Moscova 07

Jean-Jacques Lévy

INRIA Rocquencourt

April 24, 2007

Research

Part 1

Type-safe communication – *Acute*

- communicating values of abstract data types
  and preserving abstraction between 2 distinct run-times;
- incompatibility is not visible on type signatures;
  concrete representation must be described in passed values.

- type theory of ML modules with hashes of implementation.
  `[Sewell, Leifer, Peskine, Zappa Nardelli -- 2 × ICFP]`
- extension to records with horizontal subtyping
  `[Leifer, Deniélou -- ICFP 06]`
- extension to nested and polymorphic modules
  `[Peskine, PhD]`
- prototype on top of FreshOcaml
  \+ dynamic linking
  \+ modules versioning
  `[Sewell, Habouzit, Leifer, Peskine, Zappa Nardelli -- ICFP]`

Type-safe communication – *Acute*

- communicating values of abstract data types
  and preserving abstraction between 2 distinct run-times;
- incompatibility is not visible on type signatures;
  concrete representation must be described in passed values.

- type theory of ML modules with hashes of implementation.
  `[Sewell, Leifer, Peskine, Zappa Nardelli -- 2 × ICFP]`
- extension to records with horizontal subtyping
  `[Leifer, Deniélou -- ICFP 06]`
- extension to nested and polymorphic modules
  `[Peskine, PhD]`
- prototype on top of FreshOcaml
  + dynamic linking
  + modules versioning
  `[Sewell, Habouzit, Leifer, Peskine, Zappa Nardelli -- ICFP]`

Process algebras — Pattern Matching

Process algebras

- equivalences in Mobile Ambients

  `[Zappa Nardelli, Mero -- JACM 06]`
- reversible processes

  `[Krivine, Danos -- Concur 05-06]`
- link graphs, bi-graphs

  `[Leifer, Milner -- MSCS 06]`

Pattern matching

- disjunctive patterns + warnings in Ocaml

  `[Maranget, JFP 07]`
- synchronization by pattern matching in Jocaml

  `[Ma Qin, PhD 05 + Concur 04]`
- pattern matching a la XML/Cduce in Jocaml (future plan)

Process algebras — Pattern Matching

Process algebras

- equivalences in Mobile Ambients

  `[Zappa Nardelli, Mero -- JACM 06]`
- reversible processes

  `[Krivine, Danos -- Concur 05-06]`
- link graphs, bi-graphs

  `[Leifer, Milner -- MSCS 06]`

Pattern matching

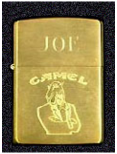
- disjunctive patterns + warnings in Ocaml

  `[Maranget, JFP 07]`
- synchronization by pattern matching in Jocaml

  `[Ma Qin, PhD 05 + Concur 04]`
- pattern matching a la XML/Cduce in Jocaml (future plan)

Jocaml

- new implementation (without mobility)
  [Maranget]
- with manual and tutorial
  [Mandel, Maranget]
- compatible with new releases of Ocaml



Join Patterns are in Polyphonic C#

Research

Part 2

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, 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

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, 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)

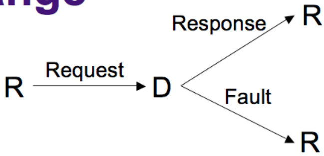
F# = Ocaml − modules + .NET

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, 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



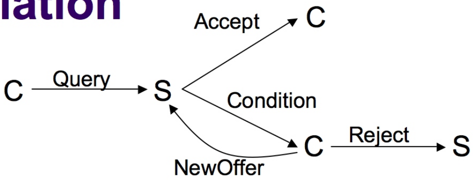
```
session S =
  role requester : int =
   !Request:string ;
   ?(Response:int + Fault:unit)

  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"
```

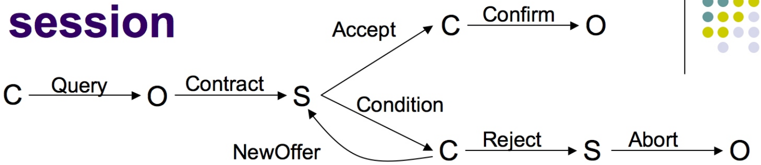
User code

Two-party negociation



```
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))
```

