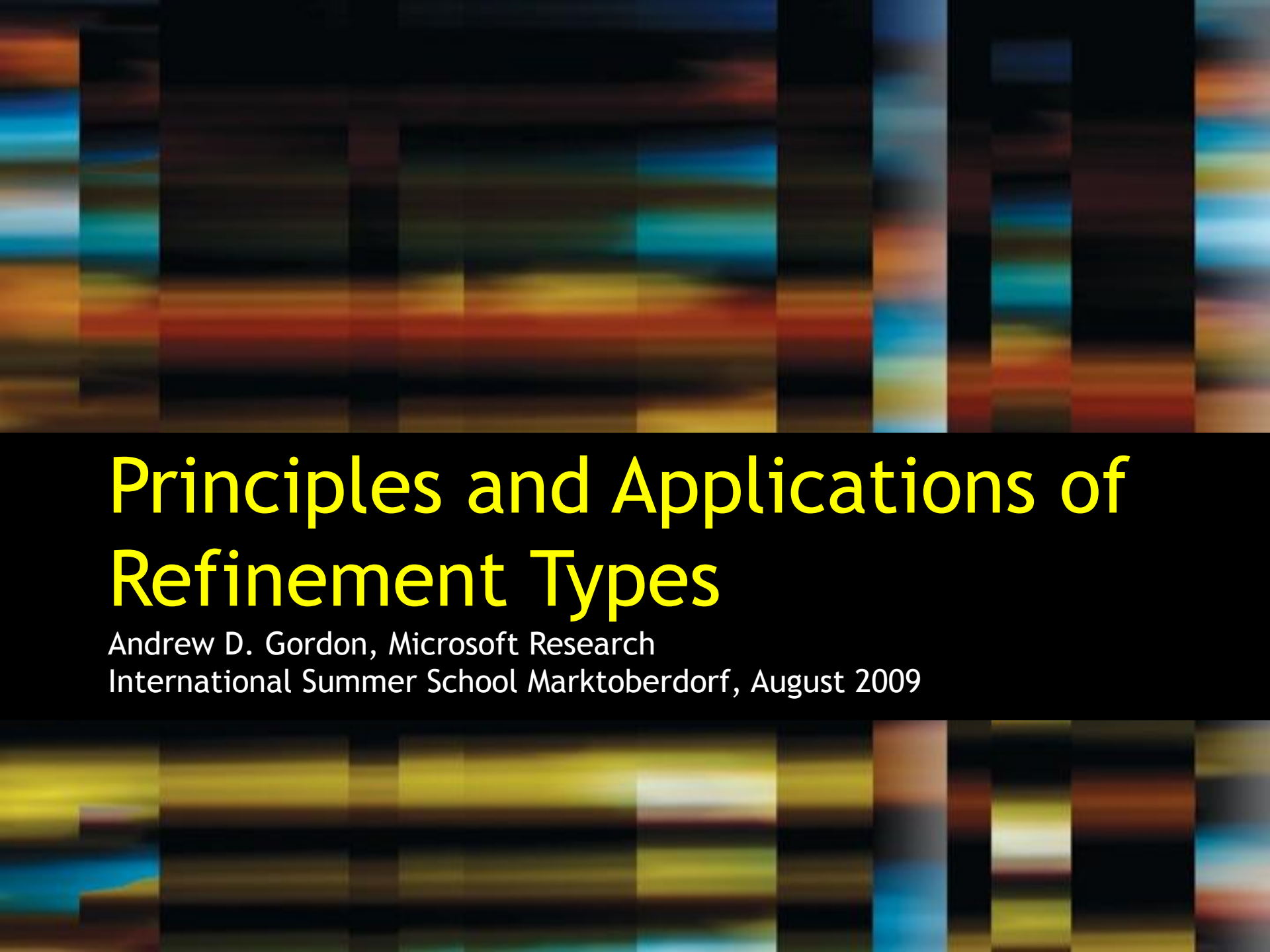


Principles and Applications of Refinement Types

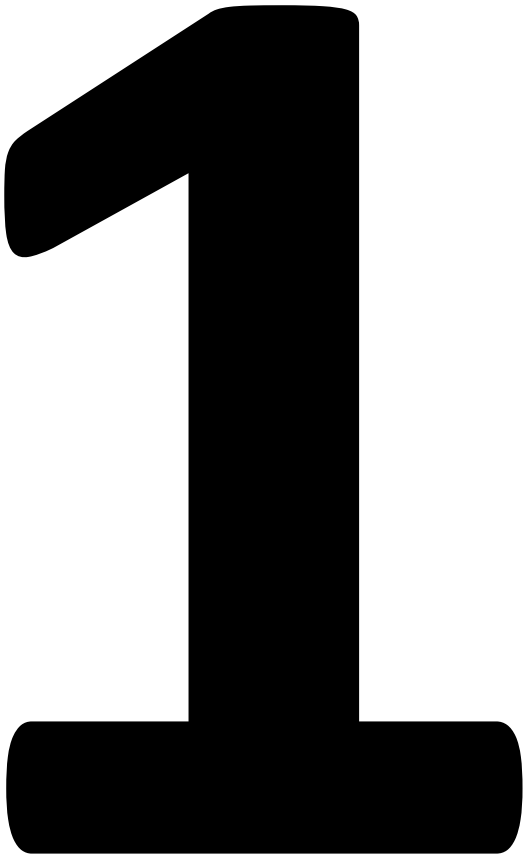
Andrew D. Gordon, Microsoft Research

International Summer School Marktoberdorf, August 2009

Syllabus and Credits

1. Overview: Refinement Types and Systems Models
2. Application: Verifying Security Protocol Code
3. A Concurrent λ-Calculus with Refinement Types

- Part 1 is largely based on unpublished joint work with G. Bierman and D. Langworthy
- Parts 2 and 3 are largely based on published papers:
  - K. Bhargavan, C. Fournet, A. Gordon, S. Tse, *Verified Interoperable Implementations of Security Protocols,* IEEE CSFW 2006.
  - J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, S. Maffeis, *Refinement Types for Secure Implementations*, IEEE CSF 2008.



A Type of Positive Numbers: Why Not?

```
fun MyFun (x:pos, y:pos): pos = if x>y then x-y else 42
```

- **Q:** No currently popular or hip language has these – why not?
- **A:** The typechecker would need to know $\forall x.\ \forall y.\ x>y \Rightarrow x-y>0$ and computers don't do arithmetic reasoning, do they?

- This is an example **refinement type** `Integer where value>0`
- Known since the 1980s, but typechecking impractical, because automated reasoning is hard, inefficient, and unreliable

Objectives

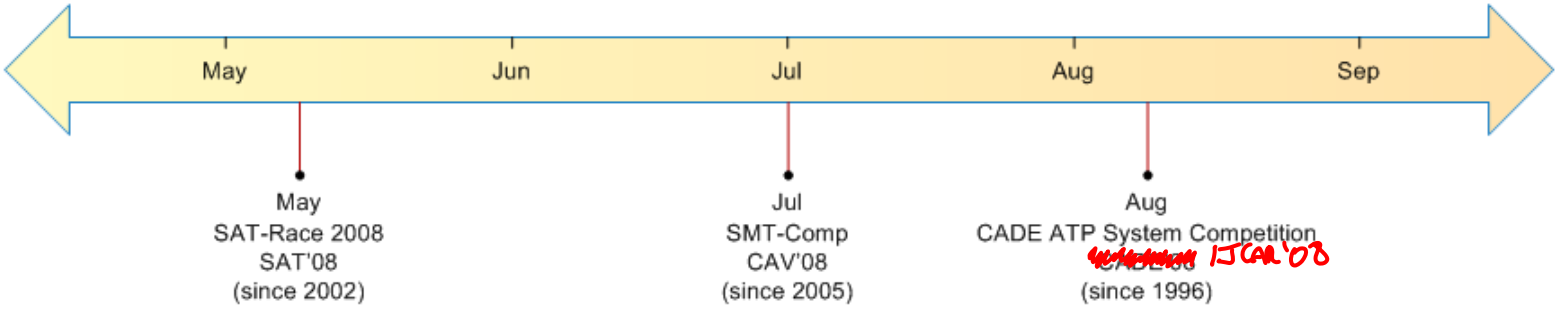
- This lecture is a primer on refinement types
- I'm assuming you know about types in standard languages like C, Java, C#, etc, but not that you're a type theory geek
- Why learn about refinement types?
- What's on offer in this lecture?
- How do I find out more?

- **Q:** How did the typechecker decide $\forall x.\ \forall y.\ x>y \Rightarrow x-y>0$?
- **A:** It didn't. It didn't even try. It asked an SMT solver.

An Opportunity: Logic as a Platform

"Satisfiability Modulo Theory (SMT) solvers decide logical satisfiability (or dually, validity) with respect to a background theory expressed in classical first-order logic with equality. These theories include: real or integer arithmetic, and theories of program or hardware structures such as bitvectors, arrays, and recursive datatypes."

- Dramatic advances in theorem proving this decade
  - Contenders include Simplify (HPL), Yices (SRI), Z3 (MSR)



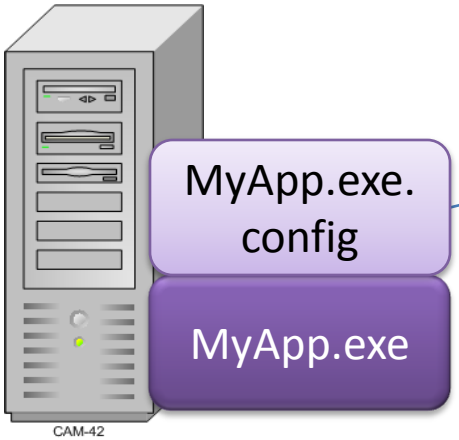
Annual competitions, standard formats for logical goals – a platform

Microsoft Research Cambridge

How typechecking based on an external solver makes type-safe systems modeling practical, and helps extend the Microsoft platform

REFINEMENT TYPES AND M

The Oslo Modeling Language




```
<?xml version="1.0" encoding="utf-8"?>
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <policy name="policy-CAM-42">
    <mutualCertificate10Security
      establishSecurityContext="false"
      messageProtectionOrder="EncryptBeforeSign">
    </mutualCertificate10Security>
  </policy>
</policies>
```

- Server stacks (eg .NET) allow post-deployment configuration
  - But as server farms scale, manual configuration becomes problematic
  - Better to drive server configurations from a central repository
- M is a new modeling language for such configuration data
  - Ad hoc modeling languages remarkably successful in Unix/Linux world
  - M is in development (first CTP at PDC'08, most recent May 2009)
  - Next, Oslo in their own words...

