. It supports personal workstations as well as servers. Although it can be used to control the whole configuration of a machine, it can also be used to control just part of the configuration; the latter is particularly useful when first adopting LCFG. LCFG can configure software within disk images, a task not addressed by the Baltic operators. On the other hand, LCFG does not support intermediaries, and uses an untyped scripting language, while Baltic introduces the idea of representing server roles as typed functions.

Like many systems, including COM and CORBA, Baltic can be seen as a component model—in the terminology of Szyperski [2002], disk images are components, and intermediaries are connectors, while imported and exported services correspond to required and provided interfaces. Component models known as Architecture Description Languages (ADLs) express a whole system as a composition of pre-existing components, interconnected by explicit connectors. Darwin [Magee et al., 1995] is a prominent and early example. ADLs are whole programming languages, while Baltic is simply a programming model within an existing language. We are not aware of any ADLs using whole VMs as components; typically, components are smaller and less isolated than VMs.

HP SmartFrog [Goldsack et al., 2003] is a domain-specific language for describing server components, together with an implementation for activating and managing them. The original version worked with JVM-based components, but a more recent version uses operating system virtualization. Like LCFG, SmartFrog can describe the structure of server roles. SmartFrog has a type system, but is not service-oriented, in the sense of treating a server role as importing and exporting typed procedures. SmartFrog has a notion of a component *lifecycle*, represented by a finite state machine, with states initialized, running, terminated, and failed.

The Service Modeling Language (SML) [Arwe et al., 2007] is an XML format for describing the resources in a data centre, to support configuration, deployment, monitoring, and other tasks. SML is not itself a tool, but is intended as a common format to support interoperability between data centre management tools. Baltic uses an ad hoc XML format to describe role metadata; it could perhaps use SML instead.

One tool set intending to use SML (and already using a precursornamed SDM [Gibson, 2005]) is Microsoft's collection of Distributed System Designers (code name "Whitehorse") [Randell and Lhotka, 2004]. These are tools for the graphical design of systems and components. They support the design of individual server roles, using a graphical notation for importing into one role a procedure exported by another.

The AppLogic grid operating system [3TERA 2006] allows VM server farms to

be constructed and managed with a graphical editor. AppLogic grids are configurable using conventional scripting languages.

In the Singularity operating system, the sole means of inter-process communication is typed message-passing, enforced by language-based techniques [Fähndrich et al., 2006]. The type safety of Baltic server farms rests on the unchecked assumption that the SOAP messages sent by each role conform to the types declared in metadata. It would be interesting future work to use Baltic to control VMs running Singularity, and to use Singularity's mechanisms to check the assumptions made by Baltic.

HPorter [Huang et al., 2007] is another combinator library written in a functional language for combining and reconfiguring software components written in lower-level languages. HPorter is aimed at robotics applications, and manages pre-existing components written in C and C++, that communicate over TCP/IP sockets. HPorter is written in the pure functional language Haskell, and relies on Haskell's higher-order type theory [Hughes, 2000] to encapsulate imperative behaviour.

Like Baltic, PiDuce [Carpineti et al., 2006] is a language and implementation for building SOAP web services, with a formal semantics. Unlike Baltic, PiDuce expresses the behaviour of individual services directly, whereas Baltic relies on pre-existing disk images to implement individual services, and concentrates on management.

OSGi [2005] is a framework for managing networked services running within JVMs. It defines APIs for composing, installing, and stopping services without restarts.

Related Formalisms. We build both our actual implementation and our formal semantics using the technique of dual concrete and abstract implementations of interfaces; this technique was introduced by Bhargavan et al. [2006].

Our use of the λ-calculus with partitions as a semantics for combinations of virtual machines is a refinement of an earlier proposal, by Gordon [2005], that operating system virtualization can usefully be formalized using process calculi. There are other process calculi with operators to snapshot, restore, and duplicate running locations, including the Kell Calculus [Schmitt and Stefani, 2005, Lienhardt et al., 2007] and the Seal Calculus [Castagna et al., 2005]. A great many formalisms—see Lapadula et al. [2007], for example, and its bibliography—have been developed to represent orchestration, choreography, and dynamic discovery of web services. We do not address these advanced topics, and instead focus on management of pre-existing systems using simple request/response patterns of SOAP messaging; such systems are the common case in today's data centres.

Conclusions We have described a set of combinators for assembling networks of virtual machines that export SOAP services. A combination of type checking together with automated allocation of addresses prevents the troublesome configuration errors that may arise with alternatives, such as untyped scripts. There is a semantics based on a typed concurrent λ-calculus with partitions, and an implementation using Virtual Server with scripts in ML. Our test scripts manage pre-existing components from a sample multi-tier web application. (Scripts can easily be written in other .NET languages such as C# and VB.)

The benefits of using ML in the Baltic implementation are that its functional syntax

is convenient for service combinators, and that ML is a good notation for writing a π-calculus semantics. Still, although the idea of functional programming for scripting goes back to the Lisp Shell of Ellis [1980], it has yet to conquer the data centre. We expect it would be straightforward to script against the Baltic combinators using .NET languages other than F#, such as those, like PowerShell [Mic, 2006], aimed at systems administrators.

The semantics in the partitioned λ-calculus provides a typed, as well as succinct and precise, description of the Baltic design. Moreover, the semantics is easily executable, as it is expressed in ML, and can be used for symbolic debugging of scripts (much as in prior work [Bhargavan et al., 2006]).

Our work brings together two parallel trends in data centres: virtualization, and service-orientation. As disk images are increasingly service-oriented, there is value in a service-oriented programming model for virtualization. VMs are increasingly significant both for data centres and multi-core processors; our combinators form a basis for exploring the benefits of typed, declarative descriptions of applications built from VMs. Another advocate of a declarative approach to data centre programming is Murthy [2007]; he reports how various well-known declarative techniques can enable significant performance optimizations.

Our framework could easily be extended to include a richer set of operations for managing machines and for handling events. Our implementation is limited to SOAP services; it should be straightforward to include other types of web services such as URIs of web applications and database connections.

Our initial implementation experiments examine small-scale deployments on a single machine. In future, we intend to look at scalability and reliability in the context of larger examples. We also intend to look at security issues, especially how to contain the behaviour of untrusted disk images.

Although our implementation uses a VMM, the Baltic combinators are not tied to virtualization. We might implement Baltic on an array of physical machines configured to boot by downloading disk images from the network [Int, 1999]. Another implementation strategy would be to implement processing nodes as components loadable into a language-based VM such as the JVM or the CLR.

Finally, Lampson [2004] argues that the reusability of big components like operating systems and databases is one of the great successes of software. Although today such components are mostly reused manually, virtualization technology increasingly makes possible their automatic assembly into systems. An API like Baltic is a way to exploit this possibility in a typeful and declarative style.

Acknowledgments. Conversations with Úlfar Erlingsson, Philippa Gardner, Galen Hunt, Dave Langworthy, Alan Schmitt, Clemens Syzperski, and Paul Watson were useful. Questions raised during a presentation of a preliminary version of Baltic at the Semantics Lunch at the University of Cambridge Computer Laboratory were extremely useful; thanks are due in particular to Andy Pitts, Peter Sewell, and Viktor Vafeiadis. Galen Hunt suggested the term “operations logic”.

A An Extended Example

This appendix presents an extended example script that generalizes the second example in Section 2 by replicating the order processing and order entry roles. Depending on load, the script grows and shrinks the number of replicas; Figure 7 depicts the state where two instances of each role have been created.

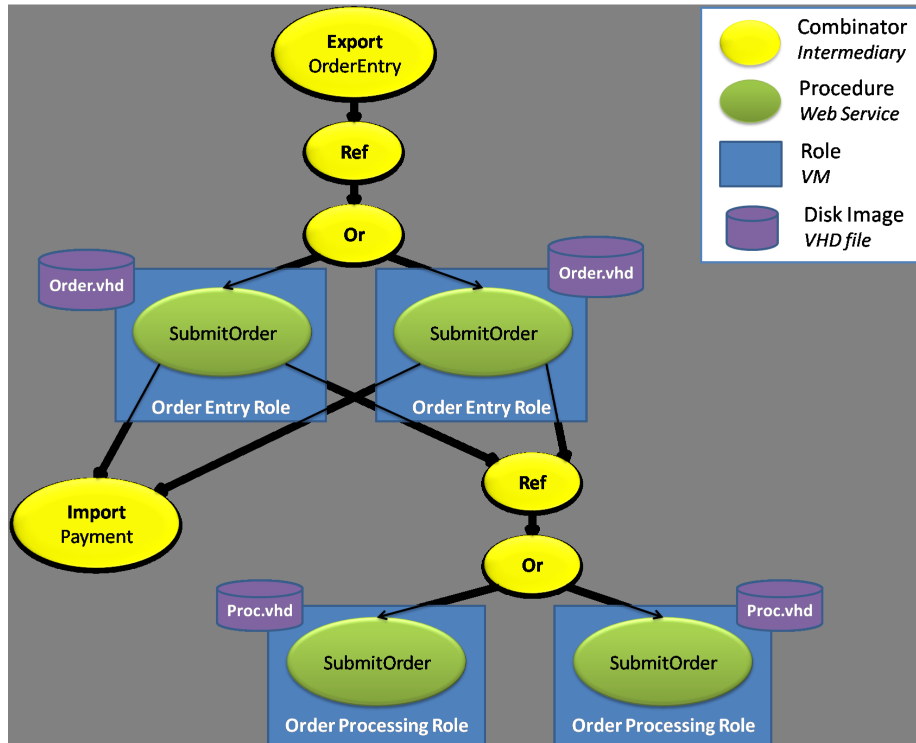

Figure 7: An example server farm built from Baltic combinators.

We first define a function replicatedVMProc that takes as its argument a function createVM that can create one instance of a VM; replicatedVMProc then manages a set of instances and balances the processing load between them. It maintains a list vms of active VMs; if a VM crashes or shuts down, it is deleted from the list; if a VM is overloaded, a new VM is created and added to the list; and if the system is underloaded, then a running VM is shut down and deleted.

```
let replicatedVMProc (createVM: unit →(vm ×(α,β) proc)): (α,β) proc =
  let vms = ref [] in
  let down : vm list ref = ref [] in
  let overloaded = ref false in
  let underloaded = ref false in
```

```
let handler vm ev =
  match ev with
      VM_Crash | VM_Shutdown → down := vm :: !down
    | VM_Underload → if !overloaded then () else underloaded := true
    | VM_Overload → overloaded := true; underloaded := false in

let rec monitor (r: (α,β) procref) () =
  vms := List.fold_left (fun l → fun a → List.remove_assoc a l) !vms !down;
  down := [];
  (match !vms with
      [] →
        let (vm,p) = createVM () in
          vms := [(vm,p)];
          eVM vm (handler vm)
    | (vm,p)::t →
        if !overloaded then
          (let (vm',p') = createVM () in
            vms := (vm',p')::(vm,p)::t;
            eVM vm' (handler vm');
            overloaded := false)
        else
          if !underloaded then
            (shutdownVM vm;
             vms := t;
             underloaded := false)
        else ());
  eRefUpdate r (orList (List.map snd !vms));
  monitor r () in

let (vm,p) = createVM () in
let (p',r') = eRef p in
  eVM vm (handler vm);
  vms := [(vm,p)];
  Pi.fork (monitor r');
  p'
```

The rest of the script creates a configuration of our running example: a payment procedure is imported; the order processing and order entry roles are replicated using replicatedVMProc; finally, the order entry procedure is exported.

```
let ePay = importPayment1 ()
let eProc = replicatedVMProc createOrderProcessingRole
let eEntry = replicatedVMProc (fun () → createOrderEntryRole ePay eProc)
let eo = exportOrderEntry eEntry
```

B Metadata for Services and Resources

First, we recall a standard metadata format for services. Second, we define our new metadata format for the resources used by a Baltic script.

