Session Types, Typestate and Security

Simon Gay
School of Computing Science
University of Glasgow

Session Types

Type-theoretic specification of communication protocols, so that protocol implementations can be verified by static type-checking.

Maths server (server side)

```
T = &{ plus: ?int.?int.!int.T,
       neg: ?int.!int.T,
       quit: end }
```

S = dual(T)

Maths server (client side)

```
S = +{ plus: !int.!int.?int.S,
       neg: !int.?int.S,
       quit: end }
```

Session Types

Makes sense in any concurrent/distributed setting with point-to-point communication channels, as long as each endpoint of a channel has a unique owner
(i.e. two-party protocols).

Multi-party protocols:

1. Channels are mobile, so protocols can be delegated.
2. Recent work on multi-party session types
   (Carbone, Honda & Yoshida 2008, and more since then).

Concentrate on two-party protocols.

Session Types

Developed by Honda (1993); Takeuchi, Honda & Kubo (1994); Honda, Vasconcelos & Kubo (1998) for process calculus.

Subtyping for session types:
Gay & Hole (1999, 2005); Gay (2008)

Session types for functional languages:
Gay, Ravara & Vasconcelos (2003,2004,2006)
Neubauer & Thiemann (2004)

Session types for operating system services:
Fähndrich et al. (2006)

Session types for object-oriented languages:
Dezani-Ciancaglini et al. (2005, 2006, 2007)
Gay et al. (2009, 2010, 2011)

+ component-based systems, security, web services, ...

Outline

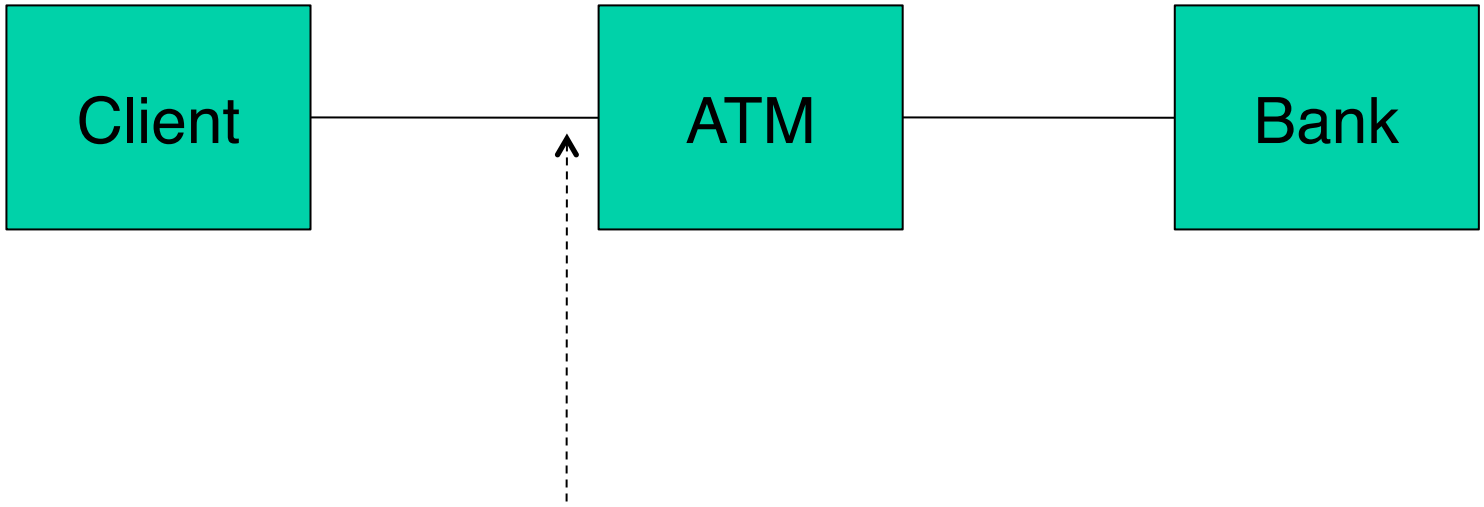
Bonelli, Compagnoni and Gunter (2003): session types + correspondence assertions for authenticity properties.

