Moscova 08

Jean-Jacques Lévy

INRIA Rocquencourt

May 15, 2008

Research team

Personal et history

- Present staff
  - 1 DR (Lévy)
  - 3 CR1 (Maranget, Leifer, Zappa Nardelli), 1 CR2 (Corin)
  - 4 PhD students (Peskine, Deniélou, Alglave, Guts)
  - 1 assistant (S. Loubressac)
  - 2 interns (Braibant, Planul from ENS Lyon)
- INRIA Rocquencourt ↔ MSR-INRIA Saclay (Fournet)
- Moscova history:
  - Para (1988), Head: Lévy
  - Moscova (2000), Head: Gonthier
  - 15 PhDs: Fournet[msr], le Fessant[saclay], Schmitt[grenoble], Melliès[pps], Pouzet[iuf], Conchon[orsay], Doligez, Maranget, ··· Laneve, Ariola.
  - in Para/Moscova: 75% Coq proof of the 4-color thm; debugging of 3 modules of Ariane-501 PV; spinoff of Polyspace [Deutsch]; etc.
  - recent departures: Gonthier[msr], Doligez[gallium], Hardin[p6]

Research themes

Research themes

- programming languages
  [safe marshalling, ott]
- concurrency
  [jocaml, separation logic/c-minor/concurrency, relaxed memory models]
- compiling security
  [secured sessions, tls, audits, history based information flow]

Research results

EXAMPLE 1 PoplMark ⟶ OTT

- formal semantics of SML or Acute are too large (40-80 pages)
  ⇒ tools for complete definitions of full languages
- problems:

1. Readability and writability	2. Consistency of definitions
3. Correctness of proofs	4. Relationship semantics/implementations

- *OTT*
  - ASCII as input
  - outputs to TeX, Isabelle, HOL, Coq
  - proofs are still manual

`[Sewell, Zappa Nardelli]`

`[demo]`

Call-by-value λ-calculus (1/4 – TeX)

termvar, x	term variable			
t	$::=$			term
	$\vert$	x		variable
	$\vert$	$\lambda x\,.\,t$	bind x in t	lambda
	$\vert$	$t\ t'$		app
v	$::=$			value
	$\vert$	$\lambda x\,.\,t$		lambda

$\boxed{t_1 \longrightarrow t_2}$ t_1 reduces to t_2

$$\frac{}{(\,\lambda x\,.\,t_{12}\,)\ v_2 \longrightarrow \{\,v_2\,/\,x\,\}\ t_{12}}\quad \text{AX}$$

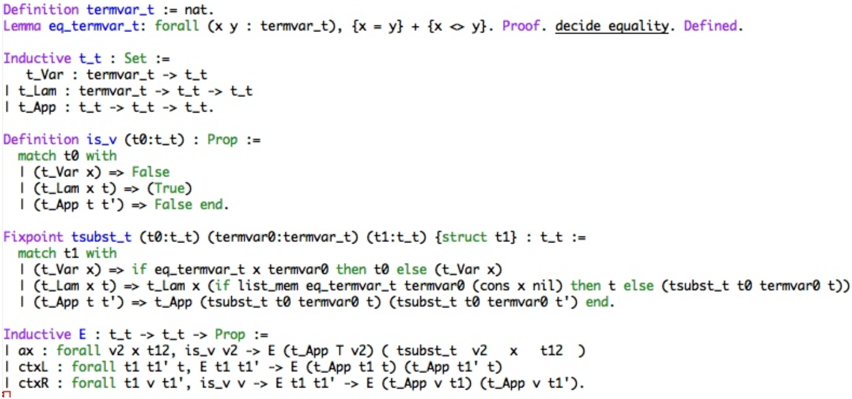
$$\frac{t_1 \longrightarrow t_1'}{t_1\ t \longrightarrow t_1'\ t}\quad \text{CTXL}$$

$$\frac{t_1 \longrightarrow t_1'}{v\ t_1 \longrightarrow v\ t_1'}\quad \text{CTXR}$$