We assume that imported and exported services are described in the WSDL metadata format [Christensen et al., 2001]. These WSDL files are generated automatically when the interface for the service is compiled, and are typically used to auto-generate proxy code for accessing the service.

A WSDL document describes a set of operations (procedures), and their input and output types. Types are typically expressed in XML Schema, though other formats are possible. We assume that the named types used within a WSDL document are captured as a set of abstract type declarations in an interface I_{ty}, and that these abstract types have some concrete implementation S_{ty} corresponding to the XML Schema definition. There are several tools that map XML Schema descriptions to programming language types. Formally, we write $\varnothing \vdash S_{ty} \rightsquigarrow I_{ty}$, meaning that the concrete types S_{ty} implement the abstract types I_{ty}.

In our example, I_{ty} consists of two abstract types:

```
type Payment
type Order
```

We use the following grammar as an abstraction of the operation descriptions in a WSDL document.

WSDL Metadata for Services:

T_{req}, T_{res}	type (see Section 4)
n, a, d, u	strings
$O ::= \{\mathsf{name} = n, \mathsf{action} = a,$ $\quad \mathsf{request_type} = T_{req}, \mathsf{response_type} = T_{res}\}$	operation
$Bd ::= \{\mathsf{name} = n, \mathsf{ops} = [O_1; \ldots; O_m]\}$	binding
$P ::= \{\mathsf{name} = n, \mathsf{url} = u, \mathsf{binding} = Bd\}$	port

We are using ML-style labelled records to represent the XML elements in a WSDL document. For example, our operations, bindings, and ports represent the WSDL elements named <operation>, <binding>, and <port>, respectively. For brevity, we sometimes elide the record labels when they are clear from context.

An *operation* $\{n, a, T_{req}, T_{res}\}$ describes a procedure, referred to as n; a SOAP request to this procedure should have a header with SOAP action a and a body encoding a value of type T_{req}, while a SOAP response from this procedure should have a body encoding a value of type T_{res}. The operation is well-formed against I_{ty} if the types T_{req} and T_{res} are well-formed against I_{ty}.

A *binding* $\{n, [O_1; \ldots; O_m]\}$ describes a service, referred to as n, with m procedures described by $O_1, \ldots, O_m$. The binding is well-formed against I_{ty} provided that the names of $O_1, \ldots, O_n$ are pairwise distinct and each O_i is well-formed against I_{ty}.

For example, the bindings for our case study are as follows. Each of these bindings is well-formed against the I_{ty} defined above.

```
let payment_ty:binding =
  {name = "Payment";
   ops = [({name = "AuthorizePayment";
          action = "http://EOP/IPayment/AuthorizePayment";
          request_type = "Payment";
```

```
          response_type = "string"}:operation)]}
let orderProc_ty:binding =
  {name = "OrderProcessing";
   ops = [({name = "SubmitOrder";
          action = "http://EOP/IOrderProcessing/SubmitOrder";
          request_type = "Order";
          response_type = "unit"}:operation)]}
let orderEntry_ty:binding =
  {name = "OrderEntry";
   ops = [({name = "SubmitOrder";
          action = "http://EOP/IOrderEntry/SubmitOrder";
          request_type = "Order";
          response_type = "string"}:operation)]}
```

The payment_ty binding describes a service, called Payment, that exposes a procedure AuthorizePayment, with a SOAP action http://EOP/IPayment/AuthorizePayment; the procedure takes as input an argument of type Payment and returns a string result.

A *port* $\{n, u, Bd\}$ describes a service, referred to as n; it is located at URI u and implements the procedures described in Bd.

For example, the following port describes an external service Payment1 that implements the payment_ty binding at URI http://creditagency1.com/CA/service.svc.

```
  {name = "Payment1";
   url = "http://creditagency1.com/CA/service.svc";
   binding = payment_ty};
```

Having defined the WSDL format for services, we introduce metadata for a complete Baltic application.

Metadata for Resources:

$r ::=$	resource
$\quad$ VM$\{$name $= n,$ disk $= d,$ inputs $= [Bd^1_{in}; \ldots; Bd^n_{in}],$ outputs $= [Bd^1_{out}; \ldots; Bd^m_{out}]\}$	virtual machine
$\quad$ Import P	imported service
$\quad$ Export P	exported service
$m ::= (I_{ty}, S_{ty}, [r_1; \ldots; r_n])$	metadata

For our case study, the following resource list rs describes all the resources available to Baltic scripts.

```
let rs:resources =
  [VM {name = "OrderEntry"; disk = "OrderW2K3.vhd";
       inputs = [payment_ty; orderProc_ty];
       outputs = [orderEntry_ty]};
   VM {name = "OrderProc"; disk = "ProcW2K3.vhd";
       inputs = [];
       outputs = [orderProc_ty]};
   VM {name = "Payment"; disk = "PaymentW2K3.vhd";
```

```
    inputs = [];
    outputs = [payment_ty]};
  Import {name = "Payment1";
          url = "http://creditagency1.com/CA/service.svc";
          binding = payment_ty};
  Import {name = "Payment2";
          url = "http://creditagency2.com/CA/service.svc";
          binding = payment_ty};
  Export {name = "OrderEntry";
          url = "http://localhost:8080/OE/service.svc";
          binding = orderEntry_ty}]
```

Recall I_{ty} from the previous section. Let S_{ty} be some implementation of the two abstract types in I_{ty}. Hence $m = (I_{ty}, S_{ty}, \mathsf{rs})$ is the metadata for our examples.

Each VM record defines a role in terms of a VM name, a disk image file accessible from the Baltic server, a list of imported bindings, and a list of exported bindings. For example, the OrderEntry role is defined by the file OrderW2K3.vhd, which holds a disk image; it takes two bindings as input, described by payment_ty and orderProc_ty, and exports a single binding described by orderEntry_ty. This metadata is compiled from an XML file config.xml that must be at the root directory of each disk image (OrderW2K3.vhd in this case). The file config.xml can be created using a text editor; it is a short file containing the file names of the WSDL descriptions of services imported or exported by the image. In turn, the WSDL files would typically be generated by the development environment used to write the code in the image.

Each Import record defines an external service port that can be used by a Baltic script. For instance, the Payment1 port at http://creditagency1.com/CA/service.svc implements the binding payment_ty.

Each Export record defines an internal service port that we wish to make available externally. Here, the port OrderEntry implements the binding orderEntry_ty and is exported at the URI http://localhost:8080/OE/service.svc.

C Metadata Compiler and Baltic Scripts

This section describes our metadata compiler. This tool, named Generator, collects metadata m describing the resources available to a Baltic script and compiles it to the following ML files:

- Em.mli: a typed interface for use in scripts; and
- Em−c.ml: a module implementing Em.mli.

This section describes the source metadata and its compilation to the interface Em.mli. The module Em−c.ml relies on the Baltic implementation and is described in Section 3.

C.1 Generating the Interface: Em.mli

Let $m = (S_{ty}, I_{ty}, [r_1; \ldots; r_n])$. Informally, E$m$.mli provides a functional interface through which an application can access the resources described in m. Generator creates Em.mli

as follows.

- It embeds I_{ty} at the beginning of Em.mli
- For every binding $Bd = \{n, [O_1; \ldots; O_n]\}$ occurring in $[r_1; \ldots; r_n]$, where each $O_i = \{n^i, a^i, T^i_{req}, T^i_{res}\}$, it defines a type t$n$:
  type tn= (T^1_{req}, T^1_{res}) proc $\times \ldots \times (T^n_{req}, T^n_{res})$ proc
- For every $r_i =$ VM$\{n, d, [Bd^1_{in}; \ldots; Bd^n_{in}], [Bd^1_{out}; \ldots; Bd^m_{out}]\}$, where each $Bd^i_{in} = \{n^i_{in}, \ldots\}$ and each $Bd^j_{out} = \{n^j_{out}, \ldots\}$, it declares a function create$n$Role:
  val create$n$Role: t$n^1_{in} \rightarrow \ldots \rightarrow$ t$n^n_{in} \rightarrow$ (vm $\times$ (t$n^1_{out} \times \ldots \times$ tn^m_{out}))
- For each $r_i =$ Import$\{n, u, Bd\}$, where $Bd = \{n_{Bd}, \ldots\}$, it declares a function import$n$:
  val import$n$: unit $\rightarrow$ tn_{Bd}
- For each $r_i =$ Export$\{n, u, Bd\}$, where $Bd = \{n_{Bd}, \ldots\}$, it declares a function export$n$:
  val export$n$: t$n_{Bd} \rightarrow$ unit

Hence, given the metadata m for our example application, Generator creates the Em.mli file shown in Figure 2.1. Given simple well-formedness constraints on the metadata, we can show that the generated interface is well-formed; we omit the details.

C.2 Generating Em−c.ml

The module Em−c.ml is our concrete implementation of the environment interface Em.mli. It is automatically generated by the metadata compiler, Generator, and is implemented using remote procedure calls, through proxy.dll, to the Baltic Server.

Let $m = (S_{ty}, I_{ty}, [r_1; \ldots; r_n])$; Generator creates the module Em−c.ml as follows:

- It embeds S_{ty} at the beginning of Em−c.ml
- For every binding $Bd = \{n, \text{ops} = [O_1; \ldots; O_n]\}$ occurring in $[r_1; \ldots; r_n]$, it generates a type definition for tn as in Em.mli
- For every $r_i =$ VM$\{n, d, [Bd^1_{in}; \ldots; Bd^n_{in}], [Bd^1_{out}; \ldots; Bd^m_{out}]\}$, where $Bd^i_{in} = \{n^i_{in}, \ldots\}$ and $Bd^j_{out} = \{n^j_{out}, \ldots\}$, it generates a function create$n$Role that takes n services, (x$_1$:tn^1_{in}),..., (x$_n$:tn^n_{in}), as arguments, and calls a function on the Baltic server with the disk image file name d, and x$_1$...x$_n$. Using the Virtual Server API, the Baltic server:
  (1) creates a differencing disk image from d;
  (2) boots a new VM, with fresh name n from this disk image and configures it with the input services x$_1$...x$_n$; and
  (3) returns n and the services (y$_1$:tn^1_{out}), ..., (y$_m$:tn^m_{out}) that the VM implements.
  create$n$Role returns the pair (n,(y$_1$,...,y$_m$)).

- For each $r_i = \mathsf{Import}\{n, u, Bd\}$, where $Bd = \{n_{Bd}, \ldots\}$, it generates a function $\mathsf{import}n$ that takes no arguments and calls a function on the Baltic server with the URI u. The Baltic server creates a new SOAP intermediary, at a fresh address e on the physical server; the intermediary forwards all calls made to e to the external address u. The function $\mathsf{import}n$ then returns the service $\mathsf{y}{:}\mathsf{t}n_{Bd}$ implemented by the intermediary.

