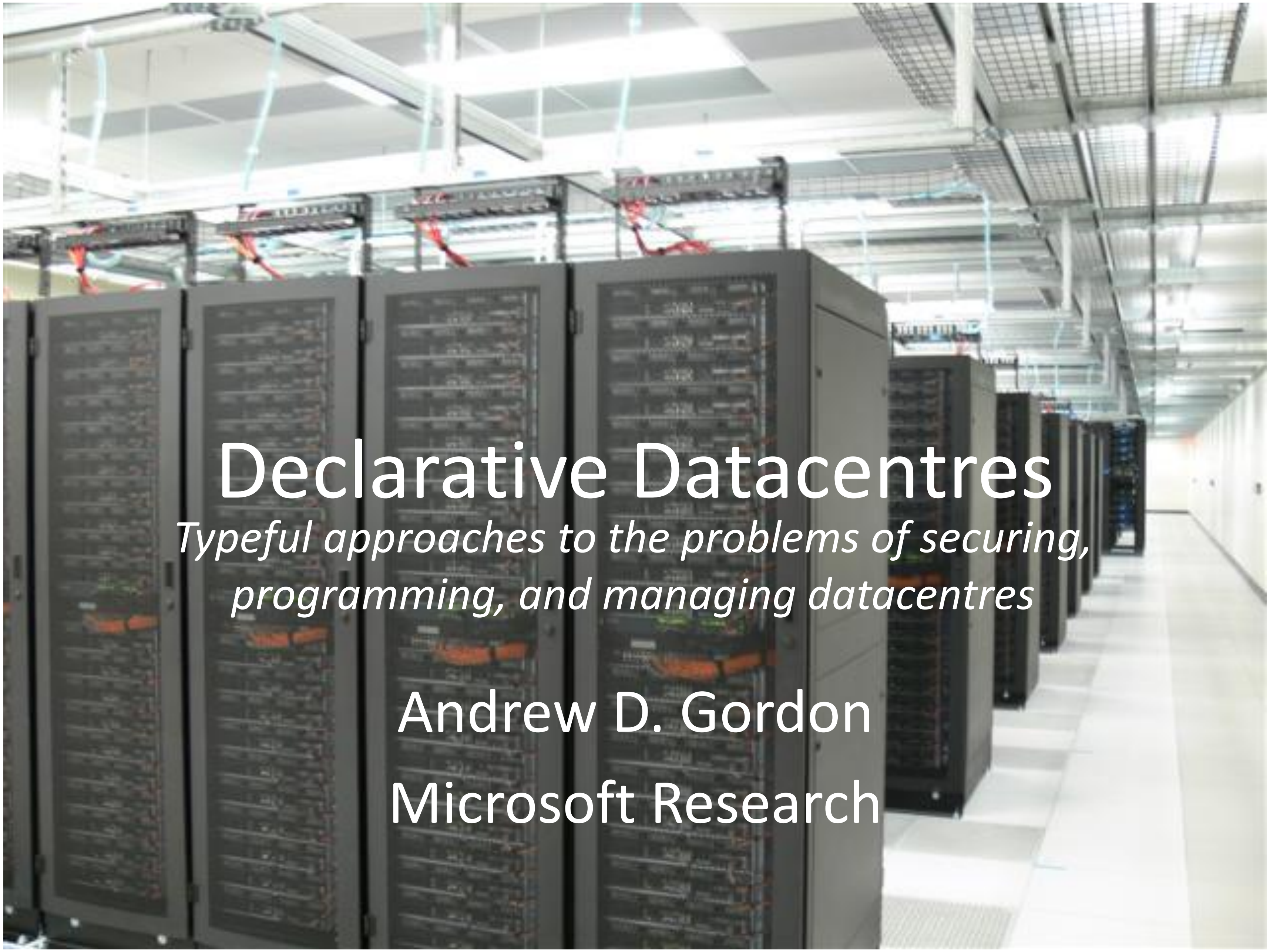


Declarative Datacentres

Typeful approaches to the problems of securing, programming, and managing datacentres

Andrew D. Gordon

Microsoft Research

GLOBAN 2008, Warsaw, September 22-26, 2008

Thursday 25th 14:00-14:45

1

Syllabus

1. V for Virtual
2. A Concurrent λ-Calculus with Refinement Types
3. Security Protocols and their Implementations

The theoretical core is the typed lambda-calculus RCF, and its implementation as an enhanced typechecker for F#; RCF supports functional programming a la ML and Haskell, concurrency in the style of process calculus, and refinement types allowing correctness properties to be stated in style of dependent type theory.

We will examine a diverse (but hardly exhaustive) range of problems in the area of programming datacentres: cryptographic security protocols, language-based access control, and the assembly and management of software components such as VMs

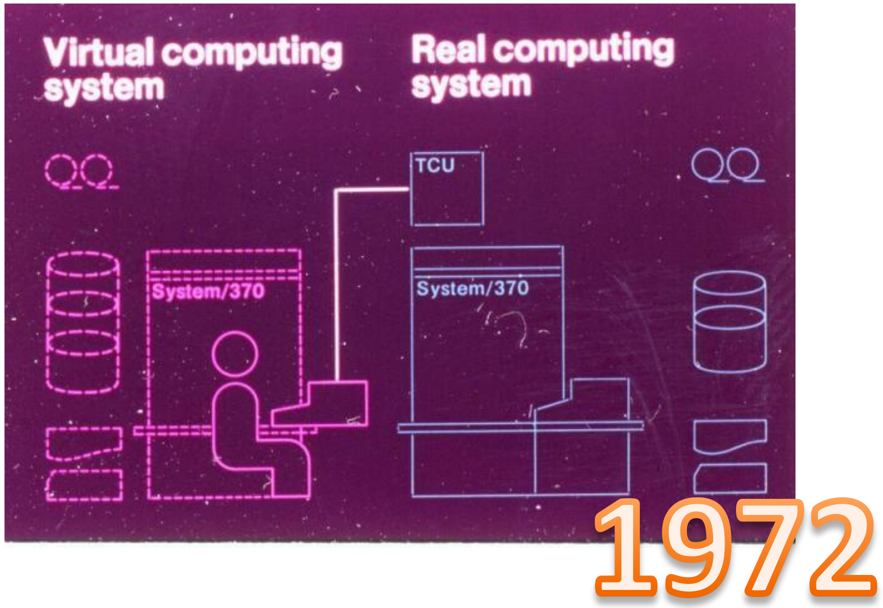
V for Virtual

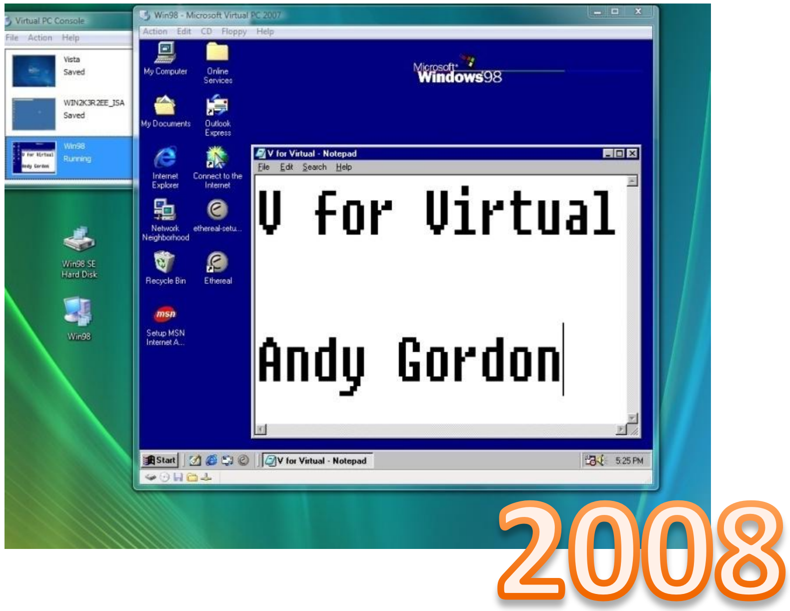
Declarative Datacentres, Part 1

VIRTUALIZATION: WHAT AND WHY?

What is a Virtual Machine?

"A virtual machine is an efficient, isolated duplicate of the real machine" (Popek and Goldberg, CACM 1974)



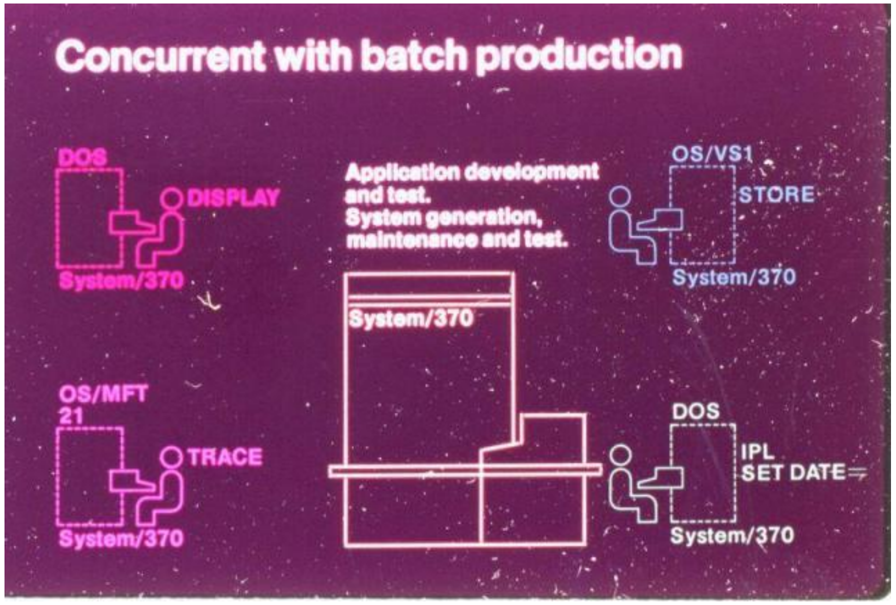


V for Virtual

Andy Gordon

- A **virtual machine monitor** (VMM) contains one or more VMs
- Each VM runs a **guest OS**, which itself runs applications
- The VMM may run under a **host OS**, or on the bare machine

Why Virtualize?


Concurrent with batch production
DOS
DISPLAY
System/370
Application development and test. System generation, maintenance and test.
OS/VS1
STORE
System/370
System/370
OS/MFT 21
TRACE
System/370
DOS
IPL
SET DATE
System/370

Comparison with Real Machines

- Better hardware utilization
- Better application isolation
- Faster provisioning
- Poorer performance

Applications of Virtualization

- Server (or client) consolidation
- Development and test
- Legacy applications
- Training and demos
- Security

What Is, and Isn't, a VMM?

- **Efficient:** An overwhelming majority of guest instructions are executed by the hardware without VMM intervention
- **Duplicate:** Software on the VMM executes identically(*) to its execution on hardware, barring timing effects
- **Isolated:** The VMM manages all hardware resources

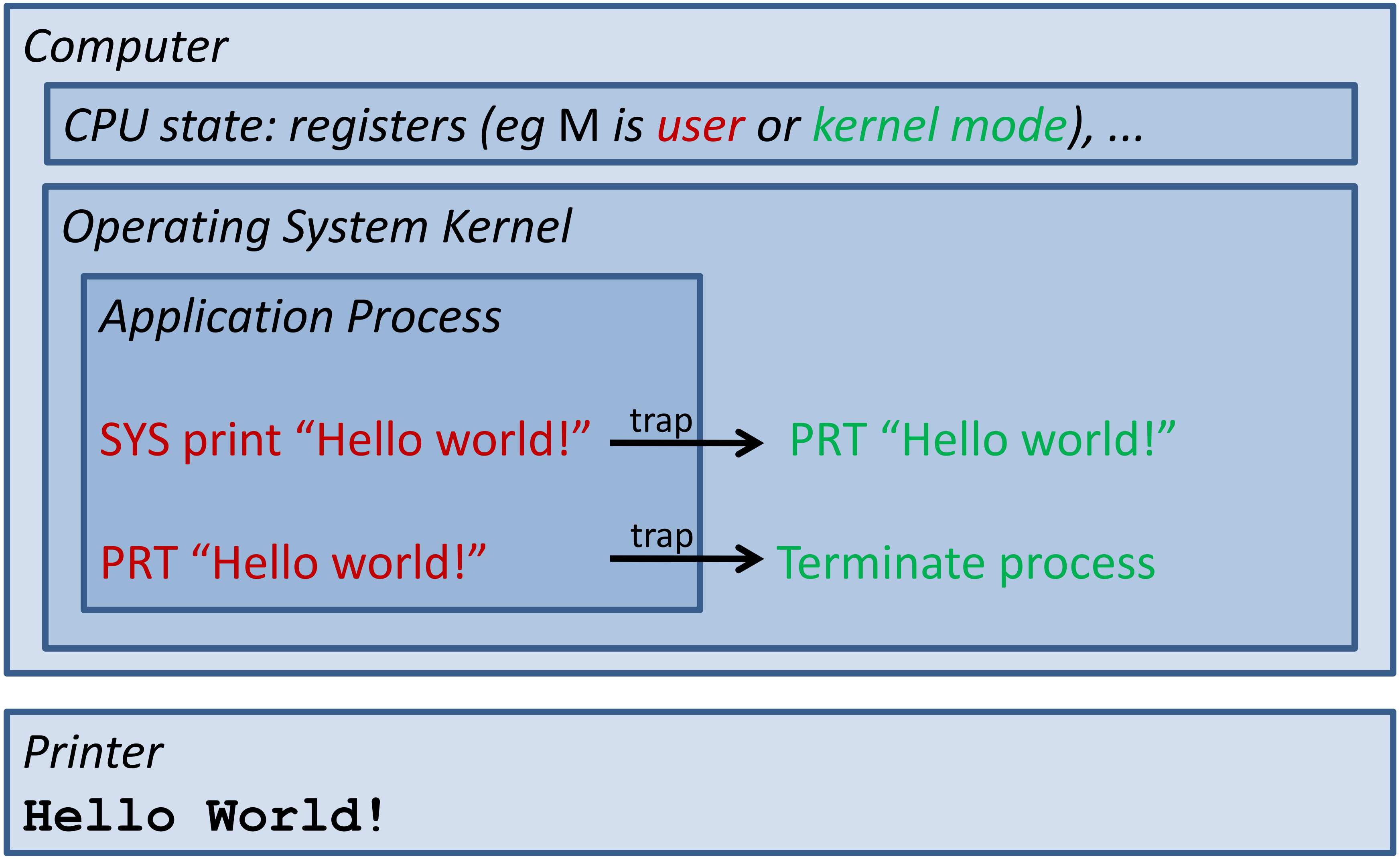
- Non-examples:
  - Language-based VMs eg Sun's JVM or Microsoft's CLR
  - Hardware emulators eg Virtual PC on Macs with PowerPC hardware

G. Popek and R. P. Goldberg (1974). *Formal requirements for virtualizable third generation architectures*. CACM 17(7):412-421.

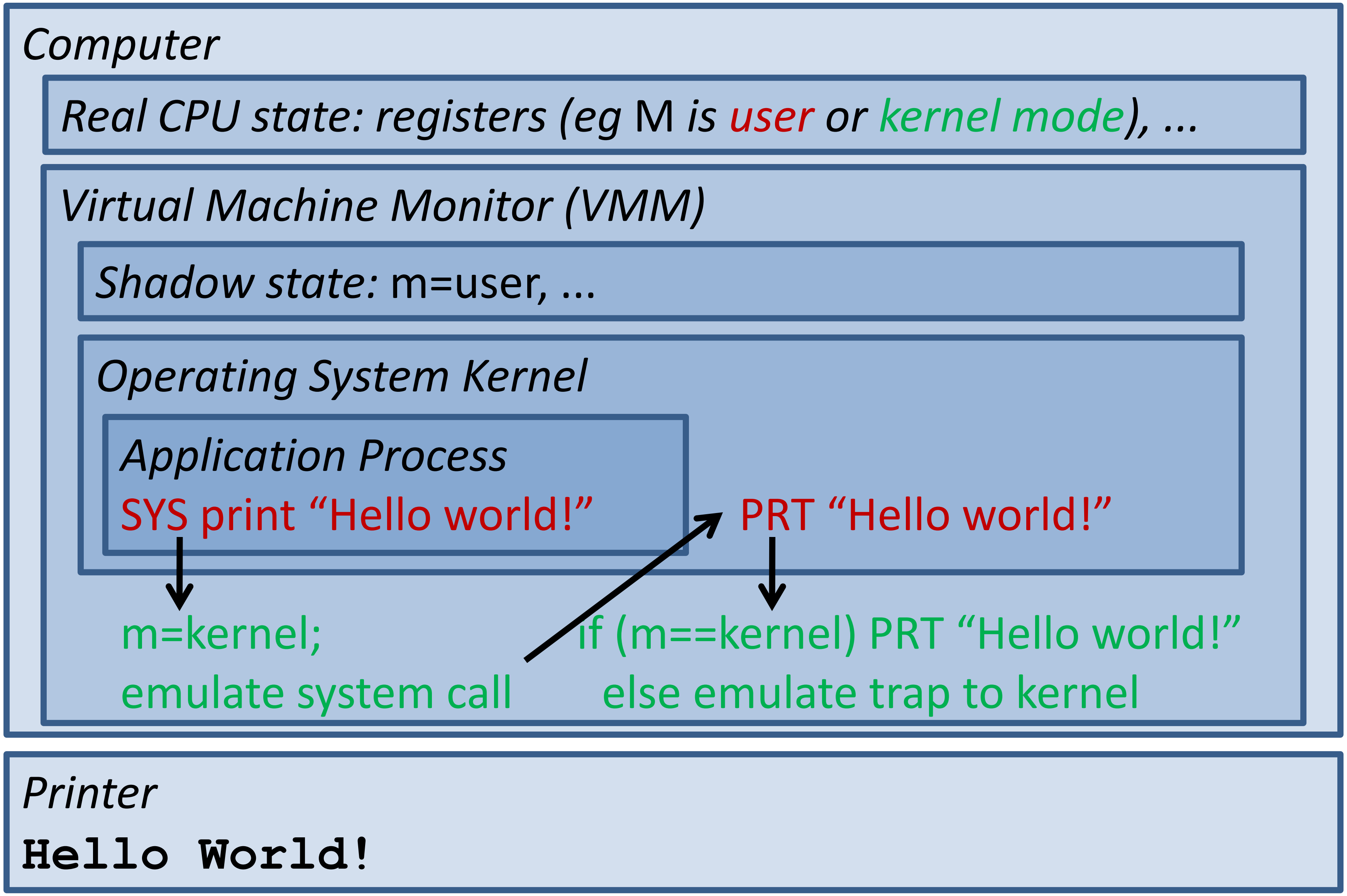
K. Adams and O. Ageson (2006). *A comparison of software and hardware techniques for x86 virtualization*. ASPLOS'06.