Corin et al. (2007): synthesis of security protocols from session types.

Generalisation from session types to typestate, and then:

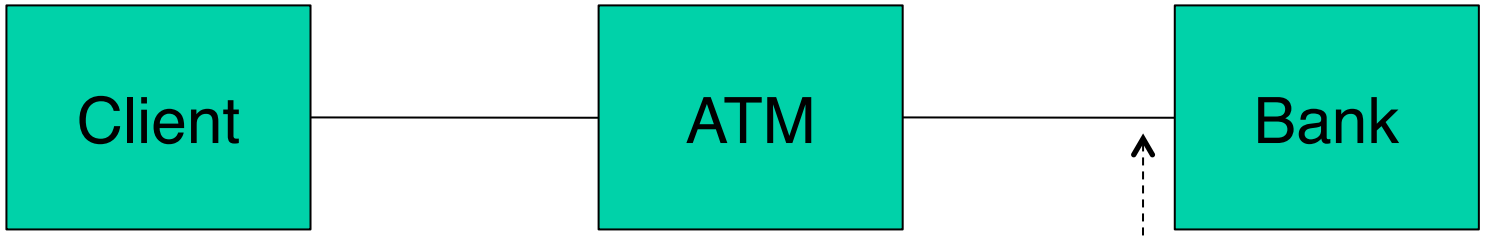
Typestate in security APIs.

Example (Bonelli et al)



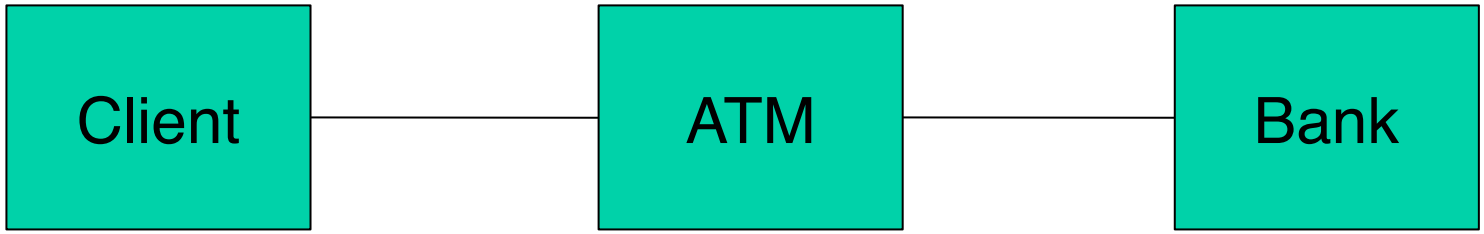
?Acct . &{ deposit: ?int . ! int . 0,
withdraw: ?int . !int . 0 }

Example (Bonelli et al)



&{ deposit: ?Acct . ?int . ! int . 0,
withdraw: ?Acct . ?int . !int . 0 }

Example (Bonelli et al)

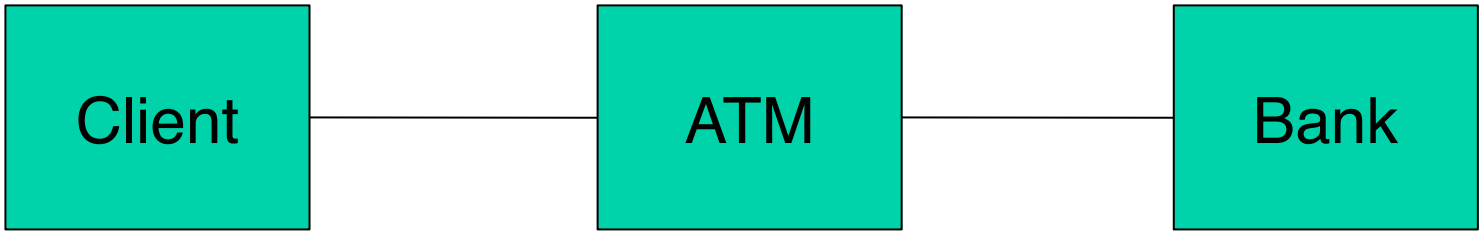


access point for ATM

```
Client(acctC,amtC,a) =
  let k = request a
  send acctC on k
  select deposit on k
  send amtC on k
  let balC = receive k
  stop
```

