Formal Logical Environments for Building Reliable, High-Performance Software

Christoph Kreitz

Department of Computer Science, Cornell University
Ithaca, NY 14853

Why Formal Formal Logical Environments?

- **Too many errors in informal arguments**
  - 40–50% of the published results turn out to be wrong
- **We can't afford errors in software development**
  - Errors are annoying (Reboot, loss of data, ...)
  - Errors are expensive (Pentium bug, Ariane 5 rocket, Mars Climate Orbiter)
  - Errors cost lives (Airbus crashes in the early 1990's)

  ... affects air traffic, banking, government, utilities, schools, e-commerce, ...
- **Current software development methods are unreliable**
  - Tested programs still contain errors
  - Correctness proofs are tedious and error-prone (if done by hand)

Need formal tools for creating reliable software

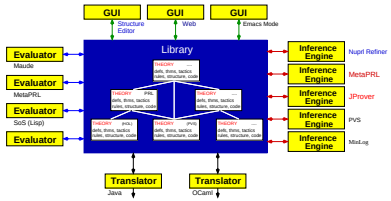
Formal methods tools are most sucessful when engaged at early stages of system design

- **Great potential**
  - clarifying critical design concepts
  - linking abstract and concrete specifications
  - detecting subtle errors in design and prototype code
  - generating code of components from specifications
  - improving system performance
- **Requires**
  - expressive formal language
  - knowledge base of formalized facts about systems concepts
  - proof environment integrating different reasoning techniques
  - collaboration with systems experts in real applications

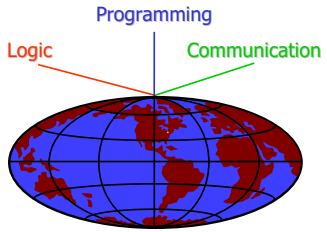
The Nuprl LPE meets these requirements

- **Computational formal logics** — TYPE THEORY

- **Proof & program development systems**
  - The NUPRL Logical Programming Environment
  - Fast inference engines + proof search techniques
  - Natural language generation from formal mathematics
  - Program extraction + automated complexity analysis



- **Application to reliable, high-performance networks**
  - Assigning precise semantics to system software
  - Performance Optimizations
  - Assurance for reliability (verification)
  - Verified System Design



Secure software infrastructure

The Nuprl Type Theory
An Instance of Martin-Löf Type Theory

- **Constructive higher-order logic**
  - Reasoning about types, members of types, propositions, functions ...
- **Functional programming language**
  - Similar to core ML: polymorphic, with partial recursive functions
- **Open-ended, expressive data type system**
  - Function, product, disjoint union, Π- & Σ-types, atoms ⇝ programming
  - Integers, lists, inductive types ⇝ inductive definition
  - Propositions as types, equality type, void, top, universes ⇝ logic
  - Subsets, subtyping, quotient types ⇝ mathematics
  - (Dependent) intersection, union, records ⇝ modules, program composition

  New types can/will be added as needed
- **Expressions separate from their types** ⇝full λ-calculus

  ... but must be typeable *in proofs* ⇝ "total" functions
- **User-defined extensions possible**

Syntax Issues

- **Uniform term syntax for all expressions**

  $opid\{p_i : F_i\}(x_{11}, .., x_{m_1 1} . t_1 ; \ldots ; x_{1n}, .., x_{m_n n} . t_n)$

  – *Operator identifier* listed in operator tables

  – Parameters for base terms (variables, numbers, tokens. . .)

  – Sub-terms may contain bound variables

  No syntactical distinction between types, members, propositions . . .

- **Separate term display form**

  – Describe visual appearance of terms in "free syntax"

  $\mapsto$ Conventional notation, information hiding, auto-parenthesizing, aliases, . . .

