Abstraction Preservation and Secure Sessions
in Distributed Languages

PhD defense of Pierre-Malo Deniélou

MOSCOVA Project (INRIA) MSR-INRIA Joint Centre

Advisors: Jean-Jacques Lévy and James Leifer

25/01/2009

Pierre-Malo.Denielou@inria.fr
http://moscova.inria.fr/~denielou/these/

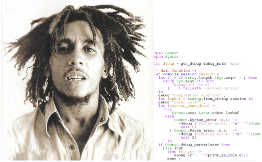
Distributed systems

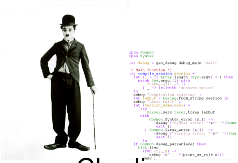
Alice

A distributed system

Independent programs that realise a global task through network interactions

Bob





Charlie

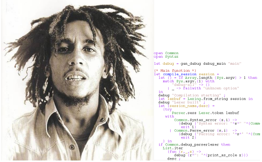
Distributed systems

Alice

A distributed system

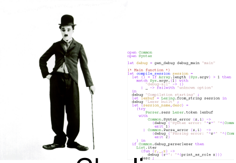
Independent programs that realise a global task through network interactions

Bob



They need to agree

- on data semantics
  Misunderstanding
- on protocols
  Miscommunication



Charlie

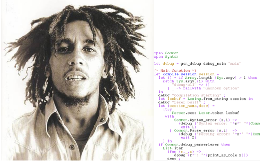
Distributed systems

Alice

A distributed system

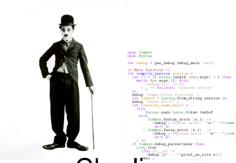
Independent programs that realise a global task through network interactions

Bob



They need to agree

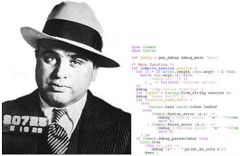
- on data semantics
  Misunderstanding
- on protocols
  Miscommunication



Charlie

Distributed systems

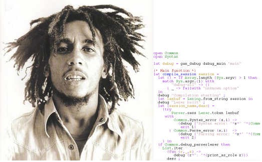
Al... Capone



A distributed system

Independent programs that realise a global task through network interactions

Bob



Charlie

They need to agree

- on data semantics
  Misunderstanding
- on protocols
  Miscommunication

There is little trust

- Errors (Safety)
  Typing system
- Corruption (Security)
  Cryptographic protocol

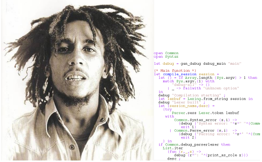
Distributed systems

Alice

A distributed system

Independent programs that realise a global task through network interactions

Bob



They need to agree

- on data semantics
  Misunderstanding
- on protocols
  Miscommunication

Charlie

There is little trust

- Errors (Safety)
  Typing system
- Corruption (Security)
  Cryptographic protocol

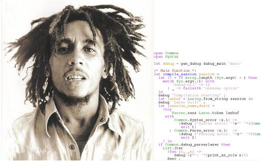
Distributed systems

Alice

A distributed system

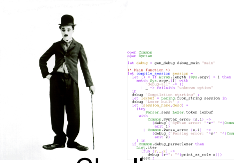
Independent programs that realise a global task through network interactions

Bob



They need to agree

- on data semantics
  Misunderstanding
- on protocols
  Miscommunication



Charlie

There is little trust

- Errors (Safety)
  Typing system
- Corruption (Security)
  Cryptographic protocol

Improving Distributed Programming

Different from sequential programming

- Independent programs need to cooperate: **safety**.
- Complicated interactive software: easier to **generate**/**prove** than to program/debug.
- No control over the execution environment (peers, network): **security**.

Improving Distributed Programming

Different from sequential programming

- Independent programs need to cooperate: **safety**.
- Complicated interactive software: easier to **generate**/**prove** than to program/debug.
- No control over the execution environment (peers, network): **security**.

Most existing tools are not well-suited

- Compilers and type systems are local.
- Security and networking libraries are low-level, binary.

Improving Distributed Programming

Different from sequential programming

- Independent programs need to cooperate: **safety**.
- Complicated interactive software: easier to **generate/prove** than to program/debug.
- No control over the execution environment (peers, network): **security**.

Most existing tools are not well-suited

- Compilers and type systems are local.
- Security and networking libraries are low-level, binary.