The Core of the M Language

- A **value** may be a **general value** (integer, text, boolean, null)
- Or a **collection** (an unordered list of values),
- Or an **entity** (a finite map from string labels to values)

- The expression

```
( from n in { 5, 4, 0, 9, 6, 7, 10}
  where n < 5
  select {Num=>n, Flag=>(n>0)} )
```

  has the type

```
{Num:Integer; Flag:Logical;}*
```

  and evaluates to

```
{{Num=>4,Flag=>true},
 {Num=>0, Flag=>false}}
```

- Semantic domain of values (in ML syntax)

```
type General = G_Integer of int | G_Logical of bool | G_Text of string | G_Null
type Value = G of General | C of Value list | E of (string * Value) list
```

Interdependent Types and Expressions

- A **refinement** type `T where e` consists of the values of type *T* such that boolean expression *e* holds
- A **typecase** expression `e in T` returns a boolean to indicate whether the value of *e* belongs to type *T*
  - `{x=1, y=2} in {x:Any}` returns true (due to subtyping)
- A **type assertion** `e : T` requires that *e* have type *T*
  - Verify statically if possible
  - Compile to `(e in T) ? e : throw "type error"` if necessary

Some Examples in M

- Example: type-safe unions
- Demo: comparison of M/MiniM
- Case study: how static typing may help Dynamic IT

Some Derived Types

- Empty type — Empty ≡ Any where false
- Singleton type — {*e*} ≡ Any where value==*e*
- Null type — Null ≡ {null}
- Union type — *T* | *U* ≡ Any where (value in *T* || value in *U*)
- Nullable type — Nullable *T* ≡ *T* | {null}

Example: Type-Safe Union Types

- Given source

```
type NullableInt : Integer | {null}
from x in ({1, null, 42, null } : NullableInt*)
where x!=null
select (x:Integer)
```

our typechecker calls the solver as follows:

```
(x!=null), x:NullableInt |- x in Integer
===
Asked Z3:
  (BG_PUSH (FORALL (x) (IFF ($NullableInt x) (OR (In_Integer x) (EQ x (v_null))))))
  (IMPLIES (AND (NOT (EQ $x (v_null))) ($NullableInt $x)) (In_Integer $x))
Z3 said : True
```

Interlude: Implementation Notes

- Expressions typed by “bidirectional rules” as in eg C#
  - But no constraint inference
- Subtyping decided semantically, by external solver
  - Term T(*e*) for each expression *e*, formula F(*T*)(*x*) for each type *T*

F([42])(x) = (x=42)
F(Integer where value < 100)(x) = (x<100)

  - Subtyping is implication: $T <: U$ iff $\forall x.\ \mathrm{F}(T)(x) \Rightarrow \mathrm{F}(U)(x)$

[42] <: (Integer where value < 100) iff $\forall$x. (x=42) $\Rightarrow$ (x<100)

```
module M {
   F() : Integer32 where value == 2 { 3 }
}
```

```
module Constraints
{
   type Person : { Name:Text; Age:Integer32; };
   type EligiblePerson : Person where value.Age > 17;
   type Marriage : { SpouseA: EligiblePerson; SpouseB: EligiblePerson; };

   PatChris(): Marriage
   {
     {SpouseA => {Name => "Pat", Age => 24},
      SpouseB => {Name => "Chris", Age => 32}}
   }

   BillySam(): Marriage
   {
     {SpouseA => {Name => "Billy", Age => 4},
      SpouseB => {Name => "Sam", Age => 5}}
   }
}
```

```
module TaggedUnions
{
   type T1 : {tag: {42}; bar: Integer32;};
   type T2 : {tag: {43}; foo: Text;};
   type U : T1 | T2;

// this fails to typecheck, because it makes insufficient checks
// Test1(xs:U*) : Text* { from x in xs select x.foo  }

   Test2(xs : U*) : Text*
   {
      from x in xs select ( x.tag==42 ? "Hello" : x.foo )
   }

   Test3(xs : U*) : Text*
   {
      from x in xs where (x.tag==43) select x.foo
   }
}
```

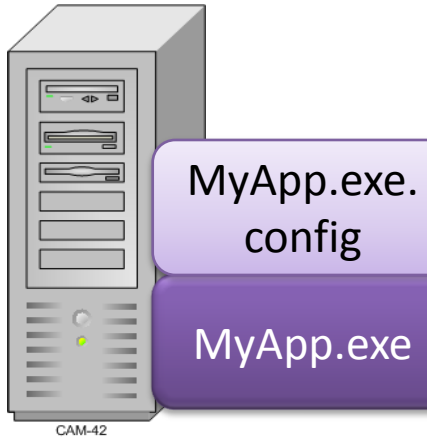
```
//typeful
module Points
{
  type Nat : Integer32 where value==0 || value>0;
  type Byte : Nat where value<256;
  type Color : {Red:Byte; Green:Byte; Blue:Byte;};
  type Point : {X: Integer32; Y:Integer32;};
  type ColorPoint : Point & {Color:Color;};
  type Points : Point*;
  type ColorPoints : ColorPoint*;

  f(x:Point) : ColorPoint { x }
}
```

```
module MinimTests
{
   type Operator : Text where
     value=="plus" || value=="minus" ||
     value=="times" || value=="div";

   type Expression :
   {kind:{"variable"}; name: Text;} |
   {kind:{"integer"}; val: Integer32;} |
   {kind:{"binary app"}; operator: Operator;arg1: Expression; arg2: Expression;};

   type Statement :
   {kind:{"assignment"}; var: Text; rhs: Expression;} |
   {kind:{"while"}; test:Expression; body:Statement;} |
   {kind:{"if"}; test:Expression; tt:Statement; ff:Statement;} |
   {kind:{"seq"}; s1:Statement; s2:Statement;} |
   {kind:{"skip"};};

  FirstExp(E:Expression) : Text
   {
      (E.kind=="variable") ? E.name : (
      (E.kind=="integer") ? "integer" :
       E.operator)
   }

   FirstStatement(S:Statement) : (Expression | {null} )
   {
      (S.kind=="assignment") ? S.rhs : (
      (S.kind=="while" || S.kind=="if") ? S.test :
      null
   }

   //Test(S:Statement) : Expression { S.rhs }  // this correctly fails to typecheck
}
```

Comparing the MiniM typechecker with the May CTP M typechecker; MiniM focuses on types, lacks significant features like extents

DEMO

Better Dynamic IT by Typing



- Many systems errors arise from misconfigurations
  - Formats often too flexible; operators make mistakes
- Numerous ad hoc tools advise on config “safety”
  - Find misconfigurations in firewalls, routers, protocol stacks, etc; check that adequate security patches have been applied
  - Tools package specialist expertise; more accessible than best practice papers; easy to update as new issues arise
- M is a general purpose platform for systems modeling
  - User-defined types can express advisories, subsuming ad hoc tools
  - Let’s look at a concrete example: WSE Policy Advisor

A Typical Config-Based Advisor

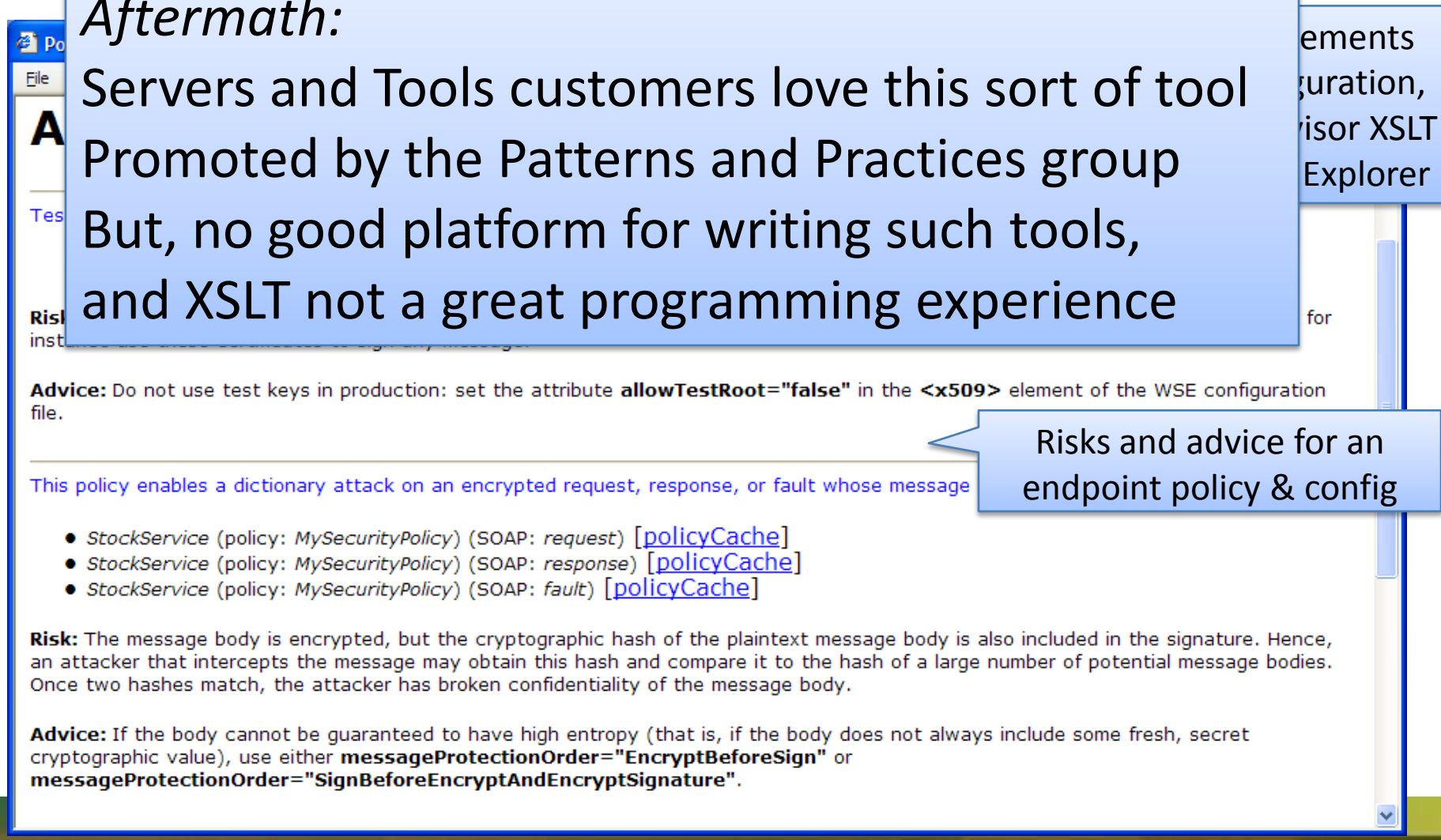
Aftermath:

Servers and Tools customers love this sort of tool

Promoted by the Patterns and Practices group

But, no good platform for writing such tools,

and XSLT not a great programming experience



1: Representing XML Data

```
<?xml version="1.0" encoding="utf-8"?>
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <policy name="policy-CAM-42">
    <mutualCertificate10Security
      establishSecurityContext="false"
      messageProtectionOrder="EncryptBeforeSign">
    </mutualCertificate10Security>
  </policy>
</policies>
```

```
{tag="policies",
 xmlns="http://schemas.microsoft.com/wse/2005/06/policy",
 body={{tag=>"policy",
        name=>"policy-CAM-42",
        body={{tag=>"mutualCertificate10Security",
               establishSecurityContext=>"false",
               messageProtectionOrder=>"EncryptBeforeSign" }}}}}
```