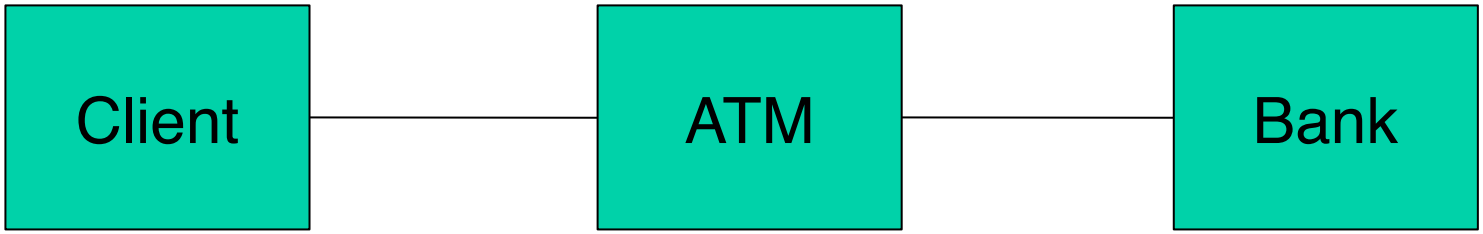
channel for this session

Example (Bonelli et al)



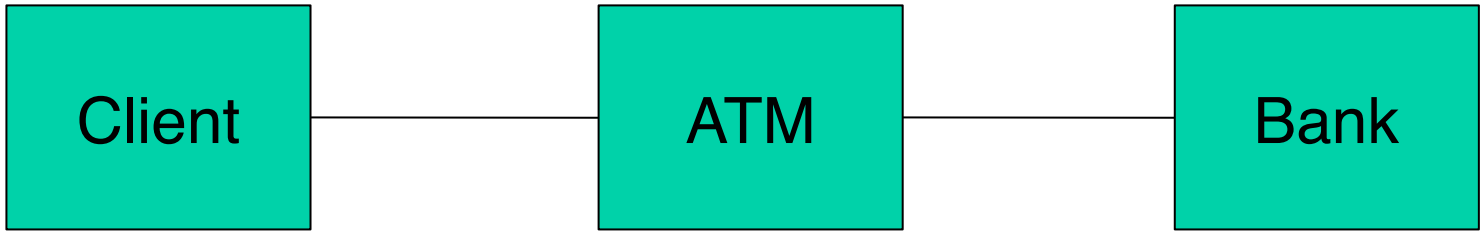

```
ATM(a,b) =
  let k = accept a
  let acctA = receive k
  offer k { deposit: let h = request b
                     let amtA = receive k
                     select deposit on h
                     send acctA on h
                     send amtA on h
                     let balA = receive h
                     send balA on k
                     ATM(a,b)
            withdraw: … }
```

Example (Bonelli et al)



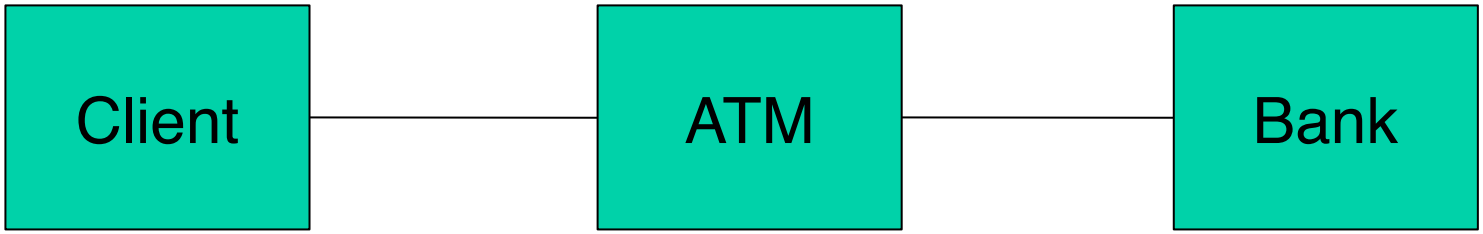

```
Bank(b) =
  let h = accept b
  offer h { deposit: let acctB = receive h
                     let amtB = receive h
                     let balB = update(acctB,amtB)
                     send balB on h
                     Bank(b)
            withdraw: … }
```