Contribution I: Abstract Type Safety

- How to enforce local semantics in a distributed environment

Improving Distributed Programming

Different from sequential programming

- Independent programs need to cooperate: **safety**.
- Complicated interactive software: easier to **generate/prove** than to program/debug.
- No control over the execution environment (peers, network): **security**.

Most existing tools are not well-suited

- Compilers and type systems are local.
- Security and networking libraries are low-level, binary.

Contribution I: Abstract Type Safety

- How to enforce local semantics in a distributed environment

Contribution II: Session Security

- How to secure a distributed execution despite compromised parties

Improving Distributed Programming

Different from sequential programming

- Independent programs need to cooperate: **safety**.
- Complicated interactive software: easier to **generate/prove** than to program/debug.
- No control over the execution environment (peers, network): **security**.

Most existing tools are not well-suited

- Compilers and type systems are local.
- Security and networking libraries are low-level, binary.

Contribution I: Abstract Type Safety

- How to enforce local semantics in a distributed environment

Contribution II: Session Security

- How to secure a distributed execution despite compromised parties

Computer science = Engineering ∩ Mathematics

- industrial objects: prototyping
- experiments and measures:

experimental method

- logical objects: mathematical definition
- theorems and proofs:

formal method

Part I

Abstraction preservation and subtyping

Abstract type preservation

Alice's counter

```
module Counter =
struct                        sig
 type t = int                  type t
 let init = 0          :       val init : t
 let incr x = x+1              val incr : t → t

 let value x = x               val value : t → int
end                           end
```

Bob's counter

```
module Counter =
struct                        sig
 type t = int                  type t
 let init = 0          :       val init : t
 let incr x = x+1              val incr : t → t
 let decr x = x-1              val decr : t → t
 let value x = x               val value : t → int
end                           end
```

Alice ↔ Bob

1. Alice sends `Counter.init` $\xrightarrow{0:\texttt{Counter.t}}$ Bob

Abstract type preservation

Alice's counter

```
module Counter =
struct                         sig
  type t = int                   type t
  let init = 0            :      val init : t
  let incr x = x+1               val incr : t → t

  let value x = x                val value : t → int
end                            end
```

Bob's counter

```
module Counter =
struct                         sig
  type t = int                   type t
  let init = 0            :      val init : t
  let incr x = x+1               val incr : t → t
  let decr x = x-1               val decr : t → t
  let value x = x                val value : t → int
end                            end
```

Alice ↔ Bob

1. Alice sends `Counter.init` $\xrightarrow{0:\texttt{Counter.t}}$ Bob

Abstract type preservation

Alice's counter

```
module Counter =
struct                             sig
  type t = int                       type t
  let init = 0               :       val init : t
  let incr x = x+1                   val incr : t → t

  let value x = x                    val value : t → int
end                                end
```

Bob's counter

```
module Counter =
struct                             sig
  type t = int                       type t
  let init = 0               :       val init : t
  let incr x = x+1                   val incr : t → t
  let decr x = x−1                   val decr : t → t
  let value x = x                    val value : t → int
end                                end
```

Alice ↔ Bob

1. Alice sends `Counter.init` $\xrightarrow{0:\texttt{Counter.t}}$ Bob
2. Bob applies `Counter.decr`

Abstract type preservation

Alice's counter

```
module Counter =
struct                      sig
 type t = int                type t
 let init = 0          :     val init : t
 let incr x = x+1            val incr : t → t

 let value x = x             val value : t → int
end                         end
```

Bob's counter

```
module Counter =
struct                      sig
 type t = int                type t
 let init = 0          :     val init : t
 let incr x = x+1            val incr : t → t
 let decr x = x-1            val decr : t → t
 let value x = x             val value : t → int
end                         end
```

Alice ↔ Bob

1. Alice sends `Counter.init` $\xrightarrow{0:\texttt{Counter.t}}$ Bob
2. Bob applies `Counter.decr`
3. Alice $\xleftarrow{-1:\texttt{Counter.t}}$ Bob sends the result

Abstract type preservation

Alice's counter

```
module Counter =
struct                        sig
 type t = int                  type t
 let init = 0          :       val init : t
 let incr x = x+1              val incr : t → t

 let value x = x               val value : t → int
end                           end
```