2: Types for Schema-Correct Configs

```
type bool : {"true"} | {"false"};
type messageProtectionOrder : {"EncryptBeforeSign"}|{"SignBeforeEncrypt"};
type mutualCertificate10Security :
   {tag:{"mutualCertificate10Security"};
    establishSecurityContext:bool;
    messageProtectionOrder:messageProtectionOrder; } ;
```

```
Policy = mutualCertificate10Security  | ...
Config = {tag:{"policies"}; body:{tag:{"policy"}; body:Policy*; }*; } ;
```

```
<?xml version="1.0" encoding="utf-8"?>
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <policy name="policy-CAM-42">
    <mutualCertificate10Security
      establishSecurityContext="false"
      messageProtectionOrder="EncryptBeforeSign">
    </mutualCertificate10Security>
  </policy>
</policies>
```

has type Config

3: Types for **Safe** Configs

```
type q_credit_taking_attack_10 :
  (mutualCertificate10Security
    where value.messageProtectionOrder == "EncryptBeforeSign") ;
type Advisory = q_credit_taking_attack_10  | ...
```

```
type SafePolicy : Policy & (!Advisory)
type SafeConfig : {tag:{"policies"}; body:{tag:{"policy"}; body:SafePolicy*; }*; } ;
```

```
<?xml version="1.0" encoding="utf-8"?>
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <policy name="policy-CAM-42">
    <mutualCertificate10Security
      establishSecurityContext="false"
      messageProtectionOrder="EncryptBeforeSign">
    </mutualCertificate10Security>
  </policy>
</policies>
```

has type Config
but **not** type SafeConfig

Refinement Types and M

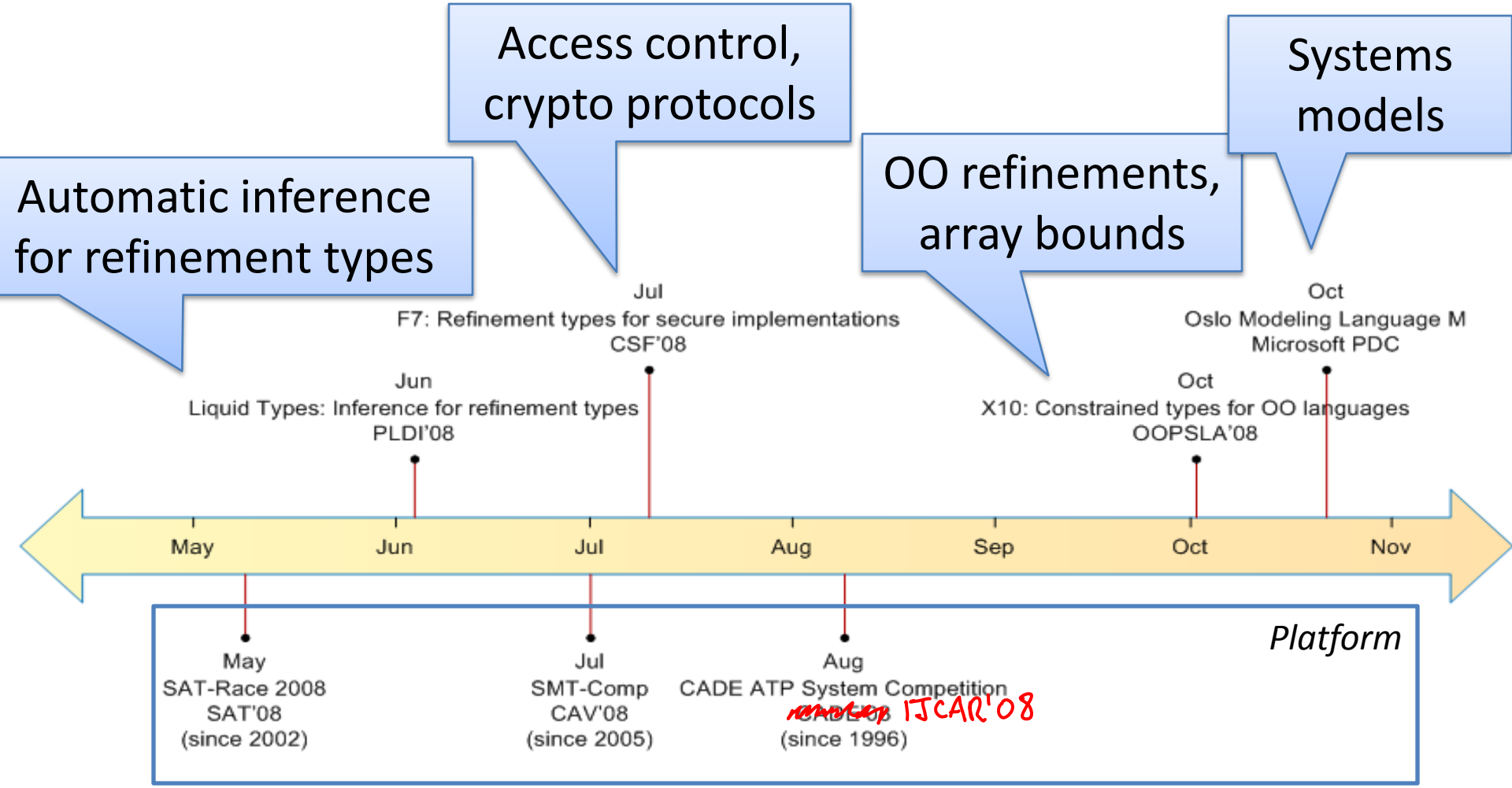
- The interdependence between typecase expressions and refinement types in M is a novel source of great expressivity
- Relying on an external solver achieves type safety for union and dependent types without complex, arbitrary rules
- Security and error checking expressible within M type system
  - Helps M extend the Microsoft platform

- Our Z3-based typechecker was jointly developed with the Oslo team in parallel with the mainline typechecker
  - We hope to merge the code-bases this year

Related Work

		Refinement	Typecase	Subtyping
1983 Nordström/Petersson	**Subset types**	{x:A | B(x)}	no	no
1986 Rushby/Owre/Shankar	**Predicate subtyping**	predicate subtype	no	limited
1989 Cardelli et al	**Modula-3 Report**	no	on references	structural
1991 Pfenning/Freeman	**Refinement types**	refined sorts	no	no
1993 Aiken and Wimmers	**Type inclusion...**	no	no	semantic
1999 Pfenning/Xi	**DML**	{x: General | e}	no	no
1999 Buneman/Pierce	**Unions for SSD**	no	yes, as pattern	structural
2000 Hosoya/Pierce	**XDuce**	no	yes, as pattern	semantic, ad hoc
2006 Flanagan et al	**SAGE**	{x: T | e}	no (but has cast)	structural, SMT
2006 Fisher et al	**PADS**	{x:T | e}	no	structural
2007 Frisch/Castagna	**CDuce**	no	e in T	semantic, ad hoc
2007 Sozeau	**Russell**	{x:T | e}	no	structural
2008 Bhargavan/Fournet/G	**F7/RCF**	{x: T | C} (formula C)	no	structural, SMT
2008 Rondon/Jhala	**Liquid Types**	{x: General | e}	no	structural, SMT
2009 Bierman/G/Langworthy	**M/Minim**	{x: T | e}	e in T	semantic, SMT

A Good Year for Refinements



Ideas to Take Away

- Remember the riddle
  - **Q:** How did the typechecker decide $\forall x.\ \forall y.\ x>y \Rightarrow x-y>0$?
  - **A:** It didn't. It didn't even try. It asked an SMT solver.
- Remember that boundaries are blurring
  - Between types, predicates, policies, patterns, schemas
  - Between typechecking and verification
- Still, SMT solvers are incomplete, often amazingly so
  - So dealing with typing errors remains a challenge

Resources

- The Microsoft Research SMT solver, Z3
  http://research.microsoft.com/en-us/um/redmond/projects/z3/
- Oslo and its modeling language, M
  http://msdn.microsoft.com/oslo
- Refinement types for security in F#
  http://research.microsoft.com/f7
- Liquid types (including online demo)
  http://pho.ucsd.edu/liquid/
- This lecture
  http://research.microsoft.com/en-us/people/adg/part.aspx

2

Application: Verifying Security Protocol Code

Principles and Applications of Refinement Types, Part 2

http://research.microsoft.com/CVK
http://research.microsoft.com/F7

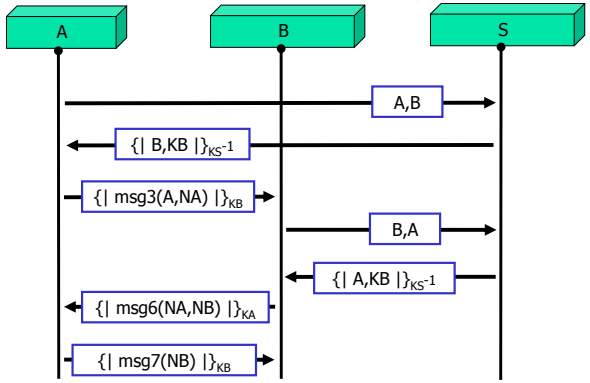
The Needham-Schroeder Problem

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.

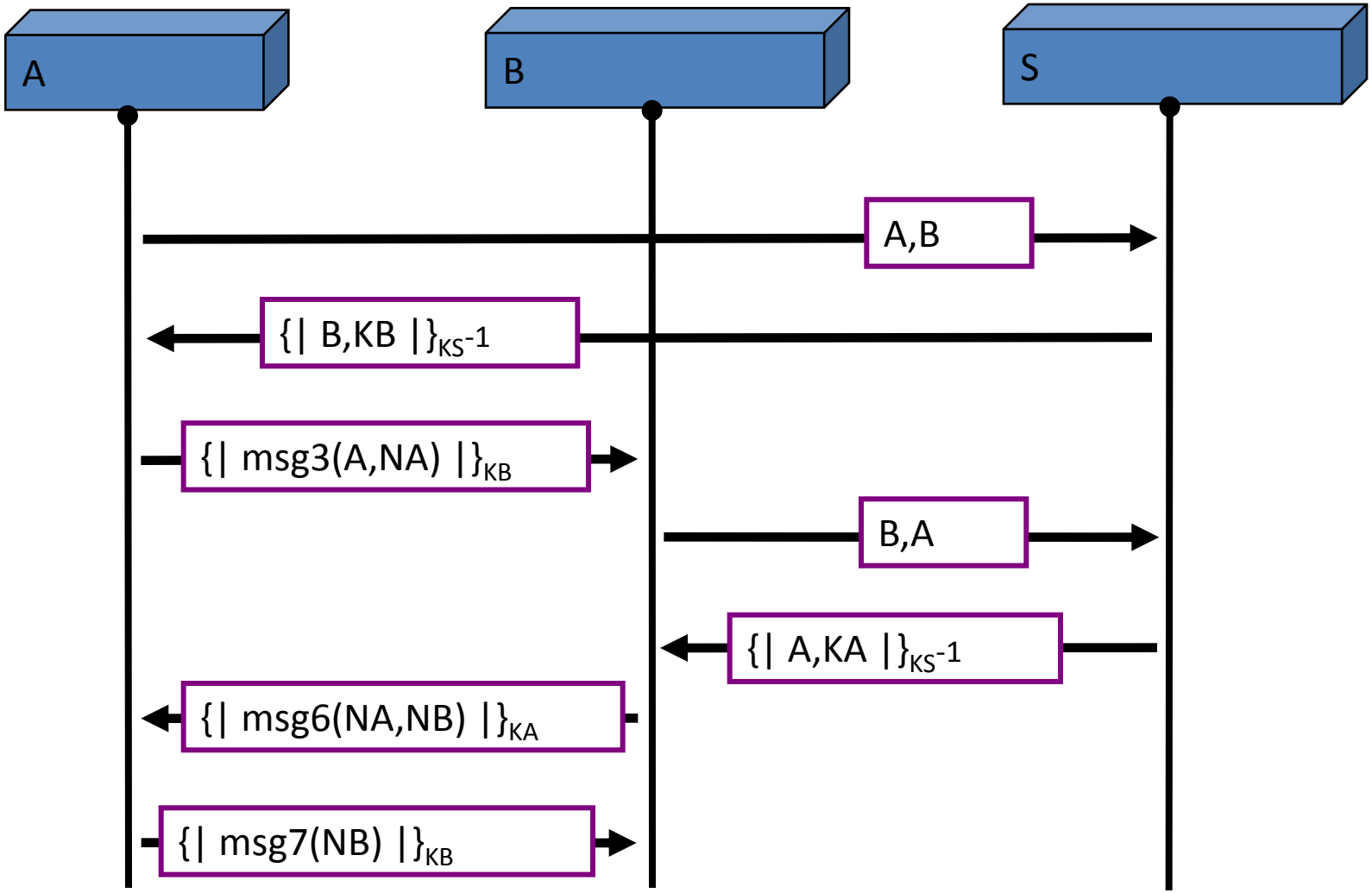
"Protocols such as those developed here are prone to extremely subtle errors that are unlikely to be detected in normal operation. The need for techniques to verify the correctness of such protocols is great, and we encourage those interested in such problems to consider this area."