- For each $r_i = \mathsf{Export}\{n, u, Bd\}$, where $Bd = \{n_{Bd}, \ldots\}$, it generates a function $\mathsf{export}n$ that takes a service $\mathsf{x}{:}\mathsf{t}n_{Bd}$ and calls a function on the Baltic server with x and URI u. The Baltic server creates a new intermediary, at the address u on the physical server; the intermediary forwards all calls made to u to the internal procedures in x.

D The Partitioned λ-Calculus

We provide the formal definition of the partitioned λ-calculus, a call-by-value λ-calculus with primitives for communication, concurrency, and starting and stopping partitioned collections of threads.

- Section D.1 defines the syntax.
- Section D.2 defines the operational semantics of expressions, and provides a proof of type preservation (Theorem 1).
- Section D.3 defines the type system.
- Section D.4 defines the semantics of modules by translation to expressions, and provides a proof of constructor safety (Theorem 2).

D.1 Syntax

The syntax of the calculus is based on the following sorts of atomic identifiers. Names are as in the π-calculus: identifiers that may be freshly created at run time.

Metavariables for Atomic Identifiers:

α	type variable
F	type constructor
a, b, k	name
x, y, z	(value) variable
f	value constructor
p	primitive function (not a value)
$v ::= a \mid x$	name or variable

The syntax of types is a subset of ML, based on abstract and algebraic types. The type of functions is a particular abstract type.

Types and Environments:

$T, U ::=$	type
$\quad \alpha$	$\quad$ type variable
$\quad (T_1, \dots, T_n)F$	$\quad$ constructed type
$\Sigma ::= (\mid f_i\ \textbf{of}\ T_{i1} \times \cdots \times T_{iw_i})^{i \in 1..m}$	type algebra, $m > 0$
$\mu ::=$	environment entry
$\quad \alpha$	$\quad$ type variable
$\quad \textbf{type}\ (\alpha_1, \dots, \alpha_n)F$	$\quad$ abstract type
$\quad \textbf{type}\ (\alpha_1, \dots, \alpha_n)F = \Sigma$	$\quad$ algebraic type
$\quad v : T$	$\quad$ type assignment
$I, E ::= \mu_1, \dots, \mu_n$	environment, interface

Domain of an Environment:

$\mathrm{dom}((\mid f_i\ \textbf{of}\ T_{i1} \times \cdots \times T_{iw_i})^{i \in 1..m}) = \{f_1, \dots, f_m\}$
$\mathrm{dom}(\alpha) = \{\alpha\}$
$\mathrm{dom}(\textbf{type}\ (\alpha_1, \dots, \alpha_n)F = \{F/n\}$
$\mathrm{dom}(\textbf{type}\ (\alpha_1, \dots, \alpha_n)F = \Sigma) = \{F/n\} \cup \mathrm{dom}(\Sigma)$
$\mathrm{dom}(\textbf{val}\ v : T) = \{v\}$
$\mathrm{dom}(\mu_1, \dots, \mu_n) = \mathrm{dom}(\mu_1) \cup \cdots \cup \mathrm{dom}(\mu_n)$

In an abstract type **type** $(\alpha_1, \dots, \alpha_n)F$, the type variables $\alpha_1, \dots, \alpha_n$ are bound, with no scope. In an algebraic type **type** $(\alpha_1, \dots, \alpha_n)F = \Sigma$, the type variables $\alpha_1, \dots, \alpha_n$ are bound, with scope Σ. We often write F alone, as an abbreviation for the type $(T_1, \dots, T_n)F$ when $n = 0$. When writing example interfaces, we prefix type assignments with the **val** keyword, as in **val** $v : T$, for example. We write $\varnothing$ for the empty environment.

The syntax of values and expressions is a subset of ML, plus additional syntax to represent configurations arising during executation: these additional notations are $\mu y.\lambda x.A$, $[A]$, $M\langle N\rangle$, $A \Rsh B$, $(\nu a)A$, and $a[A]$. The syntax of modules is a subset of ML.

Syntax of Values, Expressions, and Modules:

$M, N ::=$	value
$\quad v$	$\quad$ name or variable
$\quad f(M_1, \dots, M_n)$	$\quad$ constructed value
$\quad \mu y.\lambda x.A$	$\quad$ recursive function
$\quad [A]$	$\quad$ partition
$A, B ::=$	expression
$\quad M$	$\quad$ value
$\quad M\ N$	$\quad$ function application
$\quad \textbf{match}\ M\ \textbf{with}\ N \to A\ \textbf{else}\ B$	$\quad$ match
$\quad \textbf{let}\ x : T = A\ \textbf{in}\ B$	$\quad$ sequential composition
$\quad p\ M \quad (p \in \{\mathsf{recv}, \mathsf{cut}, \mathsf{paste}\})$	$\quad$ primitive operator
$\quad M\langle N\rangle$	$\quad$ message
$\quad A \Rsh B$	$\quad$ parallel composition

$(\nu a)A$	restriction
$a[A]$	running partition
$D ::=$	module
type $(\alpha_1,\ldots,\alpha_n)F = \Sigma$	algebraic type
let $x = A$	value declaration
$D_1\ D_2$	concatenation

Auxiliary Definitions: fv**,** fn

fv(T) is the set of free (type and value) variables in T.
fv(A) is the set of free (type and value) variables in A.
fn(A) is the set of free names in A.

Let a *pattern* be a value containing only variables and constructors. In a value $\mu y.\lambda x.A$, the variables y and x are bound, with scope A. In an expression **match** M **with** $N \to A$ **else** B, the term N is a pattern, and the variables fv(N) are bound in A. (We sometimes omit the type annotation T, as it can be inferred by the typechecker.) In an expression **let** $x : T = A$ **in** B, the variable x is bound, with scope B. In an expression $(\nu a)A$, the name a is bound, with scope A.

The entries in the environment, E_p, are always available during typechecking.

Pervasive Environment: E_p

type (α,β)func
type part
type name
type (α)id

type $(\alpha_1,\ldots,\alpha_n)$tuple$n$ = Tuple$n$ **of**$\alpha_1 \times \cdots \times \alpha_n$
(for each $n < b$, where b is an upper bound on the width of tuples.)
type bool = True | False
type int = Zero | Succ **of** int
type (α)list = Nil | Cons **of** $\alpha \times (\alpha)$list
type string = String **of** int list
type (α)chan = G **of** (α)id | L **of** (α)id

A value of the abstract type (T,U)func is a recursive function abstraction. A value of the abstract type part is a partition $[A]$. A value of the abstract type name is a name. A value of the abstract type (T)id is a name used as the basis of an (T)chan.

A channel c is any value of an algebraic type (T)chan, for any type T, and is of the form G(a) or L(a), where a is a (T)id. A channel of the form G(a) is *global*; one of the form L(a) is *local*. Global channels are global names for communication channels, while local channels represent cells in a heap local to a particular partition.

Channel: c

$c ::=$ G(a) | L(a)	global or local channel

We can interpret the following ML syntax within our core calculus. By appeal to these abbreviations we can interpret a large class of ML programs within the calculus.

Derived Forms:

$(T \to U) \triangleq (T,U)\mathsf{func}$ $\quad$ $\textbf{fun}\ x \to A \triangleq \mu y.\lambda x.A \quad (y \notin \mathrm{fv}(A))$
$(T_1 \times \cdots \times T_n) \triangleq (T_1,\ldots,T_n)\mathsf{tuple}n$ $\quad$ $(M_1,\ldots,M_n) \triangleq \mathsf{tuple}n(M_1,\ldots,M_n)$
$\mathsf{unit} \triangleq ()\mathsf{tuple}0$ $\quad$ $A;B \triangleq \textbf{let}\ x = A\ \textbf{in}\ B \quad (x \notin \mathrm{fv}(B))$
$(T)\mathsf{ref} \triangleq (T)\mathsf{chan}$ $\quad$ $\textbf{let rec}\ y\ x = A \triangleq \textbf{let}\ y = \mu y.\lambda x.A$

In addition, ML records can be seen as tuples. The syntactic forms $\mathsf{recv}\ M$, $\mathsf{cut}\ M$, and $\mathsf{paste}\ M$ are primitive. The other pervasive applications are definable as follows.

Derived Expressions for Pervasive Operators: $p\ M$

$\mathsf{ref}\ M \triangleq (\nu a)\textbf{let}\ r = \mathsf{L}(a)\ \textbf{in}\ (r\langle M\rangle \upharpoonright r)$ $\quad$ $\mathsf{send}\ (c,M) \triangleq c\langle M\rangle$
$\mathsf{get}\ M \triangleq \textbf{let}\ x = \mathsf{recv}\ M\ \textbf{in}\ (M\langle x\rangle \upharpoonright x)$ $\quad$ $\mathsf{fork}\ M \triangleq M\ () \upharpoonright ()$
$\mathsf{set}\ M\ N \triangleq \textbf{let}\ x = \mathsf{recv}\ M\ \textbf{in}\ (M\langle N\rangle \upharpoonright ())$ $\quad$ $\mathsf{name}\ M \triangleq (\nu a)a$
$\mathsf{chan}\ M \triangleq (\nu a)\mathsf{G}(a)$ $\quad$ $\mathsf{box}\ M \triangleq [M()]$

As in ML, we write $(M := N)$ for $\mathsf{set}\ M\ N$, and $!M$ for $\mathsf{get}\ M$.

D.2 Operational Semantics of Expressions

We present our operational semantics in the style of a process calculus, using the standard techniques of evaluation contexts [Felleisen and Friedman, 1986], and a reduction relation defined in terms of a structural equivalence relation [Berry and Boudol, 1990, Milner, 1999]. To support a reduction semantics of direct-style expressions, as opposed to continuation-passing processes in the π-calculus, we rely on a parallel composition $A \upharpoonright B$ that is not, in general, commutative [Gordon and Hankin, 1998].

Evaluation Contexts: $\mathscr{E}$

$\mathscr{E} ::= \{\} \mid (\textbf{let}\ x : T = \mathscr{E}\ \textbf{in}\ B) \mid (\mathscr{E} \upharpoonright B) \mid (B \upharpoonright \mathscr{E}) \mid (\nu a)\mathscr{E} \mid a[\mathscr{E}]$

$\mathrm{bn}(\mathscr{E})$ is the set of restricted names whose scope includes the hole $\{\}$ in $\mathscr{E}$.
A *partition-free* evaluation context is one derived without the clause for $a[\mathscr{E}]$.

As an exception to our general rule, we do not identify evaluation contexts up to renaming of bound names.