Call-by-value λ-calculus (2/4 – Ott)

```
metavar termvar, x ::=   {{ tex \mathit{[[termvar]]} }}  {{ com  term variable }}
    {{ isa string}} {{ coq nat}} {{ hol string}} {{ coq-equality }}
    {{ ocaml int}} {{ lex alphanum}}

grammar
t :: ’t_’ ::=                               {{ com term    }}
  | x              ::   :: Var                  {{ com variable}}
  | \ x . t        ::   :: Lam (+ bind x in t +) {{ com lambda   }}
  | t t’           ::   :: App                  {{ com app      }}
  | ( t )          :: M :: Paren                {{ icho [[t]]   }}
  | { t / x } t’   :: M :: Tsub                 {{ icho (tsubst_t [[t]] [[x]] [[t’]])}}

v :: ’v_’ ::=                               {{ com value   }}
  | \ x . t        ::   :: Lam                  {{ com lambda   }}

subrules v <:: t           substitutions single t x :: tsubst

defns Jop :: ’’ ::=
  defn t1 --> t2 :: :: reduce :: ’’ {{ com [[t1]] reduces to [[t2]]}} by

                                      t1 --> t1’                t1 --> t1’
-------------------------  :: ax      -------------- :: ctxL    -------------- :: ctxR
(\x.t12) v2 -->  {v2/x}t12            v t1 --> v t1’            t1 t --> t1’ t
```

Call-by-value λ-calculus (3/4 – Coq)



```
Definition termvar_t := nat.
Lemma eq_termvar_t: forall (x y : termvar_t), {x = y} + {x <> y}. Proof. decide equality. Defined.

Inductive t_t : Set :=
   t_Var : termvar_t -> t_t
 | t_Lam : termvar_t -> t_t -> t_t
 | t_App : t_t -> t_t -> t_t.

Definition is_v (t0:t_t) : Prop :=
  match t0 with
  | (t_Var x) => False
  | (t_Lam x t) => (True)
  | (t_App t t') => False end.

Fixpoint tsubst_t (t0:t_t) (termvar0:termvar_t) (t1:t_t) {struct t1} : t_t :=
  match t1 with
  | (t_Var x) => if eq_termvar_t x termvar0 then t0 else (t_Var x)
  | (t_Lam x t) => t_Lam x (if list_mem eq_termvar_t termvar0 (cons x nil) then t else (tsubst_t t0 termvar0 t))
  | (t_App t t') => t_App (tsubst_t t0 termvar0 t) (tsubst_t t0 termvar0 t') end.

Inductive E : t_t -> t_t -> Prop :=
| ax : forall v2 x t12, is_v v2 -> E (t_App T v2) ( tsubst_t  v2   x   t12  )
| ctxL : forall t1 t1' t, E t1 t1' -> E (t_App t1 t) (t_App t1' t)
| ctxR : forall t1 v t1', is_v v -> E t1 t1' -> E (t_App v t1) (t_App v t1').
```

Call-by-value λ-calculus (4/4 – Isabelle)

...and for Isabelle...

```
theory out = Main:
types termvar = "string"

datatype t =
   t_Var "termvar"
 | t_Lam "termvar" "t"
 | t_App "t" "t"

consts is_v :: "t => bool"
primrec
"is_v ((t_Var x)) = False"
"is_v ((t_Lam x t)) = (True)"
"is_v ((t_App t t’)) = False"

consts tsubst_t :: "t => termvar => t => t"
primrec
"tsubst_t t0 termvar0 (t_Var x) = (if x=termvar0 then t0 else (t_Var x))"
"tsubst_t t0 termvar0 (t_Lam x t) = (t_Lam x (if termvar0 mem [x] then t else (tsubst_t t0 termvar0 t)))"
"tsubst_t t0 termvar0 (t_App t t’) = (t_App (tsubst_t t0 termvar0 t) (tsubst_t t0 termvar0 t’))"

consts E :: "(t*t) set"  inductive E
intros
ax_appI: "[|is_v v2|] ==> ( (t_App T v2) , ( tsubst_t  v2   x   t12  ) ) : E"
ctx_app_funI: "[| ( t1 , t1’ ) : E|] ==> ( (t_App t1 t) , (t_App t1’ t) ) : E"
ctx_app_argI: "[|is_v v ; ( t1 , t1’ ) : E|] ==> ( (t_App v t1) , (t_App v t1’) ) : E"
end
```