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)

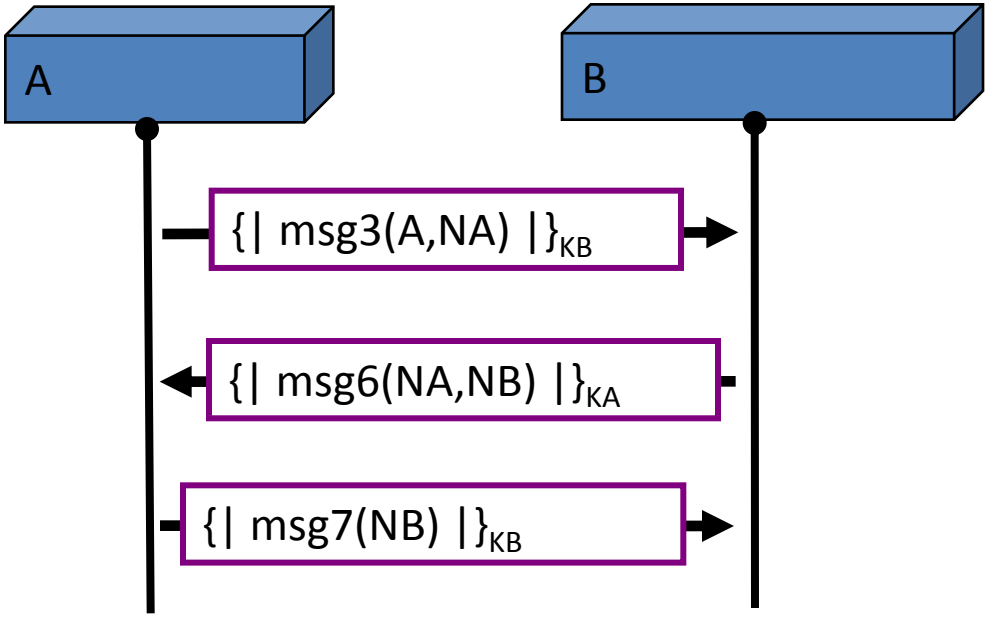


Principal A initiates a session with principal B

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:

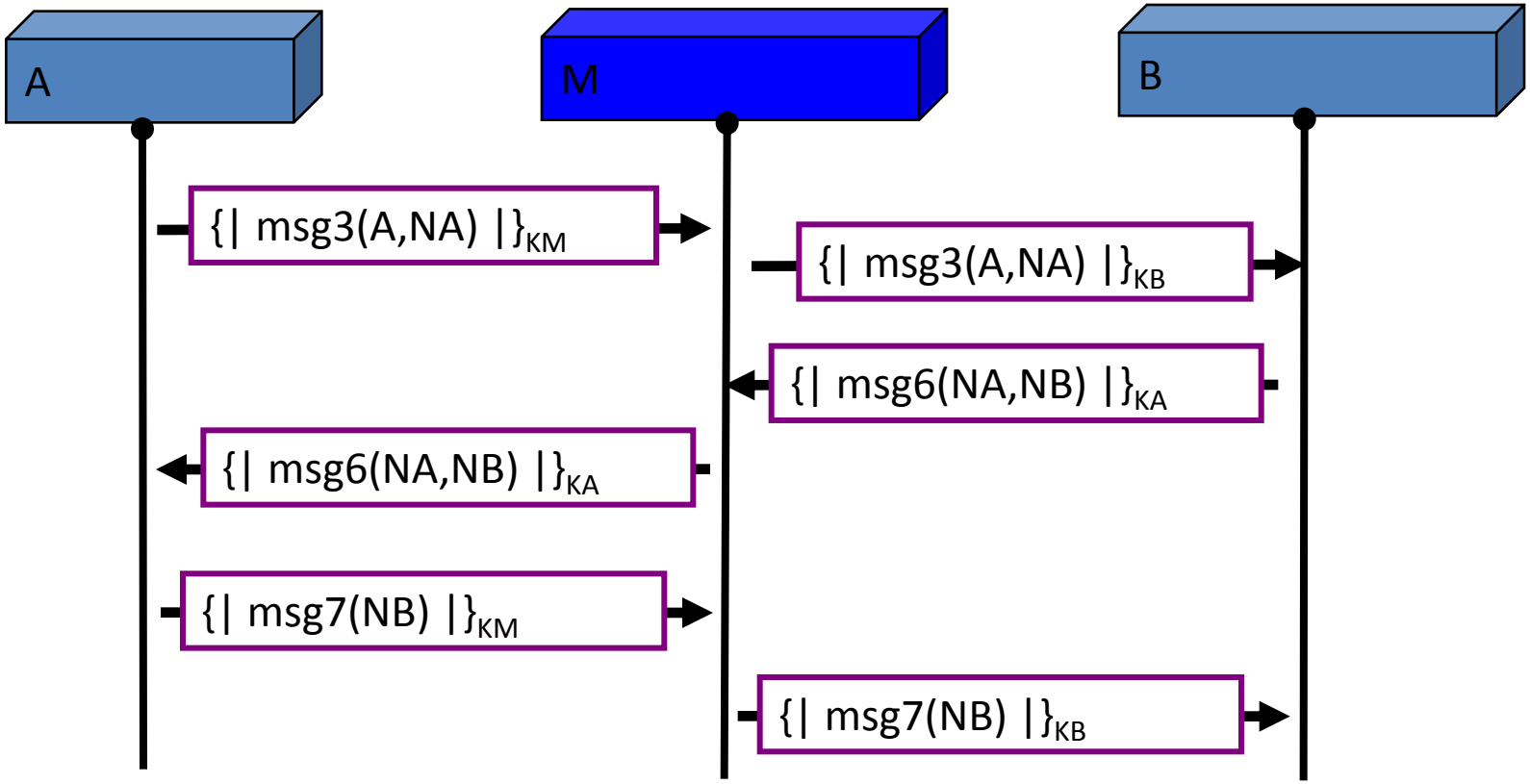


More precisely, the goals of the protocol are:

- 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 of old messages

- 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.

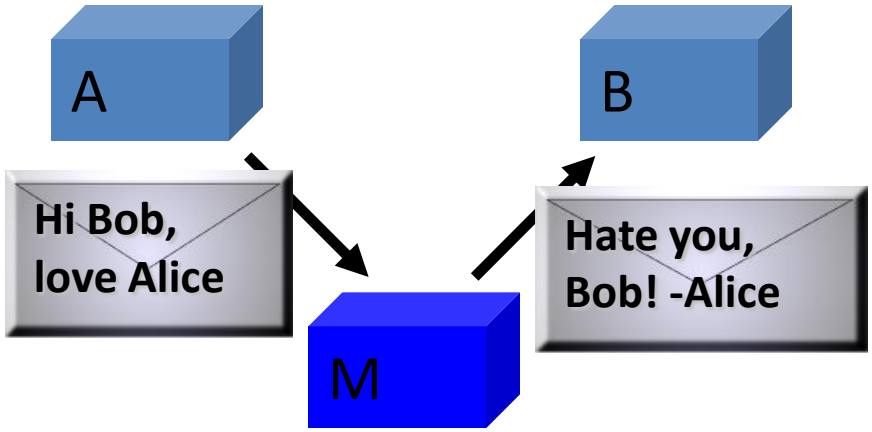
Anderson and Needham *Programming Satan's Computer* 1995

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: $\{ \{M\}_K \}_K\text{-}1 = M$
  - Public key decryption: $\{| \{| M |\}_{KA} |\}_{KA}\text{-}1 = M$
  - Their work now widely recognised, but at the time, few 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
  - Incomplete, 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
2000: Abadi and Rogaway initiate connections between formal and computational models of crypto
circa 2005: Many FMs now developed; several (eg ProVerif) deliver both accuracy and automation
2005: Cervesato et al find same insider attack as Lowe on proposed public-key Kerberos
2005: Goubault-Larrecq and Parrennes pioneer direct verification of implementation code in C

Problem of Verifying Protocol Code

- The problem of vulnerabilities in security protocol code is remarkably resistant to the success of formal methods
- Perhaps, tools for verifying the actual protocol code will help
  - Csur (VMCAI'05), fs2pv (CSFW'06), F7 (CSF'08), Aspier (CSF'09), etc etc
- Currently, fs2pv most developed, but hitting a wall
  - Translates libraries and protocol code from F#/OCaml to ProVerif
  - ProVerif does whole-program analysis of code versus symbolic attacker
  - Long, unpredictable run times on Cardspace (ASIACCS'08), TLS (CCS'08)