HOW VIRTUALIZATION WORKS

How an Operating System Works



How Classic Virtualization Works



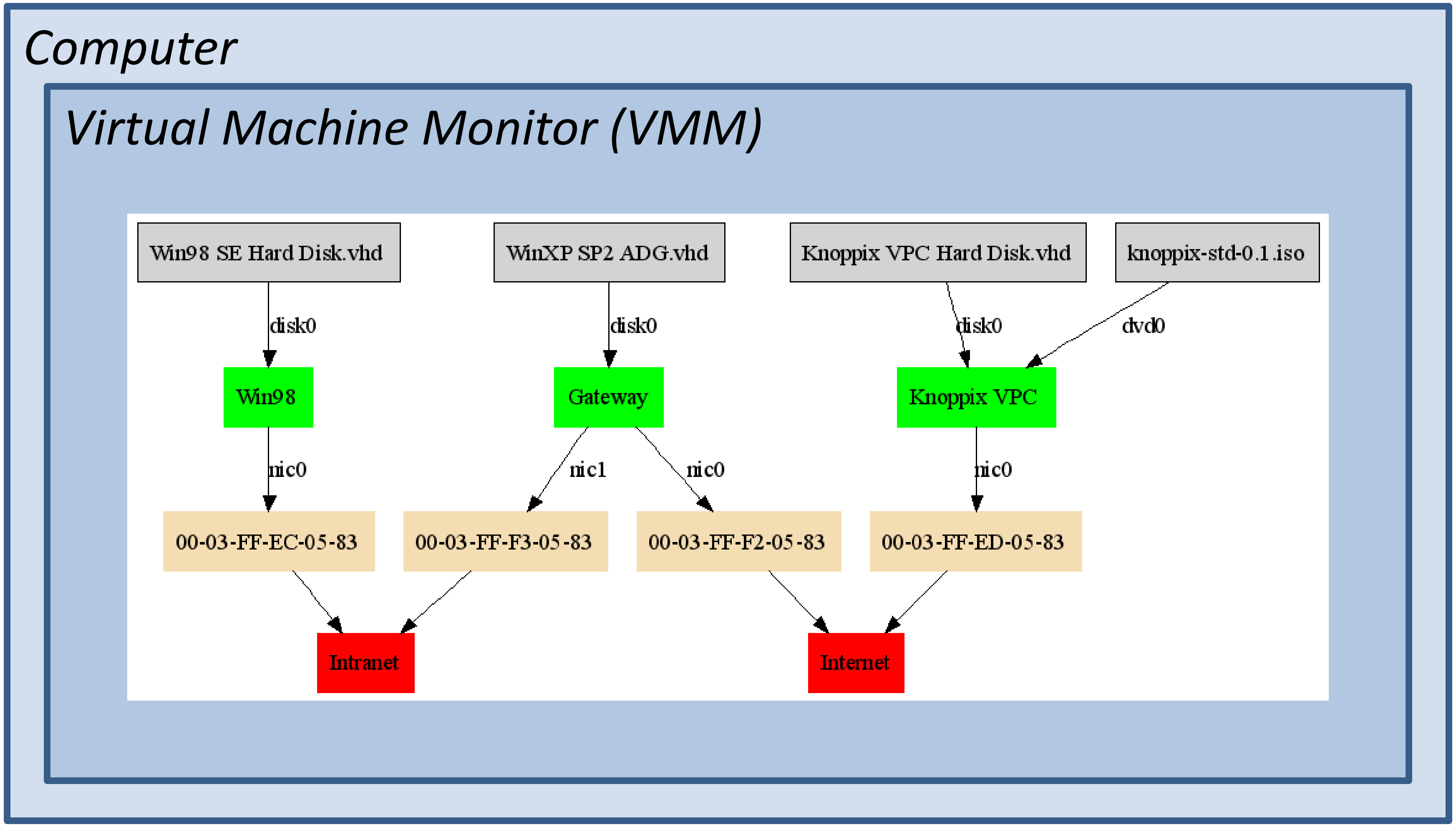
Computer
Real CPU state: registers (eg M is user or kernel mode), ...
Virtual Machine Monitor (VMM)
Shadow state: m=user, ...
Operating System Kernel
Application Process
SYS print "Hello world!"
PRT "Hello world!"
m=kernel;
emulate system call
if (m==kernel) PRT "Hello world!"
else emulate trap to kernel
Printer
Hello World!

But x86 is Not Classically Virtualizable

- In a **classically virtualizable** architecture, all instructions that access **privileged state** can be set to trap if run in user mode
- In x86, instructions can access privileged state in user mode without trapping
  - For example, in kernel mode **popf** modifies the interrupt-related IF flag; in user mode, modifications to IF are suppressed, with no trap
- Hence, VMMs for x86 rely on **binary translation (BT)**
  - To run kernel mode guest code, the VMM inserts additional instructions to emulate the kernel mode behaviour in user mode
  - **popf** turns into a short instruction sequence to access shadow state
- Since 2005, AMD's SVM & Intel's VT provide hardware support

J.S. Robin and C.E. Irvine (2000). *Analysis of the Intel Pentium's ability to support a secure virtual machine monitor*. In 9th USENIX Security Symposium.

The Computer is the Network



An ambient calculus guy programs some real virtual machines

Joint work with Karthikeyan Bhargavan and Iman Narasamdya

SERVICE COMBINATORS FOR FARMING VIRTUAL MACHINES

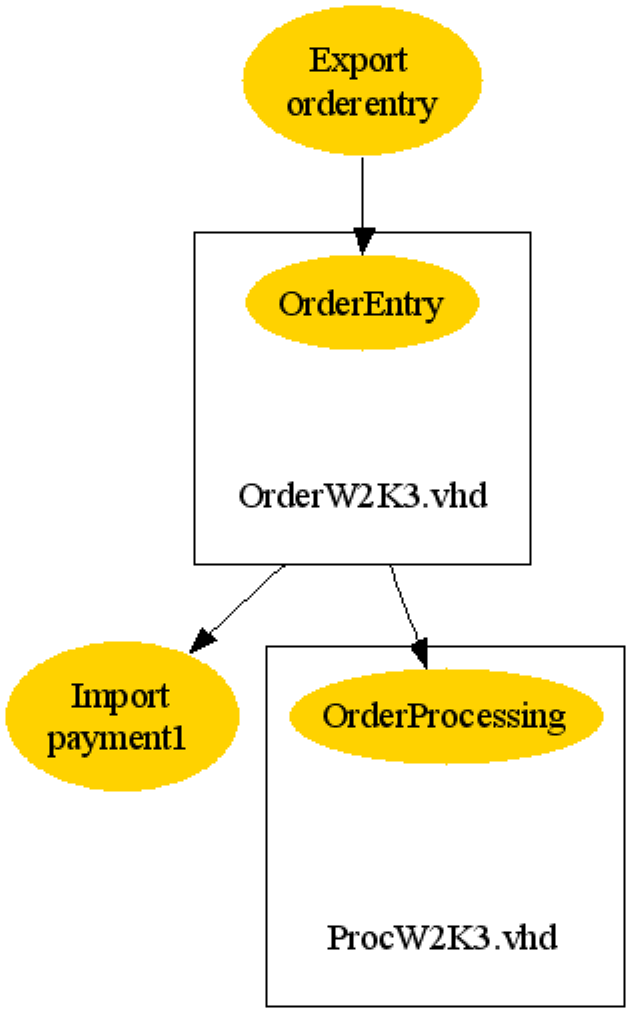
A Programming Problem

- As input, we are given:
  - Disk images for each server role in a multi-tier website
  - Addresses for external services we depend upon
  - Addresses for the services we are to export

- A human operator could run this application by:
  - Provisioning (virtual) machines
  - Configuring machines with suitable addresses
  - Monitoring the machines and taking remedial actions

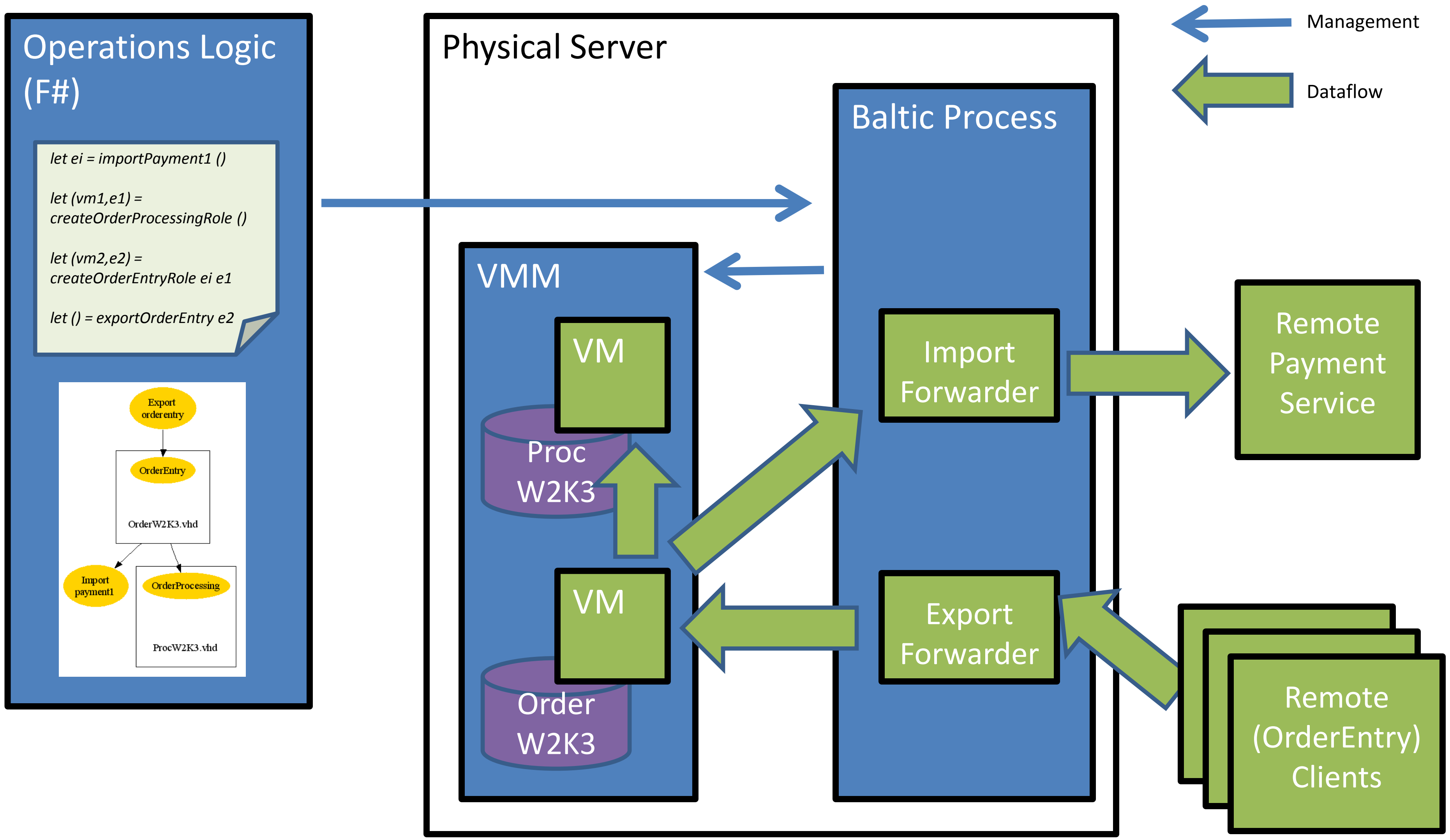
- The problem is to automate these tasks as **operations logic**
  - The standard solution is to use low-level scripting
  - Our solution (Baltic) is to use functional programming (F#, a dialect of ML), and write code to manipulate **abstract states** of the application

Abstract States are Call Graphs



```
let ei = importPayment1 ()
let (vm1,e1) = createOrderProcessingRole ()
let (vm2,e2) = createOrderEntryRole ei e1
let () = exportOrderEntry e2
```

Concrete Implementation



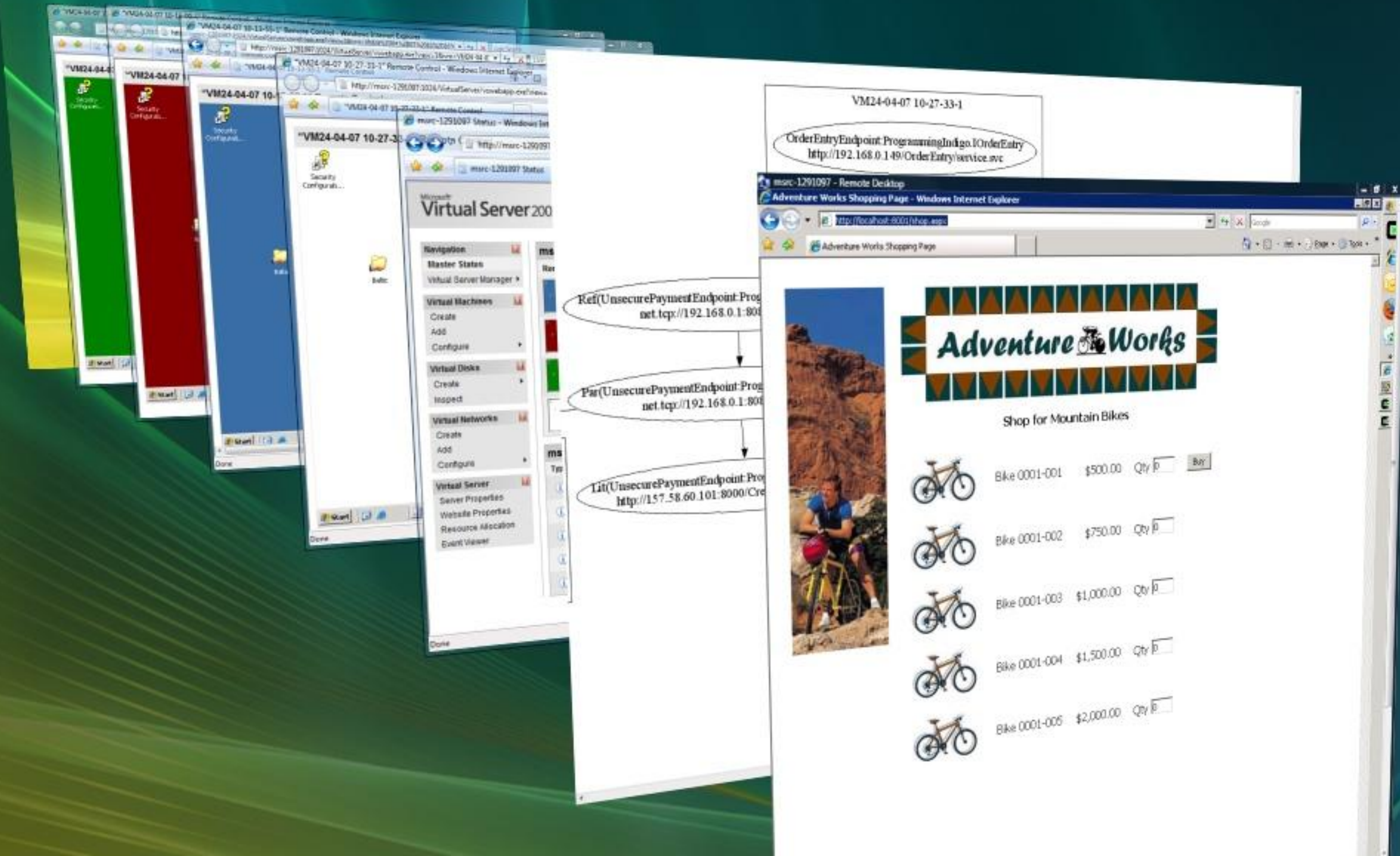
```
let ei = importPayment1 ()

let (vm1,e1) =
createOrderProcessingRole ()

let (vm2,e2) =
createOrderEntryRole ei e1

let () = exportOrderEntry e2
```

4 VMs, VMM, Baltic, External Client



VM24-04-07 10-27-33-1

OrderEntryEndpoint:ProgrammingIndigo.IOrderEntry
http://192.168.0.149/OrderEntry/service.svc