Internal Term Structure	*Display Form*
function$\{\}(S;\ .T)$	$S \longrightarrow T$
variable$\{x : \mathrm{v}\}()$	x
lambda$\{\}(x . t)$	$\lambda x . t$
apply$\{\}(f ; t)$	$f\ t$
⋮	⋮

Semantics models proof, not denotation

- **(Lazy) evaluation of expressions**
  - Identify canonical expressions (values)
  - Define reducible non-canonical expressions (redex)
  - Define reduction steps in redex–contracta table

canonical	*non-canonical*	*Redex*		*Contractum*
$S \to T$				
$\lambda x.t$	$\boxed{f}\ t$	$\boxed{\lambda x.u}\ t$	$\xrightarrow{\beta}$	$u[t/x]$

- **Judgments: semantical truths about expressions**
  - 4 categories: Typehood (T Type), Type Equality ($S = T$),
    Membership ($t \in T$), Member equality ($s = t \in T$)
  - Defined for values of expressions in semantics tables

$S_1 \to T_1 = S_2 \to T_2$ iff $S_1 = S_2$ and $T_1 = T_2$

$\lambda x_1.t_1 = \lambda x_2.t_2 \in S \to T$ iff $S \to T$ Type and $t_1[s_1/x_1] = t_2[s_2/x_2] \in T$ for all s_1, s_2 with $s_1 = s_2 \in S$

⋮ ⋮

NUPRL'S PROOF THEORY

- **Sequent** $x_1{:}T_1, \ldots, x_n{:}T_n \vdash C$ ⌊ext t⌋

  "If x_i are variables of type T_i then C has a (yet unknown) member t"

  – A judgment $t \in T$ is represented as T ⌊ext t⌋ ⇝ proof term construction

  – Equality is represented as type $s{=}t \in T$ ⌊ext Ax⌋ ⇝ propositions as types

  – Typehood represented by (cumulative) universes $\mathbf{U}_i$ ⌊ext T⌋

- **Refinement calculus**

  – Top-down decomposition of proof goal ⇝ interactive proof development

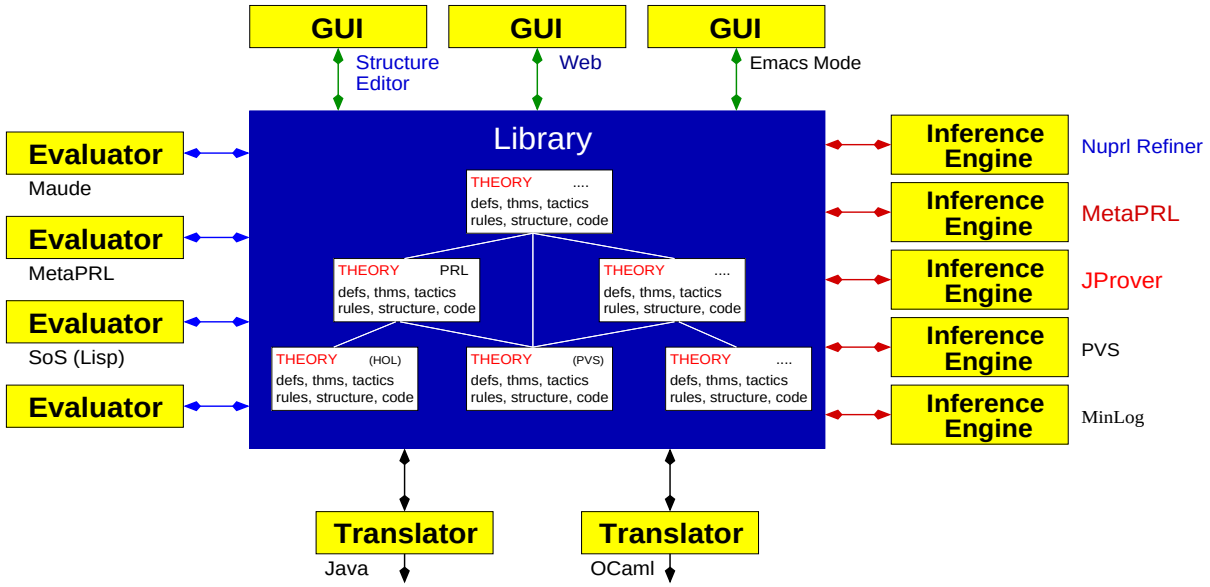
  – Bottom-up construction of proof terms ⇝ program extraction

  $\Gamma \vdash S \rightarrow T$ ⌊ext $\lambda x.e$⌋ **by** `lambda-formation` x
  $\Gamma, x{:}S \vdash T$ ⌊ext e⌋
  $\Gamma \vdash S{=}S \in \mathbf{U}_i$ ⌊ext Ax⌋