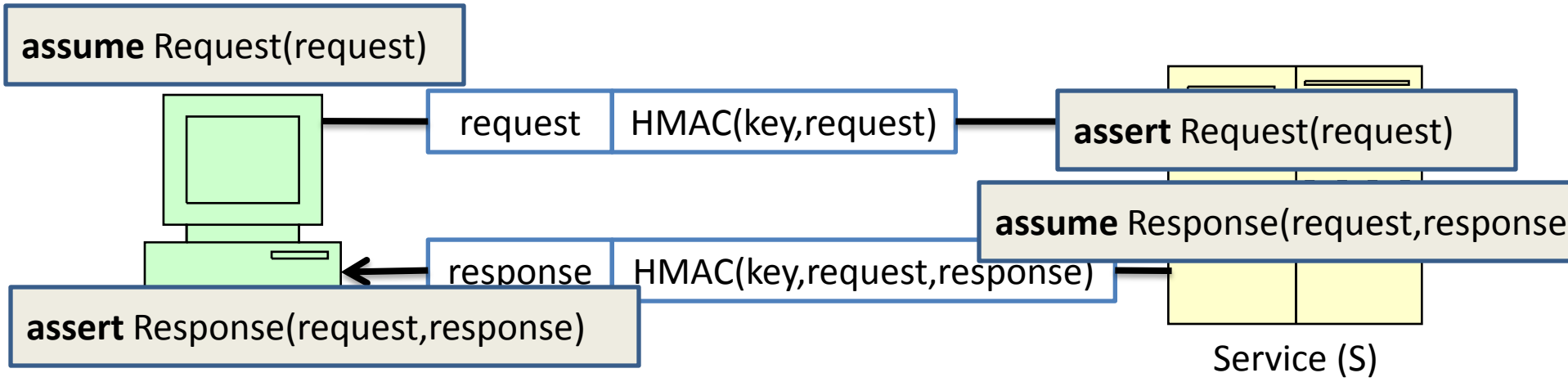
- In these lectures we'll develop a compositional analysis for protocol libraries and code, based on refinement types

Agenda for Rest of This Lecture

- How to represent protocols and their correctness within a concurrent functional language (F#/OCaml):
  - Correspondence assertions as assume/assert
  - Message-passing concurrency as in the pi-calculus
  - Crypto modelled using Morris' seal abstraction
  - Protocol roles as functions (we'll see the code in action)
  - Opponent (attacker) is an arbitrary untyped expression
  - Correctness as robust program safety

- Overall, we reduce crypto protocol verification to a program verification problem

Example: A Secure Protocol for RPC



- Security goals are authenticity and correlation of request and response, but not confidentiality or freshness
- Shows essence of problem, with simplifying assumptions
  - Assume one key, shared between two, fixed principals
  - Assume principals use keys only in compliance with protocol

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 Hoare logic)
  - Such predicates usefully represent security-related concepts like roles, permissions, events, compromises

Message-Passing and Concurrency

```
val fork : (unit → unit) → unit
type name
val name : unit → name
type α chan
val chan : unit → α chan
val send : α chan → α → unit
val recv : α chan → a
```

Concurrency in style of the **pi-calculus** (Milner, Parrow, Walker 1989) but within a functional language (like 80s languages PFL, Poly/ML, Concurrent ML)

```
let model (a: int chan) =
    fork (fun() → send a 42);
    let c:int chan = chan() in
    fork (fun() → let x = recv a in assume Sent(x); send c x);
    let x=recv c in assertSent(x)
```

Symmetric Crypto

```
type α pickled (*byte array representation of α*)
val pickle : (α → α pickled)
val unpickle : (α pickled → α)

type α hkey (*hash key*)
type hmac (*keyed hash*)
val mkHKey : (unit → α hkey)
val hmacsha1 : (α hkey → (α pickled → hmac))
val hmacsha1Verify : (α hkey → (β pickled → (hmac → α pickled)))

type α symkey (*symmetric encryption key*)
type enc (*ciphertext*)
val mkEncKey : (unit → α symkey)
val aesEncrypt : (α symkey → (α pickled → enc))
val aesDecrypt : (α symkey → (enc → α pickled))
```

Asymmetric Crypto

type α sigkey `(*private signing key*)`
type α verifkey `(*public verification key*)`
type dsig `(*signature block*)`
val rsasha1 : $(\alpha$ sigkey $\rightarrow (\alpha$ pickled $\rightarrow$ dsig$))$
val rsasha1Verify : $(\alpha$ verifkey $\rightarrow (\beta$ pickled $\rightarrow$ (dsig $\rightarrow \alpha$ pickled$)))$

type β deckey `(*private decryption key*)`
type β enckey `(*public encryption key*)`
type penc `(*ciphertext*)`
val rsaEncrypt : $(\beta$ enckey $\rightarrow (\beta$ pickled $\rightarrow$ penc$))$
val rsaDecrypt : $(\beta$ deckey $\rightarrow$ (penc $\rightarrow \beta$ pickled$))$

Morris' Seal Abstraction

A *seal* k for a type T is a pair of functions:

- the *seal function for* k, of type $T \rightarrow \mathsf{Un}$
- the *unseal function for* k, of type $\mathsf{Un} \rightarrow T$

The type Un consists of untrusted, public bitstrings known to the attacker.

The seal function, applied to M, wraps up its argument as a *sealed value*, written $\{M\}_k$.
There is no other way to construct $\{M\}_k$.

The unseal function, applied to $\{M\}_k$, unwraps its argument and returns M.
There is no other 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 $\{M_i\}_k$.

J.H. Morris, Jr, *Protection in Programming Languages*, CACM 1973

Coding Seals using Pi Library

```
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() 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"

type α Seal = (α → Un) * (Un → α)

let mkSeal (n:string) : α Seal =
  let s:α SealRef = ref [] in (seal s, unseal s)
```

Coding Crypto Library 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.

```
type α symkey = Sym of α pickled Seal
type enc = AES of Un
```

Duplex Communication

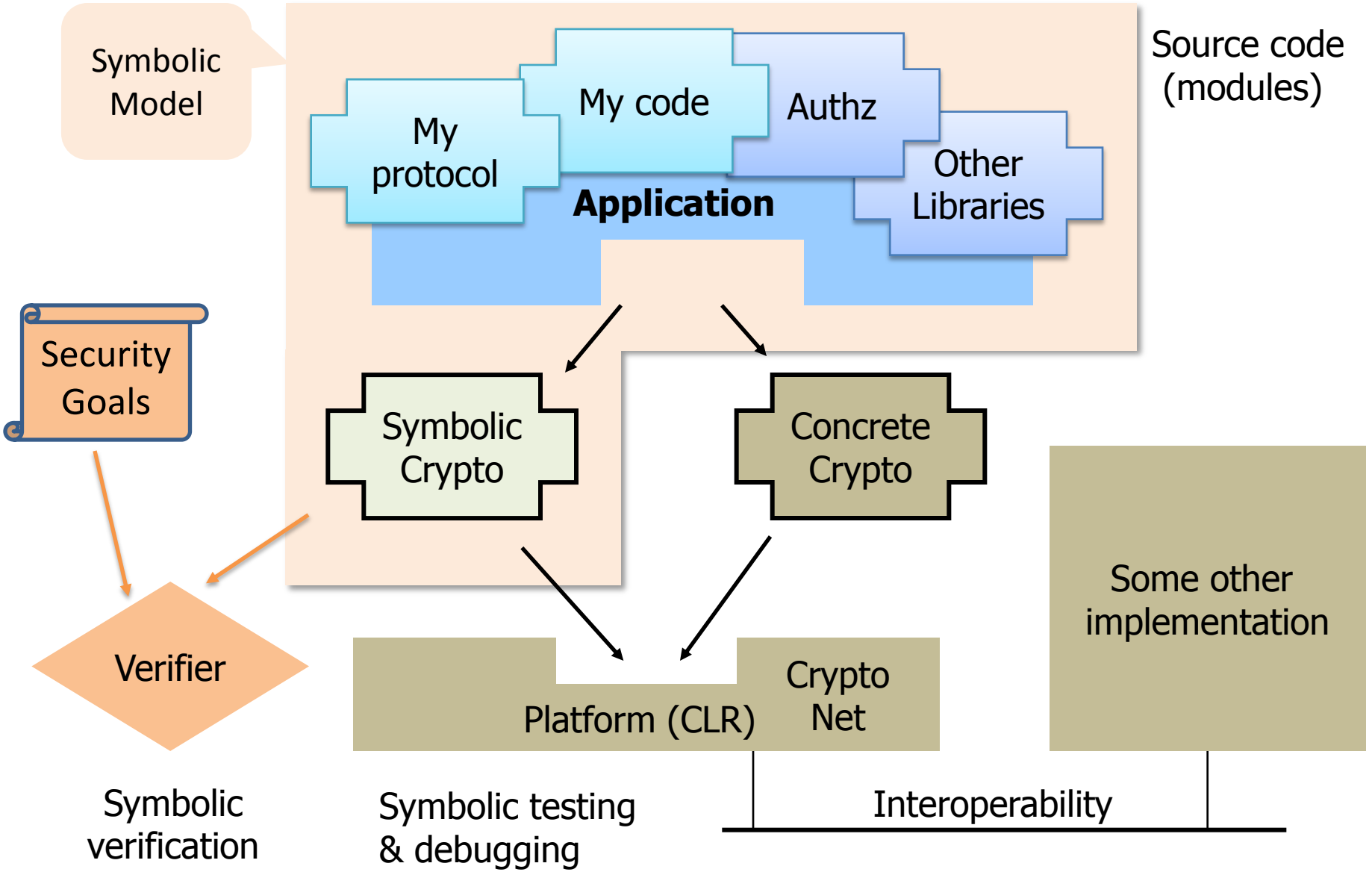
type (α, β) addr
type (α, β) conn

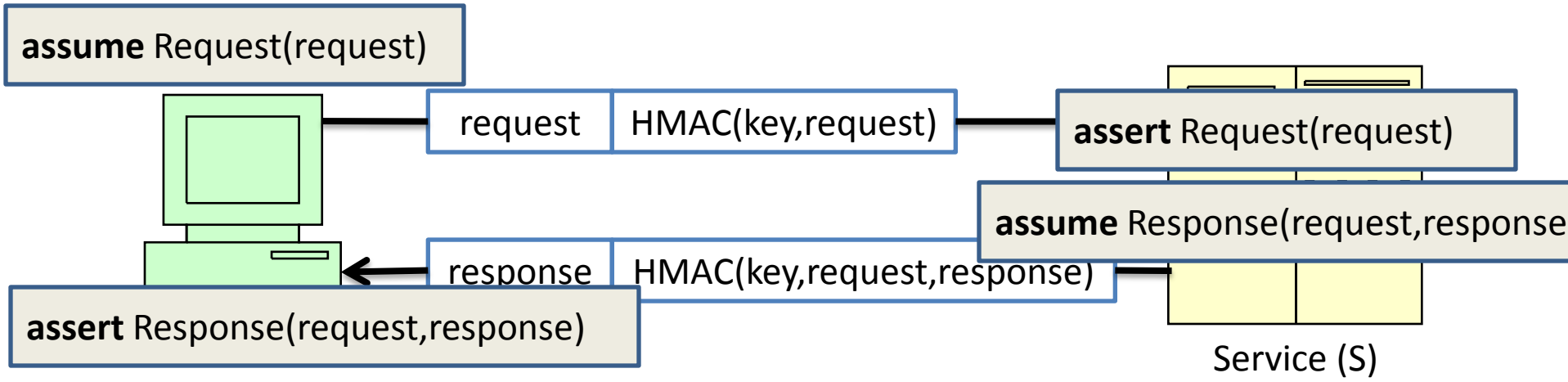
val http : string $\rightarrow$ string $\rightarrow (\alpha, \beta)$ addr
val connect : (α, β) addr $\rightarrow (\alpha, \beta)$ conn
val listen : (α, β) addr $\rightarrow (\beta, \alpha)$ conn
val close : (α, β) conn $\rightarrow$ unit
val send : (α, β) conn $\rightarrow \alpha$ pickled $\rightarrow$ unit
val recv : (α, β) conn $\rightarrow \beta$ pickled

Exercise: Implement these functions over the message-passing API.
Exercise: Implement these functions over a TCP sockets API.