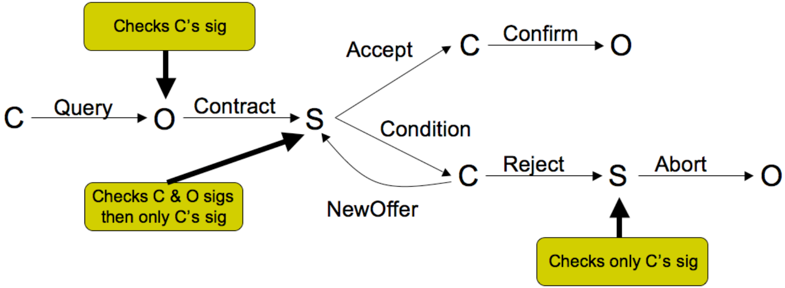
Three-party session



```
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:



Checks C's sig

C Query O Contract S

Accept C Confirm O

Condition C Reject S Abort O

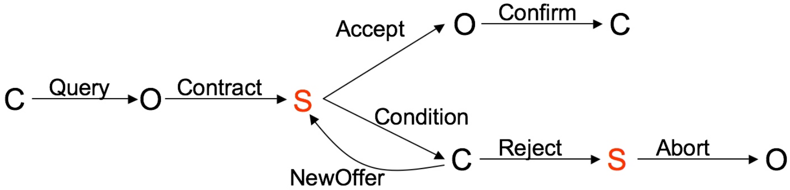
NewOffer

Checks C & O sigs then only C's sig

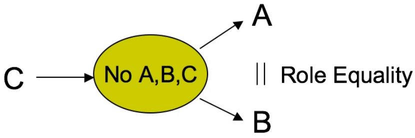
Checks only C's sig

No Blind Fork

- Some forks in protocols represent a security threat.



- Property



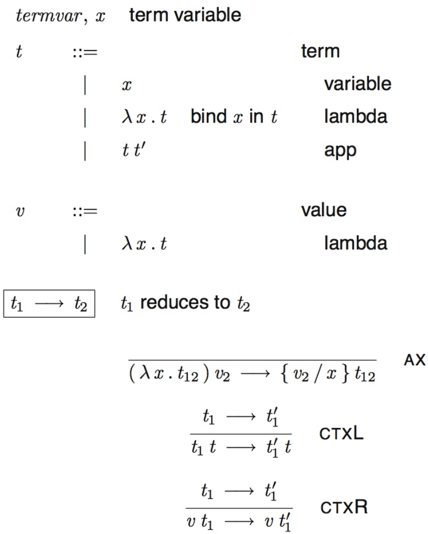
PoplMark ⟶ OTT

- 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 (2/4 – 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

$$\frac{}{(\,\lambda\, x\, .\, t_{12}\,)\; v_2 \longrightarrow \{\, v_2\, /\, x\,\}\, t_{12}}\ \textsc{ax}$$

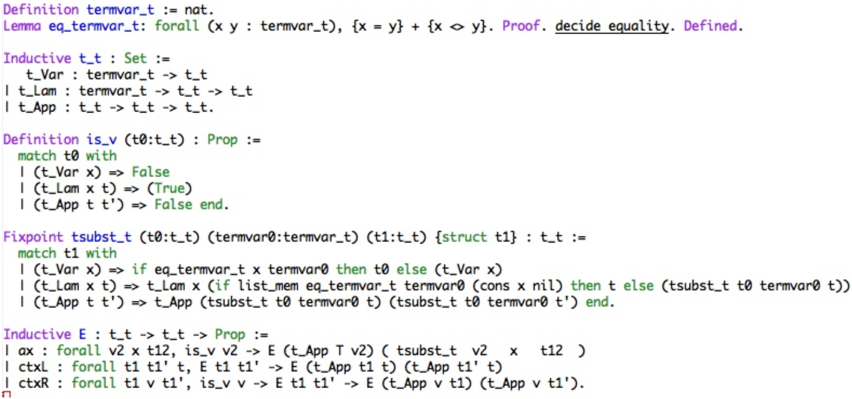
$$\frac{t_1 \longrightarrow t_1'}{t_1\; t \longrightarrow t_1'\; t}\ \textsc{ctxL}$$

$$\frac{t_1 \longrightarrow t_1'}{v\; t_1 \longrightarrow v\; t_1'}\ \textsc{ctxR}$$

Call-by-value λ-calculus (1/4 – 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)



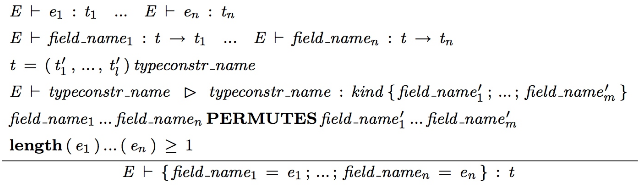
```
Definition termvar_t := nat.
Lemma eq_termvar_t: forall (x y : termvar_t), {x = y} + {x <> y}. Proof. decide equality. Defined.