  – Computation rules ⇝ program evaluation

About 8–10 inference rules for each data type in NUPRL

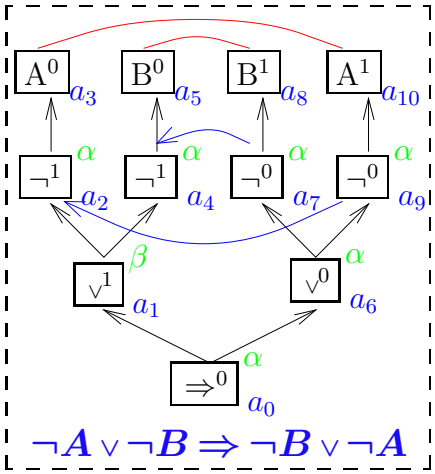
NUPRL'S AUTOMATED REASONING ENVIRONMENT



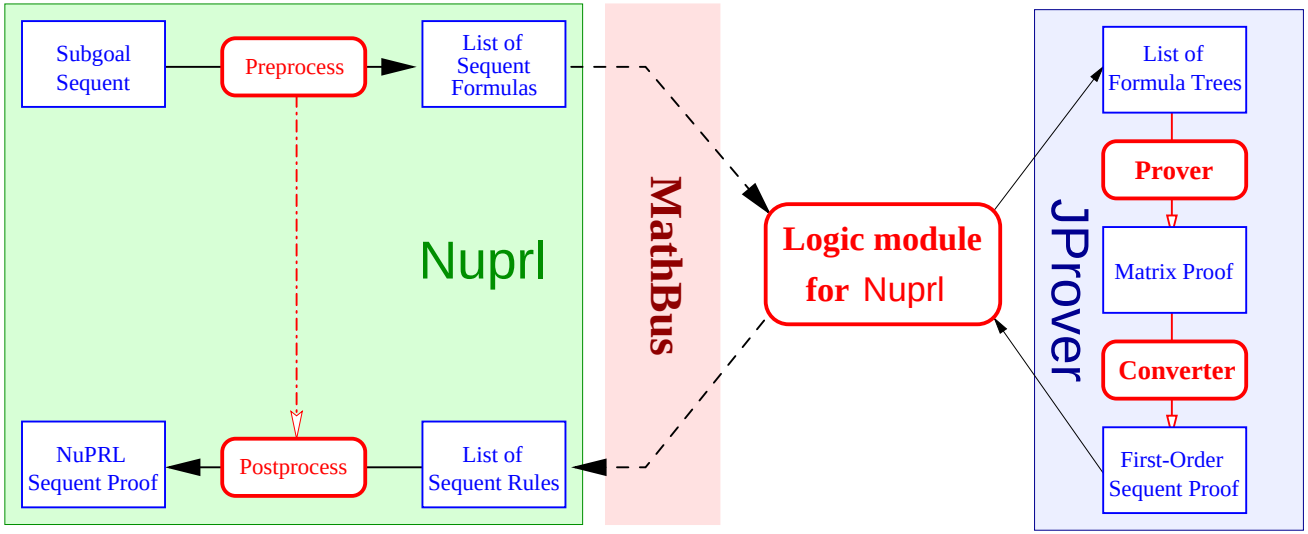
- **Interactive proof development**
  - Supports program extraction and evaluation
  - Proof automation through tactics & decision procedures
  - Highly customizable: conservative language extensions, term display, ...
- **Cooperating processes centered around knowledge base**
  - Library of formal algorithmic knowledge
  - Multiple user interfaces
  - External proof engines
  - Asynchronous & collaborative theorem proving