One Source, Three Tasks

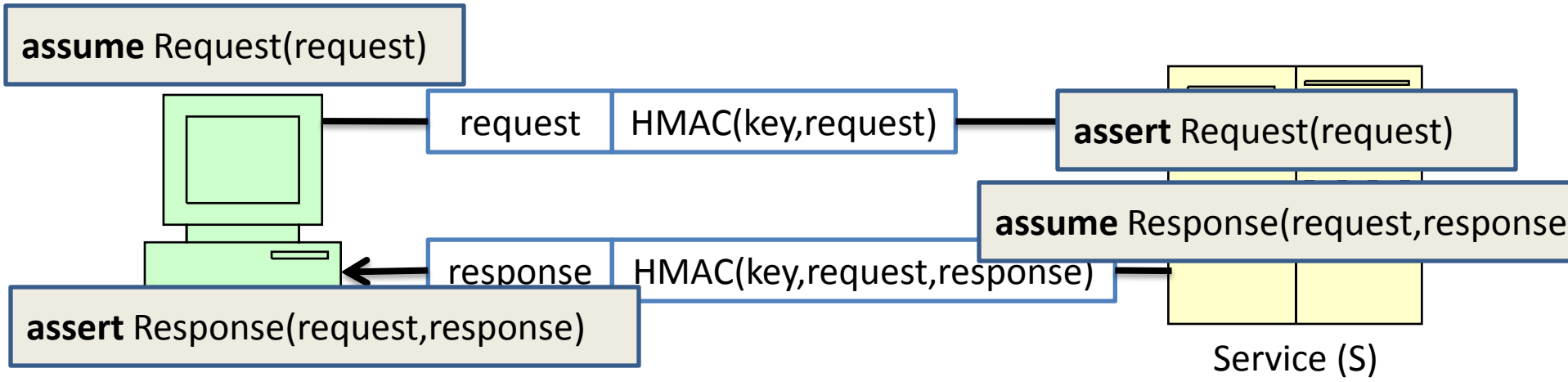
Bhargavan, Fournet, Gordon, Tse, *Verified Interoperable Implementations of Security Protocols,* IEEE Computer Security Foundations Workshop 2006





Demo: Run with F#

```
type content = (string * hmac)
type message = content pickled

val addr : (content, content) Net.addr
val client : (string -> string)
val server : (unit -> unit)
```



Preview

```
type request = r:string {Request(r)}
type response = r:string * s:string {Response(r,s)}

type info = i:string
   { (∀r. i=Concat("Request",r) ⇒ Request(r)) ∧
     (∀r,s. i=Concat("Response",Concat(r,s)) ⇒ Response(r,s)) }

private val k: info hkey
```

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)