Lists: a more typical not-so-mini example

$$\frac{\begin{array}{l} E \vdash e_1 : t_1 \quad \ldots \quad E \vdash e_n : t_n \\ E \vdash \mathit{field\_name}_1 : t \rightarrow t_1 \quad \ldots \quad E \vdash \mathit{field\_name}_n : t \rightarrow t_n \\ t = (\, t'_1 \,, \ldots , \, t'_l \,)\ \mathit{typeconstr\_name} \\ E \vdash \mathit{typeconstr\_name} \ \triangleright\ \mathit{typeconstr\_name} : \mathit{kind}\,\{\,\mathit{field\_name}'_1 ; \ldots ; \mathit{field\_name}'_m\,\} \\ \mathit{field\_name}_1 \ldots \mathit{field\_name}_n\ \mathbf{PERMUTES}\ \mathit{field\_name}'_1 \ldots \mathit{field\_name}'_m \\ \mathbf{length}\,(\,e_1\,)\ldots(\,e_n\,) \geq 1 \end{array}}{E \vdash \{\,\mathit{field\_name}_1 = e_1 ; \ldots ; \mathit{field\_name}_n = e_n\,\} : t}$$

```
E |- e1 : t1 ... E |- en : tn
E |- field_name1 : t->t1 ... E |- field_namen : t->tn
t = (t1’, ..., tl’) typeconstr_name
E |- typeconstr_name gives typeconstr_name:kind {field_name1’; ...; field_name
field_name1...field_namen PERMUTES field_name1’...field_namem’
length (e1)...(en)>=1
---------------------------------------------------------------------- ::
E |- {field_name1=e1; ...; field_namen=en} : t
```

- proof of the subject reduction theorem for Ocaml without objects + modules in 7 weeks

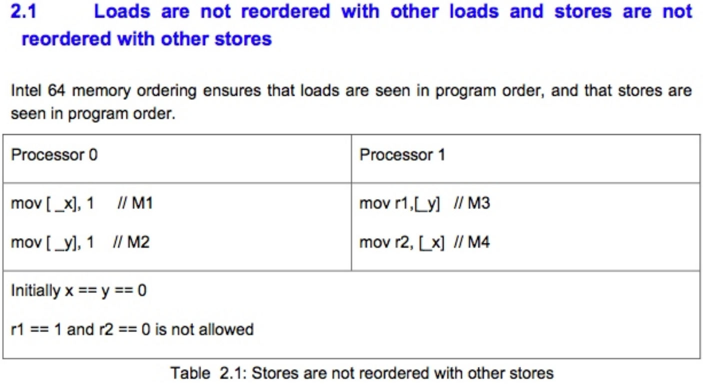
  (3 Harper-years)

Relaxed memory models

- description of Intel [Cambridge] / PPC [INRIA] memory model constraints using event structures
- formalisation in Isabelle/Coq, symbolic evaluator, test wrt processor behaviour
- formal proof of simple concurrent code (eg. Linux spinlocks)
- operational reasoning: data-race freedom, separation logic
- certified compiler back-ends for concurrent primitives

`[Zappa Nardelli, Alglave, Braibant, Sewell et al]`

Intel whitepaper (1/3)



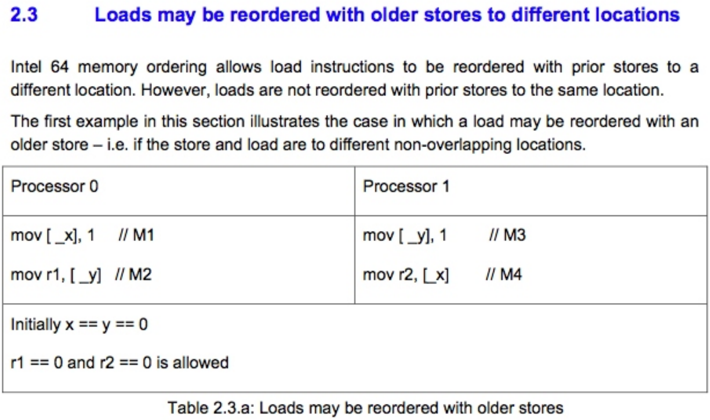
2.1 Loads are not reordered with other loads and stores are not reordered with other stores

Intel 64 memory ordering ensures that loads are seen in program order, and that stores are seen in program order.