Bob's counter

```
module Counter =
struct                        sig
 type t = int                  type t
 let init = 0          :       val init : t
 let incr x = x+1              val incr : t → t
 let decr x = x−1              val decr : t → t
 let value x = x               val value : t → int
end                           end
```

Alice ↔ Bob

1.	Alice sends `Counter.init`	$\xrightarrow{0:\texttt{Counter.t}}$	Bob
2.			Bob applies `Counter.decr`
3.	Alice	$\xleftarrow{-1:\texttt{Counter.t}}$	Bob sends the result
4.	Alice applies `Counter.value`		

Abstract type preservation

Alice's counter

```
module Counter =
struct                    sig
 type t = int              type t
 let init = 0          :   val init : t
 let incr x = x+1          val incr : t → t

 let value x = x           val value : t → int
end                       end
```

Bob's counter

```
module Counter =
struct                    sig
 type t = int              type t
 let init = 0          :   val init : t
 let incr x = x+1          val incr : t → t
 let decr x = x−1          val decr : t → t
 let value x = x           val value : t → int
end                       end
```

Alice ↔ Bob

1. Alice sends `Counter.init` $\xrightarrow{0:\texttt{Counter.t}}$ Bob
2. Bob applies `Counter.decr`
3. Alice $\xleftarrow{-1:\texttt{Counter.t}}$ Bob sends the result
4. Alice applies `Counter.value`
5. **Alice fails!** (broken invariant)

Abstract type preservation

Alice's counter

```
module Counter =
struct                          sig
 type t = int                    type t
 let init = 0          :         val init : t
 let incr x = x+1                val incr : t → t

 let value x = x                 val value : t → int
end                             end
```

Bob's counter

```
module Counter =
struct                          sig
 type t = int                    type t
 let init = 0          :         val init : t
 let incr x = x+1                val incr : t → t
 let decr x = x-1                val decr : t → t
 let value x = x                 val value : t → int
end                             end
```

Alice ↔ Bob

1.	Alice sends `Counter.init`	$\xrightarrow{0:\texttt{Counter.t}}$	Bob
2.			Bob applies `Counter.decr`
3.	Alice	$\xleftarrow{-1:\texttt{Counter.t}}$	Bob sends the result
4.	Alice applies `Counter.value`		
5.	**Alice fails!** (broken invariant)		

Abstract types refer to *local* modules.

Type safety requires more than comparing names.

- different internal invariants
- different concrete types
- different dependencies

A solution using hashes and colour brackets

- Leifer, Peskine, Sewell, Wansbrough:
  "Global abstraction-safe marshalling with hash types", ICFP 2003
- ...used in Acute (ICFP 2005) and HashCaml ('ML 2006).

Idea: hash the source code of modules

- We use the hash as a unique identifier for each abstract type.
- Thus, the compiler replaces the local type name `Counter.t` by the global `h.t` where `h` is the hash of `Counter` (recursively dealing with dependencies).
- Each change yields a new hash.
- We can easily compare abstract types dynamically at unmarshall time by a simple equality check on hashes. Thus, type errors are detected at the earliest possible moment.
- Coloured brackets are used to track abstract values during evaluation.

Present contributions

Motivation: More flexibility

- We want to exchange values between executables running different versions of modules (upgrades, bug fixes, ...).
- Compatibility after a module upgrade is not necessarily symmetric!

$\Longrightarrow$ We model this by a subtyping relation.

Our contributions:

We give a sound semantics for subtyping with hashing, coloured brackets and marshalling.

1. Records and structural subtyping for concrete types
2. User-declared subtyping between abstract types
3. Partial abstract types (bounded existentials)

User-declared Subtyping

Alice's counter

```
module CounterA =
struct                          sig
 type t = int                    type t
 let init = 0             :      val init : t
 let incr x = x+1                val incr : t → t

 let value x = x                 val value : t → int
end                             end
```

Bob's counter

```
module CounterB =
struct                          sig
 type t = int                    type t
 let init = 0             :      val init : t
 let incr x = x+1                val incr : t → t
 let decr x = x-1                val decr : t → t
 let value x = x                 val value : t → int
end                             end
```

The invariants of `CounterA.t` and `CounterB.t` are different but they are compatible in one direction.

Problem: No way in general to infer the invariant compatibility, thus preventing potentially useful and safe communications. Solution:

Bob's counter

```
module CounterB extends CounterA =
                ...
```

Then we'll only be able to use `CounterA.t <: CounterB.t`.