Structural Equivalence: $A \equiv A'$

$A \equiv A$ (Struct Refl)
$A' \equiv A \quad \text{if } A' \equiv A$ (Struct Symm)
$A \equiv A'' \quad \text{if } A \equiv A' \text{ and } A' \equiv A''$ (Struct Trans)
$\mathscr{E}\{A\} \equiv \mathscr{E}\{A'\} \quad \text{if } A \equiv A'$ (Struct $\mathscr{E}$)

$(A \upharpoonright A') \upharpoonright A'' \equiv A \upharpoonright (A' \upharpoonright A'')$ (Struct Assoc)

$() \upharpoonright A \equiv A$ (Struct Fork Unit)
$a[A] \upharpoonright () \equiv a[A]$ (Struct Part Unit)
$M\langle N\rangle \upharpoonright () \equiv M\langle N\rangle$ (Struct Msg Unit)

$\textbf{let}\ x:T\ =\ (A \upharpoonright A')\ \textbf{in}\ B \equiv A \upharpoonright \textbf{let}\ x:T\ =\ A'\ \textbf{in}\ B$ (Struct Let)
$(\nu a)\mathscr{E}\{B\} \equiv \mathscr{E}\{(\nu a)B\}$ if $a \notin \mathrm{fn}(\mathscr{E}) \cup \mathrm{bn}(\mathscr{E})$ (Struct Res)
$a[A] \upharpoonright \mathscr{E}\{B\} \equiv \mathscr{E}\{a[A] \upharpoonright B\}$ if $\mathrm{fn}(a[A]) \cap \mathrm{bn}(\mathscr{E}) = \varnothing$ (Struct Part)
$M\langle N\rangle \upharpoonright \mathscr{E}\{B\} \equiv \mathscr{E}\{M\langle N\rangle \upharpoonright B\}$ if $\mathrm{fn}(M\langle N\rangle) \cap \mathrm{bn}(\mathscr{E}) = \varnothing$ (Struct Msg)
and either M is a global channel or $\mathscr{E}$ is partition-free

The rule (Struct Part) allows us to flatten hierarchies of running partitions; for example, we can derive $a[A] \upharpoonright b[B] \equiv b[a[A] \upharpoonright B]$ by taking $\mathscr{E} = b[\{\}]$. More generally, we can place any expression in a normal form defined as follows.

Normal Forms:

$\mathbf{G} ::= M_1\langle N_1\rangle \upharpoonright \ldots \upharpoonright M_n\langle N_n\rangle \upharpoonright ()$ ($n \geq 0$, each M_i is a global channel)
$\mathbf{L} ::= M_1\langle N_1\rangle \upharpoonright \ldots \upharpoonright M_n\langle N_n\rangle \upharpoonright ()$ ($n \geq 0$, each M_i is not a global channel)
$th ::= M \mid (M\ N) \mid (\textbf{match}\ M\ \textbf{with}\ N \to A\ \textbf{else}\ B) \mid (p\ M) \mid (\textbf{let}\ x:T\ =\ th\ \textbf{in}\ B)$
$\mathbf{T} ::= () \upharpoonright th_1 \upharpoonright \ldots \upharpoonright th_n$ ($n \geq 0$)
$\mathbf{P} ::= a[\mathbf{L} \upharpoonright \mathbf{T}]$
$\mathbf{N} ::= (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \upharpoonright \mathbf{L} \upharpoonright \mathbf{P}_1 \upharpoonright \ldots \upharpoonright \mathbf{P}_m \upharpoonright \mathbf{T})$ ($m,n \geq 0$)

Proposition 1 (Normal Form) *For every expression A, there is a normal expression $\mathbf{N}$ such that $A \equiv \mathbf{N}$.*

Proof The proof is by structural induction on A.

- In case A is one of M, $M\ N$, or $p\ M$ where $p \in \{\mathsf{recv}, \mathsf{cut}, \mathsf{paste}\}$, then A is a thread th, and a thread is a normal expression.
- In case $A = M\langle N\rangle$ let $\mathbf{M} = M\langle N\rangle \upharpoonright ()$ and $\mathbf{T} = ()$, and we have $A \equiv \mathbf{M} \upharpoonright \mathbf{T}$ by (Struct Msg Unit), (Struct Fork Unit), and (Struct Assoc).
- In case $A = a[A_1]$, by induction hypothesis there is $\mathbf{N}_1$ such that $A_1 \equiv \mathbf{N}_1$. Suppose that $\mathbf{N}_1 = (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \upharpoonright \mathbf{L} \upharpoonright \mathbf{P}_1 \upharpoonright \ldots \upharpoonright \mathbf{P}_m \upharpoonright \mathbf{T})$. We may assume that the bound names $a_1, \ldots, a_n$ are distinct from a. By (Struct $\mathscr{E}$), (Struct Res) (with $a \notin \{a_1, \ldots, a_n\}$), (Struct Msg), (Struct Part), (Struct Part Unit), we calculate a normal form for A as follows:

$$\begin{aligned}
A &\equiv a[\mathbf{N}_1] \\
&= a[(\nu a_1)\ldots(\nu a_n)(\mathbf{G} \upharpoonright \mathbf{L} \upharpoonright \mathbf{P}_1 \upharpoonright \ldots \upharpoonright \mathbf{P}_m \upharpoonright \mathbf{T})] \\
&\equiv (\nu a_1)\ldots(\nu a_n)a[\mathbf{G} \upharpoonright \mathbf{L} \upharpoonright \mathbf{P}_1 \upharpoonright \ldots \upharpoonright \mathbf{P}_m \upharpoonright \mathbf{T}] \\
&\equiv (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \upharpoonright a[\mathbf{L} \upharpoonright \mathbf{P}_1 \upharpoonright \ldots \upharpoonright \mathbf{P}_m \upharpoonright \mathbf{T}]) \\
&\equiv (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \upharpoonright \mathbf{P}_1 \upharpoonright \ldots \upharpoonright \mathbf{P}_m \upharpoonright a[\mathbf{L} \upharpoonright \mathbf{T}]) \\
&\equiv (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \upharpoonright \mathbf{P}_1 \upharpoonright \ldots \upharpoonright \mathbf{P}_m \upharpoonright a[\mathbf{L} \upharpoonright \mathbf{T}] \upharpoonright ())
\end{aligned}$$

- In case $A = (\textbf{let } x : T = A_1 \textbf{ in } A_2)$, by induction hypothesis there is $\mathbf{N}_1$ such that $A_1 \equiv \mathbf{N}_1$. Suppose that $\mathbf{N}_1 = (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \Rsh \mathbf{L} \Rsh \mathbf{P}_1 \Rsh \ldots \Rsh \mathbf{P}_m \Rsh \mathbf{T})$. By (Struct $\mathscr{E}$), (Struct Res), (Struct Msg), (Struct Part), we calculate as follows:

$$\begin{aligned}
A &\equiv \textbf{let } x : T = (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \Rsh \mathbf{L} \Rsh \mathbf{P}_1 \Rsh \ldots \Rsh \mathbf{P}_m \Rsh \mathbf{T}) \textbf{ in } A_2 \\
&\equiv (\nu a_1)\ldots(\nu a_n)\textbf{let } x : T = (\mathbf{G} \Rsh \mathbf{L} \Rsh \mathbf{P}_1 \Rsh \ldots \Rsh \mathbf{P}_m \Rsh \mathbf{T}) \textbf{ in } A_2 \\
&\equiv (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \Rsh \mathbf{L} \Rsh \textbf{let } x : T = (\mathbf{P}_1 \Rsh \ldots \Rsh \mathbf{P}_m \Rsh \mathbf{T}) \textbf{ in } A_2) \\
&\equiv (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \Rsh \mathbf{L} \Rsh \mathbf{P}_1 \Rsh \ldots \Rsh \mathbf{P}_m \Rsh \textbf{let } x : T = \mathbf{T} \textbf{ in } A_2)
\end{aligned}$$

  If $\mathbf{T} = ()$ we have found a normal form for A. Otherwise, suppose $\mathbf{T} = \mathbf{T}' \Rsh th$ for some $\mathbf{T}'$ and we continue as follows by appeal to (Struct Let).

$$\begin{aligned}
A &\equiv (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \Rsh \mathbf{L} \Rsh \mathbf{P}_1 \Rsh \ldots \Rsh \mathbf{P}_m \Rsh \textbf{let } x : T = (\mathbf{T}' \Rsh th) \textbf{ in } A_2) \\
&\equiv (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \Rsh \mathbf{L} \Rsh \mathbf{P}_1 \Rsh \ldots \Rsh \mathbf{P}_m \Rsh \mathbf{T}' \Rsh \textbf{let } x : T = th \textbf{ in } A_2)
\end{aligned}$$

- In case $A = (B \Rsh B')$, by induction hypothesis there are $\mathbf{N}$ and $\mathbf{N}'$ such that $B \equiv \mathbf{N}$ and $B' \equiv \mathbf{N}'$. Suppose that $\mathbf{N} = (\nu a_1)\ldots(\nu a_n)(\mathbf{G} \Rsh \mathbf{L} \Rsh \mathbf{P}_1 \Rsh \ldots \Rsh \mathbf{P}_m \Rsh \mathbf{T})$ and $\mathbf{N}' = (\nu a'_1)\ldots(\nu a'_{n'})(\mathbf{G}' \Rsh \mathbf{L}' \Rsh \mathbf{P}'_1 \Rsh \ldots \Rsh \mathbf{P}'_{m'} \Rsh \mathbf{T}')$. Let $\vec{a} = a_1, \ldots, a_n, a'_1, \ldots, a'_{n'}$. We may assume all the bound names $\vec{a}$ are distinct. By (Struct $\mathscr{E}$), (Struct Res), (Struct Msg), (Struct Part), we calculate a normal form for A as follows:

$$\begin{aligned}
A &\equiv (\nu\vec{a})((\mathbf{G} \Rsh \mathbf{L} \Rsh \mathbf{P}_1 \Rsh \ldots \Rsh \mathbf{P}_m \Rsh \mathbf{T}) \Rsh (\mathbf{G}' \Rsh \mathbf{L}' \Rsh \mathbf{P}'_1 \Rsh \ldots \Rsh \mathbf{P}'_{m'} \Rsh \mathbf{T}')) \\
&\equiv (\nu\vec{a})(\mathbf{G} \Rsh \mathbf{G}' \Rsh \mathbf{L} \Rsh \mathbf{L}' \Rsh (\mathbf{P}_1 \Rsh \ldots \Rsh \mathbf{P}_m \Rsh \mathbf{T}) \Rsh (\mathbf{P}'_1 \Rsh \ldots \Rsh \mathbf{P}'_{m'} \Rsh \mathbf{T}')) \\
&\equiv (\nu\vec{a})(\mathbf{G} \Rsh \mathbf{G}' \Rsh \mathbf{L} \Rsh \mathbf{L}' \Rsh \mathbf{P}_1 \Rsh \ldots \Rsh \mathbf{P}_m \Rsh \mathbf{P}'_1 \Rsh \ldots \Rsh \mathbf{P}'_{m'} \Rsh (\mathbf{T} \Rsh \mathbf{T}'))
\end{aligned}$$

- In case $A = (\nu a)A_1$, by induction hypothesis there is $\mathbf{N}_1$ such that $A_1 \equiv \mathbf{N}_1$. By (Struct $\mathscr{E}$), we have $A \equiv (\nu a)\mathbf{N}_1$ and $(\nu a)\mathbf{N}_1$ is a normal expression.

In each case, we have found a normal expression structurally equivalent to A. ☐

We define a class of *copyable expressions*. Intuitively, a copyable expression contains no top-level restrictions, running partitions, or global channel. We do not intend these items to be duplicated by stopping and restarting a partition multiple times, so we only allow a running partition $a[A]$ to be stopped when A is copyable.