Ref(UnsecurePaymentEndpoint:Prog
net.tcp://192.168.0.1:80

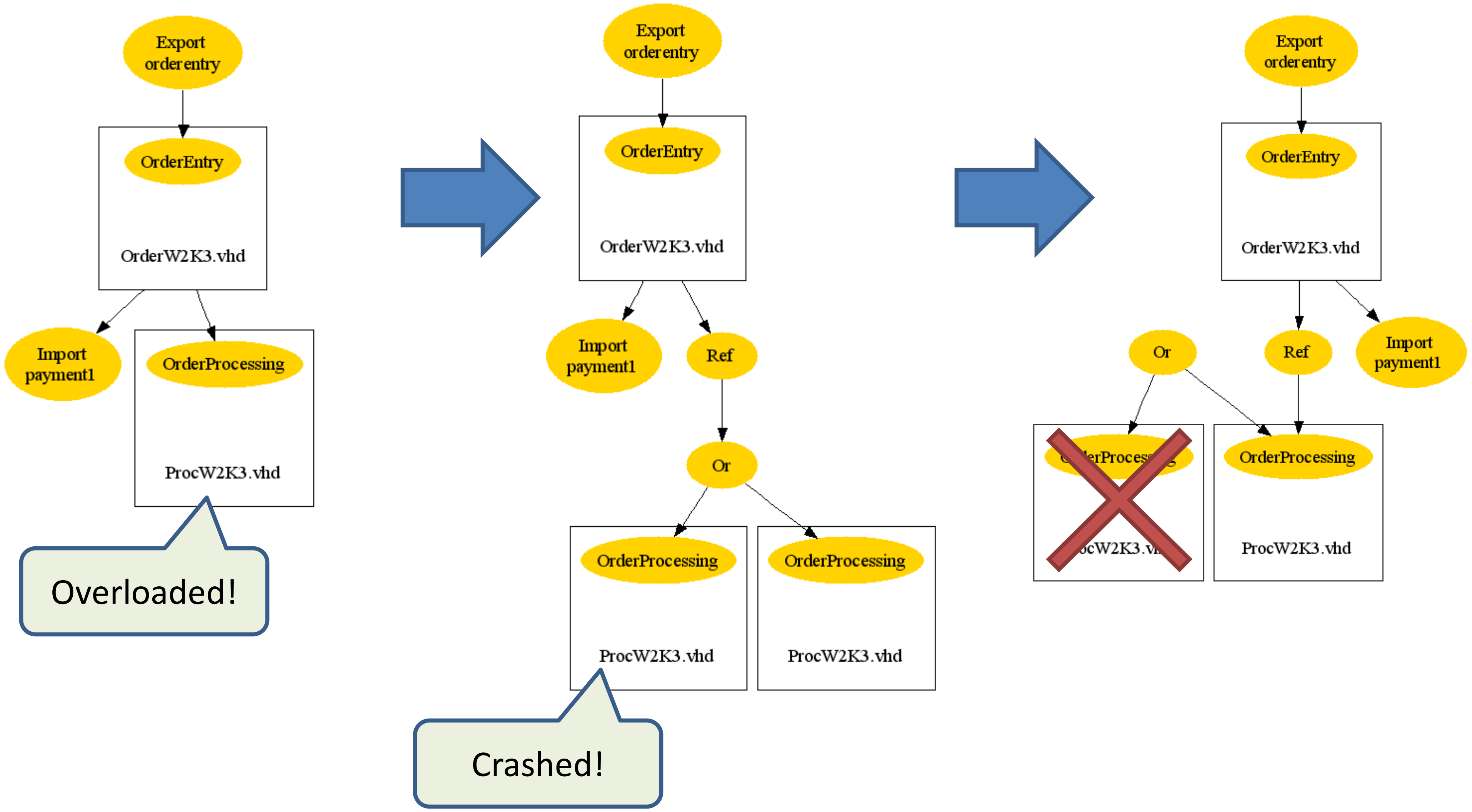
Par(UnsecurePaymentEndpoint:Prog
net.tcp://192.168.0.1:80

Lit(UnsecurePaymentEndpoint:Pro
http://157.58.60.101:8000/Cre

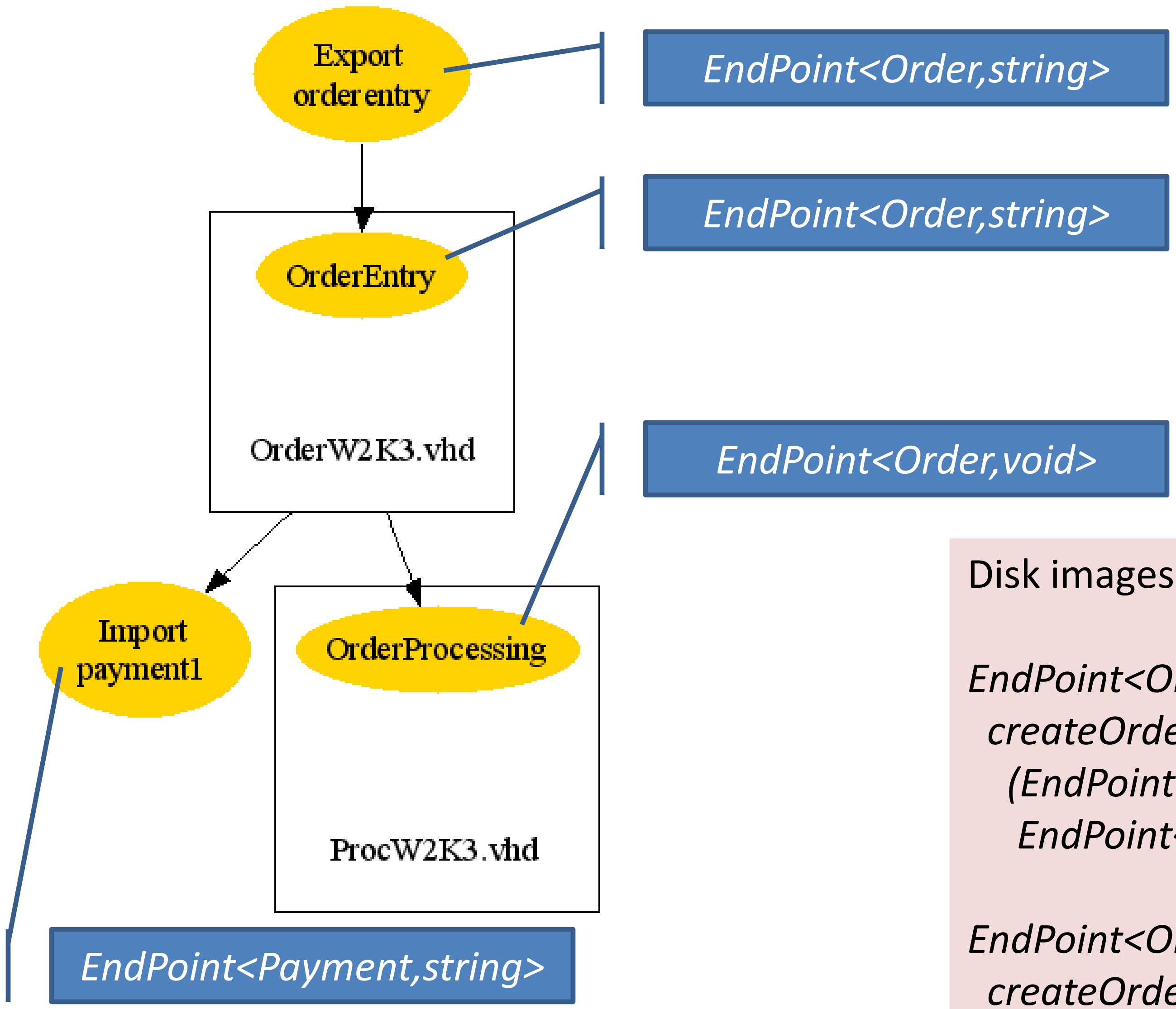
Shop for Mountain Bikes

Bike 0001-001	$500.00	Qty 0	Buy
Bike 0001-002	$750.00	Qty 0	
Bike 0001-003	$1,000.00	Qty 0	
Bike 0001-004	$1,500.00	Qty 0	
Bike 0001-005	$2,000.00	Qty 0	

States Evolve in Response to Events



Abstract States are Well-Typed



Disk images become typed functions:

EndPoint<Order,string>
createOrderEntryRole
(EndPoint<Payment,string> ep1,
EndPoint<Order,void> ep2)

EndPoint<Order,void>
createOrderProcessingRole ()

An Executable Formal Semantics

- We build a process model of a Baltic script in terms of a typed, concurrent, partitioned, lambda calculus
- To model VMs, processes look like: $a_1[P_1] | \ldots | a_n[P_n] \mid Q$
  - where $P_1, \ldots, P_n$, Q are expressions in the calculus
- To model VM snapshots, we allow partitions [P] as values
- We can execute this model to generate symbolic traces and pictorial call graphs, for debugging

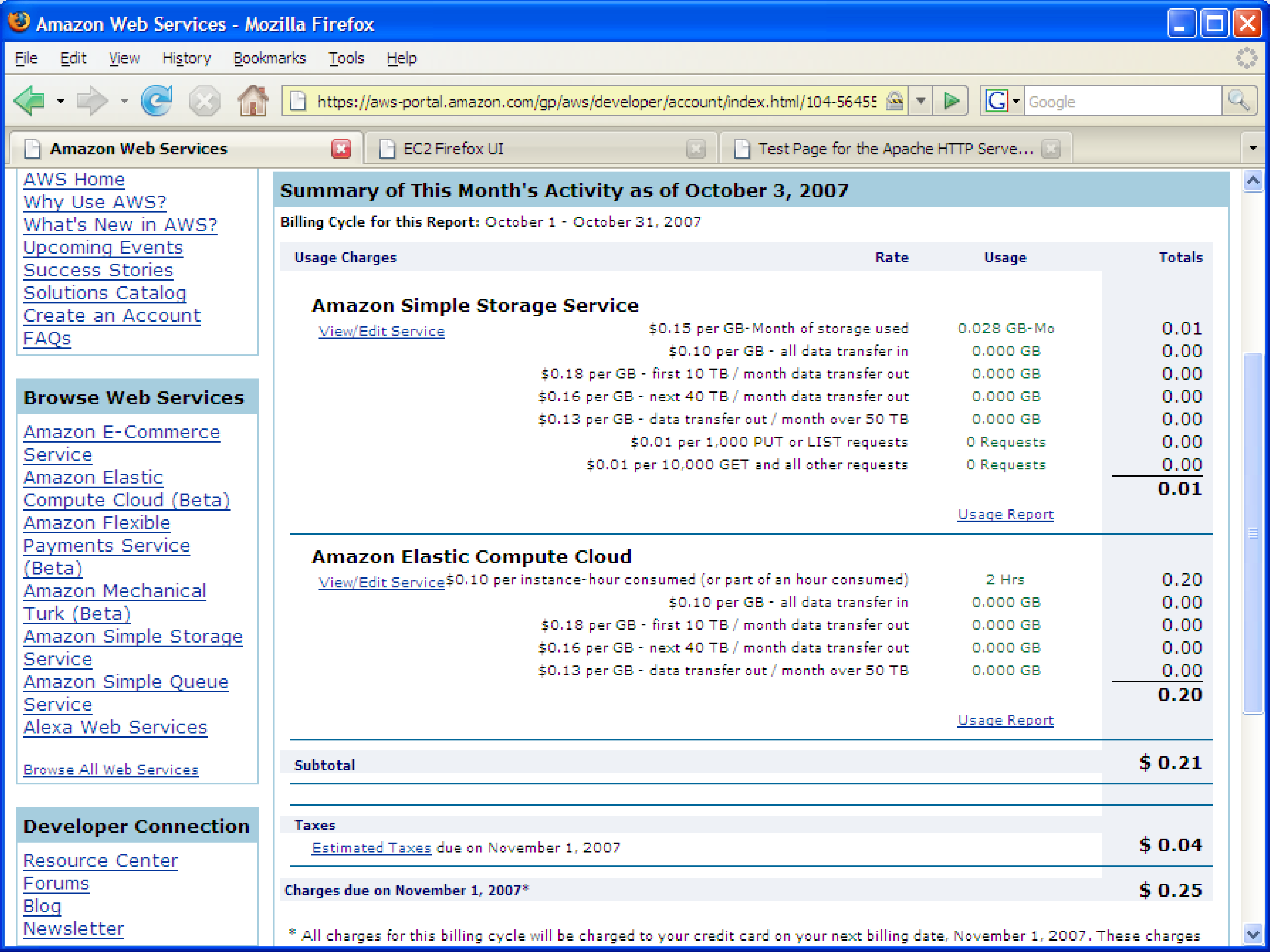
- We prove that (1) well-typed programs map to well-typed processes, and (2) process computation preserves typing
- **Well-typed processes never send ill-formed messages; hence, typing stops (some) interconnection errors**

Baltic – Summary

- Various trends are turning **datacentre management** into a programming problem
- How to code **business logic** is well understood, but how to code **operations logic** is becoming a Hot Topic

- We explore a **typed** approach to operations logic within a **functional** language (F#, a dialect of ML)
- As a first study, we focus on managing the operations of **service-oriented virtual machines** on a single host
- We develop the design, implementation, and formal semantics for a **call-graph**-based management API
- By assigning **function types** to whole **disk images**, we catch some errors statically, but much more could be done...

REVOLUTION

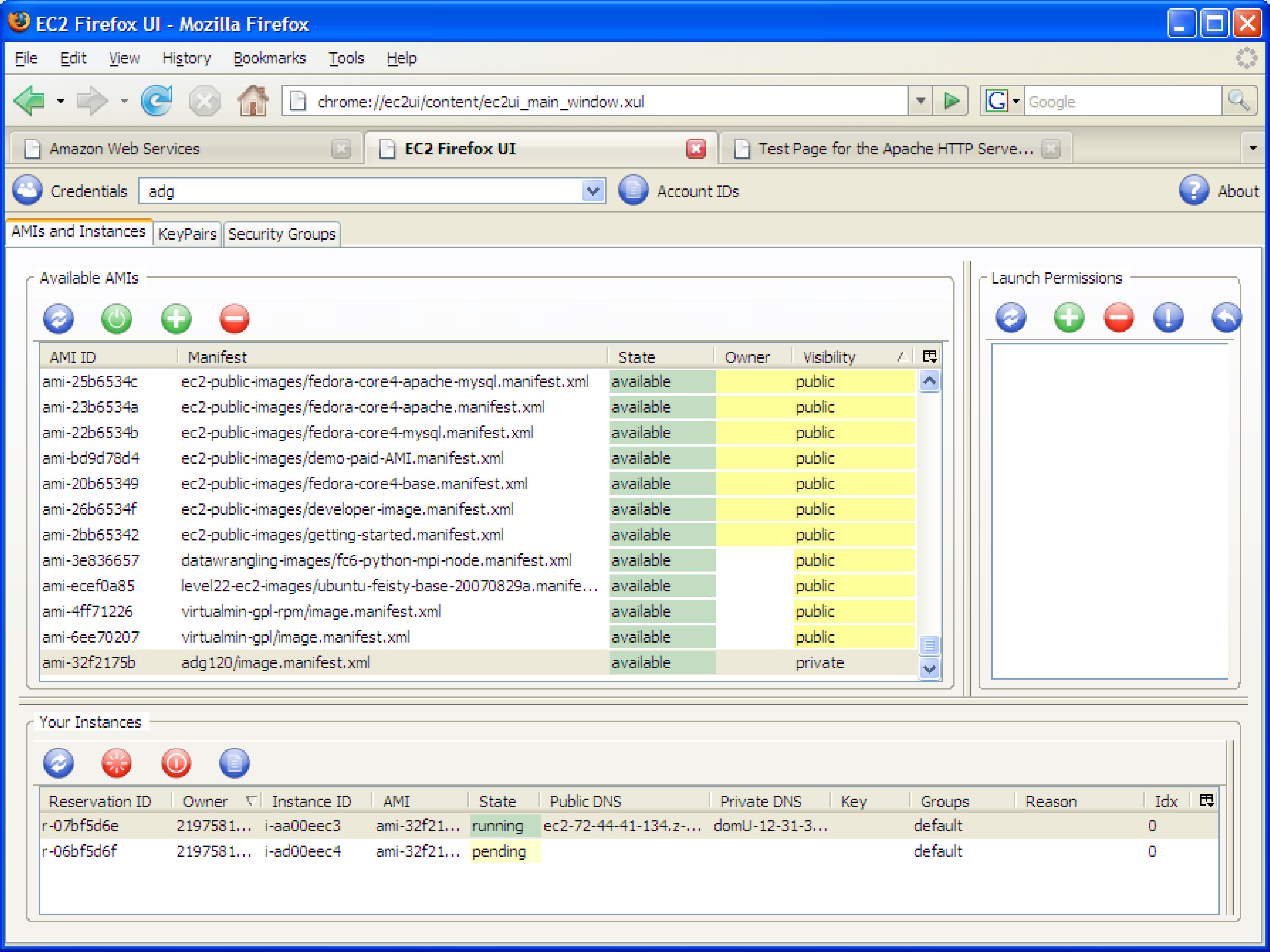

Amazon Web Services - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://aws-portal.amazon.com/gp/aws/developer/account/index.html/104-56455
Google
Amazon Web Services
EC2 Firefox UI
Test Page for the Apache HTTP Serve...
AWS Home
Why Use AWS?
What's New in AWS?
Upcoming Events
Success Stories
Solutions Catalog
Create an Account
FAQs
Browse Web Services
Amazon E-Commerce Service
Amazon Elastic Compute Cloud (Beta)
Amazon Flexible Payments Service (Beta)
Amazon Mechanical Turk (Beta)
Amazon Simple Storage Service
Amazon Simple Queue Service
Alexa Web Services
Browse All Web Services
Developer Connection
Resource Center
Forums
Blog
Newsletter