Summary (1/2): final semantics

Type system (85 rules)

- Singleton kinds (à la Harper & Lillibridge) and bounded kinds
- Subtyping
- Type equivalence
- ...

Operational semantics (30 rules)

- Machines (compilation): $H, m \rightarrow_c H', m'$ (2 rules)
- Expressions (run-time execution): $H, e \rightarrow_c H', e$ (21 rules)
- Networks (communication): $n \rightarrow n'$ (7 rules)

Summary (2/2): Theorems

Abstraction preservation is a combination of two results.

Type Preservation

If $\vdash_c^H e : T$ and $H, e \rightarrow_c H', e'$ then $\vdash_c^{H'} e' : T$.

Typing Unicity

If $\vdash_c^H e : T_0$ and $\vdash_c^H e : T_1$, then $\vdash_c^H T_0 == T_1$

Progress

If $\vdash_c^H e : T$ then one of the following holds:

- e is a value in the colour c, blocked on I/O, or an exception.
- e reduces, i.e. there exist e' and H' such that $H, e \rightarrow_c H', e'$.

Part II

Compiler for secure sessions

Securing distributed languages

Uncertainty over the execution environment

The programmer has little control over:

- the network
- the remote peers

$\Rightarrow$

Securing distributed languages

Uncertainty over the execution environment

The programmer has little control over:

- the network
- the remote peers

$\Rightarrow$

Only realistic security assumption

Everyone is potentially malicious.

Securing distributed languages

Uncertainty over the execution environment

The programmer has little control over:

- the network
- the remote peers

$\Rightarrow$

Only realistic security assumption

Everyone is potentially malicious.

Designing a (correct) security protocol is hard

- Involves low-level, error-prone coding below communication abstractions.
- Depends on global message choreography.
- Should handle compromised peers.

Securing distributed languages

Uncertainty over the execution environment

The programmer has little control over:

- the network
- the remote peers

⇒

Only realistic security assumption

Everyone is potentially malicious.

Designing a (correct) security protocol is hard

- Involves low-level, error-prone coding below communication abstractions.
- Depends on global message choreography.
- Should handle compromised peers.

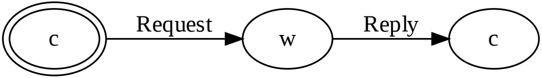
Our goal

- To automatically generate taylored cryptographic protocols protecting against the network and compromised peers;
- To hide implementation details with a *clear* semantics and proofs of correctness

Sessions (protocols, contracts, conversations, workflows, ...)

How do we specify a message flow between several roles?

- They can be represented as global graphs;



- or as local processes (our concrete syntax).

```
session Rpc =
  role c : int =
    send Request : string ;
    recv Reply : int
  role w : unit =
    recv Request : string →
    send Reply : int
```

Active area of research

- Pi-calculus, web services, operating systems
- Common strategy: type systems enforce protocol compliance
  if every site program is well-typed, sessions follow their specification

Secure compilation of session abstractions

Contributions

- Design of a high-level session language
- Automated generation of a secure implementation from the specification

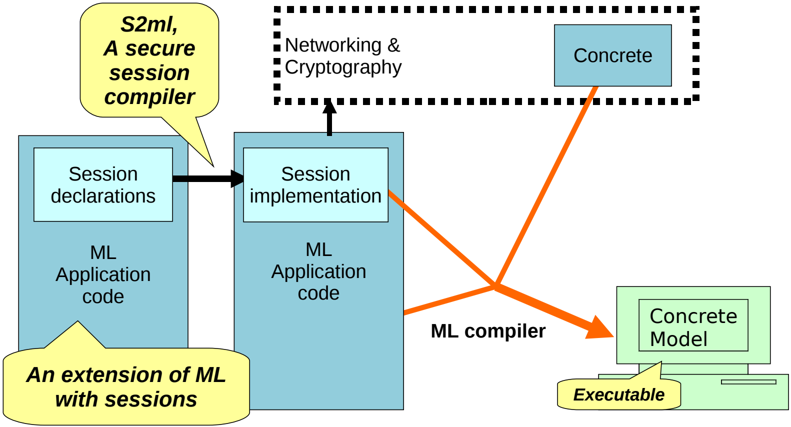
Results

- **Functional result**: Well-typed programs play their role
- **Security theorem**: A role using our generated implementation can assume that remote peers play their role without having to trust them.