Example (Bonelli et al)



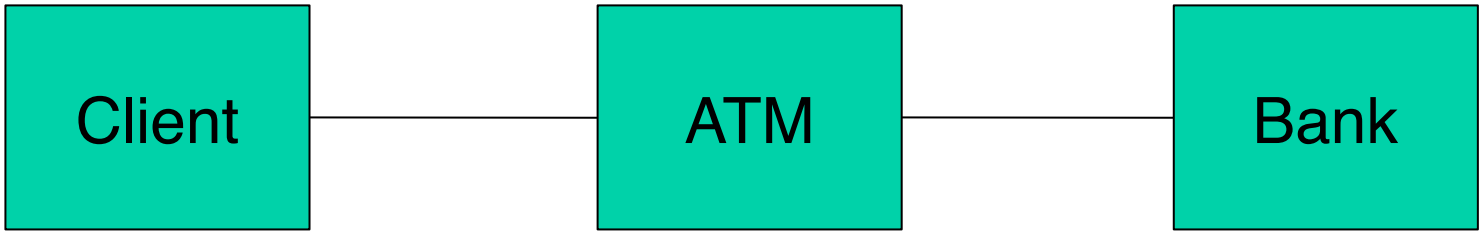
Standard session typing guarantees that all messages are sent and received as expected. This can be verified by static type checking.

Example (Bonelli et al)




```
ATM(a,b) =
  let k = accept a
  let acctA = receive k
  offer k { deposit: let h = request b
                     let amtA = receive k
                     select deposit on h
                     send acctA on h
                     send amtA on h
                     let balA = receive h
                     send balA on k
                     ATM(a,b)
            withdraw: … }
```

Example (Bonelli et al)




```
ATM(a,b) =
  let k = accept a
  let acctA = receive k
  offer k { deposit: let h = request b
                     let amtA = receive k
                     select deposit on h
                     send acctA on h
                     send amtA-10 on h
                     let h' = request b
                     select deposit on h'
                     send diffAcct on h'
                     send 10 on h' …
```

This version is also typable.

Example (Bonelli et al)

We can type an ATM that follows the protocols correctly but does not do correct banking.

This doesn't mean that the type system is wrong – it just means that the type system doesn't address questions of correct banking, only questions of correct communication.

Bonelli et al. extend the type system with correspondence assertions, following Gordon & Jeffrey (2002). The idea of correspondence assertions themselves comes from Woo & Lam (1993).

Correspondence Assertions

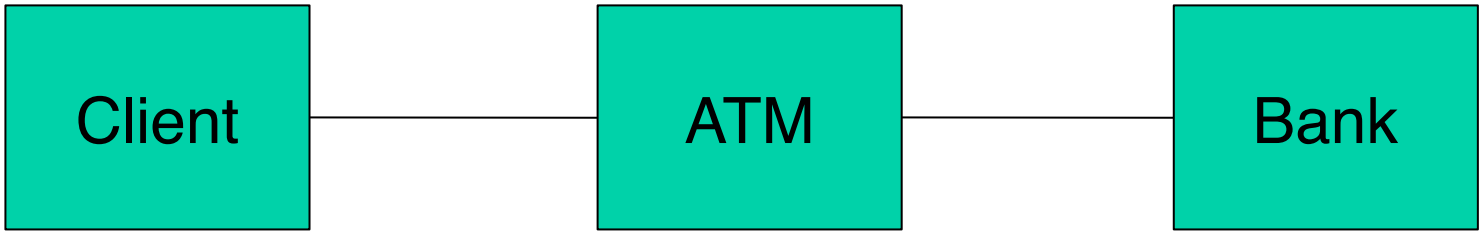
The idea is to introduce assertions begin L and end L (where L ranges over some set of labels) and then prove that in every run of a system, every end L is preceded by a corresponding begin L .