Summary of This Month's Activity as of October 3, 2007

Billing Cycle for this Report: October 1 - October 31, 2007

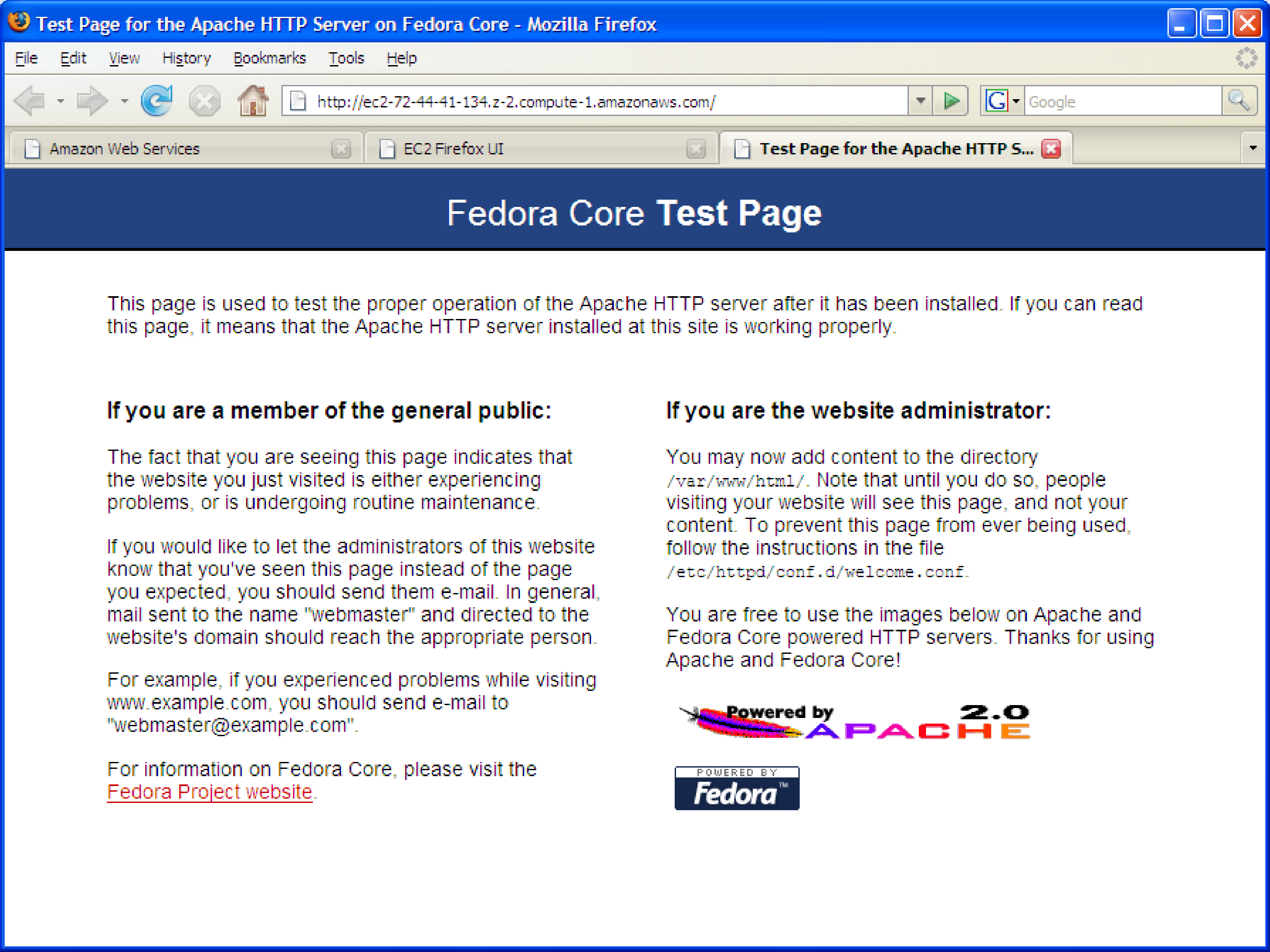
Usage Charges	Rate	Usage	Totals
Amazon Simple Storage Service			
View/Edit Service	$0.15 per GB-Month of storage used	0.028 GB-Mo	0.01
	$0.10 per GB - all data transfer in	0.000 GB	0.00
	$0.18 per GB - first 10 TB / month data transfer out	0.000 GB	0.00
	$0.16 per GB - next 40 TB / month data transfer out	0.000 GB	0.00
	$0.13 per GB - data transfer out / month over 50 TB	0.000 GB	0.00
	$0.01 per 1,000 PUT or LIST requests	0 Requests	0.00
	$0.01 per 10,000 GET and all other requests	0 Requests	0.00
			0.01
		Usage Report	
Amazon Elastic Compute Cloud			
View/Edit Service	$0.10 per instance-hour consumed (or part of an hour consumed)	2 Hrs	0.20
	$0.10 per GB - all data transfer in	0.000 GB	0.00
	$0.18 per GB - first 10 TB / month data transfer out	0.000 GB	0.00
	$0.16 per GB - next 40 TB / month data transfer out	0.000 GB	0.00
	$0.13 per GB - data transfer out / month over 50 TB	0.000 GB	0.00
			0.20
		Usage Report	
Subtotal			**$ 0.21**
Taxes			
Estimated Taxes due on November 1, 2007			**$ 0.04**
Charges due on November 1, 2007*			**$ 0.25**

* All charges for this billing cycle will be charged to your credit card on your next billing date, November 1, 2007. These charges


EC2 Firefox UI - Mozilla Firefox
File Edit View History Bookmarks Tools Help
chrome://ec2ui/content/ec2ui_main_window.xul
Google
Amazon Web Services
EC2 Firefox UI
Test Page for the Apache HTTP Serve...
Credentials
adg
Account IDs
About
AMIs and Instances
KeyPairs
Security Groups
Available AMIs
Launch Permissions
Your Instances

AMI ID	Manifest	State	Owner	Visibility
ami-25b6534c	ec2-public-images/fedora-core4-apache-mysql.manifest.xml	available		public
ami-23b6534a	ec2-public-images/fedora-core4-apache.manifest.xml	available		public
ami-22b6534b	ec2-public-images/fedora-core4-mysql.manifest.xml	available		public
ami-bd9d78d4	ec2-public-images/demo-paid-AMI.manifest.xml	available		public
ami-20b65349	ec2-public-images/fedora-core4-base.manifest.xml	available		public
ami-26b6534f	ec2-public-images/developer-image.manifest.xml	available		public
ami-2bb65342	ec2-public-images/getting-started.manifest.xml	available		public
ami-3e836657	datawrangling-images/fc6-python-mpi-node.manifest.xml	available		public
ami-ecef0a85	level22-ec2-images/ubuntu-feisty-base-20070829a.manife...	available		public
ami-4ff71226	virtualmin-gpl-rpm/image.manifest.xml	available		public
ami-6ee70207	virtualmin-gpl/image.manifest.xml	available		public
ami-32f2175b	adg120/image.manifest.xml	available		private

Reservation ID	Owner	Instance ID	AMI	State	Public DNS	Private DNS	Key	Groups	Reason	Idx
r-07bf5d6e	2197581...	i-aa00eec3	ami-32f21...	running	ec2-72-44-41-134.z-...	domU-12-31-3...		default		0
r-06bf5d6f	2197581...	i-ad00eec4	ami-32f21...	pending				default		0


Test Page for the Apache HTTP Server on Fedora Core - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://ec2-72-44-41-134.z-2.compute-1.amazonaws.com/
Google
Amazon Web Services
EC2 Firefox UI
Test Page for the Apache HTTP S...

Fedora Core **Test Page**

This page is used to test the proper operation of the Apache HTTP server after it has been installed. If you can read this page, it means that the Apache HTTP server installed at this site is working properly.

If you are a member of the general public:

The fact that you are seeing this page indicates that the website you just visited is either experiencing problems, or is undergoing routine maintenance.

If you would like to let the administrators of this website know that you've seen this page instead of the page you expected, you should send them e-mail. In general, mail sent to the name "webmaster" and directed to the website's domain should reach the appropriate person.

For example, if you experienced problems while visiting www.example.com, you should send e-mail to "webmaster@example.com".

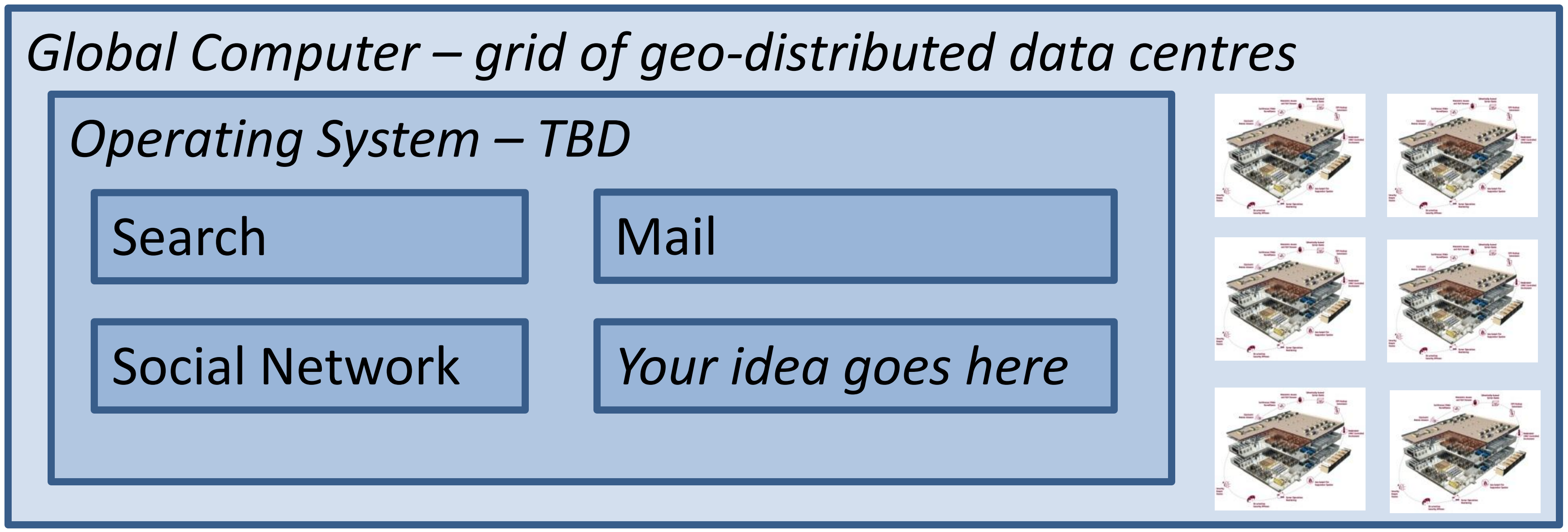
For information on Fedora Core, please visit the Fedora Project website.

If you are the website administrator:

You may now add content to the directory `/var/www/html/`. Note that until you do so, people visiting your website will see this page, and not your content. To prevent this page from ever being used, follow the instructions in the file `/etc/httpd/conf.d/welcome.conf`.

You are free to use the images below on Apache and Fedora Core powered HTTP servers. Thanks for using Apache and Fedora Core!

Stored-Program Global Computers



- EC2 is **imperfect**, has **research precursors**, but from a user perspective it's **revolutionary** – users can run their own programs in the cloud – eg Hadoop
- Like the 1940s, transition from fixed-program to stored-program computers
- We know the basic parts (eg VMs), but what are the OSs, the languages?
- What do we need to enable a single coder to write a global app?
- This is a Hot Question in industry today, and platforms like EC2 allow any grad student to have a go at implementing an answer

Resources for Part 1

- Hypervisor verification research at Saarbruecken
  http://www.microsoft.com/emic/verisoft.mspx
- Microsoft Virtual Server (free download)
  http://www.microsoft.com/windowsserversystem/virtualserver/
- Amazon EC2 (VMs at 10c per instance per hour)
  http://aws.amazon.com/ec2
- James Hamilton’s recent talks (source of my datacentre picture)
  http://research.microsoft.com/~jamesrh/

Thursday 25th 15:00-15:45

2

A Concurrent λ-Calculus with Refinement types

Declarative Datacentres, Part 2

Based on joint work with Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, and Sergio Maffeis

Why Study this Calculus?

- RCF is an assembly of standard parts, generalizing some ad hoc constructions in language-based security
  - **FPC** (Plotkin 1985, Gunter 1992) – core of ML and Haskell
  - Concurrency in style of the **pi-calculus** (Milner, Parrow, Walker 1989) but for a lambda-calculus (like 80s languages PFL, Poly/ML, CML)
  - Formal crypto is derivable by coding up **seals** (Morris 1973, Sumii and Pierce 2002), not primitive as in eg spi calculus(Abadi and Gordon, 1997)
  - Security specs via **assume/assert** (Floyd, Hoare, Dijkstra 1970s), generalizing eg correspondences (Woo and Lam 1992)
  - To check assertions statically, rely on dependent functions and pairs with subtyping (Cardelli 1988) and **refinement types** (Pfenning 1992, ...) aka **predicate subtyping** (as in PVS, and more recently Russell)
  - **Public/tainted kinds** to track data that may flow to or from the opponent, as in Cryptyc (Gordon, Jeffrey 2002)
- For experiment, there is a downloadable implementation F7

RCF PART 1:
SYNTAX AND SEMANTICS

The Core Language (FPC):

x, y, z	variable
$h ::=$	value constructor
inl	left constructor of sum type
inr	right constructor of sum type
fold	constructor of iso-recursive type
$M, N ::=$	value
x	variable
$()$	unit
fun $x \rightarrow A$	function (scope of x is A)
(M, N)	pair
$h\ M$	construction
$A, B ::=$	expression
M	value
$M\ N$	application
$M = N$	syntactic equality
let $x = A$ **in** B	let (scope of x is B)
let $(x, y) = M$ **in** A	pair split (scope of x, y is A)
match M **with** $h\ x \rightarrow A$ **else** B	constructor match (scope of x is A)

The Reduction Relation: $A \rightarrow A'$

$$(\mathbf{fun}\, x \rightarrow A)\ N \rightarrow A\{N/x\}$$

$$(\mathbf{let}\ (x_1, x_2) = (N_1, N_2)\ \mathbf{in}\ A) \rightarrow A\{N_1/x_1\}\{N_2/x_2\}$$

$$(\mathbf{match}\ M\ \mathbf{with}\ h\ x \rightarrow A\ \mathbf{else}\ B) \rightarrow \begin{cases} A\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ B & \text{otherwise} \end{cases}$$

$$M = N \rightarrow \begin{cases} \mathsf{inl}() & \text{if } M = N \\ \mathsf{inr}() & \text{otherwise} \end{cases}$$

$$\mathbf{let}\ x = M\ \mathbf{in}\ A \rightarrow A\{M/x\}$$

$$A \rightarrow A' \Rightarrow \mathbf{let}\ x = A\ \mathbf{in}\ B \rightarrow \mathbf{let}\ x = A'\ \mathbf{in}\ B$$

Exercise: These rules implement call-by-value functions and strict constructors. Adapt the semantics to call-by-name functions and non-strict constructors.
Exercise: Can pairs, constructions, and equality $M = N$ be encoded with just functions, that is, within the pure untyped λ-calculus?

Example: Booleans and Conditional Branching:

$\textbf{false} \triangleq \mathsf{inl}\ ()$
$\textbf{true} \triangleq \mathsf{inr}\ ()$
$\textbf{if}\ A\ \textbf{then}\ B\ \textbf{else}\ B' \triangleq$
$\quad \textbf{let}\ x = A\ \textbf{in match}\ x\ \textbf{with true}\ \to B\ \textbf{else match}\ x\ \textbf{with false}\ \to B'$

Exercise: Derive arithmetic, that is, value zero, functions succ, pred, and iszero.

Exercise: Derive list processing, that is, value nil, functions cons, hd, tl, and null.

Exercise: Write down an expression Ω that diverges, that is, $\Omega \to A_1 \to A_2 \to \ldots$.

Exercise: Derive a fixpoint function fix so that we can define recursive function definitions as follows: $\textbf{let rec}\ f x = A \triangleq \textbf{let}\ f = \mathsf{fix}\ (\textbf{fun}\, f \to \textbf{fun}\, x \to A)$.

The Heating Relation $A \Rrightarrow A'$:

Axioms $A \equiv A'$ are read as both $A \Rrightarrow A'$ and $A' \Rrightarrow A$.

$A \Rrightarrow A$

$A \Rrightarrow A''$ if $A \Rrightarrow A'$ and $A' \Rrightarrow A''$

$A \Rrightarrow A' \Rightarrow \textbf{let } x = A \textbf{ in } B \Rrightarrow \textbf{let } x = A' \textbf{ in } B$

$A \rightarrow A'$ if $A \Rrightarrow B, B \rightarrow B', B' \Rrightarrow A'$

Heating is an auxiliary relation; its purpose is to enable reductions, and to place every expression in a normal form, known as a *structure*.