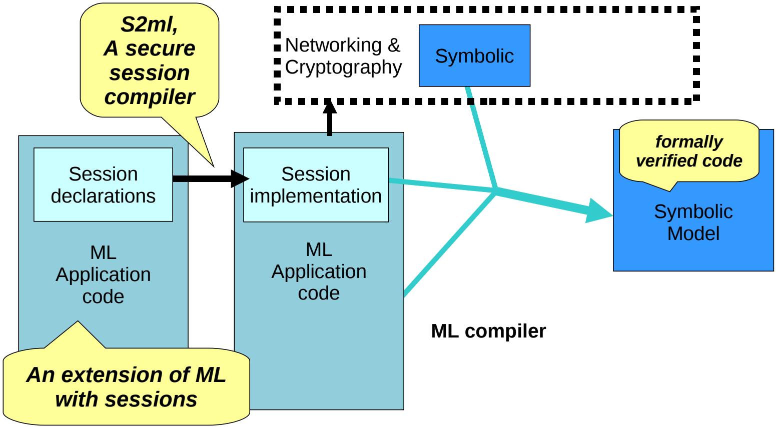
Outline:

1. Session programming & examples
2. Security threats
3. Generated protocol
4. Theorem
5. Performance evaluation

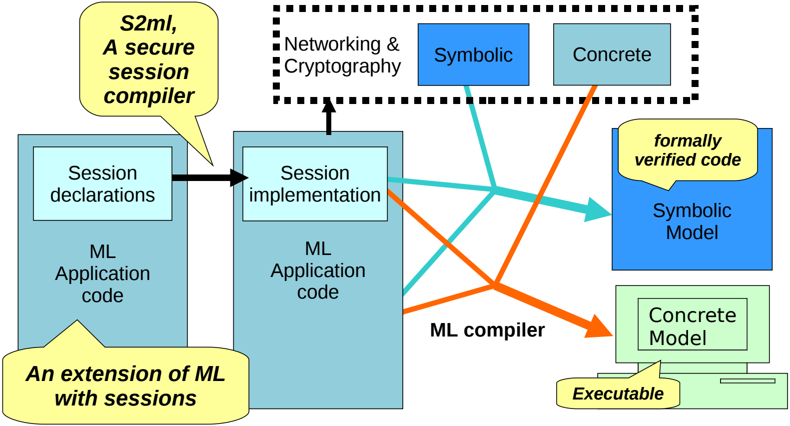
Architecture



Architecture

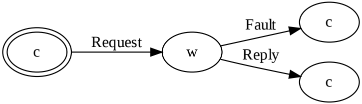


Architecture

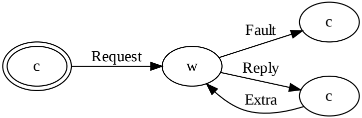


Session expressiveness

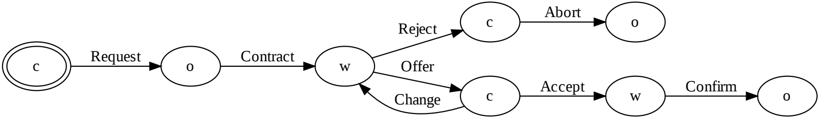
- Ws: 2 roles, 3 messages, 1 choice



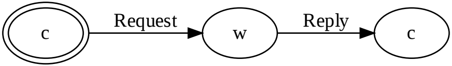
- Wsn: 2 roles, 4 messages, 1 choice, 1 loop



- Shopping: 3 roles, 8 messages, 1 choice, 1 loop



Programming with continuations



File Rpc.mli

```
(* Function for role w *)
type result_w = unit
type msg3 = {
  hRequest : (prins * string → msg4)}
and msg4 =
  Reply of (int * result_w)
val w : principal → msg3 → result_w
[...]
```

Arbitrary ML code can be used to run the session and produce the message content.