For example, begin <acct,amt> interpreted as “client acct asked to deposit amt” and end <acct,amt> interpreted as “ATM asked to deposit amt for client acct”.

Saying that every end has a matching begin is a safety property.

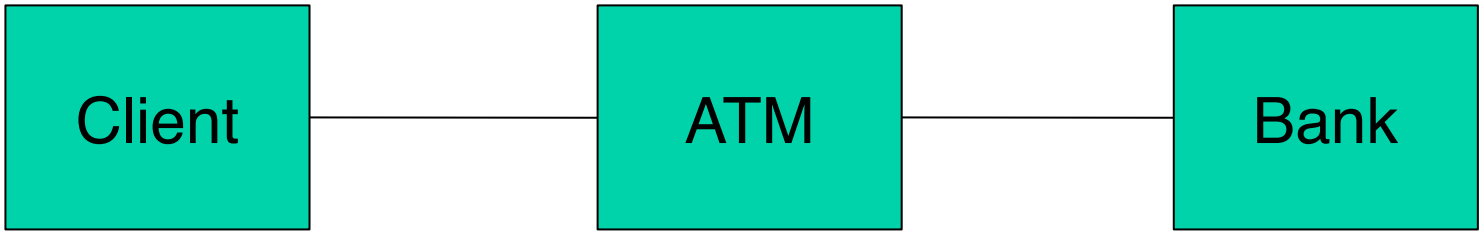
Could prove it by model-checking, but Bonelli et al. (following Gordon & Jeffrey) do it by typing.

Client with Assertion




```
Client(acctC,amtC,a) =
  let k = request a
  begin <acctC,amtC>
  send acctC on k
  select deposit on k
  send amtC on k
  let balC = receive k
  stop
```

Bank with Assertion




```
Bank(b) =
  let h = accept b
  offer h { deposit: let acctB = receive h
                     let amtB = receive h
                     end <acctB,amtB>
                     let balB = update(acctB,amtB)
                     send balB on h
                     Bank(b)
            withdraw: … }
```

Verifying Correspondence Assertions

The idea of the type system is that every program fragment has a typing of the form

$$P : [\ L1,\ldots Ln\ ]$$

The interpretation is that the multiset [L1,…,Ln] is an upper bound on the labels that P might end but not begin.
A soundness theorem states that this interpretation is obeyed by the semantics.

The aim is to give the top-level program the type [ ] .

Verifying Correspondence Assertions

For sequential programs the typing rules are very simple, e.g.

… end L : [L]

begin L … end L : [ ]

but this is not very interesting. The key is dealing with parallel composition.

Sending a message can also transfer a begin obligation to the receiver. So the bank can match its end <acct,amt> with the begin <acct,amt> received from the ATM, which in turn is received from the Client.

A Note on Type Checking Algorithms

Bonelli et al.'s type system can be implemented, although the communications (in the session type) that transfer begin obligations must be explicitly annotated. These annotations can be seen as useful documentation of the protocol.

In order to type check the banking example, where we want to know that the amount of the deposit is correct, the type checker must be able to prove (in)equalities in first order linear arithmetic, e.g. amtA-10 ≠ amtA.

Comparing Bonelli et al. with Gordon & Jeffrey

Bonelli et al.'s type system is simpler because the session types already give a lot of information about the relative order of events.

```
?Acct . &{ deposit: ?int . ! int . 0,
           withdraw: ?int . !int . 0 }
```


everything in Client before this message is before everything in ATM after this message

Gordon & Jeffrey don't have session types, so instead they use a "nonce handshake protocol" to synchronize between components, and attach begin obligations to the nonce.

Comparing Bonelli et al. with Gordon & Jeffrey

Gordon & Jeffrey have a much stronger safety property: the correspondence assertions match in the presence of any attacker (and the attacker is untyped).