The problem: can any attacker break any assertion, given:

```
val addr : (content, content) Net.addr
val client : (string -> string)
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.

Limits of Symbolic Models

- Dolev-Yao style **symbolic models** (including seals) have effective proof techniques, but make strong assumptions:
  - Message length is only partially observable
  - No collisions: $\{M\}_K=\{M'\}_{K'}$ implies M=M' and K=K'
  - Non-malleability: from $\{M\}_K$ cannot construct $\{M'\}_K$
  - No partial information: that attacker cannot guess half the bits of a message, or know half in advance
  - Keys are unguessable, even passwords
- Cryptographers rely on probabilistic **computational models**, making fewer assumptions, but with fewer automated reasoning techniques
- Justifying symbolic models via computational models (where possible), or simply developing automation for the latter, is a growing research area

Summary of Part 2

- The problem of vulnerabilities in security protocol code is remarkably resistant to the success of formal methods
- Verifying the actual protocol code may help
- We have recast prior work on modelling protocols within process calculi (spi, applied pi) in the setting of ML with concurrency
- Security properties (authenticity, but secrecy too) are expressed using program assertions

- In Part 3, we develop RCF – a formal foundation for ML with concurrency – and its system of refinement types
- RCF is the basis for F7, a scalable verifier for protocol code

http://research.microsoft.com/en-us/people/adg/part.aspx

3

RCF: Refined Concurrent FPC

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

- RCF is the theoretical basis for F7, but there is also a direct implementation (done at Saarbruecken)
- My goal is to explain from first principles how we can show the following RCF example is safe by typechecking:

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

RCF PART 1:
SYNTAX AND SEMANTICS

The Fixpoint Calculus (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 $x \to A$	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** $h\ x \to A$ **else** B	constructor match (scope of x is A)

The Reduction Relation: $A \longrightarrow A'$

$(\textbf{fun}\, x \to A)\; N \to A\{N/x\}$

$(\textbf{let}\; (x_1, x_2) = (N_1, N_2)\; \textbf{in}\; A) \to A\{N_1/x_1\}\{N_2/x_2\}$

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

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

$\textbf{let}\; x = M\; \textbf{in}\; A \to A\{M/x\}$

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

Example: Booleans and Conditional Branching:

$\mathbf{false} \stackrel{\triangle}{=} \mathsf{inl}\ ()$
$\mathbf{true} \stackrel{\triangle}{=} \mathsf{inr}\ ()$
$\mathbf{if}\ A\ \mathbf{then}\ B\ \mathbf{else}\ B' \stackrel{\triangle}{=}$
$\quad \mathbf{let}\ x = A\ \mathbf{in}\ \mathbf{match}\ x\ \mathbf{with}\ \mathsf{inr}(\_) \rightarrow B\ \mathbf{else}\ \mathbf{match}\ x\ \mathbf{with}\ \mathsf{inl}(\_) \rightarrow B'$

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

Exercise: What is the reduction of: $\mathbf{if\ true\ then}\ B\ \mathbf{else}\ B'$

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

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

Exercise: Derive a fixpoint function fix so that we can define recursive function definitions as follows: $\mathbf{let\ rec}\ f x = A \stackrel{\triangle}{=} \mathbf{let}\ f = \mathsf{fix}\ (\mathbf{fun}\ f \rightarrow \mathbf{fun}\ x \rightarrow A)$.

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*.
(Process calculi often use a symmetric version, called *structural equivalence*.)

Parallel Composition:

$A, B ::=$ expression

$\ldots$ as before

$A \Rsh B$ fork

$() \Rsh A \equiv A$

$(A \Rsh A') \Rsh A'' \equiv A \Rsh (A' \Rsh A'')$

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

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

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

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

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

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

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

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: $a!3 \Rsh$ **let** $x = a?$ **in** $F\ x$

Exercise: What are the reductions of the expression: $a!$**true** $\Rsh a!$**false**

Name Generation:

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

$A \equiv\!\!\Rightarrow A' \Rightarrow (\nu a)A \equiv\!\!\Rightarrow (\nu a)A'$

$a \notin fn(A') \Rightarrow A' \Rsh ((\nu a)A) \equiv\!\!\Rightarrow (\nu a)(A' \Rsh A)$

$a \notin fn(A') \Rightarrow ((\nu a)A) \Rsh A' \equiv\!\!\Rightarrow (\nu a)(A \Rsh A')$

$a \notin fn(B) \Rightarrow \textbf{let}\ x = (\nu a)A\ \textbf{in}\ B \equiv\!\!\Rightarrow (\nu a)\textbf{let}\ x = A\ \textbf{in}\ B$

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

Exercise: What are the reductions of the following expression:

$\textbf{let}\ x = (\nu a)a \Rsh (\nu b)b\ \textbf{in}\ F\ x$

Origins of 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)

Example: Concurrent ML:

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

$\mathsf{chan} \triangleq \mathbf{fun}\ \_ \to (\nu a)(\mathbf{fun}\, x \to a!x, \mathbf{fun}\ \_ \to a?)$

$\mathsf{send} \triangleq \mathbf{fun}\, c\, x \to \mathbf{let}\ (s,r) = c\ \mathbf{in}\ s\, x$	send x on c
$\mathsf{recv} \triangleq \mathbf{fun}\, c \to \mathbf{let}\ (s,r) = c\ \mathbf{in}\ r\ ()$	block for x on c
$\mathsf{fork} \triangleq \mathbf{fun}\, f \to (f() \Rsh ())$	run f in parallel

Example: Mutable State:

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

$\mathbf{ref}\, M \triangleq \mathbf{let}\ r = \mathsf{chan}()\ \mathbf{in}\ \ \mathsf{send}\ r\ M; r$	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 IMP programs within RCF.

Consider a global set of formulas, the *log*, drawn from some logic.

A General Class of Logics:

$C ::= p(M_1, \ldots, M_n) \mid M = M' \mid \ldots$
$\{C_1, \ldots, C_n\} \vdash C$ deducibility relation

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*.

Assume and Assert:

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

Exercise: What are the reductions of our running example:

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

Structures and Static Safety:

$e ::= M \mid MN \mid M = N \mid \textbf{let}\ (x,y) = M\ \textbf{in}\ B \mid$
$\quad \textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B \mid M? \mid \textbf{assert}\ C$
$\prod_{i \in 1..n} A_i \stackrel{\triangle}{=} () \Rsh A_1 \Rsh \ldots \Rsh A_n$
$\mathscr{L} ::= \{\} \mid (\textbf{let}\ x = \mathscr{L}\ \textbf{in}\ B)$

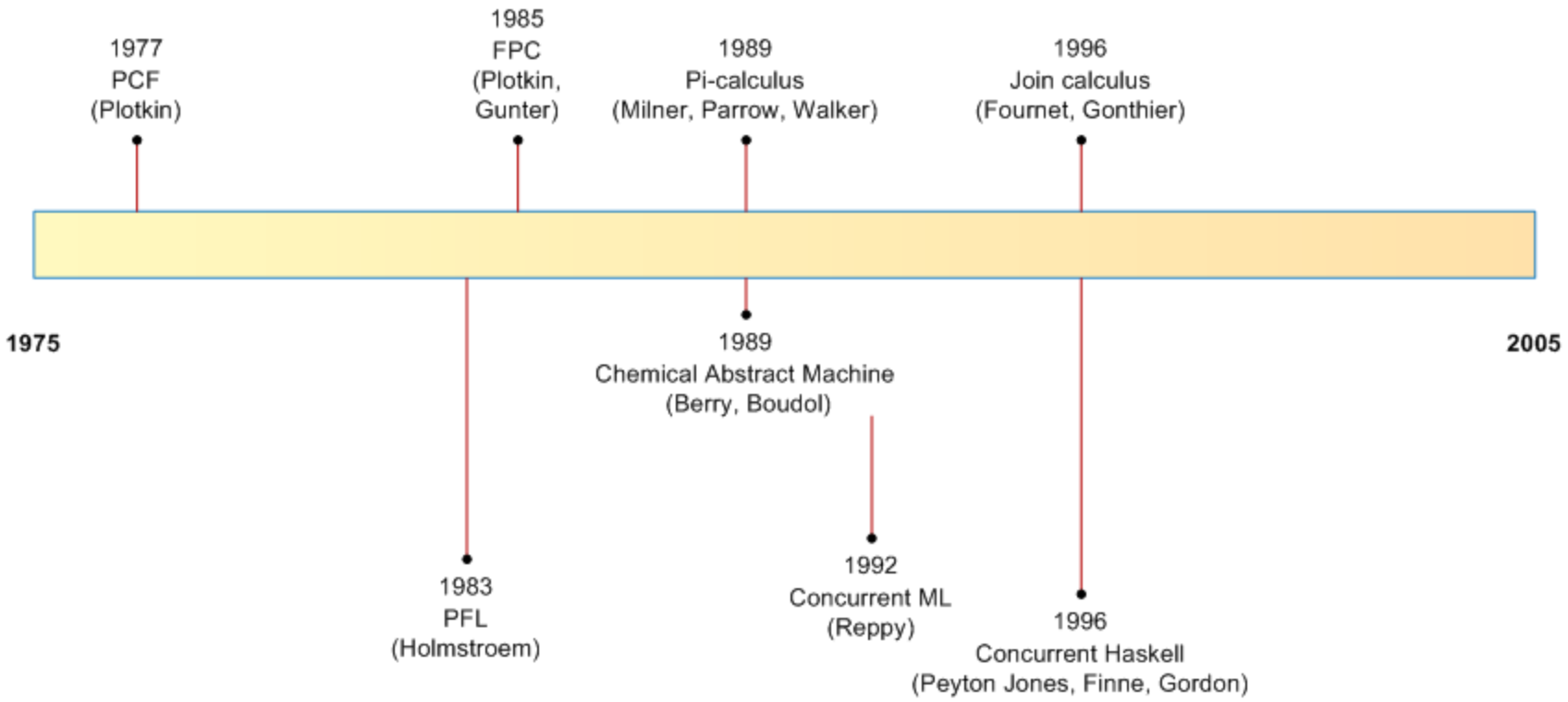
$$\mathbf{S} ::= (\nu a_1) \ldots (\nu a_\ell) \left((\prod_{i \in 1..m} \textbf{assume}\ C_i) \Rsh (\prod_{j \in 1..n} c_j!M_j) \Rsh (\prod_{k \in 1..o} \mathscr{L}_k\{e_k\}) \right)$$

Let structure $\mathbf{S}$ 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 $\mathbf{S}$ such that $A \Rrightarrow \mathbf{S}$.

Expression Safety:

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



FUNCTIONAL PROGRAMMING AND CONCURRENCY

RCF PART 2:
TYPES FOR SAFETY

Starting Point: The Type System for FPC:

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

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

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

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

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

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

Exercise: Write types of Booleans, numbers, and lists.
Exercise: Write a well-typed fixpoint combinator.

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: A Dependent Type System for FPC:

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

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

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

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

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

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

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 $\mathsf{forms}(x : \{C\})$?

We write $E \vdash C$ to mean that C follows from the refinement formulas in C.
For example, $x : \{x : \mathsf{int} \mid x > 0\}, b : \{b : \mathsf{bool} \mid x < 2\} \vdash x = 1$.
(In F7, did we try to implement this directly?)

Rules for Assume and Assert:

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

Subtyping Rules for Refinement Types:

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

Exercise: How would we derive $\vdash \{x : \mathsf{int} \mid x > 0\} <: \mathsf{int}$.
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 (Dependent) 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: Understand why:

$\vdash \{x : \mathsf{int} \mid x > 0\} <: \mathsf{int}$

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

but not:

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

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.

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

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

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

$$\overline{(\nu a)A} = (\exists a.\overline{A})$$

$$\overline{A_1 \Rsh A_2} = (\overline{A_1} \wedge \overline{A_2})$$

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

$$\overline{\textbf{assume } C} = C$$

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

Exercise: Find types to typecheck the following code:

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

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 $\mathbf{S}$ 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 $\mathbf{S}$ such that $A \rightarrow^* A'$ and $A' \Rrightarrow \mathbf{S}$ we need that $\mathbf{S}$ is statically safe.)

RCF III: TYPES FOR ROBUST SAFETY

Safety Versus an Untyped Adversary

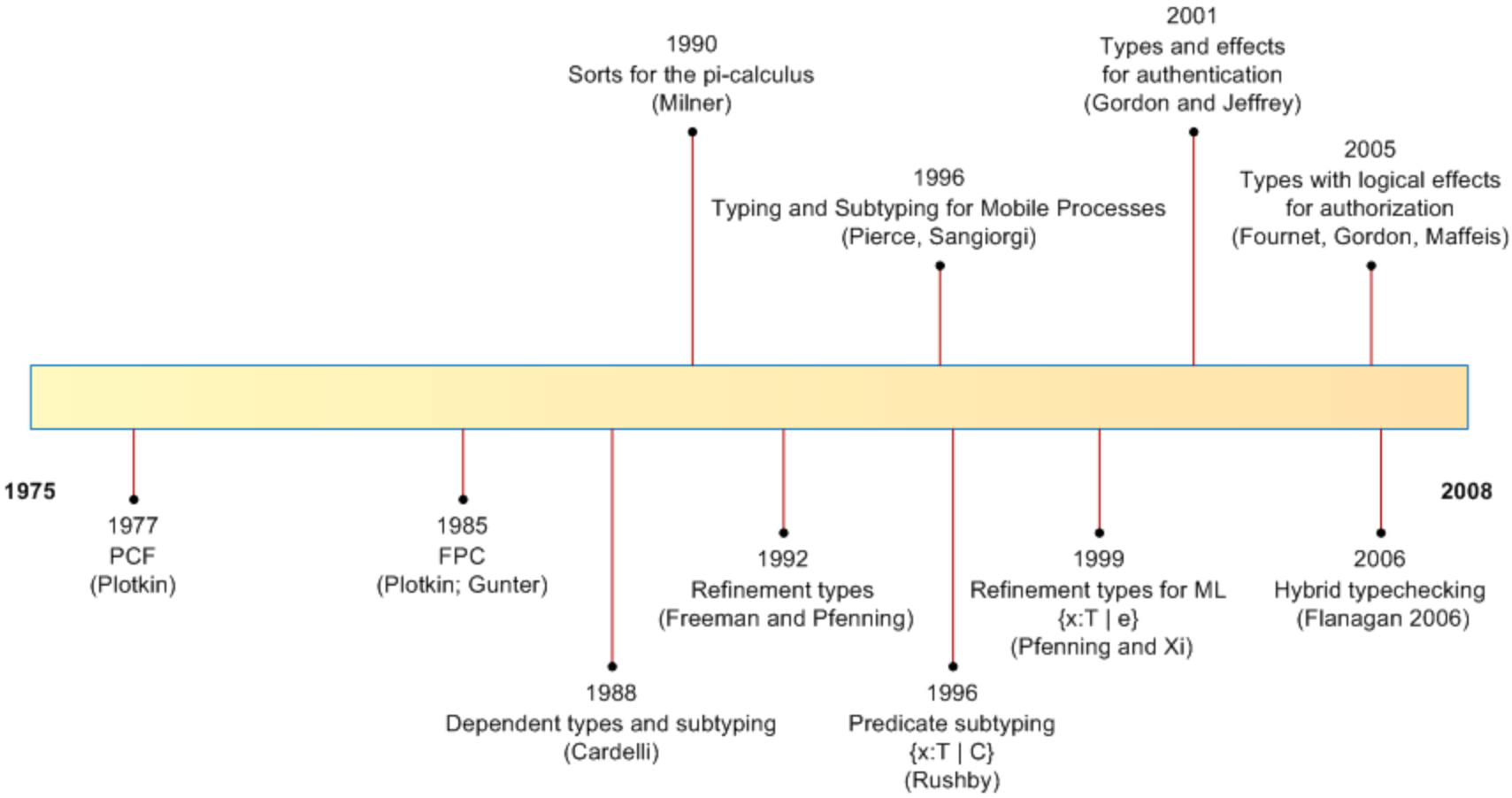
Closed expression A is *robustly safe* iff the application $O\ A$ is safe, for all opponents O.
Well-typed expressions are safe, but not in general robustly safe.
Consider $\textbf{fun}\, x : \mathsf{pos} \to (\textbf{assert}\ x > 0)$ where $\mathsf{pos} \triangleq \{x : \mathsf{int} \mid x > 0\}$.
Type T is *public* iff all refinements occur positively.

- pos
- $\mathsf{int} \to \mathsf{pos}$
- $\mathsf{pos} \to \mathsf{int}$
- $(\mathsf{pos} \to \mathsf{int}) \to \mathsf{int}$

We extend the type system with a type Un and public/tainted rules to get:

Lemma 1 (Opponent Typability) *If O is any opponent then $\varnothing \vdash O : \mathsf{Un}$.*

Theorem 1 (Robust Safety) *If $\varnothing \vdash A : T$ and T is public then A is robustly safe.*

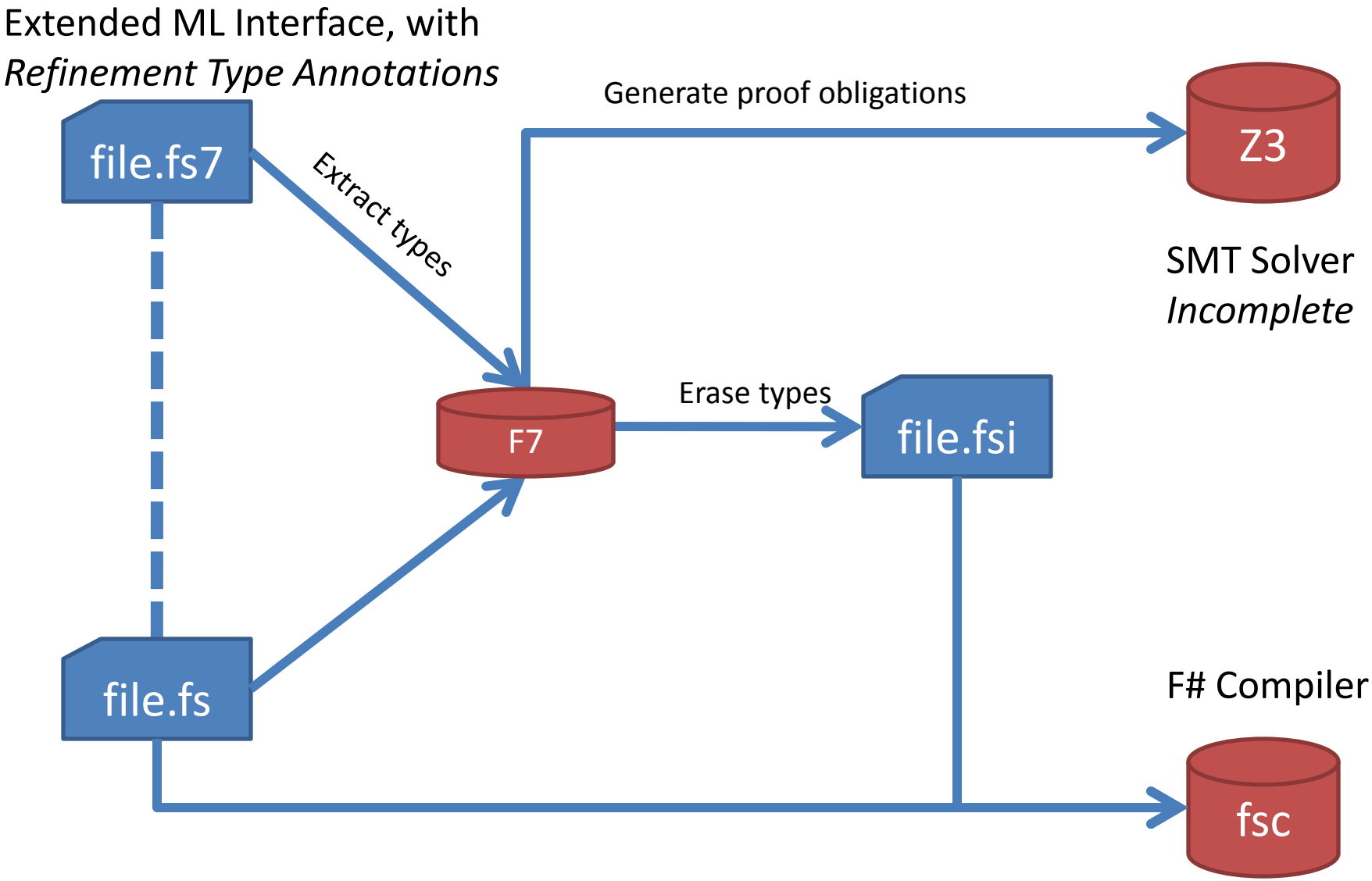


TYPE THEORIES BEHIND RCF

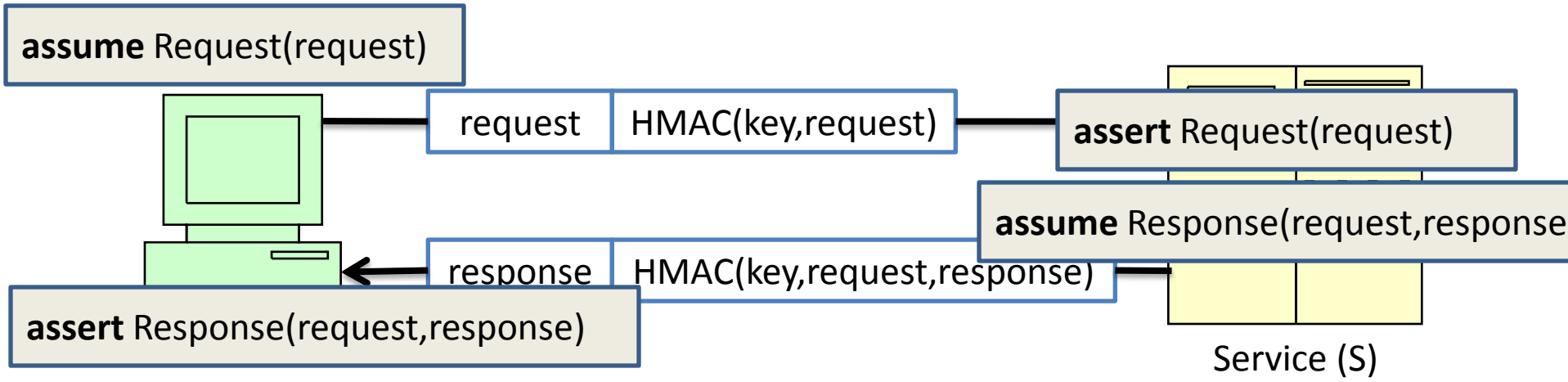
F7: AN IMPLEMENTATION OF RCF

http://research.microsoft.com/F7

F7 Typechecker Implementation



- 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



Demo: Check with F7