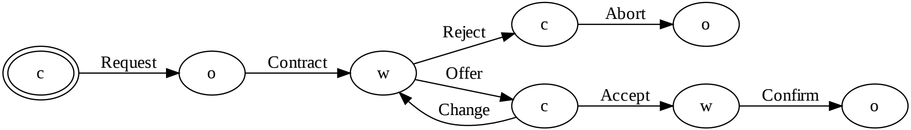
Sample user file to play w's role

```
Rpc.w "Bob"
    {hRequest = function (_,x) → match x with "Cheese" → Reply(24,())
                                            | "Wine"   → Reply(53,())}
```

Threats against session integrity

Powerful Attacker model

- can spy on transmitted messages
- can join a session as any role
- can initiate sessions
- can access the librairies (networking, crypto)
- cannot forge signatures



Attacks against an unsecure implementation

- Message integrity (`Offer` by `Reject`)
- Message replay (`Offer` triggers a new iteration)
- Control integrity (from `Reject` to `Change`)
- Sender authentication (`c` could send `Confirm` to `o`)

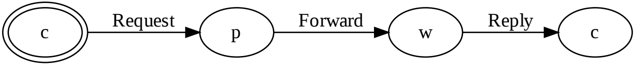
Protocol outline

Principles of our protocol generation

1. Each edge is implemented by a unique concrete message.
2. We want static message handling for efficiency.

Against replay attacks

- between session executions: session nonces
- between loop iterations: time stamps
- at session initialisations: anti-replay caches



Against session flow attacks

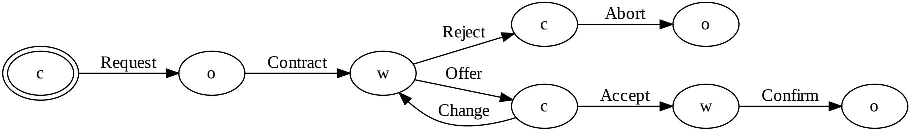
- Signatures of the entire message history (optimisations possible ...)

Visibility

Optimising the protocol

Signing and countersigning the full history

1. Using time stamps to avoid countersigning
2. Using local states to remember past achievements



Execution paths: which signatures to convince the receiver?

- `Request-Contract-`**`Reject-Abort`**
- `Request-Contract-Offer-Change-Offer-`**`Change`**
- `Request-Contract-(Offer-Change)`^{n}`-`**`Reject-Abort`**

Visibility: at most one signature from each of the previous roles is enough.

Session integrity

Our formalism:

- F+S is our high-level language where sessions are primitive;
- F is our low-level language without sessions (ie ML);
- F$\subseteq$ F+S.

Theorem (Session integrity)

If $L\ M_{\widetilde{S}}\ U\ O'$ may fail in F *then $L\ \widetilde{S}\ U\ O$ may fail in* F+S*.*

Intuition

- L is the set of libraries.
- $\widetilde{S}$ is a set of session declarations and $M_{\widetilde{S}}$ their generated session implementation.
- Failure is a barb raised by the user code U.
- U is the same code in F+S and F.
- O cannot make U see an observable difference between F+S and F.

Evaluation

Performance of the code generation

Session S	Rôles	Fichier .session (loc)	Application (loc)	Graphe (loc)	Graphes Locaux (loc)	S.mli (loc)	S.ml (loc)	Compilation (s)
Single	2	5	21	8	12	19	247	1.26
Rpc	2	7	25	10	18	23	377	1.35
Forward	3	10	33	12	25	34	632	1.66
Auth	4	15	45	16	38	49	1070	1.86
Ws	2	7	33	12	24	25	481	1.36
Wsn	2	15	44	13	42	29	782	1.50
Wsne	2	19	45	15	48	31	881	1.90
Shopping	3	29	70	21	85	49	1780	2.43
Conf	3	48	86	37	181	78	3451	3.32
Loi	6	101	189	57	310	141	7267	6.29

Performance of the generated code (10000 messages)

Authentication using	signatures	MACs
Total execution time	93.92 s	1.77 s
Without verification	90.80 s	1.66 s
Without cryptography	1.43 s	
Unprotected	1.31 s	

Conclusion

I. Abstraction preservation

- Design of a distributed language with abstract data types and subtyping.
- Semantics to ensure abstract type safety.
- Soundness, typing unicity and progress proofs.

II. Compiler for secure session

- Design of a high-level session language
- Automated generation of a secure implementation from the specification
- Generic proof of the security protocol correctness

Conclusion

I. Abstraction preservation

- Design of a distributed language with abstract data types and subtyping.
- Semantics to ensure abstract type safety.
- Soundness, typing unicity and progress proofs.

II. Compiler for secure session

- Design of a high-level session language
- Automated generation of a secure implementation from the specification
- Generic proof of the security protocol correctness

Thank you!