There is some work to do in following up this difference, which might also be related to the next point…

Also, correspondence assertions in multi-party session types have not been investigated.

Secure Implementations for Typed Session Abstractions

Corin, Deniélou, Fournet, Bhargavan, Leifer
CSF 2007 + later work

Work on session types for programming languages assumes that the whole system will be type checked.

Corin et al. take a different view, and regard (multi-party) session types as specifications for security protocols: the specified sequence of messages must be sent securely, with authentication of the sender.

E.g. security protocols typically contain mechanisms for keeping track of which session the parties are in – the idea is to use session types to specify this.

Secure Implementations for Typed Session Abstractions

Corin et al. have produced a complete compiler system based on an extension of F#.

> *Our main result is that, when reasoning about programs that use our session implementation, one can safely assume that all session peers comply with their roles—without trusting their remote implementations.*

If you are interested in specifying and/or implementing security protocols then session types are relevant. They might also be a good starting point for further analysis.

Typestate and Security

Typestate is the idea that the type of an object should specify not only which operations are possible, but also when they are possible.

Roughly speaking: some sequences of method calls are not allowed.

The aim is to develop type systems that verify typestate properties statically.

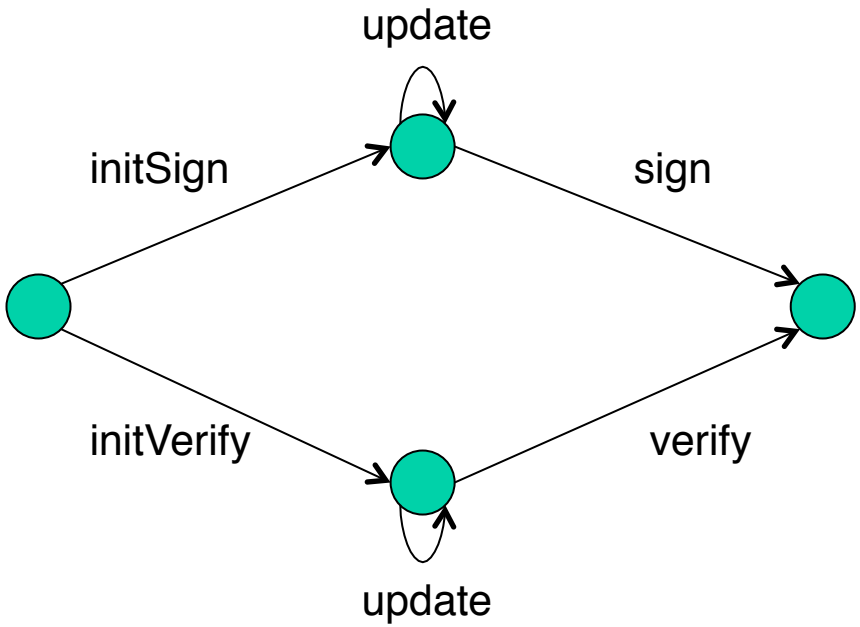
References: Yemini & Strom 1986;
Vault & Fugue (DeLine & Fahndrich 2001-);
Plural / Plaid (Aldrich et al. 2007-);
Gay et al. 2009, 2010, 2011; Hanoi (McGinniss 2010)

Typestate and Session Types

The connection between session types and typestate is that a channel with a session type can be viewed as an object with non-uniform method availability: send and receive are only available when the session type says that they should be available.

Example: java.security.Signature

An interface for objects that work with digital signatures: either signing data or verifying a signature.



Example: java.security.Signature

It should be possible to specify this transition diagram in a typestate system such as Plural or Hanoi.

(Aldrich et al. have done a lot of work with naturally-occurring APIs; I don't know whether or not they have looked at java.security).

If you are interested in security APIs, you might find typestate systems interesting, or you might be able to give me other naturally-occurring examples.

Conclusion

Several connections between session types (and the related topic of typestate) and security have emerged from recent work.

There should be possibilities for further work in several different directions.