Processor 0	Processor 1
mov [_x], 1 // M1 mov [_y], 1 // M2	mov r1,[_y] // M3 mov r2, [_x] // M4
Initially x == y == 0 r1 == 1 and r2 == 0 is not allowed	

Table 2.1: Stores are not reordered with other stores

Intel whitepaper (2/3)



2.3 Loads may be reordered with older stores to different locations

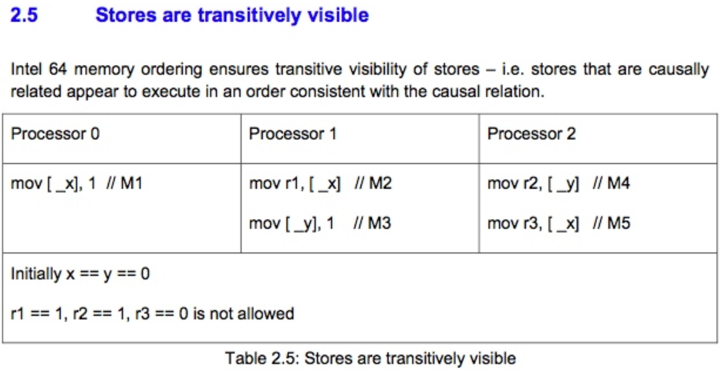
Intel 64 memory ordering allows load instructions to be reordered with prior stores to a different location. However, loads are not reordered with prior stores to the same location.

The first example in this section illustrates the case in which a load may be reordered with an older store – i.e. if the store and load are to different non-overlapping locations.

Processor 0	Processor 1
mov [_x], 1 // M1 mov r1, [_y] // M2	mov [_y], 1 // M3 mov r2, [_x] // M4
Initially x == y == 0 r1 == 0 and r2 == 0 is allowed	

Table 2.3.a: Loads may be reordered with older stores

Intel whitepaper (3/3)



2.5 Stores are transitively visible

Intel 64 memory ordering ensures transitive visibility of stores – i.e. stores that are causally related appear to execute in an order consistent with the causal relation.

Processor 0	Processor 1	Processor 2
mov [_x], 1 // M1	mov r1, [_x] // M2 mov [_y], 1 // M3	mov r2, [_y] // M4 mov r3, [_x] // M5
Initially x == y == 0 r1 == 1, r2 == 1, r3 == 0 is not allowed		

Table 2.5: Stores are transitively visible

EXAMPLE 3 Secure Communication – INRIA/MSR

- passing authenticated (signed) values between 2 *run-times*;
- design of a mini F# + primitives for authentication
  + global contract with sessions types;
  `[Corin, Deniélou, Leifer, Fournet, Bhargavan, CSFW’07]`
- compiling scheme into a low-level language ($\simeq$ pi-calculus)
  to describe authentication protocols;
- formal proof of its correctness, with security property induced by
  strong typing of F# + usage of authentication primitives.
- extension to other security properties
  (privacy, integrity, sessions, etc)

F# = Ocaml − modules + .NET

EXAMPLE 3 Secure Communication – INRIA/MSR

- passing authenticated (signed) values between 2 *run-times*;
- design of a mini F# + primitives for authentication
  + global contract with sessions types;
  `[Corin, Deniélou, Leifer, Fournet, Bhargavan, CSFW'07]`
- compiling scheme into a low-level language ($\simeq$ pi-calculus)
  to describe authentication protocols;
- formal proof of its correctness, with security property induced by strong typing of F# + usage of authentication primitives.
- extension to other security properties
  (privacy, integrity, sessions, etc)

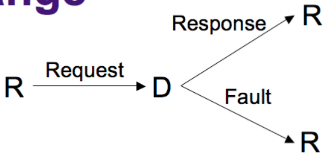
F# = Ocaml − modules + .NET

EXAMPLE 3 Secure Communication – INRIA/MSR

- passing authenticated (signed) values between 2 *run-times*;
- design of a mini F# + primitives for authentication
  + global contract with sessions types;
  `[Corin, Deniélou, Leifer, Fournet, Bhargavan, CSFW'07]`
- compiling scheme into a low-level language ($\simeq$ pi-calculus)
  to describe authentication protocols;
- formal proof of its correctness, with security property induced by strong typing of F# + usage of authentication primitives.
- extension to other security properties
  (privacy, integrity, sessions, etc)