Copyable Expressions: A copyable

A is *copyable* if and only if for all $\mathscr{E}, a, T, B, M$:
$A \neq \mathscr{E}\{(\nu a)B\}$ and $A \neq \mathscr{E}\{a[B]\}$ and $A \neq \mathscr{E}\{\mathsf{G}(a)\langle M\rangle\}$

Lemma 1 *Expression A is copyable if and only if $A \equiv \mathbf{L} \Rsh \mathbf{T}$ for some $\mathbf{L}$ and $\mathbf{T}$.*

Reduction: $A \rightarrow A'$

$A \rightarrow A'$ if $A \equiv B, B \rightarrow B', B' \equiv A'$	(Red $\equiv$)
$\mathscr{E}\{A\} \rightarrow \mathscr{E}\{A'\}$ if $A \rightarrow A'$	(Red $\mathscr{E}$)

$\textbf{let}\ x : T\ = M\ \textbf{in}\ A \rightarrow A\{M/x\}$ (Red Let)
$(\mu y.\lambda x.A)\ N \rightarrow A\{\mu y.\lambda x.A/y\}\{N/x\}$ (Red Fun)
$(\textbf{match}\ M\ \textbf{with}\ N \rightarrow A\ \textbf{else}\ B) \rightarrow A\sigma$ if $M = N\sigma \wedge \mathrm{dom}(\sigma) = \mathrm{fv}(N)$ (Red Match)
$(\textbf{match}\ M\ \textbf{with}\ N \rightarrow A\ \textbf{else}\ B) \rightarrow B$ if $\neg\exists\sigma.M = N\sigma$ (Red Mismatch)

$c\langle M\rangle \upharpoonright \mathsf{recv}\ c \rightarrow M$ (Red Comm)
$a[A] \upharpoonright \mathsf{cut}\ a \rightarrow [A]$ if A copyable (Red Cut)
$\mathsf{paste}\ (a,[A]) \rightarrow a[A]$ (Red Paste)

For example, here is a reduction sequence showing a message exchange between two partitions.

```
paste(a,[send(g,42)]); cut a; paste a [let x = recv g in send(h,x)]; cut a
⟶ ⟶ a[send(g,42)] ↱ cut a; paste a [let x = recv g in send(h,x)]; cut a
≡ send(g,42) ↱ a[()] ↱ cut a; paste a [let x = recv g in send(h,x)]; cut a
⟶ send(g,42) ↱ paste a [let x = recv g in send(h,x)]; cut a
⟶ ⟶ send(g,42) ↱ a[let x = recv g in send(h,x)] ↱ cut a
≡ a[send(g,42) ↱ let x = recv g in send(h,x)] ↱ cut a
≡ a[let x = (send(g,42) ↱ recv g) in send(h,x)] ↱ cut a
⟶ a[let x = 42 in send(h,x)] ↱ cut a
⟶ a[send(h,42)] ↱ cut a
≡ send(h,42) ↱ a[()] ↱ cut a
⟶ send(h,42) ↱ [()]
```

D.3 Type System

We present a type system for the partitioned λ-calculus.

Main Judgments of the Type System:

$E \vdash \diamond$	environment E is good
$E \vdash T$	type T is good
$E \vdash f : T_1, \ldots, T_n \rightarrow T$	term constructor f has an instance
$E \vdash A : T$	expression A returns type T
$E \vdash D \rightsquigarrow I$	module D implements interface I

By a convention of Abadi and Cardelli [1996], if the indexing set I is empty, and $E \vdash \mathscr{J}_i$ is a judgment indexed by i, then the schema $E \vdash \mathscr{J}_i\ \forall i \in I$ means $E \vdash \diamond$.

Good Environments: $E \vdash \diamond$

(Env $\varnothing$)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env α)
$$\frac{E \vdash T \quad \alpha \notin \mathrm{dom}(E_p, E)}{E, \alpha \vdash \diamond}$$

(Env v)
$$\frac{E \vdash T \quad v \notin \mathrm{dom}(E_p, E)}{E, v : T \vdash \diamond}$$

(Env Abs) (α_i distinct)
$$\frac{E \vdash \diamond \quad \neg\exists m.F/m \in \mathrm{dom}(E_p, E)}{E, (\alpha_1, \ldots, \alpha_n)F \vdash \diamond}$$

(Env Alg)

$$\frac{\mathrm{dom}(\Sigma) \cap \mathrm{dom}(E_p, E) = \varnothing \quad \Sigma = (| f_i \ \textbf{of}\ T_{i1} \times \cdots \times T_{iw_i})^{i \in 1..m} \quad E, (\alpha_1, \ldots, \alpha_n)F, \alpha_1, \ldots, \alpha_n \vdash T_{ij} \quad \forall j \in 1..w_i \quad \forall i \in 1..m}{E, (\alpha_1, \ldots, \alpha_n)F = \Sigma \vdash \diamond}$$

Good Types: $E \vdash T$

(Good α)

$$\frac{E \vdash \diamond \quad \alpha \in \mathrm{dom}(E_p, E)}{E \vdash \alpha}$$

(Good Type)

$$\frac{E \vdash T_i \quad \forall i \in 1..n \quad F/n \in \mathrm{dom}(E_p, E)}{E \vdash (T_1, \ldots, T_n)F}$$

Constructor Instance: $E \vdash f : T_1, \ldots, T_n \to T$

(Constructor f)

$$\frac{(E_p, E) = (E', (\alpha_1, \ldots, \alpha_n)F = (| f_i \ \textbf{of}\ T_{i1} \times \cdots \times T_{iw_i})^{i \in 1..m}, E'') \quad E \vdash U_j \quad \forall j \in 1..n \quad \sigma = \{U_1/\alpha_1\} \ldots \{U_n/\alpha_n\}}{E \vdash f_i : T_{i1}\sigma, \ldots, T_{iw_i}\sigma \to (U_1, \ldots, U_n)F}$$

Intuitively, the generative types are those whose values may be scoped by a restriction $(\nu a)A$. Let a type constructor F be *generative* if and only if $F \in \{\mathsf{name}, \mathsf{id}\}$. Let a type T be *generative* if and only if $T = (T_1, \ldots, T_n)F$ for some types $T_1, \ldots, T_n$, and F is generative. Let an environment E be *generative* if and only if v is a name and T is generative whenever $(v : T) \in E$.

Expression Typing: $E \vdash A : T$

(Exp v)

$$\frac{E \vdash \diamond \quad (v : T) \in (E_p, E)}{E \vdash v : T}$$

(Exp Fun)

$$\frac{E, y : T \to U, x : T \vdash A : U}{E \vdash \mu y.\lambda x.A : T \to U}$$

(Exp Passive)

$$\frac{E \vdash A : \mathsf{unit}}{E \vdash [A] : \mathsf{part}}$$

(Exp f)

$$\frac{E \vdash f : T_1, \ldots, T_n \to T \quad E \vdash M_i : T_i \quad \forall i \in 1..n}{E \vdash f(M_1, \ldots, M_n) : T}$$

(Exp Defn Appl)

$$\frac{E \vdash M : T \to U \quad E \vdash N : T}{E \vdash M\ N : U}$$

(Exp Let)

$$\frac{E \vdash A : T \quad E, x : T \vdash B : U}{E \vdash \textbf{let}\ x : T = A\ \textbf{in}\ B : U}$$

(Exp Prim Appl)

$$\frac{p : T_p \quad T_p\{U/\alpha\} = (T \to T') \quad E \vdash M : T \quad E \vdash T'}{E \vdash p\ M : T'}$$

(Exp Match)

$$\frac{\mathrm{fv}(N) = \{x_1, \ldots, x_n\} \quad E, x_1 : T_1, \ldots, x_n : T_n \vdash N : T \quad E \vdash M : T \quad E, x_1 : T_1, \ldots, x_n : T_n \vdash A : U \quad E \vdash B : U}{E \vdash \textbf{match}\ M\ \textbf{with}\ N \to A\ \textbf{else}\ B : U}$$

(Exp Msg)
$$\frac{E \vdash M : (T)\mathsf{chan} \quad E \vdash N : T}{E \vdash M\langle N \rangle : \mathsf{unit}}$$

(Exp Active)
$$\frac{E \vdash a : \mathsf{name} \quad E \vdash A : \mathsf{unit}}{E \vdash a[A] : \mathsf{unit}}$$

(Exp Fork)
$$\frac{E \vdash A : \mathsf{unit} \quad E \vdash B : T}{E \vdash A \upharpoonright B : T}$$

(Exp Res)
$$\frac{E, a : T \vdash A : U \quad T \text{ generative}}{E \vdash (\nu a)A : U}$$

We state without proof a series of basic lemmas.

Lemma 2 *If* $E \vdash T$ *then* $\mathrm{fv}(T) \subseteq \mathrm{dom}(E_p, E)$ *and* $E \vdash \diamond$.

Lemma 3 *Suppose* $\mu = \mathbf{type}\ (\alpha_1, \ldots, \alpha_n)F = \Sigma$ *and* $\mu' = \mathbf{type}\ (\alpha'_1, \ldots, \alpha'_{n'})F' = \Sigma'$. *If* $E \vdash \diamond$ *and* $\mu, \mu' \in (E_p, E)$ *and* $F = F'$ *then* $\mu = \mu'$.

Lemma 4 *If* $E \vdash f : T_1, \ldots, T_n \to T$ *then* $E \vdash T$ *and, for each* $i \in 1..n$, $E \vdash T_i$.

Lemma 5 *If* $E \vdash A : T$ *then* $\mathrm{fn}(A) \cup \mathrm{fv}(A) \subseteq \mathrm{dom}(E)$ *and* $E \vdash T$.

Lemma 6 *If* $E, x : T, E' \vdash A : U$ *and* $E \vdash M : T$ *then* $E, E' \vdash A\{M/x\} : U$.

Lemma 7 *If* $E \vdash v : T$ *and* $E \vdash v : T'$ *then* $T = T'$.

Lemma 8 *If* $E, x_1 : T_1, \ldots, x_n : T_n \vdash N : T$ *and* $E \vdash N\{N_1/x_1\} \ldots \{N_n/x_n\} : U$ *and* $\mathrm{fv}(N) = \{x_1, \ldots, x_n\}$ *and* $\mathrm{fv}(N_1, \ldots, N_n) \subseteq \mathrm{dom}(E)$ *then* $E \vdash N_i : T_i$ *for each* $i \in 1..n$.

Lemma 9 *If* $E, \mu, E' \vdash \mathscr{J}$ *and* $\mathrm{dom}(\mu) \cap (\mathrm{fnfv}(E') \cup \mathrm{fnfv}(\mathscr{J})) = \varnothing$ *then* $E, E' \vdash \mathscr{J}$.

Lemma 10 *If* $E, \mu, \mu', E \vdash \mathscr{J}$ *and* $\mathrm{dom}(\mu) \cap \mathrm{fnfv}(\mu') = \varnothing$ *then* $E, \mu', \mu, E \vdash \mathscr{J}$.

The judgment $E \vdash D \leadsto I$ is defined from two auxiliary judgments.