Parallel Composition:

$A, B ::=$	expression
$\quad \ldots$	as before
$\quad A \Rsh B$	fork

$() \Rsh A \equiv A$
$(A \Rsh A') \Rsh A'' \equiv A \Rsh (A' \Rsh A'')$
$(A \Rsh A') \Rsh A'' \Rrightarrow (A' \Rsh A) \Rsh A''$
$\textbf{let}\ x = (A \Rsh A')\ \textbf{in}\ B \equiv A \Rsh (\textbf{let}\ x = A'\ \textbf{in}\ B)$

$A \Rrightarrow A' \Rightarrow (A \Rsh B) \Rrightarrow (A' \Rsh B)$
$A \Rrightarrow A' \Rightarrow (B \Rsh A) \Rrightarrow (B \Rsh A')$

$A \longrightarrow A' \Rightarrow (A \Rsh B) \longrightarrow (A' \Rsh B)$
$B \longrightarrow B' \Rightarrow (A \Rsh B) \longrightarrow (A \Rsh B')$

Exercise: Which parameter is passed to the function F by the following expression:
$\textbf{let}\ x = (1 \Rsh (2 \Rsh 3))\ \textbf{in}\ F x$

Name Generation:

$A, B ::=$	expression
$\quad \ldots$	as before
$\quad (\nu a)A$	fork

$A \Rrightarrow A' \Rightarrow (\nu a)A \Rrightarrow (\nu a)A'$
$a \notin fn(A') \Rightarrow A' \upharpoonright ((\nu a)A) \Rrightarrow (\nu a)(A' \upharpoonright A)$
$a \notin fn(A') \Rightarrow ((\nu a)A) \upharpoonright A' \Rrightarrow (\nu a)(A \upharpoonright A')$
$a \notin fn(B) \Rightarrow \textbf{let } x = (\nu a)A \textbf{ in } B \Rrightarrow (\nu a)\textbf{let } x = A \textbf{ in } B$

$A \rightarrow A' \Rightarrow (\nu a)A \rightarrow (\nu a)A'$

Exercise: For π-calculus experts, which common rules of structural equivalence for restriction are missing?
Exercise: What are the reductions of the following expression:
$\textbf{let } x = (\nu a)a \upharpoonright (\nu b)b \textbf{ in } F\ x$

Input and Output:

$A, B ::=$	expression
$\ldots$	as before
$a!M$	transmission of M on channel a
$a?$	receive message off channel

$$a!M \Rrightarrow a!M \Rsh ()$$
$$a!M \Rsh a? \rightarrow M$$

Exercise: What are the reductions of the expression: $a!3 \Rsh a? \Rsh a!5$

Exercise: What are the reductions of the expression: $a!3 \Rsh \textbf{let}\ x = a?\ \textbf{in}\ F\ x$

Exercise: What are the reductions of the expression: $a!\textbf{true} \Rsh a!\textbf{false}$

Example: Concurrent ML:

$(T)\mathsf{chan} \triangleq (T \to \mathsf{unit}) * (\mathsf{unit} \to T)$

$\mathsf{chan} \triangleq \mathbf{fun}\, x \to (\nu a)(\mathbf{fun}\, x \to a!x, \mathbf{fun}\, \_ \to a?)$	
$\mathsf{send} \triangleq \mathbf{fun}\, c\; x \to \mathbf{let}\ (s,r) = c\ \mathbf{in}\ s\; x$	send x on c
$\mathsf{recv} \triangleq \mathbf{fun}\, c \to \mathbf{let}\ (s,r) = c\ \mathbf{in}\ r\ ()$	block for x on c
$\mathsf{fork} \triangleq \mathbf{fun}\, f \to (f() \urcorner\!\!\!\!\rightarrow ())$	run f in parallel

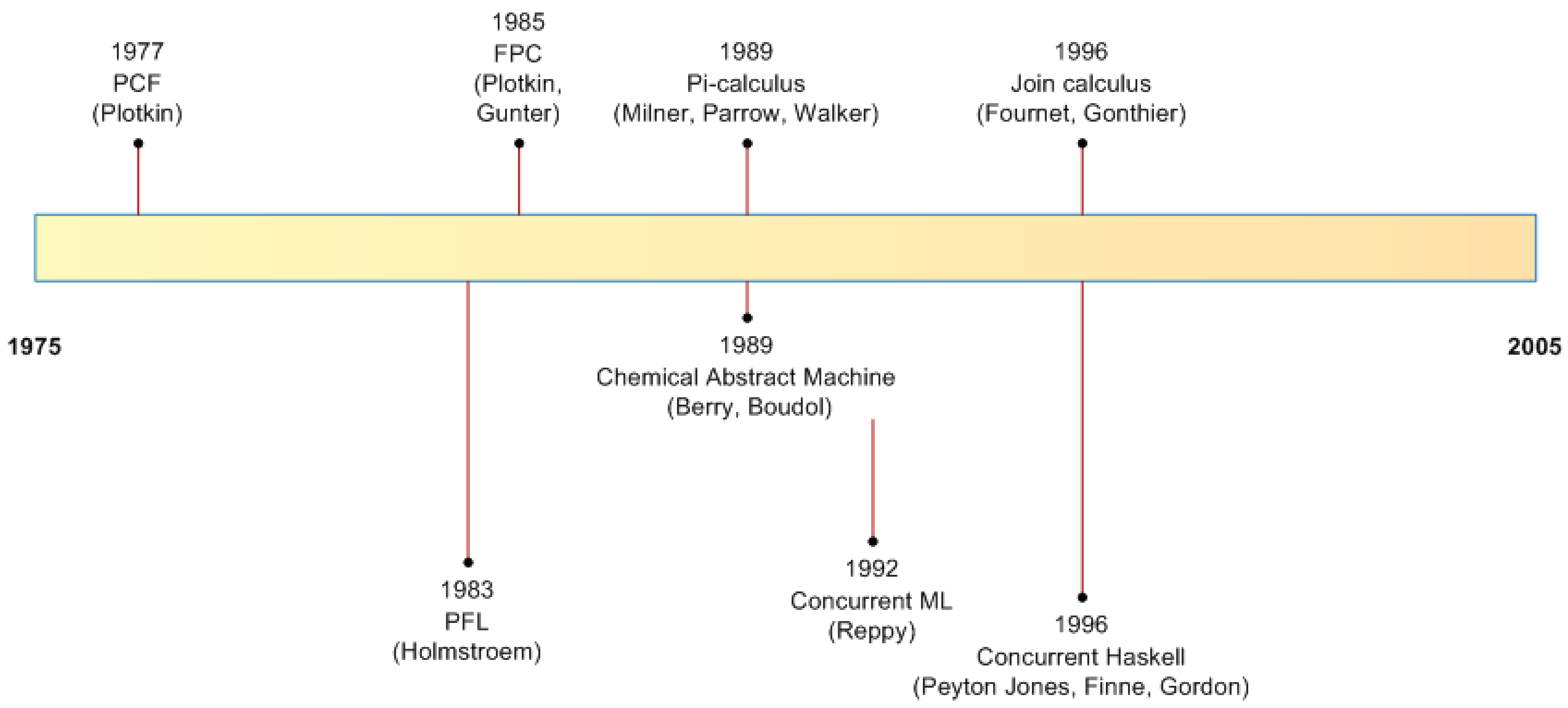
Example: Mutable State:

$(T)\mathbf{ref} \triangleq (T)\mathsf{chan}$

$\mathbf{ref}\ M \triangleq \mathbf{let}\ r = \mathsf{chan}\ \texttt{"r"}\ \mathbf{in}\ \mathsf{send}\ r\ M; r$	new reference to M
$\mathsf{deref}\ M \triangleq \mathbf{let}\ x = \mathsf{recv}\ M\ \mathbf{in}\ \mathsf{send}\ M\ x; x$	dereference M
$M := N \triangleq \mathbf{let}\ x = \mathsf{recv}\ M\ \mathbf{in}\ \mathsf{send}\ M\ N$	update M with N

Exercise: What are the reductions of the expression: $\mathbf{let}\ x = \mathbf{ref}\ 5\ \mathbf{in}\ x := 7$

Exercise: Encode concurrent while-programs within RCF.



FUNCTIONAL PROGRAMMING AND CONCURRENCY

Assume and Assert

- Suppose there is a global set of formulas, the **log**
- To evaluate **assume** *C*, add *C* to the log, and return ().
- To evaluate **assert** *C*, return ().
  - If *C* logically follows from the logged formulas, we say the assertion **succeeds**; otherwise, we say the assertion **fails**.
  - The log is only for specification purposes; it does not affect execution

- **assume** Foo(); **assert** Bar(); **assume** Foo()$\Rightarrow$Bar(); **assert** Bar()

- Our use of first-order logic predicates (like Foo()) generalizes conventional assertions (like **assert** i>0 in eg JML, Spec#)
  - Such predicates usefully represent security-related concepts like roles, permissions, events, compromises

A General Class of Logics:

$C ::= p(M_1,\ldots,M_n) \mid M = M' \mid C \wedge C' \mid C \vee C' \mid \neg C \mid C \Rightarrow C' \mid \forall x.C \mid \exists x.C$

$\{C_1,\ldots,C_n\} \vdash C$	deducibility relation

Assume and Assert:

$A, B ::=$	expression
$\ldots$	as before
assume C	assumption of formula C
assert C	assertion of formula C

assume $C \Rrightarrow$ **assume** $C \Rsh ()$

assert $C \rightarrow ()$

Exercise: What are the reductions of the expression:
assume Foo(); **assert** Bar(); **assume** Foo() $\Rightarrow$ Bar(); **assert** Bar()

Structures and Static Safety:

$e ::= M \mid MN \mid M = N \mid \textbf{let}\ (x,y) = M\ \textbf{in}\ B \mid$
$\quad \textbf{match}\ M\ \textbf{with}\ h\ x \rightarrow A\ \textbf{else}\ B \mid M? \mid \textbf{assert}\ C$

$\prod_{i \in 1..n} A_i \triangleq () \upharpoonright A_1 \upharpoonright \ldots \upharpoonright A_n$

$\mathscr{L} ::= \{\} \mid (\textbf{let}\ x = \mathscr{L}\ \textbf{in}\ B)$

$$\mathbf{S} ::= (\nu a_1)\ldots(\nu a_\ell)\left((\prod_{i \in 1..m} \textbf{assume}\ C_i) \upharpoonright (\prod_{j \in 1..n} c_j!M_j) \upharpoonright (\prod_{k \in 1..o} \mathscr{L}_k\{e_k\}) \right)$$

Let structure **S** be *statically safe* if and only if,
for all $k \in 1..o$ and C, if $e_k = \textbf{assert}\ C$ then $\{C_1, \ldots, C_m\} \vdash C$.

Lemma For every expression A, there is a structure **S** such that $A \Rrightarrow \mathbf{S}$.

Expression Safety:

Let expression A be *safe* if and only if,
for all A' and **S**, if $A \rightarrow^* A'$ and $A' \Rrightarrow \mathbf{S}$, then **S** is statically safe.

Friday 26th 11:15-12:00

3

APPLICATION OF RCF:
ACCESS CONTROL BY TYPING

Motivating Example: Access Control

- **Example:** untrusted code calling into a trusted library
- Trusted code expresses security policy with assumes and asserts
- Every policy violation causes an assertion failure
- **Idea:** rule out assertion failures statically

```
type facts = CanRead of string | CanDelete of string

let read file = assert(CanRead(file)); ...
let delete file = assert(CanDelete(file)); ...

let pwd = "C:/etc/password"
let tmp = "C:/temp/tempfile"

let _ = assume (CanDelete(tmp))
assume ∀x. CanDelete(x) → CanRead(x)
```

```
let untrusted() =
    let v1 = read tmp in // ok, by global assumption
    let v2 = read pwd in // assertion fails
    delete tmp; // ok
    delete pwd // assertion fails
```

Logging Dynamic Events

- Security policies often stated in terms of dynamic events such as role activations or data checks
- We mark such events by adding formulas to the log with **assume**

```
type facts = ... | PublicFile of string
let read file = assert(CanRead(file)); ...
let readme = "C:/public/README"

// Dynamic validation:
let publicfile f =
  if f = "C:/public/README" || ...
  then assume (PublicFile(f))
  else failwith "not a public file"

assume ∀x. PublicFile(x) ⇒ CanRead(x)
```

```
let untrusted() =
  let v2 = read readme in // assertion fails
  publicfile readme; // validate the filename
  let v3 = read readme in () // now, ok
```

Access Control by Typing

```
val read: (file: string {CanRead(file)}) → string
val delete: (file:string {CanDelete(file)}) → unit
val publicfile: (file : string) → unit { PublicFile(file) }
```

- Preconditions express access control requirements
- Postconditions express results of validation
- We typecheck partially trusted code to guarantee all preconditions (and hence all asserts) hold at run time
- To do so, we have an enhanced function type:
  - (x1: T1) {C1} → (x2:T2) {C2}
  - In RCF, these boil down to dependent functions plus refinement types
- Related work: eg types for stack inspection (Pottier, Skalka, Smith), Aura (Zdancevic et al)

RCF PART 2: TYPES FOR SAFETY

Starting Point: The Type System for FPC:

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

$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}} \qquad \frac{E \vdash M : T \quad E \vdash N : U}{E \vdash M = N : \mathsf{unit} + \mathsf{unit}}$$

$$\frac{E, x:T \vdash A : U}{E \vdash \textbf{fun}\, x \to A : (T \to U)} \qquad \frac{E \vdash M : (T \to U) \quad E \vdash N : T}{E \vdash M\ N : U}$$

$$\frac{E \vdash M : T \quad E \vdash N : U}{E \vdash (M,N) : (T \times U)} \qquad \frac{E \vdash M : (T \times U) \quad E, x:T, y:U \vdash A : V}{E \vdash \textbf{let}\ (x,y) = M\ \textbf{in}\ A : V}$$

$$\frac{h : (T,U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\ M : U} \qquad \frac{E \vdash M : T \quad h : (H,T) \quad E, x:H \vdash A : U \quad E \vdash B : U}{E \vdash \textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B : U}$$

$\mathsf{inl}{:}(T, T+U) \quad \mathsf{inr}{:}(U, T+U) \quad \mathsf{fold}{:}(T\{\mu\alpha.T/\alpha\}, \mu\alpha.T)$

Exercise: Write types of Booleans, numbers, and lists.

Exercise: Write a well-typed fixpoint combinator.

Three Steps Toward Safety by Typing

1. We include **refinement types** {*x* : *T* | *C*}, whose values are those of *T* that satisfy *C*
2. To exploit refinements, we add a judgment *E* |- *C*, meaning that *C* follows from the refinement types in *E*
3. To manage refinement formulas, we need (1) dependent versions of the function and pair types, and (2) subtyping

- A value of $\Pi x : T.\ U$ is a function M such that if N has type T, then $M\ N$ has type $U\{N/x\}$.
- A value of $\Sigma x : T.\ U$ is a pair (M,N) such that M has type T and N has type $U\{M/x\}$.
- If $A : T$ and $T <: U$ then $A : U$.

Syntax of RCF Types:

$H, T, U, V ::=$	type
unit	unit type
$\Pi x : T.\ U$	dependent function type (scope of x is U)
$\Sigma x : T.\ U$	dependent pair type (scope of x is U)
$T + U$	disjoint sum type
$\mu \alpha . T$	iso-recursive type (scope of α is T)
α	iso-recursive type variable
$\{x : T \mid C\}$	refinement type (scope of x is C)

$\{C\} \triangleq \{\_ : \mathsf{unit} \mid C\}$	ok-type
$\mathsf{bool} \triangleq \mathsf{unit} + \mathsf{unit}$	Boolean type

Starting Point: ~~The~~ A Dependent Type System for FPC:

$$\frac{E \vdash \diamond \quad (x:T) \in E}{E \vdash x : T} \qquad \frac{E \vdash A : T \quad E, x:T \vdash B : U}{E \vdash \mathbf{let}\ x = A\ \mathbf{in}\ B : U}\ \ x \notin fv\, U$$

$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}} \qquad \frac{E \vdash M : T \quad E \vdash N : U}{E \vdash M = N : \mathsf{unit} + \mathsf{unit}}$$

$$\frac{E, x:T \vdash A : U}{E \vdash \mathbf{fun}\, x \to A : \underset{\Pi x:T.U}{(\cancel{T \to U})}} \qquad \frac{E \vdash M : \overset{\Pi x:T.U}{(\cancel{T \to U})} \quad E \vdash N : T}{E \vdash M\, N : U[N/x]}$$

$$\frac{E \vdash M : T \quad E \vdash N : U[M/x]}{E \vdash (M,N) : \underset{\Sigma x:T,U}{(\cancel{T \times U})}} \qquad \frac{E \vdash M : \overset{\Sigma x:T.U}{(\cancel{T \times U})} \quad E, x:T, y:U \vdash A : V}{E \vdash \mathbf{let}\ (x,y) = M\ \mathbf{in}\ A : V}\ \ x,y \notin fv\, V$$

$$\frac{h : (T,U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\, M : U} \qquad \frac{E \vdash M : T \quad h : (H,T) \quad E, x:H \vdash A : U \quad E \vdash B : U}{E \vdash \mathbf{match}\ M\ \mathbf{with}\ h\, x \to A\ \mathbf{else}\ B : U}$$

$\mathsf{inl}{:}(T, T+U) \qquad \mathsf{inr}{:}(U, T+U) \qquad \mathsf{fold}{:}(T\{\mu\alpha.T/\alpha\}, \mu\alpha.T)$

Exercise: Write types of Booleans, numbers, and lists.

Exercise: Write a well-typed fixpoint combinator.

Rules for Formula Derivation:

$$\mathsf{forms}(E) \triangleq \begin{cases} \{C\{y/x\}\} \cup \mathsf{forms}(y:T) & \text{if } E = (y:\{x:T \mid C\}) \\ \mathsf{forms}(E_1) \cup \mathsf{forms}(E_2) & \text{if } E = (E_1, E_2) \\ \varnothing & \text{otherwise} \end{cases}$$

$$\frac{E \vdash \diamond \quad \mathit{fnfv}(C) \subseteq \mathit{dom}(E) \quad \mathsf{forms}(E) \vdash C}{E \vdash C}$$

Exercise: What is $\mathsf{forms}(E)$ if $E = x_1 : \{y_1 : \mathsf{int} \mid \mathsf{Even}(y_1)\}, x_2 : \{y_2 : \mathsf{int} \mid \mathsf{Odd}(x_1)\}$?