F# = Ocaml − modules + .NET

Simple Exchange


R
Request
D
Response
R
Fault
R

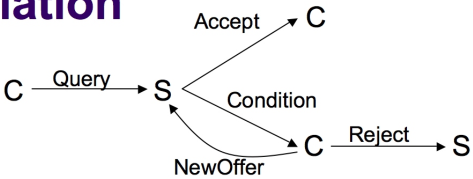
```
session S =
   role requester : int =
    !Request:string ;
    ?(Response:int + Fault:unit)

   role directory : string =
    ?Request:string;
    !(Response:int + Fault:unit)
```

Session declaration

```
let lookup name =
 S.requester ["client";"server"]
      (Request
          (name,
           {hResponse = (fun _ q → q) ;
            hFault = (fun _ x → failwith "Failed")
           }))
in lookup "Ricardo"
```

User code

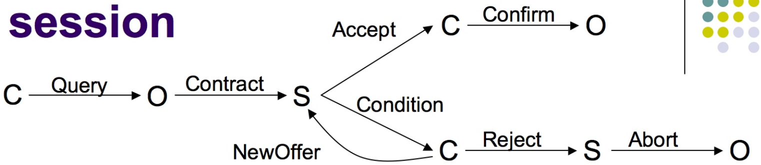
Two-party negociation



C
Query
S
Accept
C
Condition
C
Reject
S
NewOffer

```
session S2 =
  role customer : string =
   !Query:int;
     mu start.?(Accept:unit +
                Condition:unit;!(NewOffer:int;start + Reject:unit))

   role store : string=
   ?Query:int;
     mu start.!(Accept:unit +
                Condition:unit;?(NewOffer:int;start + Reject:unit))
```

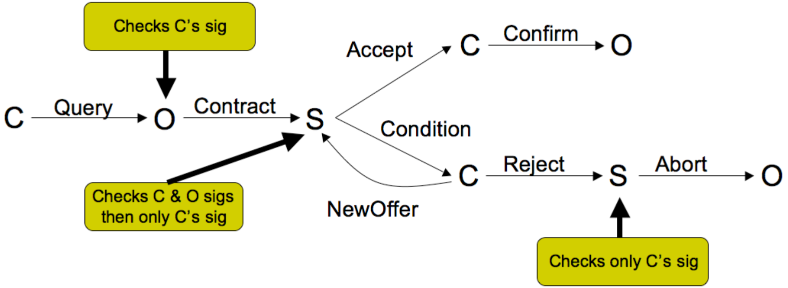
Three-party session


C Query O Contract S
Accept C Confirm O
Condition C Reject S Abort O
NewOffer

```
session S3 =
   role customer :string =
    !Query:int;
     mu start.?(Accept:unit;!Confirm:unit +
                    Condition:unit; !(Newoffer:int;start + Reject:unit;))

   role store :string=
    ?Contract:int;
     mu start.!(Accept:unit +
                    Condition:unit; ?(Newoffer:int;start + Reject:unit;!Abort:unit))

   role officer :string=
    ?Query:int;!Contract:int;?(Confirm:unit + Abort:unit)
```

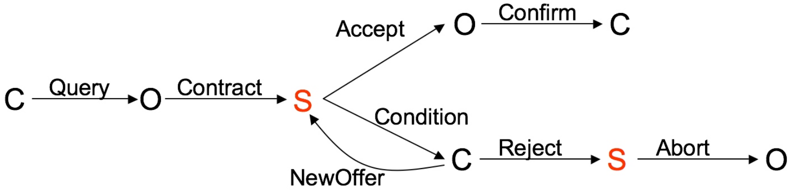
Visibility

- Minimal sequence of signatures that guarantee session compliance.
- Example:



No Blind Fork

- Some forks in protocols represent a security threat.