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)

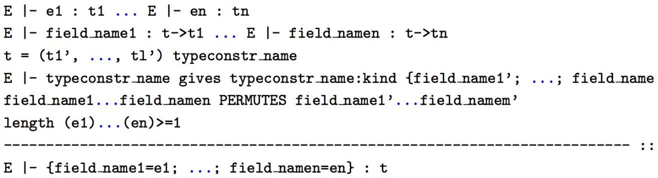
...and for Isabelle...

```
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



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



```
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)

OBJECTIVE 3 Separation logic, C-minor and concurrency

Existing

- Coq library for Peter O'Hearn's logic

  [Yonezawa et al]
- for very simple imperative languages
  (no types, no functions, no recursivity)
- POH developed a separation logic for concurrency, on top of a unrealistic model (not implementable)

  ⇒ need for relaxing the model

To do

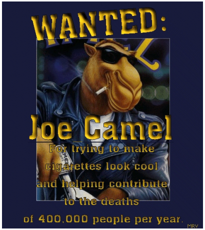
- building a new framework for formal proofs
- example: prove the correctness of *reverse* in C minor

- make proofs of *lock-free* programs

  [Appel, Blazy, Zappa Nardelli]

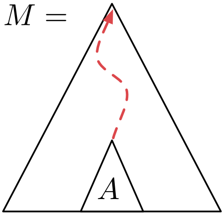
Jocaml

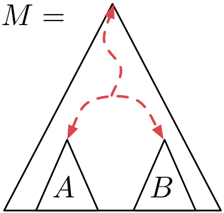
- maintaining the implementation;
- better design of active mobility;
- transform Jocaml in a platform for implementing various kinds of distributed processing.



Information flow in the λ-calculus with history

- stack inspection for JVM/CLR
  [Fournet, Gordon, Blanc]
- relate flow analysis and theory of history in the λ-calculus
  [Blanc, Lévy]





Software

and

Extras

Extra softwares – Contracts

- 5% Ocaml (pattern matching)
  `[Maranget]`
- Hévéa: an efficient translator of Tex into Html
  `[Maranget]`
- Advix: efficient previewer of Dvi
  `[Rémy, Zappa Nardelli]`

(not enough many)

- Joint Centre with Microsoft Research
- ANR Parsec with Mimosa, Everest, Lande, PPS

Teaching

- MPRI (master course at Paris 7)
- Ecole polytechnique

  `[Lévy on leave 1/1/06 -- 1/1/08, 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.

(too *undergraduate*)

Personal et history

- 1 DR (Lévy), 2 CR1 (Maranget, Leifer), 1 CR2 (Zappa Nardelli)
- 2 PhD students: Peskine, Deniélou
- 1 post-doc: Mandel
- 1 invited professor: Appel (Princeton)
- 1 assistant (S. Loubressac), also Head of SAPR

- Moscova history:
  - Para (en 88), Head: Lévy
  - Moscova (en 00), Head: Gonthier
  - 15 PhDs: Fournet[msr], le Fessant[futurs], Schmitt[grenoble], Melliès[pps], Pouzet[orsay], 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], 3 PhD students have just finished.

Conclusion

Conclusion

- Moscova once more in reconfiguration phase
- need for new researchers
- need for new PhD students
- Moscova should be more involved in softwares