Exercise: A handy abbreviation is $\{C\} \triangleq \{\_ : \mathsf{unit} \mid C\}$, where $\_$ is fresh. What is $\mathsf{forms}(x : \{C\})$?

Assume and Assert

$$\frac{E \vdash \diamond \quad \mathit{fnfv}(C) \subseteq \mathit{dom}(E)}{E \vdash \mathbf{assume}\ C : \{\_ : \mathsf{unit} \mid C\}}$$

$$\frac{E \vdash C}{E \vdash \mathbf{assert}\ C : \mathsf{unit}}$$

Rules for Refinement Types:

$$\frac{E \vdash \{x : T \mid C\} \quad E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'}$$

$$\frac{E \vdash T <: T' \quad E, x : T \vdash C}{E \vdash T <: \{x : T' \mid C\}}$$

$$\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x : T \mid C\}}$$

Exercise: Derive the following subtyping rules:

$$\frac{E \vdash T <: T' \quad E, x : \{x : T \mid C\} \vdash C'}{E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}} \qquad \frac{E \vdash C \Rightarrow C'}{E \vdash \{C\} <: \{C'\}}$$

Standard Rules of Subtyping:

$$\frac{E \vdash A : T \quad E \vdash T <: T'}{E \vdash A : T'}$$

$$\frac{E \vdash \diamond}{E \vdash \mathsf{unit} <: \mathsf{unit}} \qquad \frac{E \vdash T' <: T \quad E, x : T' \vdash U <: U'}{E \vdash (\Pi x : T.\ U) <: (\Pi x : T'.\ U')}$$

$$\frac{E \vdash T <: T' \quad E, x : T \vdash U <: U'}{E \vdash (\Sigma x : T.\ U) <: (\Sigma x : T'.\ U')} \qquad \frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash (T + U) <: (T' + U')}$$

$$\frac{E \vdash \diamond \quad (\alpha <: \alpha') \in E}{E \vdash \alpha <: \alpha'} \qquad \frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \notin \mathit{fnfv}(T') \quad \alpha' \notin \mathit{fnfv}(T)}{E \vdash (\mu\alpha.T) <: (\mu\alpha'.T')}$$

Exercise: Prove that $E \vdash T <: T'$ is decidable, assuming an oracle for $E \vdash C$.

Exercise: (Hard.) Prove that $E \vdash T <: T'$ is transitive.

Exercise: Assume that $\vdash (x = 0) \Rightarrow \mathsf{Even}(x)$ but not the converse. Which are true?

$\vdash (\Pi x : \{x : \mathsf{int} \mid x = 0\}.\ \mathsf{bool}) <: (\Pi x : \{x : \mathsf{int} \mid \mathsf{Even}(x)\}.\ \mathsf{bool})$

$\vdash (\Sigma x : \{x : \mathsf{int} \mid x = 0\}.\ \mathsf{bool}) <: (\Sigma x : \{x : \mathsf{int} \mid \mathsf{Even}(x)\}.\ \mathsf{bool})$

$\vdash (\Sigma x : \{x : \mathsf{int} \mid x = 0\}.\ \mathsf{bool}) <: (\Pi x : \{x : \mathsf{int} \mid \mathsf{Even}(x)\}.\ \mathsf{bool})$

Rules for Restriction, I/O, and Parallel Composition:

$$\frac{E, a \updownarrow T \vdash A : U \quad a \notin fn(U)}{E \vdash (\nu a)A : U} \qquad \frac{E \vdash M : T \quad (a \updownarrow T) \in E}{E \vdash a!M : \mathsf{unit}} \qquad \frac{E \vdash \diamond \quad (a \updownarrow T) \in E}{E \vdash a? : T}$$

$$\frac{E, \_ : \{\overline{A_2}\} \vdash A_1 : T_1 \quad E, \_ : \{\overline{A_1}\} \vdash A_2 : T_2}{E \vdash (A_1 \upharpoonright A_2) : T_2}$$

$\overline{(\nu a)A} = (\exists a.\overline{A})$ $\qquad$ $\overline{A_1 \upharpoonright A_2} = (\overline{A_1} \wedge \overline{A_2})$

$\overline{\textbf{let } x = A_1 \textbf{ in } A_2} = \overline{A_1}$ $\qquad$ $\overline{\textbf{assume } C} = C$

$\overline{A} = \mathsf{True}$ $\quad$ if A matches no other rule

Exercise: Find types to typecheck the following code:

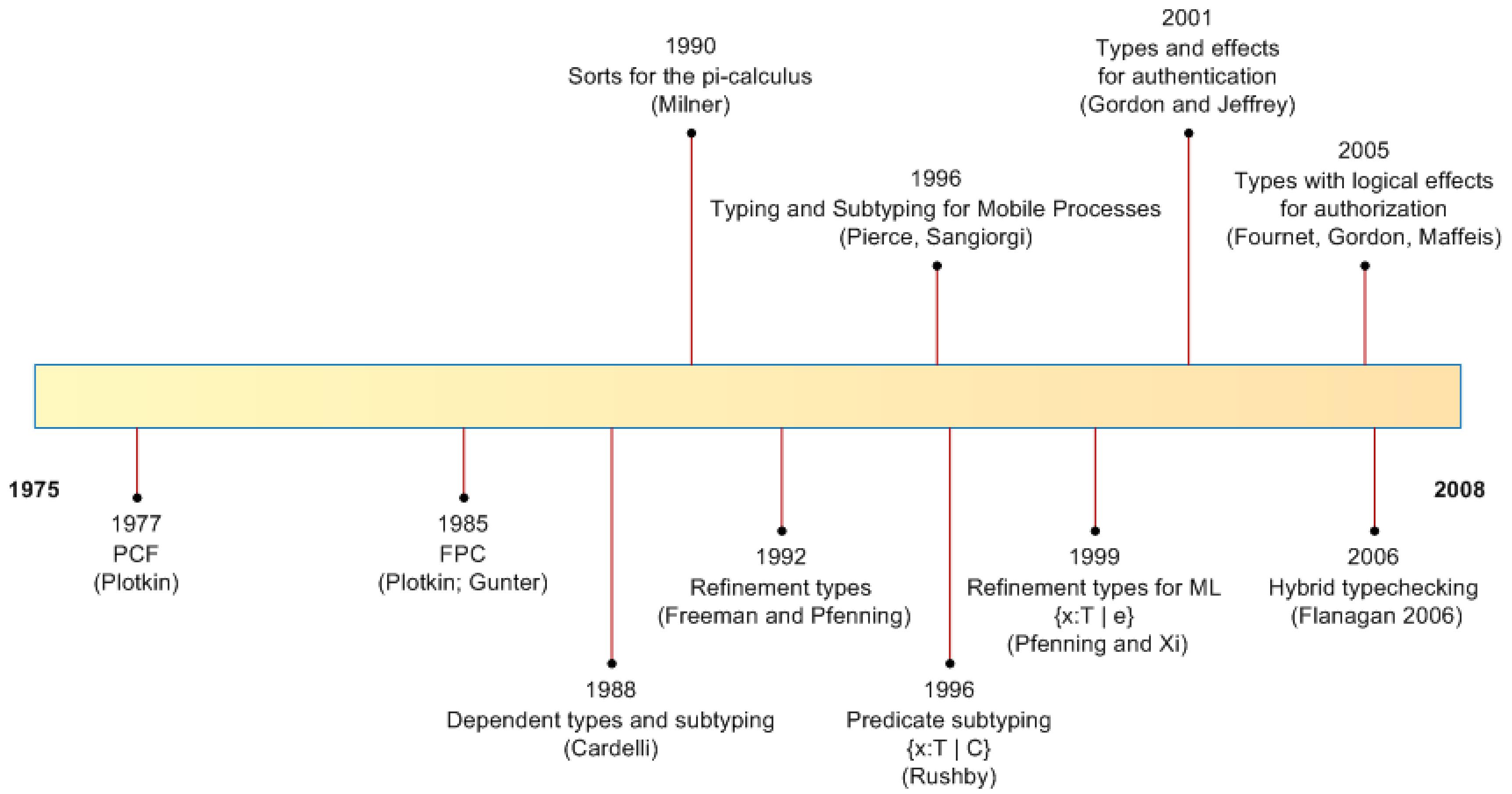
$$a!42 \upharpoonright (\nu c)((\textbf{let } x = a? \textbf{ in assume } \mathsf{Sent}(x) \upharpoonright c!x) \upharpoonright (\textbf{let } x = c? \textbf{ in assert } \mathsf{Sent}(x)))$$

Type System and Theorem

$E ::= x_1 : T_1, \ldots, x_n : T_n$	environment
$E \vdash \diamond$	E is syntactically well-formed
$E \vdash T$	in E, type T is syntactically well-formed
$E \vdash C$	formula C is derivable from E
$E \vdash T <: U$	in E, type T is a subtype of type U
$E \vdash A : T$	in E, expression A has type T

Lemma If $\varnothing \vdash \mathbf{S} : T$ then **S** is statically safe.
Lemma If $E \vdash A : T$ and $A \Rrightarrow A'$ then $E \vdash A' : T$.
Lemma If $E \vdash A : T$ and $A \rightarrow A'$ then $E \vdash A' : T$.

Theorem If $\varnothing \vdash A : T$ then A is safe.
(For any A' and **S** such that $A \rightarrow^* A'$ and $A' \Rrightarrow \mathbf{S}$
we need that **S** is statically safe.)



TYPE THEORIES BEHIND RCF

Summary of Part 2

- RCF supports functional programming a la ML and Haskell,
- concurrency in the style of process calculus,
- and refinement types allowing correctness properties to be stated in the style of dependent type theory.

- Next, we will develop applications of RCF, and describe our implementation http://research.microsoft.com/F7

- By embedding our theory of concurrency within an existing language, we obtain a programming environment at once

- There are many open questions around RCF: partitions, equivalences, type inference, mutable state, information flow

Friday 26th 12:15-13:00

4

Security Protocols and their Implementations

Declarative Datacentres, Part 3

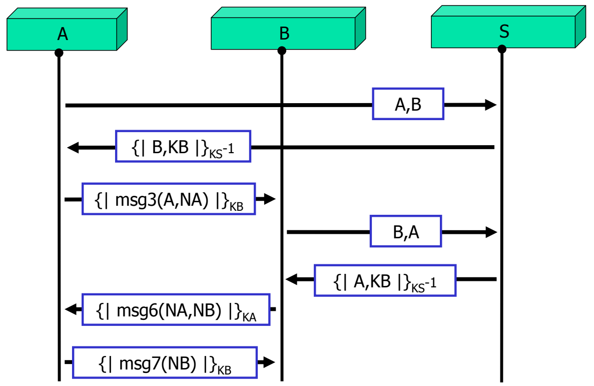
The Needham-Schroeder Problem

In **Using encryption for authentication in large networks of computers (CACM 1978)**, Needham and Schroeder didn't just initiate a field that led to widely deployed protocols like Kerberos, SSL, SSH, IPSec, etc.

They threw down a gauntlet.

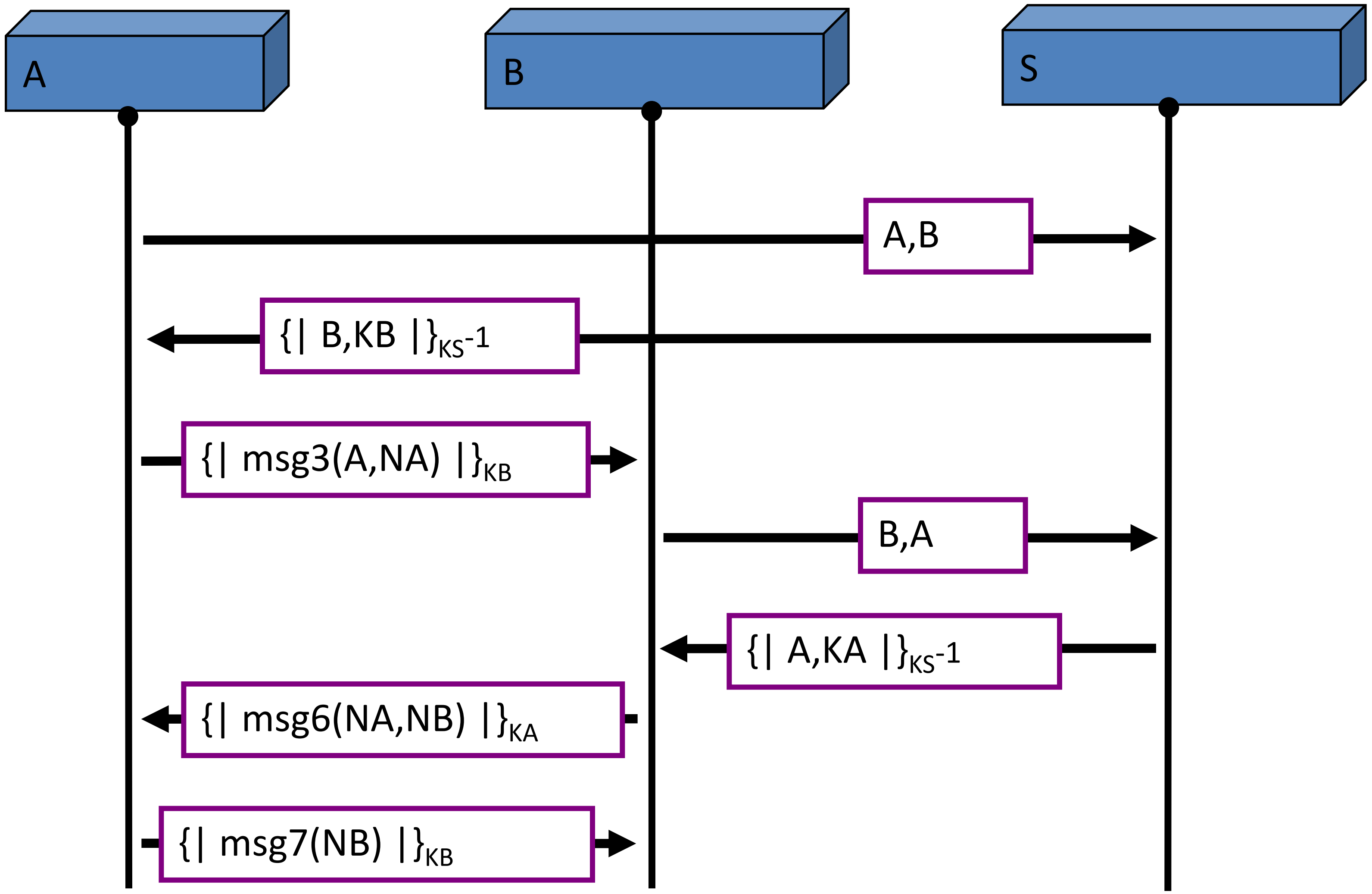
"Protocols such as those developed here are prone to extremely subtle errors that are unlikely to be detected in normal operation. The need for techniques to verify the correctness of such protocols is great, and we encourage those interested in such problems to consider this area."