```
type request = r:string {Request(r)}
type response = r:string * s:string {Response(r,s)}

type info = i:string
  { (∀r. i=Concat("Request",r) ⇒ Request(r)) ∧
    (∀r,s. i=Concat("Response",Concat(r,s)) ⇒ Response(r,s)) }

private val k: info hkey
```

Limits of the F7 Model

- As usual, formal security guarantees hold only within the boundaries of the model
  - We keep model and implementation in sync – they're the same!
  - 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 – ongoing

Performance on Larger Protocols

Example	F# Program		F7 Typechecking		Fs2Pv Verification	
	Modules	Lines of Code	Interface	Checking Time	Queries	Verifying Time
Cryptographic Patterns	1	158 lines	100 lines	17.1s	4	3.8s
Basic Protocol (Section 2)	1	76 lines	141 lines	8s	4	4.1s
Otway-Rees (Section 4.2)	1	265 lines	233 lines	1m.29.9s	10	8m 2.2s
Otway-Rees (No MACs)	1	265 lines	-	(Type Incorrect)	10	2m 19.2s
Secure Conversations (Section 4.3)	1	123 lines	111 lines	29.64s	-	(Not Verified)
Web Services Security Library	5	1702	475	48.81s	(Not Verified Separately)	
X.509-based Client Auth (Section 5.1)	+ 1	+ 88 lines	+ 22 lines	+ 10.8s	2	20.2s
Password-X.509 Mutual Auth(Section 5.2)	+ 1	+ 129 lines	+ 44 lines	+ 12s	15	44m
X.509-based Mutual Auth	+ 1	+ 111 lines	+ 53 lines	+ 10.9s	18	51m
Windows Cardspace (Section 5.3)	1	1429 lines	309 lines	6m3s	6	66m 21s

Table 1. Verification Times and Comparison with ProVerif

- F7's compositional type-checking is scaling better than ProVerif's whole-program analysis on these examples
- Still, ProVerif can find attack traces;
  maybe ProVerif's analysis can be modularized?

Cryptographic Verification Kit

MICROSOFT SECURITY DEVELOPMENT LIFECYCLE

SECURITY ENGINEERING & COMMUNITY

OCTOBER 1, 2008

Version 4.1

> Innovative use of cryptographic constructs often results in subtle (or not so subtle) mistakes. Using standard algorithms in standard ways, or getting expert advice from the crypto board greatly reduces the odds of a problem.

Our goal is a toolkit to verify reference implementations of standardized and custom cryptographic protocols

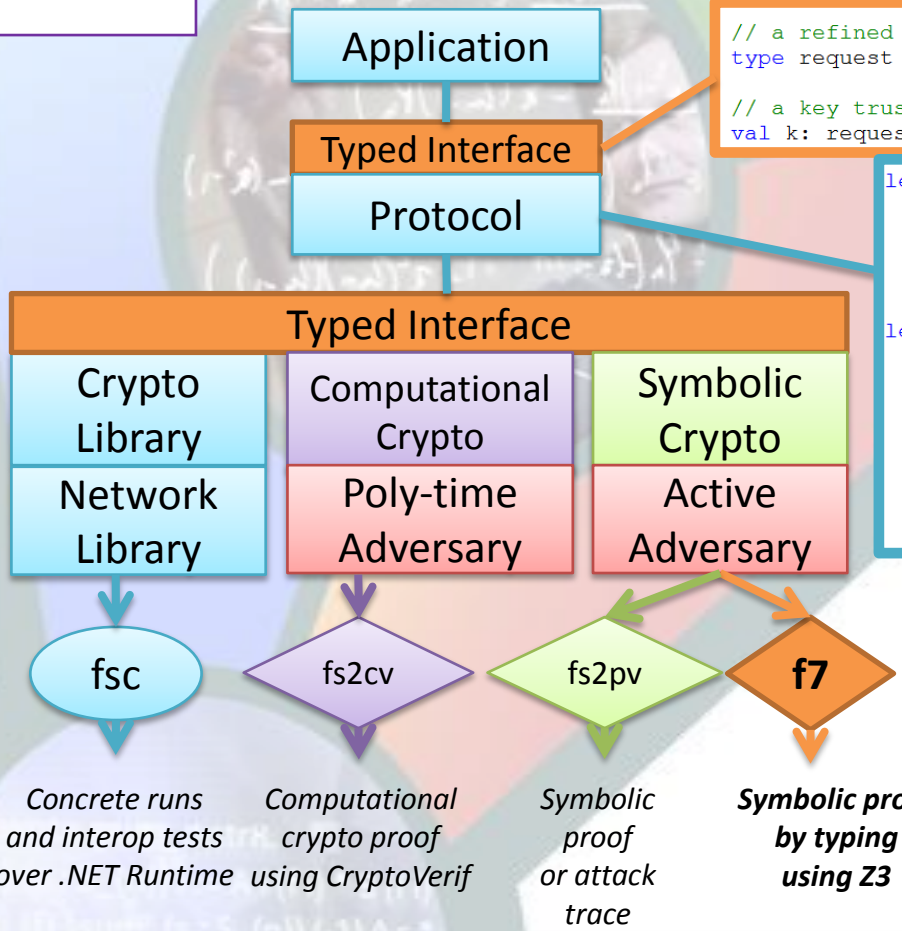
Case Studies

WS-Security
1750 lines
fs2pv [MSRC'06]

CardSpace
1420 lines
fs2pv [MSRC'08]

TLS 1.0
2940 lines
fs2pv, fs2cv [MSR-INRIA'08]

Multi-party Sessions
2180 lines
f7 [MSR-INRIA'08]




```
// a refined type for strings that are genuine requests
type request = s:string {Request(s)}

// a key trusted for authenticating requests
val k: request hkey
```

```
let verify message =
  let payload, mac = unpickle message in
  // is this the right cryptographic check?
  let request = hmacVerify k payload mac in
  request

let server() =
  let socket = Net.listen addr in
  let message = Net.recv socket in
  let request = verify message in
  // can I be sure this request is authentic?
  assert Request(request);
  let response = handle request in
  let msg2 = emitResponse request response in
  Net.send socket msg2
```

Verification Tools for F#

- Statically verify security assertions
- Different techniques, cryptographic models

K Bhargavan, C Fournet, AD Gordon (MSR Cambridge), R Corin, P-M Deniélou, JJ Leifer, E Zalinescu (MSR-INRIA)

Microsoft Research

Centre de Recherche Commun INRIA Microsoft Research

Summary of Part 3

- RCF is an assembly of standard parts, generalizing some ad hoc constructions in language-based security
- It underpins F7, a scalable verifier for security code
- There are many open questions around RCF: equivalences, type inference, mutable state, information flow

- http://research.microsoft.com/F7

More Ideas to Take Away

- Remember the riddle
  - **Q:** How did the typechecker decide $\forall x.\ \forall y.\ x>y \Rightarrow x-y>0$?
  - **A:** It didn't. It didn't even try. It asked an SMT solver.
- Remember that boundaries are blurring
  - Between types, predicates, policies, patterns, schemas
  - Between typechecking and verification
- There are several automatic tools for analyzing protocol models
  - Blanchet's ProVerif and CryptoVerif are prominent examples
- There are also several tools for analyzing reference implementations
  - FS2PV and FS2CV compile F# code to Blanchet's tools
  - F7 typechecks F# directly, relies on Z3 for local refinement checking

http://research.microsoft.com/en-us/people/adg/part.aspx

THE END