Auxiliary Judgments:

$E \vdash D \mapsto I$	module D declares interface I exactly
$I \leq I'$	interface I can be used in place of I'

Rules for the Auxiliary Judgments:

(Module Alg Type)
$$\frac{E, \mu \vdash \diamond \quad \mu = (\mathbf{type}\ (\alpha_1, \ldots, \alpha_n)F = \Sigma)}{E \vdash \mu \mapsto \mu}$$

(Module Value)
$$\frac{E \vdash A : T \quad E, x : T \vdash \diamond}{E \vdash \mathbf{let}\ x = A \mapsto (x : T)}$$

(Module Comp)
$$\frac{E \vdash D_1 \mapsto I_1 \quad E, I_1 \vdash D_2 \mapsto I_2}{E \vdash D_1\ D_2 \mapsto I_1, I_2}$$

(Order $\varnothing$)
$$\frac{}{\varnothing \leq \varnothing}$$

(Order Copy)
$$\frac{I \leq I'}{I, \mu \leq I', \mu}$$

(Order Drop)
$$\frac{I \leq I'}{I, \mu \leq I'}$$

(Order Abstract)
$$\frac{I \leq I'}{I, ((\alpha_1, \ldots, \alpha_n)F = \Sigma) \leq I', (\alpha_1, \ldots, \alpha_n)F}$$

Lemma 11 *If* $E \vdash D \mapsto I$ *then* $E, I \vdash \diamond$.

Lemma 12 *The relation* $I \leq I'$ *is reflexive and transitive.*

Lemma 13 *If* $E \vdash \mathscr{J}$ *and* $E' \leq E$ *and* $E' \vdash \diamond$ *then* $E' \vdash \mathscr{J}$.

Interface Implementation: $E \vdash D \rightsquigarrow I$

(Implements)

$$\frac{E \vdash D \mapsto I \quad I \leq I' \quad E, I' \vdash \diamond}{E \vdash D \rightsquigarrow I'}$$

Lemma 14 *If* $E \vdash \mathscr{E}\{A\} : T$ *then there is generative* E' *with* $\text{dom}(E') = \text{bn}(\mathscr{E})$, *and* U, *such that (1)* $E, E' \vdash A : U$ *and (2) for all* B, *if* $E, E' \vdash B : U$ *then* $E \vdash \mathscr{E}\{B\} : T$.

Proof The proof is by induction on the structure of $\mathscr{E}$. □

Lemma 15 *If* $E \vdash \mathscr{E}\{(\nu a)A\} : T$ *and* $a \notin \text{fn}(\mathscr{E}) \cup \text{bn}(\mathscr{E})$ *then* $E \vdash (\nu a)\mathscr{E}\{A\} : T$.

Proof The proof is by induction on the structure of $\mathscr{E}$. □

The definition of constructor safety for expressions is in Section 4.

Constructor Safety for Expressions (Reprise):

Let μ be the type definition $\mu = (\textbf{type}\ (\alpha_1, \ldots, \alpha_n)F = (|\ f_i\ \textbf{of}\ T_{i1} \times \cdots \times T_{iw_i})^{i \in 1..m})$. An expression A is *constructor safe* for μ iff, whenever $A = \mathscr{E}[\textbf{let}\ x : T\ =\ M\ \textbf{in}\ B]$ where $T = (T_1, \ldots, T_n)F$, for some $\mathscr{E}$, x, $T_1, \ldots, T_n$, M, and B, then M takes the form $f_i(N_1, \ldots, N_{w_i})$ for some $i \in 1..m$.

Lemma 16 *If* E *is generative and* $E \vdash A : T$ *and type definition* $\mu \in E$ *then expression* A *is constructor safe for* μ.

Proof Suppose that μ takes the form:

$$\textbf{type}\ (\alpha_1, \ldots, \alpha_n)F = (|\ f_i\ \textbf{of}\ T_{i1} \times \cdots \times T_{iw_i})^{i \in 1..m}$$

Consider any $\mathscr{E}$, x, $T_1, \ldots, T_n$, M, and B, such that $A = \mathscr{E}[\textbf{let}\ x : (T_1, \ldots, T_n)F\ =\ M\ \textbf{in}\ B]$. To show that A is constructor safe for μ we must establish that M takes the form $f_i(N_1, \ldots, N_{w_i})$ for some $i \in 1..m$. We have $E \vdash \mathscr{E}[\textbf{let}\ x : (T_1, \ldots, T_n)F\ =\ M\ \textbf{in}\ B] : T$. By Lemma 14, there is generative E' with $\text{dom}(E') = \text{bn}(\mathscr{E})$, and U, such that (1) $E, E' \vdash \textbf{let}\ x : (T_1, \ldots, T_n)F\ =\ M\ \textbf{in}\ B : U$ and (2) for all B, if $E, E' \vdash B : U$ then $E \vdash \mathscr{E}[B] : T$. There must be an instance of (Exp Let) such that $E \vdash M : (T_1, \ldots, T_n)F$ and $E, x : (T_1, \ldots, T_n)F \vdash B : U$. We proceed by a case analysis of the rule with which $E \vdash M : (T_1, \ldots, T_n)F$ is derived.

(Exp v) We have $M = v$ and $(v : (T_1,\ldots,T_n)F) \in E$. The environment E is generative, so the type constructor F must be generative. But all the generative type constuctors are declared in the pervasive environment E_p, while F is declared in E, a contradiction.

(Exp Fun), (Exp Passive) Here the type constructor F is either func or part. In either case F is declared in the pervasive environment E_p, and yet F is declared in E, a contradiction.

(Exp f) We have $M = f(M_1,\ldots,M_n)$ and $E \vdash f : U_1,\ldots,U_n \to (T_1,\ldots,T_n)F$ and $E \vdash M_j : U_j$ for each $j \in 1..n$. There must be an instance of (Constructor f) such that there is a declaration $\mu' \in (E_p, E)$ of F with f as one of the value constructors of μ'. By Lemmas 2 and 5, $E \vdash A : T$ implies $E \vdash \diamond$. By Lemma 3, $\mu = \mu'$. Hence, there is $i \in 1..m$ such that $f = f_i$. □

Lemma 17 *If $A \equiv A'$ then: $E \vdash A : T$ if and only if $E \vdash A' : T$.*

Proof The proof is by induction on the derivation of $A \equiv A'$.

(Struct $\mathscr{E}$) We have $\mathscr{E}\{B\} \equiv \mathscr{E}\{B'\}$ derived from $B \equiv B'$.

For the forwards direction, suppose that $E \vdash \mathscr{E}\{B\} : T$. By Lemma 14, there is generative E' with $\text{dom}(E') = \text{bn}(\mathscr{E})$, and U, such that (1) $E, E' \vdash B : U$ and (2) for all B', if $E, E' \vdash B' : U$ then $E \vdash \mathscr{E}\{B'\} : T$. By induction hypothesis, (1) and $B \equiv B'$ imply $E, E' \vdash B' : U$. By (2), this implies $E \vdash \mathscr{E}\{B'\} : T$.

The backwards direction follows by a symmetric argument.

(Struct Res) Assuming $a \notin \text{fn}(\mathscr{E}) \cup \text{bn}(\mathscr{E})$ we have $(\nu a)\mathscr{E}\{B\} \equiv \mathscr{E}\{(\nu a)B\}$.

For the forwards direction, suppose that $E \vdash (\nu a)\mathscr{E}\{B\} : T$. There must be an instance of (Exp Res) such that $E, a : U \vdash \mathscr{E}\{B\} : T$ for some generative type U. By Lemma 14, there is generative E' with $\text{dom}(E') = \text{bn}(\mathscr{E})$, and U', such that (1) $E, a : U, E' \vdash B : U'$ and (2) for all B', if $E, a : U, E' \vdash B' : U'$ then $E, a : U \vdash \mathscr{E}\{B'\} : T$. By repeated applications of Lemma 10, $E, a : U, E' \vdash B : U'$ and $\{a\} \cap \text{fnfv}(E') = \varnothing$ imply $E, E', a : U \vdash B : U'$. By (Exp Res), this implies $E, E' \vdash (\nu a)B : U'$. By Lemma 13, $E, a : U, E' \vdash \diamond$ and $E, a : U, E' \leq E, E'$ and $E, E' \vdash (\nu a)B : U'$ imply $E, a : U, E' \vdash (\nu a)B : U'$. By (2), this implies $E, a : U \vdash \mathscr{E}\{(\nu a)B\} : T$. By Lemma 9, this and $a \notin \text{fn}(\mathscr{E}\{(\nu a)B\})$ imply $E \vdash \mathscr{E}\{(\nu a)B\} : T$.

For the backwards direction, suppose that $E \vdash \mathscr{E}\{(\nu a)B\} : T$. By Lemma 15, we have $E \vdash (\nu a)\mathscr{E}\{B\} : T$.

(Struct Part) Assuming $\text{fn}(a[B_1]) \cap \text{bn}(\mathscr{E}) = \varnothing$ we have $a[B_1] \Rsh \mathscr{E}\{B_2\} \equiv \mathscr{E}\{a[B_1] \Rsh B_2\}$.

For the forwards direction, suppose that $E \vdash a[B_1] \Rsh \mathscr{E}\{B_2\} : T$. There must be an instance of (Exp Fork) such that $E \vdash a[B_1] :$ unit and $E \vdash \mathscr{E}\{B_2\} : T$. By Lemma 14, there is generative E' with $\text{dom}(E') = \text{bn}(\mathscr{E})$, and U, such that (1) $E, E' \vdash B_2 : U$ and (2) for all B, if $E, E' \vdash B : U$ then $E \vdash \mathscr{E}\{B\} : T$. By Lemma 13,

$E, E' \vdash \diamond$ and $E, E' \leq E$ and $E \vdash a[B_1] : \mathsf{unit}$ imply $E, E' \vdash a[B_1] : \mathsf{unit}$. By (Exp Fork), $E, E' \vdash a[B_1] : \mathsf{unit}$ and $E, E' \vdash B : U$ imply $E, E' \vdash a[B_1] \upharpoonright B_2 : U$. By (2), this implies $E \vdash \mathscr{E}\{a[B_1] \upharpoonright B_2\} : T$, as required.

For the backwards direction, suppose that $E \vdash \mathscr{E}\{a[B_1] \upharpoonright B_2\} : T$. By Lemma 14, there is generative E' with $\mathrm{dom}(E') = \mathrm{bn}(\mathscr{E})$, and U, such that (1) $E, E' \vdash a[B_1] \upharpoonright B_2 : U$ and (2) for all B, if $E, E' \vdash B : U$ then $E \vdash \mathscr{E}\{B\} : T$. There must be an instance of (Exp Fork) such that $E, E' \vdash a[B_1] : \mathsf{unit}$ and $E, E' \vdash B_2 : U$. By (2), this implies $E \vdash \mathscr{E}\{B_2\} : T$. By Lemma 9, $E, E' \vdash a[B_1] : \mathsf{unit}$ and $\mathrm{dom}(E') \cap \mathrm{fnfv}(a[B_1]) = \varnothing$ imply $E \vdash a[B_1] : \mathsf{unit}$. By (Exp Fork), this and $E \vdash \mathscr{E}\{B_2\} : T$ imply $E \vdash a[B_1] \upharpoonright \mathscr{E}\{B_2\} : T$, as required.

(Struct Msg) Assuming $\mathrm{bn}(\mathscr{E}) \cap \mathrm{fn}(M\langle N\rangle) = \varnothing$, and either M is a global channel or $\mathscr{E}$ is partition-free, we have $M\langle N\rangle \upharpoonright \mathscr{E}\{B\} \equiv \mathscr{E}\{M\langle N\rangle \upharpoonright B\}$.