The Needham-Schroeder public-key authentication protocol (CACM 1978)



Principal A initiates a session with principal B
S is a trusted server returning public-key certificates eg {| A,KA |}$_{KS}$-1
NA,NB serve as nonces to prove freshness of messages 6 and 7

The Needham-Schroeder public-key authentication protocol (CACM 1978)

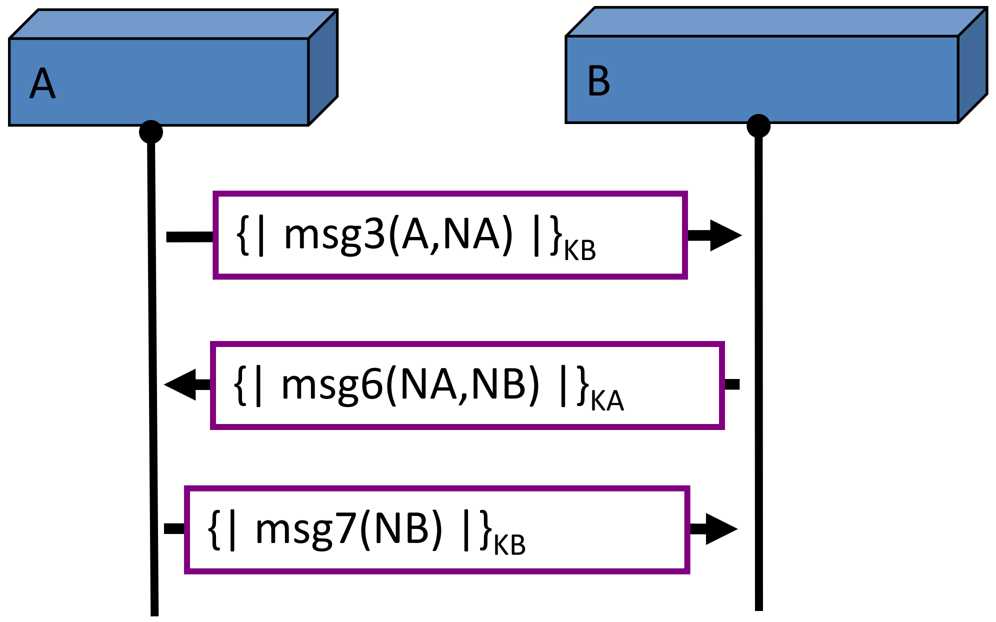


Principal A initiates a session with principal B

S is a trusted server returning public-key certificates eg {| A,KA |}$_{KS}$-1

NA,NB serve as nonces to prove freshness of messages 6 and 7

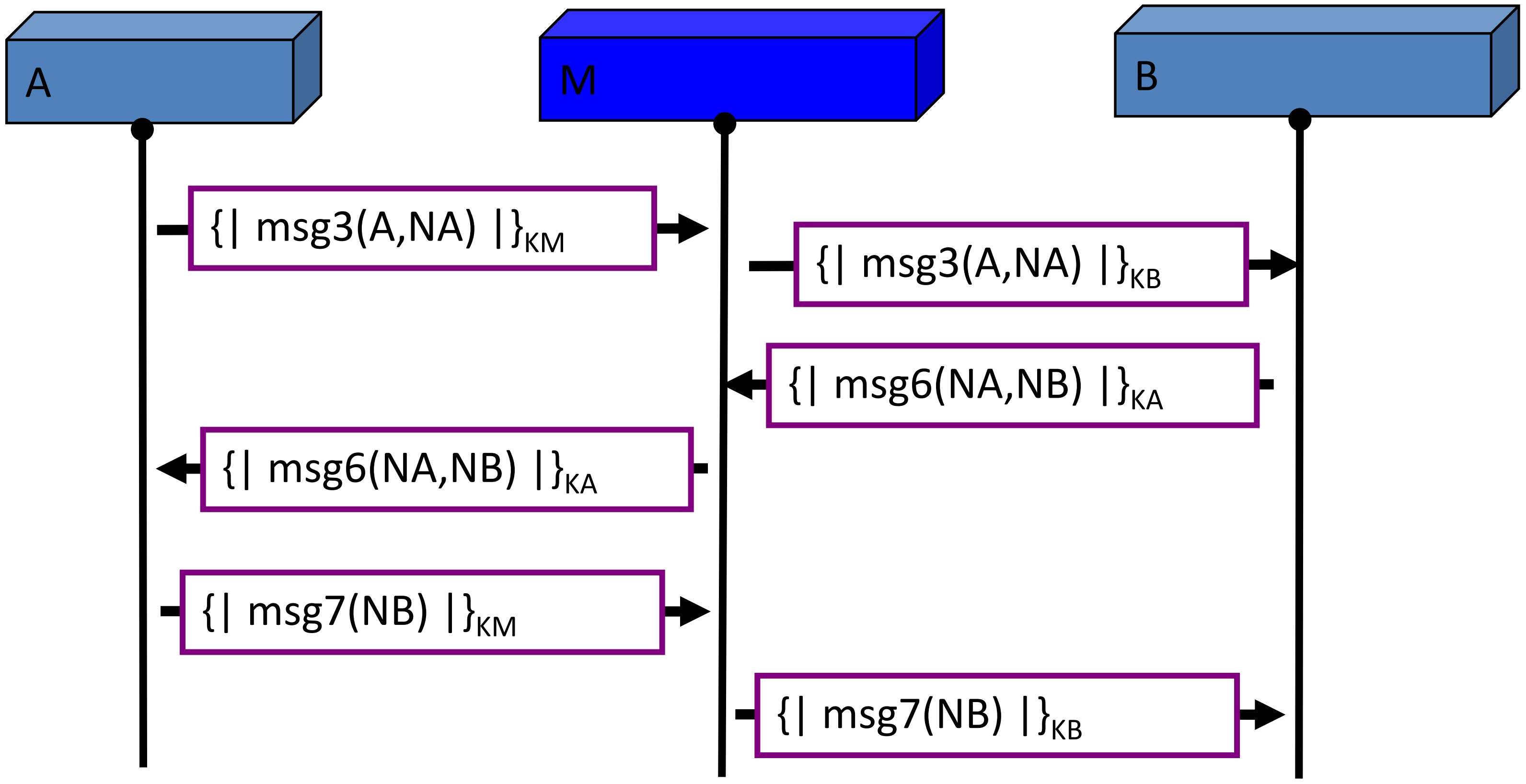
Assuming A knows KB and B knows KA, we get the core protocol:



More precisely, the goals of the protocol are:
- After receiving message 6, A believes NA,NB shared just with B
- After receiving message 7, B believes NA,NB shared just with A

If these goals are met, A and B can subsequently rely on keys derived from NA,NB to efficiently secure subsequent messages

A certified user M can play a man-in-the-middle attack (Lowe 1995)



This run shows a certified user M can violate the protocol goals:

- After receiving message 6, A believes NA,NB shared just with M
- After receiving message 7, B believes NA,NB shared just with A

(Writing in the 70s, Needham and Schroeder assumed certified users would not misbehave; we know now they do.)

Cryptographic Protocols

- Principals communicate over an untrusted network
  - Our focus is on Internet protocols, but same principles apply to banking, payment, and telephony protocols
- A range of security and privacy objectives is possible
  - Message confidentiality – against release of contents
  - Identity protection – against release of principal identities
  - Message authentication – against impersonated access
  - Message integrity – against tampering
  - Message correlation – that a response matches a request
  - Message freshness – against replays
- To achieve these goals, principals rely on applying cryptographic algorithms to parts of messages, but also on including message identifiers, nonces (unpredictable quantities), and timestamps

Informal Methods

Informal lists of prudent practices enumerate common patterns in the extensive record of flawed protocols, and formulate positive advice for avoiding each pattern.

(eg Abadi and Needham 1994, Anderson and Needham 1995)

The Explicitness Principle

Robust security is about explicitness. A cryptographic protocol should make any necessary naming, typing and freshness information explicit in its messages; designers must also be explicit about their starting assumptions and goals, as well as any algorithm properties which could be used in an attack.

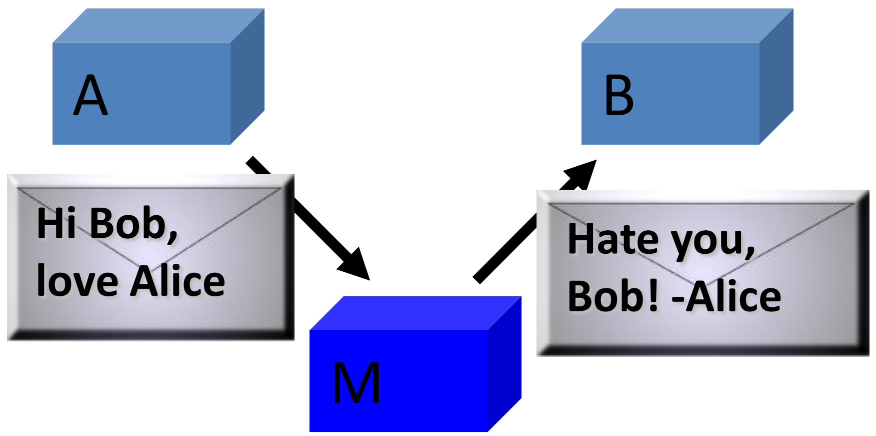
Anderson and Needham *Programming Satan's Computer* 1995

For instance, Lowe's famous fix of the Needham-Schroeder PK protocol makes explicit that message 6, {|NA,**B**,NB|}KA, is sent by B, who is not mentioned in the original version of the message.

Formal Methods

- Dolev&Yao first formalize N&S problem in early 80s
  - Shared key decryption: $\{ \{M\}_K \}_K$-1 = M
  - Public key decryption: $\{| \{| M |\}_{KA} |\}_{KA}$-1 = M
  - Their work now widely recognised, but at the time, no proof techniques, so little applied
- In 1987, Burrows, Abadi and Needham (BAN) propose a systematic rule-based logic for reasoning about protocols
  - If P believes that he shares a key K with Q, and sees the message M encrypted under K, then he will believe that Q once said M
  - If P believes that the message M is fresh, and also believes that Q once said M, then he will believe that Q believes M
  - Neither sound nor complete, but useful; hugely influential

A Potted History: 1978-2005



We assume that an intruder can interpose a computer on all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material. While this may seem an extreme view, it is the only safe one when designing authentication protocols.

Needham and Schroeder CACM (1978)

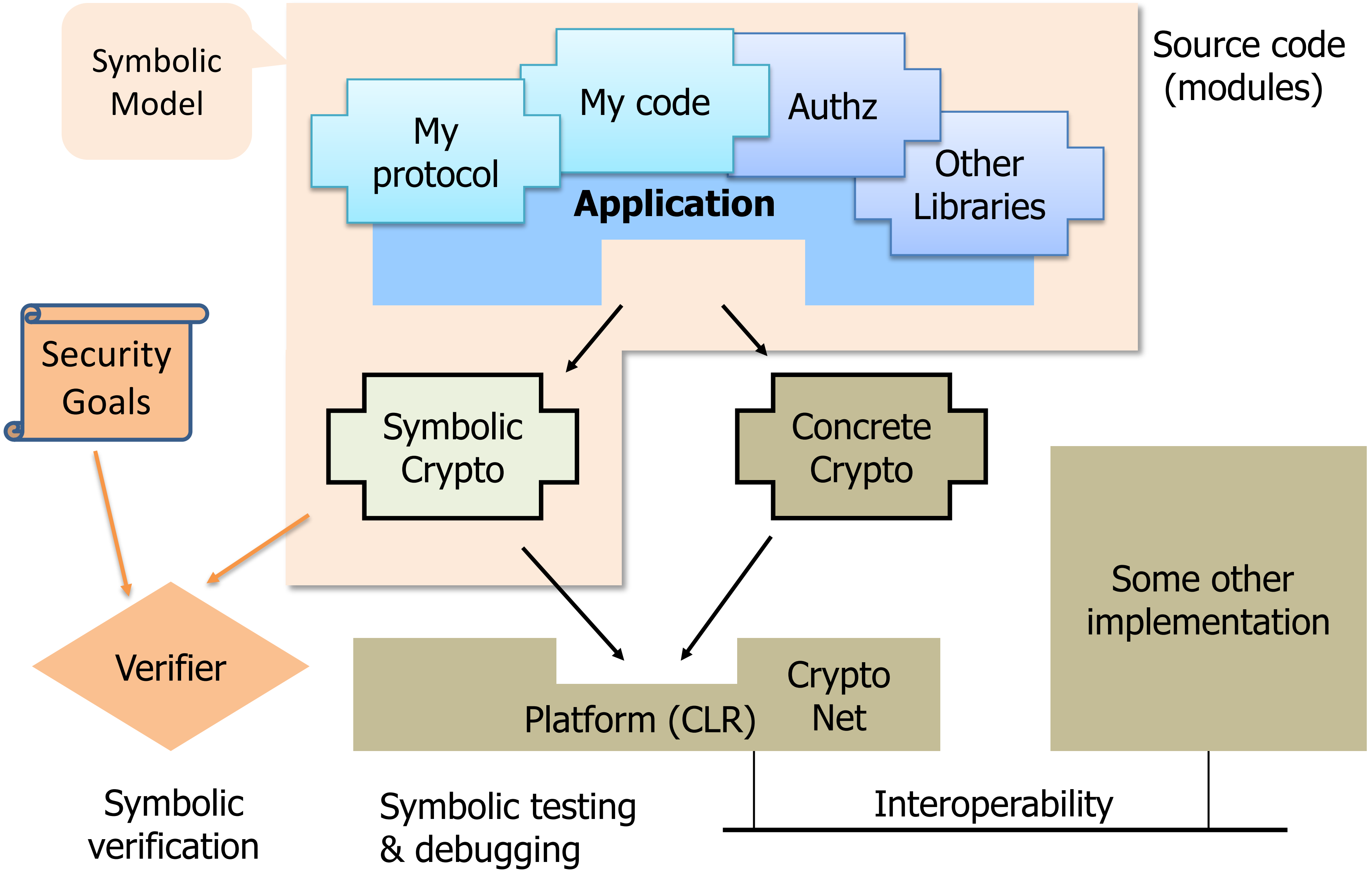
1978: N&S propose authentication protocols for "large networks of computers"
1981: Denning and Sacco find attack found on N&S symmetric key protocol
1983: Dolev and Yao first formalize secrecy properties wrt N&S threat model, using formal algebra
1987: Burrows, Abadi, Needham invent authentication logic; incomplete, but useful
1994: Hickman (Netscape) invents SSL; holes in v2, but v3 fixes these, very widely deployed
1994: Ylonen invents SSH; holes in v1, but v2 good, very widely deployed
1995: Abadi, Anderson, Needham, et al propose various informal "robustness principles"
1995: Lowe finds insider attack on N&S asymmetric protocol; rejuvenates interest in FMs
circa 2000: Several FMs for "D&Y problem": tradeoff between accuracy and approximation
circa 2005: Many FMs now developed; several deliver both accuracy and automation
2005: Cervesato et al find same insider attack as Lowe on proposed public-key Kerberos

Verifying Security Protocol Code

C	Goubault-Larrecq, Parrennes	2005	Csur|SPASS	FM	NSL (self-written)
Java	O'Shea	2006	LysaTool	FM	NSL, Otway-Rees (self-written)
F#	Bhargavan, Fournet, Gordon, Tse, Swamy	2006	FS2PV|PV	FM	WS protocols et al (self-written, but interoperable)
Java	Poll, Schubert	2007	JML	FSA	MIDP-SSH (independent)
F#	Bhargavan, Corin, Fournet	2007	FS2CV|CV	CM	Self-written examples

This table omits work on deriving code from models, and tools to check for insecure configurations of security protocols

One Source, Three Tasks



Source Language: F#

- F# is a dialect of ML running on the CLR developed by Don Syme at MSR Cambridge
  - First release around 2005; preview of product by 2008!
- An F# subset supports protocol programming, and model extraction
  - Simple formal semantics
  - Modular programming based on typed interfaces
  - Algebraic data types with pattern matching are useful for symbolic crypto and XML processing
- Still, few protocols are written in F#…

Pi Calculus Verifier: ProVerif

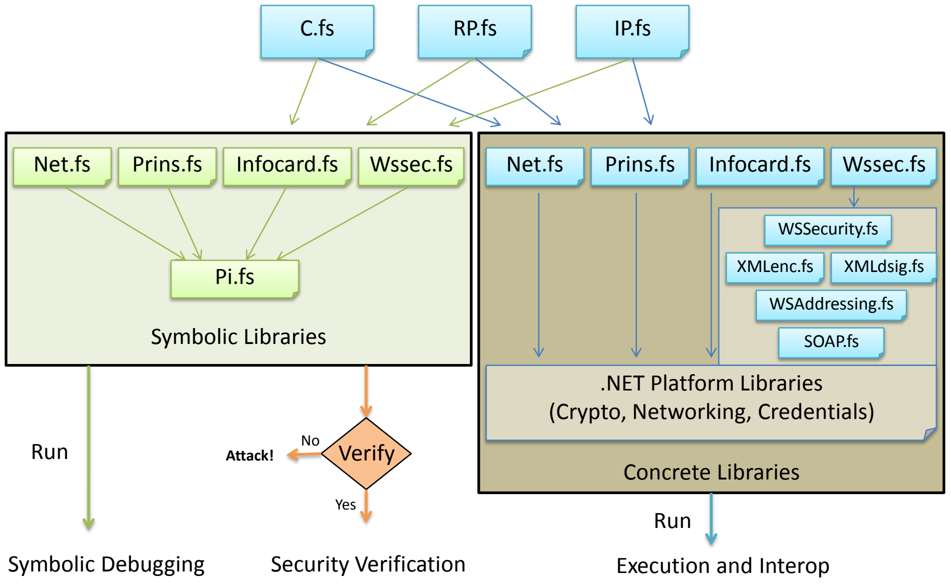
- ProVerif is an automated cryptographic protocol verifier developed by Bruno Blanchet
- What it can prove:
  - Secrecy, authenticity (correspondence properties)
  - Equivalences (e.g., secrecy properties)
- How it works:
  - Internal representation based on Horn clauses
  - Resolution-based algorithm, with clever selection rules
  - Attack reconstruction
- Automatic, but source must be tuned for efficient verification

B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. **CSFW 2001**
B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. **LICS 2005.**

A Motivation for Refinement Types

- Even with some aggressive abstraction, FS2PV|PV is hitting some long and unpredictable run times
- We think we may do better with source-level security types

A Reference InfoCard Implementation



Safety Results

Name	LOC	Crypto Ops	Auth	Secrecy	Verif Time
SelfIssued-SOAP	1410 (80)	9,3	A1-A3	S1,S2	38s
UserPassword-TLS	1426(96)	0,5,17,6	A1-A3	S1,S2	24m40s
UserPassword-SOAP	1429(99)	9,11,17,6	A1-A3	S1,S2	20m53s
UserCertificate-SOAP	1429(99)	13,7,11,6	A1-A3	S1-S3	66m21s
UserCertificate-SOAP-v	1429(99)	7,5,7,4	**A3 Fails!**	S1-S3	10s

MODELLING CRYPTOGRAPHIC ALGORITHMS IN RCF

Morris' Seal Abstraction

In our notation, a *seal* k for a type T is a pair of functions: the *seal function for* k, of type $T \rightarrow \mathsf{Un}$, and the *unseal function for* k, of type $\mathsf{Un} \rightarrow T$.

The seal function, when applied to M, wraps up its argument as a *sealed value*, informally written $\{M\}_k$ in this discussion. This is the only way to construct $\{M\}_k$.

The unseal function, when applied to $\{M\}_k$, unwraps its argument and returns M. This is the only way to retrieve M from $\{M\}_k$.

Sealed values are opaque; in particular, the seal k cannot be retrieved from $\{M\}_k$.

To implement a seal k, we maintain a list of pairs $[(M_1, a_1); \ldots ; (M_n, a_n)]$. The list records all the values M_i that have so far been sealed with k. Each a_i is a fresh name representing the sealed value $\{M_i\}_k$.