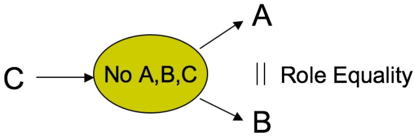
C Query O Contract S

S Accept O Confirm C

S Condition C Reject S Abort O

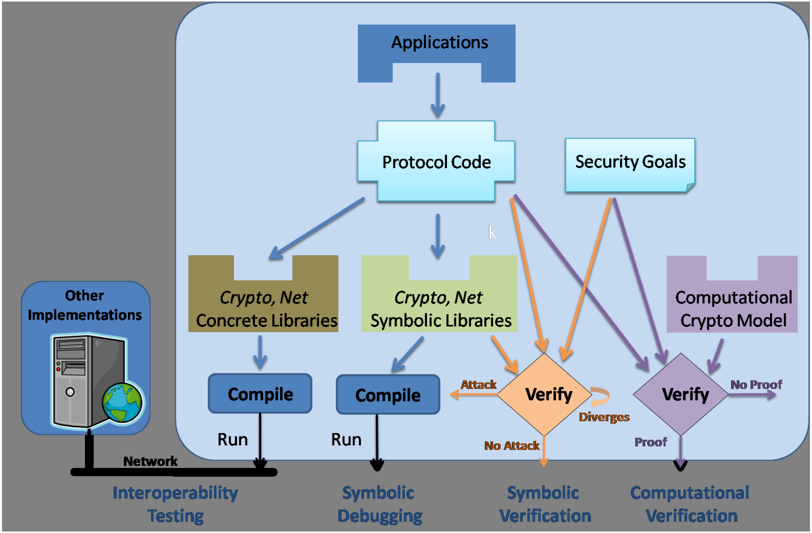
C NewOffer S

- Property



C → No A,B,C → A || Role Equality → B

EXAMPLE 4 Certified implementation of SSL/TLS



SSL/TLS

- simple TLS client C and server S written in Ocaml
- checking interoperability with other clients/servers
- proof of secured implementation of C/S in Proverif (formal security)
- same with Cryptoverif (computational security)

`[Corin, Zalinescu, Fournet]`

Other works

- Jocaml (version 3; more portable, documentation)

  `[Maranget, Mandel]`

- Security through logs

  `[Guts, Fournet, Zappa Nardelli]`

- Acute – type safed marshalling

  `[Leifer, Peskine, Zappa Nardelli]`

- Pattern-matching in Ocaml

  `[Maranget]`

- Process calculi (bigraphs, reversible processes)

  `[Leifer, Krivine]`

- History based flow analysis

  `[Blanc, Lévy]`

Miscellaneous

Links

- through Joint Centre with Microsoft Research
- ANR Parsec with Mimosa, Everest, Lande, PPS
- Gallium for general discussion about programming languages
- several projects with Computer Lab in Cambridge University
- Andrew Appel, Princeton

Competitors

- POPL/ICFP community, ...
- formal security (MSR – Abadi, Bhargavan, Gordon, etc)
- concurrency and formal proofs (Milner, Peter O'Hearn, Sewell)
- ⋯ many others

Extra softwares – Admin

- Jocaml `[Maranget, Mandel]`
- 5% Ocaml (pattern matching)
  `[Maranget]`
- Hévéa: an efficient translator of Tex into Html
  `[Maranget]`
- Advix: efficient previewer of Dvi
  `[Rémy, Zappa Nardelli]`
- Burfiks: bayesian filter for the web `[burfiks.gforge.inria.fr]`
  `[Zappa Nardelli + several indian interns]`
- Lévy as Director of the MSR-INRIA Joint Centre

Teaching

- MPRI (master course at Paris 7)
- Ecole polytechnique
  `[Lévy on leave 1/1/06 -- ??, Maranget]`
  lecture notes + web pages + book
  “Introduction à la théorie des langages de programmation”
  with `[Dowek]`, similar plan with `[Cori]`
- Entrance examination at Polytechnique
  `[Maranget (4 years), Lévy since beginning]`
- Bertinoro, Bangalore, etc.

Objectives for next years

Future

- Ott, Jocaml widely used
- easy binders in Ott
- concurrent secured sessions
- proofs of concurrent algorithms with relaxed memory models
- security with logs
- programming languages with secure primitives safely compiled.