For the forwards direction, suppose that $E \vdash M\langle N\rangle \upharpoonright \mathscr{E}\{B\} : T$. There must be an instance of (Exp Fork) such that $E \vdash M\langle N\rangle : \mathsf{unit}$ and $E \vdash \mathscr{E}\{B\} : T$. By Lemma 14, there is generative E' with $\mathrm{dom}(E') = \mathrm{bn}(\mathscr{E})$, and U, such that (1) $E, E' \vdash B : U$ and (2) for all B', if $E, E' \vdash B' : U$ then $E \vdash \mathscr{E}\{B'\} : T$. By (Exp Fork), $E, E' \vdash M\langle N\rangle \upharpoonright B : U$. By (2), $E \vdash \mathscr{E}\{M\langle N\rangle \upharpoonright B\} : T$, as required.

For the backwards direction, suppose that $E \vdash \mathscr{E}\{M\langle N\rangle \upharpoonright B\} : T$. By Lemma 14, there is generative E' with $\mathrm{dom}(E') = \mathrm{bn}(\mathscr{E})$, and U, such that (1) $E, E' \vdash M\langle N\rangle \upharpoonright B : U$ and (2) for all B', if $E, E' \vdash B' : U$ then $E \vdash \mathscr{E}\{B'\} : T$. There must be an instance of (Exp Fork) such that $E, E' \vdash M\langle N\rangle : \mathsf{unit}$ and $E, E' \vdash B : T$. By (2), $E \vdash \mathscr{E}\{B\} : T$. By Lemma 9, $E, E' \vdash M\langle N\rangle : \mathsf{unit}$ and $\mathrm{dom}(E') \cap \mathrm{fn}(M\langle N\rangle) = \varnothing$ imply $E \vdash M\langle N\rangle : \mathsf{unit}$. By (Exp Fork), $E \vdash M\langle N\rangle \upharpoonright \mathscr{E}\{B\} : T$, as required.

The cases for (Struct Assoc), (Struct Let), (Struct Fork Unit), (Struct Part Unit), (Struct Msg Unit) follow by similar arguments. The cases for (Struct Refl), (Struct Symm), and (Struct Trans) are routine. □

Lemma 18 *If $E \vdash A : T$ and $A \to A'$ then $E \vdash A' : T$.*

Proof The proof is by induction on the derivation of $A \to A'$. Suppose that $E \vdash A : T$.

(Red $\equiv$) We have $A \equiv B$ and $B \to B'$ and $B' \equiv A'$, leading to $A \to A'$. By Lemma 17, $A \equiv B$ and $E \vdash A : T$ imply $E \vdash B : T$. By induction hypothesis, $E \vdash B : T$ and $B \to B'$ imply $E \vdash B' : T$. By Lemma 17, $B' \equiv A'$ and $E \vdash B' : T$ imply $E \vdash A' : T$.

(Red $\mathscr{E}$) We have $\mathscr{E}\{B\} \to \mathscr{E}\{B'\}$ derived from $B \to B'$. By Lemma 14, $E \vdash \mathscr{E}\{B\} : T$ then there is generative E' with $\mathrm{dom}(E') = \mathrm{bn}(\mathscr{E})$, and U, such that (1) $E, E' \vdash B : U$ and (2) for all B', if $E, E' \vdash B' : U$ then $E \vdash \mathscr{E}\{B'\} : T$. By induction hypothesis, (1) and $B \to B'$ imply $E, E' \vdash B' : U$. By (2), this implies $E \vdash \mathscr{E}\{B'\} : T$.

(Red Let) We have:

$$\mathbf{let}\ x : U = M\ \mathbf{in}\ B \to B\{M/x\}$$

There must be an instance of (Exp Let), such that $E \vdash M : U$ and $E, x : U \vdash B : T$. By Lemma 6, $E \vdash B\{M/x\} : T$.

(Red Fun) We have:

$$(\mu y.\lambda x.B)\ N \rightarrow B\{\mu y.\lambda x.B/y\}\{N/x\}$$

It must be that $E \vdash (\mu y.\lambda x.B)\ N : T$ derives from an instance of (Exp Defn Appl) such that $E \vdash (\mu y.\lambda x.B) : U \rightarrow T$ and $E \vdash N : U$ for some U, and an instance of (Exp Fun) such that $E, y : U \rightarrow T, x : U \vdash B : T$. By Lemma 6, $E, x : U \vdash B\{\mu y.\lambda x.B/y\} : T$. By Lemma 6, $E, \vdash B\{\mu y.\lambda x.B/y\}\{N/x\} : T$.

(Red Match) Assuming $\exists\sigma.M = N\sigma \wedge \mathrm{dom}(\sigma) = \mathrm{fv}(N)$ we have:

$$(\textbf{match}\ M\ \textbf{with}\ N \rightarrow B\ \textbf{else}\ B') \rightarrow B\sigma$$

There must be an instance of (Exp Match) such that $E, x_1 : T_1, \ldots, x_n : T_n \vdash N : U$ and $E \vdash M : U$ and $E, x_1 : T_1, \ldots, x_n : T_n \vdash B : T$ and $E \vdash B' : T$ where $\mathrm{fv}(N) = \{x_1, \ldots, x_n\}$. We know $\mathrm{fv}(M) \subseteq \mathrm{dom}(E)$ and that $\mathrm{dom}(E) \cap \mathrm{fv}(N) = \varnothing$; so since $M = N\sigma$ we may assume that $\sigma = \{N_1/x_1\} \ldots \{N_n/x_n\}$ where $\mathrm{fv}(N_i) \subseteq \mathrm{fv}(M)$ for each $i \in 1..n$. By Lemma 8, $E \vdash N\{N_1/x_1\} \ldots \{N_n/x_n\} : U$ and $E, x_1 : T_1, \ldots, x_n : T_n \vdash N : T$ and $\mathrm{fv}(N) = \{x_1, \ldots, x_n\}$ and $\mathrm{fv}(N_1, \ldots, N_n) \subseteq \mathrm{dom}(E)$ imply that $E \vdash N_i : T_i$ for each $i \in 1..n$. By multiple applications of Lemma 6, $E \vdash B\{N_1/x_1\} \ldots \{N_n/x_n\} : T$.

(Red Mismatch) Assuming $\neg\exists\sigma.M = N\sigma$ we have:

$$(\textbf{match}\ M\ \textbf{with}\ N \rightarrow B\ \textbf{else}\ B') \rightarrow B'$$

There must be an instance of (Exp Match) such that $E, x_1 : T_1, \ldots, x_n : T_n \vdash N : U$ and $E \vdash M : U$ and $E, x_1 : T_1, \ldots, x_n : T_n \vdash B : T$ and $E \vdash B' : T$ where $\mathrm{fv}(N) = \{x_1, \ldots, x_n\}$.

(Red Comm) Assuming $\mathrm{fn}(c, M) \cap (\mathrm{bn}(\mathscr{E}_1, \mathscr{E}_2)) = \varnothing$ we have:

$$\mathscr{E}_1\{c\langle M\rangle\} \upharpoonright \mathscr{E}_2\{\mathsf{recv}\ c\} \rightarrow \mathscr{E}_1\{()\} \upharpoonright \mathscr{E}_2\{M\}$$

There must be an instance of (Exp Fork) such that $E \vdash \mathscr{E}_1\{c\langle M\rangle\} : \mathsf{unit}$ and $E \vdash \mathscr{E}_2\{\mathsf{recv}\ c\} : T$. By Lemma 14, $E \vdash \mathscr{E}_1\{c\langle M\rangle\} : \mathsf{unit}$ implies there is generative E_1 with $\mathrm{dom}(E_1) = \mathrm{bn}(\mathscr{E}_1)$ and U_1 such that (1) $E, E_1 \vdash c\langle M\rangle : U_1$ and (2) for all B, if $E, E_1 \vdash B : U_1$ then $E \vdash \mathscr{E}_1\{B\} : \mathsf{unit}$. Given (1) there must be an instance of (Exp Msg) such that $E, E_1 \vdash c : (U)\mathsf{chan}$ and $E, E_1 \vdash M : U$ and $U_1 = \mathsf{unit}$ for some type U. Lemma 9 implies $E \vdash M : U$ since $\mathrm{dom}(E_1) = \mathrm{bn}(\mathscr{E}_1)$ and $\mathrm{fn}(M) \cap \mathrm{bn}(\mathscr{E}_1) = \varnothing$. By Lemma 14, $E \vdash \mathscr{E}_2\{\mathsf{recv}\ c\} : T$ implies there is generative E_2 with $\mathrm{dom}(E_2) = \mathrm{bn}(\mathscr{E}_2)$ and U_2 such that (3) $E, E_2 \vdash \mathsf{recv}\ c : U_2$ and (4) for all B, if $E, E_2 \vdash B : U_2$ then $E \vdash \mathscr{E}_2\{B\} : A$. Given (3) there must be an instance of (Exp Prim Appl) such that $\mathsf{recv} : T_p$ where $T_p = (\alpha)\mathsf{chan} \rightarrow \alpha$ and $T_p\{U_4/\alpha\} = (U_3 \rightarrow U_2)$ and $E \vdash c : U_3$. Hence, by Lemma 7, we have $U_3 = (U)\mathsf{chan}$ and $U_4 = U_2 = U$. By (2), $E, E_1 \vdash () : \mathsf{unit}$ implies $E \vdash \mathscr{E}_1\{()\} : \mathsf{unit}$. By Lemma 13, $E, E_2 \vdash \diamond$ and $E, E_2 \leq E$ implies $E, E_2 \vdash M : U$. By (4), $E, E_2 \vdash M : U$ implies $E \vdash \mathscr{E}_2\{M\} : A$. By (Exp Fork), we have $E \vdash \mathscr{E}_1\{()\} \upharpoonright \mathscr{E}_2\{M\} : A$.

(Red Cut) Assuming B copyable, we have:

$$a[B] \upharpoonright \mathsf{cut}\ a \longrightarrow [B]$$

There must be an instance of (Exp Fork) such that $E \vdash a[B] : \mathsf{unit}$ and $E \vdash \mathsf{cut}\ a : A$. There must be an instance of (Exp Active) such that $E \vdash a : \mathsf{name}$ and $E \vdash B : \mathsf{unit}$. Given $E \vdash \mathsf{cut}\ a : A$, there must be an instance of (Exp Prim Appl) such that $\mathsf{cut} : T_p$ where $T_p = \mathsf{name} \to \mathsf{part}$ and $T_p\{U/\alpha\} = (\mathsf{name} \to \mathsf{part})$ and $E \vdash a : \mathsf{name}$ and $A = \mathsf{part}$. By (Exp Passive), $E \vdash B : \mathsf{unit}$ implies $E \vdash [B] : \mathsf{part}$.

(Red Paste) We have:

$$\mathsf{paste}\ (a, [B]) \longrightarrow a[B]$$

There must be an instance of (Exp Prim Appl) such that $\mathsf{paste} : T_p$ where $T_p = (\mathsf{name} \times \mathsf{part}) \to \mathsf{unit}$ and $T_p\{U/\alpha\} = (\mathsf{name} \times \mathsf{part}) \to \mathsf{unit}$ and $E \vdash (a, [B]) : (\mathsf{name} \times \mathsf{part})$ and $T = \mathsf{unit}$. The notation $(a, [B])$ stands for $\mathsf{tuple}2(a, [B])$, while the notation $(\mathsf{name} \times \mathsf{part})$ stands for $(\mathsf{name}, \mathsf{part})\mathsf{tuple}2$. There must be an instance of (Exp f) such that $E \vdash \mathsf{tuple}2 : \mathsf{name}, \mathsf{part} \to (\mathsf{name}, \mathsf{part})\mathsf{tuple}2$ and $E \vdash a : \mathsf{name}$ and $E \vdash [B] : \mathsf{part}$. There must be an instance of (Exp Passive), such that $E \vdash B : \mathsf{unit}$. By (Exp Active), $E \vdash a : \mathsf{name}$ and $E \vdash B : \mathsf{unit}$ imply $E \vdash a[B] : \mathsf{unit}$.