Seals within RCF

```
type α SealRef = ((α ∗ Un) list) ref
let seal: α SealRef → α → Un = fun s m →
   let state = deref s in match first (left m) state with
   | Some(a) → a
   | None →
      let a: Un = Pi.name "a" in
      s := ((m,a)::state); a
let unseal: α SealRef → Un → α = fun s a →
   let state = deref s in match first (right a) state with
   | Some(m) → m
   | None → failwith "not a sealed value"
let mkSeal (n:string) : α Seal =
   let s:α SealRef = ref [] in
      (seal s, unseal s)
```

Representing Crypto with Seals

```
type α hkey = HK of α pickled Seal
type hmac = HMAC of Un

let mkHKey ():α hkey = HK (mkSeal "hkey")
let hmacsha1 (HK key) text = HMAC (fst key text)
let hmacsha1Verify (HK key) text (HMAC h) =
   let x:α pickled = snd key h in
      if x = text then x else failwith "hmac verify failed"
```

Exercise: Implement shared key encryption, public-key encryption, and digital signatures using seals.

A TINY PROTOCOL

$A \rightarrow B$: M, hmac(sk$_{AB}$,M)

```
type event = Send of string
type message = (string * hmac) pickled

let make hk s = pickle (s,hmacsha1 hk (pickle s))

let check hk m =
  let s,h = unpickle m in
  let sv = hmacsha1Verify hk (pickle s) h in unpickle sv

let addr = http "http://localhost:7000/hmac"
let hk = mkHKey()

let client text = assume (Send(text));
                  let c  = connect addr in
                  send c (make hk text)

let server () =  let c = listen addr in
                 let text = check hk (recv c) in
                 assert (Send text)
```

TinySym

```
type 'a Seal = ('a -> Un) * (Un -> 'a)
val mkSeal: string -> 'a Seal
type 'a SealRef = (('a* Un) list) ref
```

PrimCrypto

```
val fork : (unit -> unit) -> unit
type  name
val name : string -> name
type 'a chan
val chan : string -> 'a chan
val send : 'a chan -> 'a -> unit
val recv : 'a chan -> ‘a
```

Pi

The **first part** of the attacker library represents our formal model, eg, the ability to compute with and communicate seals

```
type str
type bytes
type 'a pickled
type 'a hkey
type hmac
val str : string -> str
val istr : str -> string
val concat : bytes -> bytes -> bytes
val pickle : 'a -> 'a pickled
val unpickle : 'a pickled -> 'a
type 'a hkey = HK of 'a pickled Seal
type hmac = HMAC of Un
val mkHKey: unit -> 'a hkey
val hmacsha1: 'a hkey -> 'a pickled -> hmac
val hmacsha1Verify:
  'a hkey -> Un -> hmac -> 'a pickled...
```

Crypto

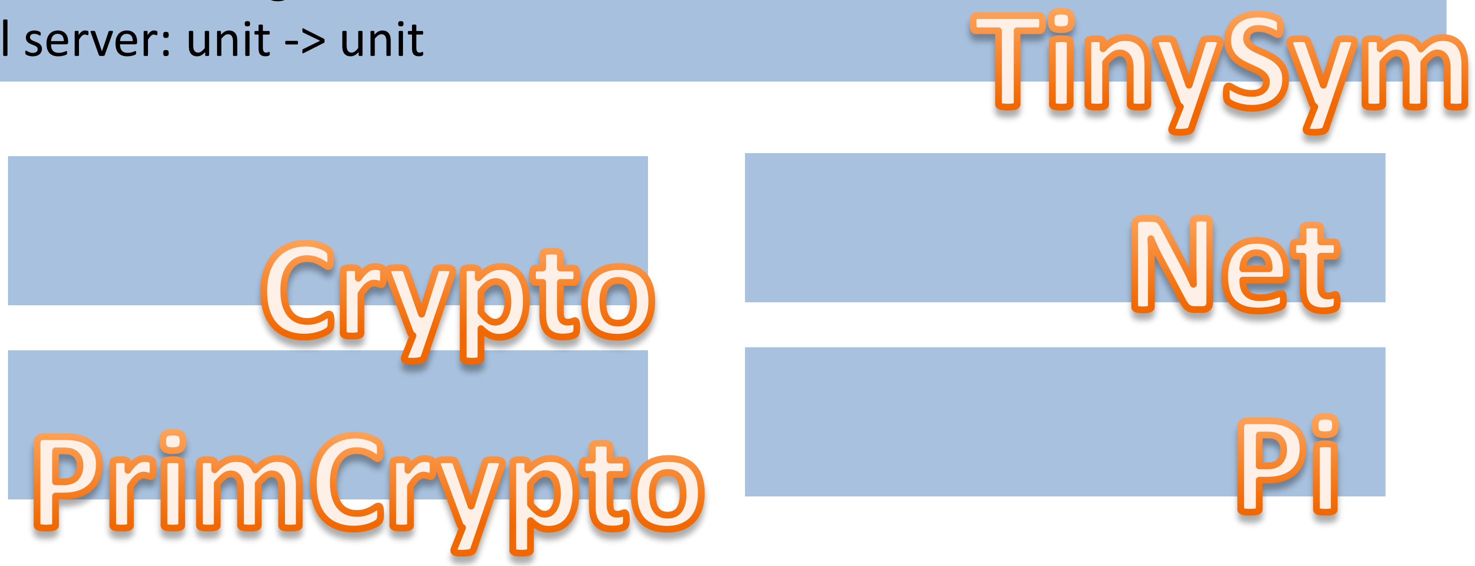
```
type ('a, 'b) addr
type ('a, 'b) conn
val http : string -> ('a, 'b) addr
val connect : ('a, 'b) addr -> ('a, 'b) conn
val listen : ('a, 'b) addr -> ('b, 'a) conn
val close : ('a, 'b) conn -> unit
val send : ('a, 'b) conn -> 'a pickled -> unit
val recv : ('a, 'b) conn -> 'b pickled
```

Net

The **second part** of the attacker library is the computing platform.
The concrete implementations use the actual platform (eg .NET).
The abstract implementations use PrimCrypto and Pi.

The problem: can any attacker break any assertion, given access to the following interfaces:

```
val addr : (string * hmac, unit) addr
val client: string -> unit
val server: unit -> unit
```



Formal Threat Model: Opponents and Robust Safety

A closed expression O is an *opponent* iff O contains no occurrence of **assert**.
A closed expression A is *robustly safe* iff application $O\ A$ is safe for all opponents O.

Hence, our problem is whether the expression (addr, client, server, . . .) robustly safe.

RCF III: TYPES FOR ROBUST SAFETY

Universal, Tainted, Public Types

To allow type-based reasoning about the opponent, we introduce a *universal type* Un of data known to the opponent. Somewhat arbitrarily, we define $\mathsf{Un} \triangleq \mathsf{unit}$. By definition, Un is type equivalent to (both a subtype and a supertype of) all of the following types: unit, $(\Pi x : \mathsf{Un}.\ \mathsf{Un})$, $(\Sigma x : \mathsf{Un}.\ \mathsf{Un})$, $(\mathsf{Un} + \mathsf{Un})$, and $(\mu\alpha.\mathsf{Un})$. Hence, we obtain *opponent typability*, that $O : \mathsf{Un}$ for all opponents O.

It is useful to characterize two *kinds* of type: *public types* (of data that may flow to the opponent) and *tainted types* (of data that may flow from the opponent).

Let a type T be *public* if and only if $T <: \mathsf{Un}$.
Let a type T be *tainted* if and only if $\mathsf{Un} <: T$.

Kinding Rules: $E \vdash T :: \nu$ for $\nu \in \{\mathbf{pub}, \mathbf{tnt}\}$

$$\frac{E \vdash \diamond}{E \vdash \mathsf{unit} :: \nu} \qquad \frac{E \vdash T :: \overline{\nu} \quad E, x : T \vdash U :: \nu}{E \vdash (\Pi x : T.\, U) :: \nu} \qquad \frac{E \vdash T :: \nu \quad E, x : T \vdash U :: \nu}{E \vdash (\Sigma x : T.\, U) :: \nu}$$

$$\frac{E \vdash T :: \nu \quad E \vdash U :: \nu}{E \vdash (T + U) :: \nu} \qquad \frac{E \vdash \diamond \quad (\alpha :: \nu) \in E}{E \vdash \alpha :: \nu} \qquad \frac{E, \alpha :: \nu \vdash T :: \nu}{E \vdash (\mu\alpha.T) :: \nu}$$

$$\frac{E \vdash \{x : T \mid C\} \quad E \vdash T :: \mathbf{pub}}{E \vdash \{x : T \mid C\} :: \mathbf{pub}} \qquad \frac{E \vdash T :: \mathbf{tnt} \quad E, x : T \vdash C}{E \vdash \{x : T \mid C\} :: \mathbf{tnt}}$$

Exercise: Which of the following are derivable?

(1) $\mathsf{int} \rightarrow \{y : \mathsf{int} \mid \mathsf{Even}(y)\} :: \mathbf{pub}$

(2) $\mathsf{int} \rightarrow \{y : \mathsf{int} \mid \mathsf{Even}(y)\} :: \mathbf{tnt}$

(3) $\{x : \mathsf{int} \mid \mathsf{Odd}(x)\} \rightarrow \mathsf{int} :: \mathbf{pub}$

(4) $(\{x : \mathsf{int} \mid \mathsf{Odd}(x)\} \rightarrow \mathsf{int}) \rightarrow \mathsf{int} :: \mathbf{pub}$

Additional Rule of Subtyping: $E \vdash T <: U$

$$\frac{E \vdash T :: \mathbf{pub} \quad E \vdash U :: \mathbf{tnt}}{E \vdash T <: U}$$

Lemma 1 (Public Tainted) *For all T and E:*

(1) $E \vdash T :: \mathbf{pub}$ *if and only if* $E \vdash T <: \mathsf{Un}$.

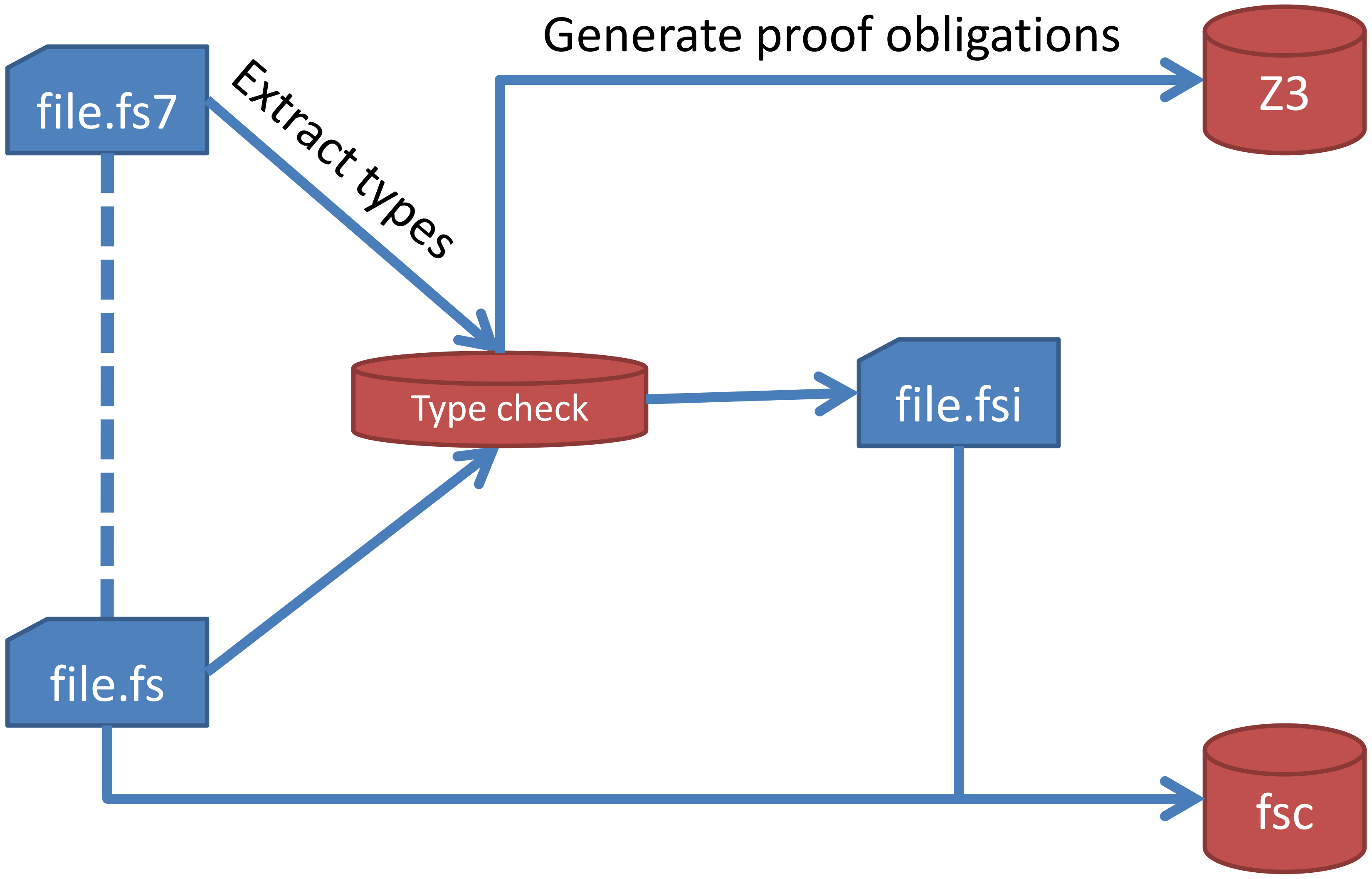
(2) $E \vdash T :: \mathbf{tnt}$ *if and only if* $E \vdash \mathsf{Un} <: T$.

Lemma 2 (Opponent Typability) *Suppose $E \vdash \diamond$. If O is an expression containing no **assert** such that $(a \updownarrow \mathsf{Un}) \in E$ for each name $a \in fn(O)$, and $(x : \mathsf{Un}) \in E$ for each variable $x \in fv(O)$, then $E \vdash O : \mathsf{Un}$.*

Theorem 1 (Robust Safety) *If $\varnothing \vdash A : \mathsf{Un}$ then A is robustly safe.*

Corollary: if $\varnothing \vdash A : T$ and $\varnothing \vdash T :: \mathbf{pub}$ then A is robustly safe.

The F7 Typechecker



- Our extended typechecker “compiles” .fs7 and .fs to .fsi
  - We typecheck .fs implementation against series of .fs7 interfaces
  - We kind-check .fs7 interfaces (every public value is indeed public)
  - We generate .fsi interfaces by erasure from .fs7
- We deal with a subset of F# larger than our core calculus
  - We treat many constructs as syntactic sugar (eg, records, patterns)
  - We support value- and type-polymorphic types
- We do some type inference
  - Plain F# types as usual
  - Refinement types typically require annotations
- We call Z3 on every non-trivial proof obligation
  - We generate type-based assumptions for data structures
  - Incomplete, but good enough for now

```
type prin = string
type event = Send of (prin ∗ prin ∗ string) | Leak of prin
type (;a:prin,b:prin) content = x:string{ Send(a,b,x) }
type message = (prin ∗ prin ∗ string ∗ hmac) pickled

private val mkContentKey: a:prin → b:prin → ((;a,b)content) hkey
private val hkDb: (prin∗prin, a:prin ∗ b:prin ∗ k:(;a,b) content hkey) Db.t
val genKey: prin → prin → unit
private val getKey: a: string → b:string → ((;a,b) content) hkey

assume ∀a,b,x. ( Leak(a) ) ⇒ Send(a,b,x)
val leak: a:prin → b:prin → (unit{ Leak(a) }) ∗ ((;a,b) content) hkey

val addr : (prin ∗ prin ∗ string ∗ hmac, unit) addr
private val check: b:prin → message → (a:prin ∗ (;a,b) content)
val server: string → unit

private val make: a:prin → b:prin → (;a,b) content → message
val client: prin → prin → string → unit
```

LittleSym

Evaluation so Far

Sample	.fs	.fs7	Time (s)	Z3 Obligations	Time per Obligation	Obligations per Line
Logs and Queries	37	16	2.8	6	0.47	0.16
MAC Protocol	40	12	2.5	3	0.83	0.08
Princs and Comp	48	26	3.1	12	0.26	0.25
Certificate Chains	61	21	3.65	19	0.19	0.31
Access Control	104	34	8.3	16	0.52	0.15
Flexible Signatures	167	52	14.6	28	0.52	0.17
Typed Libraries	440	146	12.1	12	1.01	0.03

- We can typecheck some non-trivial functions
- Not yet comparable with ProVerif

Limits of the Model

- As usual, formal security guarantees hold only within the boundaries of the model
  - We keep model and implementation in sync
  - We automatically deal with very precise models
  - We can precisely “program” the attacker model
- We verify our own implementations, not legacy code
- We trust the compiler, runtime, typechecker
  - Our method only finds bugs in the protocol code in F#
  - Independent certification is possible, but a separate problem
- We trust our symbolic model of cryptography
  - Partial computational soundness results may apply
  - Further verification tools may use a concrete model

Summary of Part 3

- We verify reference implementations of security protocols
- Our implementations run with both concrete and symbolic cryptographic libraries.
  - Concrete implementation for production and interop testing
  - Symbolic implementation for debugging and verification
- We develop our approach for protocols written in F#
  - We show its correctness for a range of security properties, against realistic classes of adversaries
  - Tool FS2PV compiles to applied pi calculus, and relies on ProVerif,
  - Tool F7 relies on refinement types, based on the RCF calculus
  - Case studies include WS security, CardSpace, SSL/TLS
- Some challenges
  - Establish computational guarantees (eg FS2CV)
  - Move from functional to imperative implementations
  - Move from F# to C

THE END