In each case, we have proved $E \vdash A' : T$, as required. □

Proof of Theorem 1 *If $A \equiv A'$ or $A \to A'$ then $E \vdash A : T$ implies $E \vdash A' : T$.*

Proof Combine Lemmas 17 and 18. □

D.4 Translational Semantics of Modules

Semantics of a Module: $[\![D]\!](E,A) = (E',A')$

$[\![\mu]\!](E,A) = ((\mu,E),A)$
$[\![\mathbf{let}\ x = B]\!](E,A) = (E, \mathbf{let}\ x = B\ \mathbf{in}\ A)$
$[\![D_1\ D_2]\!](E,A) = [\![D_1]\!]([\![D_2]\!](E,A))$

Lemma 19 *If $[\![D]\!](E_2,A) = (E_{12},B)$ then $E_{12} = E_1,E_2$ where $[\![D]\!](\varnothing,A) = (E_1,B)$. Moreover, E_1 consists entirely of algebraic type definitions.*

Proof The proof is by induction on the structure of the module D. □

Lemma 20 *If $E \vdash D \mapsto I$ and $E,I \vdash A : T$ and $[\![D]\!](\varnothing,A) = (E',A')$ then $E,E' \vdash A' : T$.*

Proof The proof is by induction on the structure of the module D. Suppose that $E \vdash D \mapsto I$ and $E,I \vdash A : T$ and $[\![D]\!](\varnothing,A) = (E',A')$.

(Module Alg Type) We have $D = I = \mu$ and $E,\mu \vdash \diamond$ where $\mu = (\mathbf{type}\ (\alpha_1,\ldots,\alpha_n)F = \Sigma)$. Hence, we have $E' = \mu$ and $A' = A$, and so $E,E' \vdash A' : T$, as required.

(Module Value) We have $D = (\textbf{let}\ x = B)$ and $E \vdash B : U$ and $E, x : U \vdash \diamond$ and $I = (x : U)$. Hence, $E' = \varnothing$ and $A' = \textbf{let}\ x = B\ \textbf{in}\ A$. By assumption, $E, x : U \vdash A : T$ since $I = (x : U)$. By (Exp Let), this and $E \vdash B : U$ imply $E \vdash \textbf{let}\ x = B\ \textbf{in}\ A : T$, as required.

(Module Comp) We have $D = (D_1\ D_2)$ and $I = I_1, I_2$ and $E \vdash D_1 \mapsto I_1$ and $E, I_1 \vdash D_2 \mapsto I_2$. Let $[\![D_2]\!](\varnothing, A) = (E_2, A_2)$ so that $[\![D]\!](E, A) = [\![D_1]\!](E_2, A_2) = (E', A')$. By induction hypothesis, $E, I_1 \vdash D_2 \mapsto I_2$ and $E, I_1, I_2 \vdash A : T$ and $[\![D_2]\!](\varnothing, A) = (E_2, A_2)$ imply $E, I_1, E_2 \vdash A_2 : T$. By Lemma 10, we obtain $E, E_2, I_1 \vdash A_2 : T$ since by construction $\text{fnfv}(E_2) = \varnothing$, because E_2 consists entirely of algebraic type definitions. By Lemma 13, $E \vdash D_1 \mapsto I_1$ and $E, E_2 \leq E$ and $E, E_2 \vdash \diamond$ imply $E, E_2 \vdash D_1 \mapsto I_1$. By Lemma 19, $[\![D_1]\!](E_2, A_2) = (E', A')$ implies $E' = E_1, E_2$ where $[\![D_1]\!](\varnothing, A_2) = (E_1, A')$. By induction hypothesis, $E, E_2 \vdash D_1 \mapsto I_1$ and $E, E_2, I_1 \vdash A_2 : T$ and $[\![D_1]\!](\varnothing, A_2) = (E_1, A')$ imply $E, E_2, E_1 \vdash A' : T$. By Lemma 10, we obtain $E, E_1, E_2 \vdash A' : T$ since by construction $\text{fnfv}(E_1) = \varnothing$, because E_1 consists entirely of algebraic type definitions.

In each case, we have proved $E, E' \vdash A' : T$, as required. □

The definition of constructor safety for modules is in Section 4.

Constructor Safety for Modules (Reprise):

A module D is *constructor safe* if and only if, if $[\![D]\!](\varnothing, ()) = (E, A)$, for some E, A, and if $A \rightarrow^* A'$, then for all type definitions $\mu \in E$, A' is constructor safe for μ.

Proof of Theorem 2 *If $\varnothing \vdash D \rightsquigarrow \varnothing$ then D is constructor safe.*

Proof Suppose that $[\![D]\!](\varnothing, ()) = (E, A)$ and that $A \rightarrow^* A'$. Consider any type definition $\mu \in E$. By definition, to show that D is constructor safe, it suffices to show that A' is constructor safe for μ. By (Implements), $\varnothing \vdash D \rightsquigarrow \varnothing$ implies there is I such that $\varnothing \vdash D \mapsto I$ and $I \leq \varnothing$ and $\varnothing \vdash \diamond$. We have $I \vdash () : \mathtt{unit}$, and so $E \vdash A : \mathtt{unit}$, by Lemma 20. By Theorem 1, $A \rightarrow^* A'$ implies that $E \vdash A' : \mathtt{unit}$. By construction, E consists entirely of algebraic type definitions, and hence is generative. By Lemma 16, A' is constructor safe for μ. □

References

AppLogic: Grid Operating System for Utility Computing. 3TERA, September 2006. At http://0301.netclime.net/1_5/8/A/8/3teraAppLogic0906.pdf.

M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

Amazon Elastic Compute Cloud (Amazon EC2) - Limited Beta. Amazon Web Services LLC, August 2006. At http://aws.amazon.com/ec2.

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.

J. Arwe et al. *Service Modeling Language*, 2007. Draft Specification, version 1.0, at http://www.serviceml.org/SML-200702.pdf.

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, 2003.

A. Bavier, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *NSDI'04*, 2004.

G. Berry and G. Boudol. The chemical abstract machine. In *17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 81–94. 1990.

K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 139–152, 2006.

D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. *Simple Object Access Protocol (SOAP) 1.1*, 2000.

L. Cardelli and R. Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25(3):309–316, 1999.

S. Carpineti, C. Laneve, and L. Padovani. Piduce—a project for experimenting web services technologies. At http://www.cs.unibo.it/PiDuce/, 2006.

G. Castagna, J. Vitek, and F. Zappa Nardelli. The seal calculus. *Information and Computation*, 201(1):1–54, 2005.

E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL) 1.1*, 2001.

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.

J. R. Ellis. A LISP shell. *ACM SIGPLAN Notices*, 15(5):24–34, May 1980.

M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of EuroSys2006*, 2006.

M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217. North Holland, 1986.

S. Garfinkel. Commodity grid computing with Amazon's S3 and EC2. *;login:*, pages 7–13, February 2007.

W. Gibson. Understanding the System Definition Model. TechNote TN–1104, Visual Studio Team System, 2005. Available at http://msdn2.microsoft.com/en-us/teamsystem/aa718852.aspx.

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/.

A. D. Gordon. V for Virtual. In *Algebraic Process Calculi: The First Twenty Five Years and Beyond*, pages 114–117, 2005. Available as BRICS Note NS–05–3, University of Aarhus.

A. D. Gordon and P. D. Hankin. A concurrent object calculus: reduction and typing. In *3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*, volume 16(3) of *ENTCS*. Elsevier, 1998.

K. Havelund and K. Larsen. The fork calculus. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *International Conference on Automata, Languages, and Programming (ICALP'93)*, volume 700 of *LNCS*. Springer, 1993.

S. Holmström. PFL: A functional language for parallel programming. In *Declarative Programming Workshop*, pages 114–139. University College, London, 1983. Extended version published as Report 7, Programming Methodology Group, Chalmers University. September 1983.

A. Hoykhet, J. Lange, and P. Dinda. Virtuoso: A system for virtual machine marketplaces. Technical Report NWU-CS-04-39, Northwestern University, 2004. URL http://virtuoso.cs.northwestern.edu/NWU-CS-04-39.pdf.

L. Huang, P. Hudak, and J. Peterson. HPorter: using arrows to compose parallel processes. In *Practical Aspects of Declarative Languages*, volume 4354 of *LNCS*, pages 275–289. Springer, 2007.

J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37: 67–111, 2000.

Preboot Execution Environment (PXE) Specification. Intel Corporation and SystemSoft, 1999. Available at http://www.pix.net/software/pxeboot/archive/pxespec.pdf.

B. Lampson. Software components: Only the giants survive. In K. Spärck-Jones and A. Herbert, editors, *Computer Systems: Theory, Technology, and Applications*, pages 137–146. Springer, 2004.

A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *European Symposium on Programming (ESOP'07)*, pages 33–47, 2007.

M. Lienhardt, A. Schmitt, and J.-B. Stefani. OZ/K: A kernel language for component-based open programming. In *Generative Programming and Component Engineering (GPCE'07)*, 2007.

J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *5th European Software Engineering Conference (ESEC 95)*, volume 989, pages 137–153. Springer, 1995.

R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199–218, 1970.

Windows PowerShell Getting Started Guide and Quick Reference. Microsoft Corporation, 2006. Available at http://go.microsoft.com/fwlink/?LinkID=64774&clcid=0x409.

R. Milner. *Communicating and Mobile Systems: the π-Calculus*. CUP, 1999.

C. Murthy. Advanced programming language design in enterprise software: A lambda-calculus theorist wanders into a datacenter. In *34th ACM Symposium on Principles of Programming Languages (POPL'07)*, pages 263–264, 2007.

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.

About the OSGi Service Platform, Technical Whitepaper Revision 4.1. OSGi, November 2005. At http://www.osgi.org/documents/collateral/TechnicalWhitePaper2005osgi-sp-overview.pdf.

D. Pallmann. *Programming "Indigo": The Code Name for the Unified Framework for Building Service-Oriented Applications on the Microsoft Windows Platform*. Microsoft Press, 2005.

B. A. Randell and R. Lhotka. Bridge the gap between development and operations with Whitehorse. *MSDN Magazine*, July 2004. Available at http://msdn.microsoft.com/msdnmag/issues/04/07/whitehorse/default.aspx.

J. H. Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation (PLDI'91)*, pages 293–305, 1991.

A. Schmitt and J.-B. Stefani. The Kell calculus: A family of higher-order distributed process calculi. In *Global Computing*, LNCS, pages 146–178. Springer, 2005.

D. Syme. *F#*, 2005. Project website at http://research.microsoft.com/fsharp/.

C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

C. Wolf and E. M Halter. *Virtualization: from the desktop to the enterprise*. Apress, 2005.