Connecting external systems to Nuprl

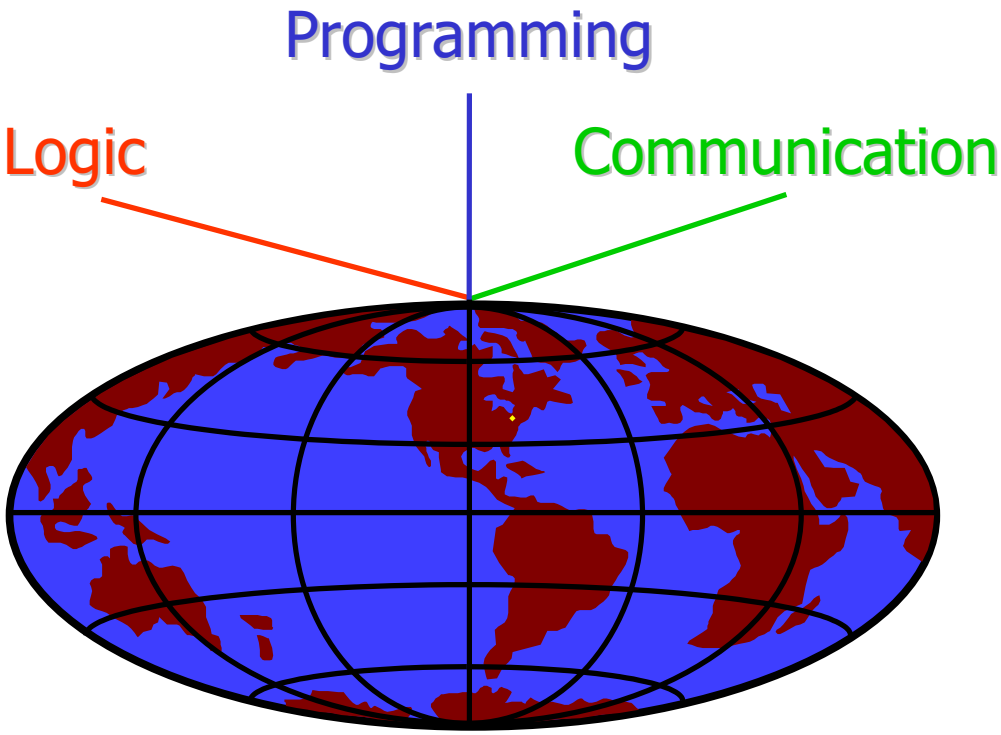
- **JProver: first-order intuitionistic logic**
  - Automatic proof search based on matrix methods
  + Proof transformation: matrix proof ⟶ sequent proof



- **Cooperation Methodology**
  - Communication of formulas in uniform format
  - Logic module converts between representations
  - Pre- and postprocessing in Nuprl widens range of applicability



Application: Reliable, High-Performance Networks



Secure software infrastructure

- **Ensemble Group Communication Toolkit**
  - System optimization and verification, formal component design
- **MediaNet Stream Computation Network** (ongoing)
  - Validation of real-time schedules wrt. resource limitations

The Ensemble Group Communication Toolkit

Modular group communication system

– Developed by Cornell's System Group (Ken Birman)

– Used commercially (BBN, JPL, Segasoft, Alier, Nortel Networks)

Architecture: stack of micro-protocols

– Select from 60 micro-protocols for specific tasks

– Modules can be stacked arbitrarily

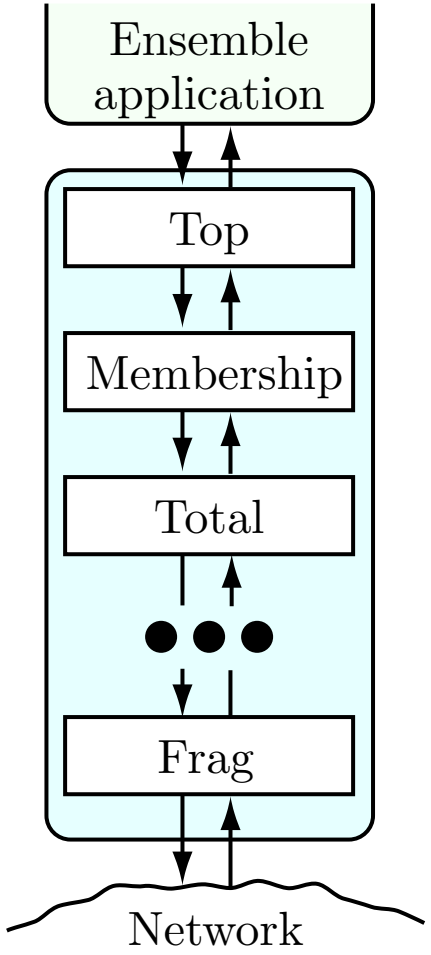
– Modeled as state/event machines

Implementation in Objective Caml (INRIA)

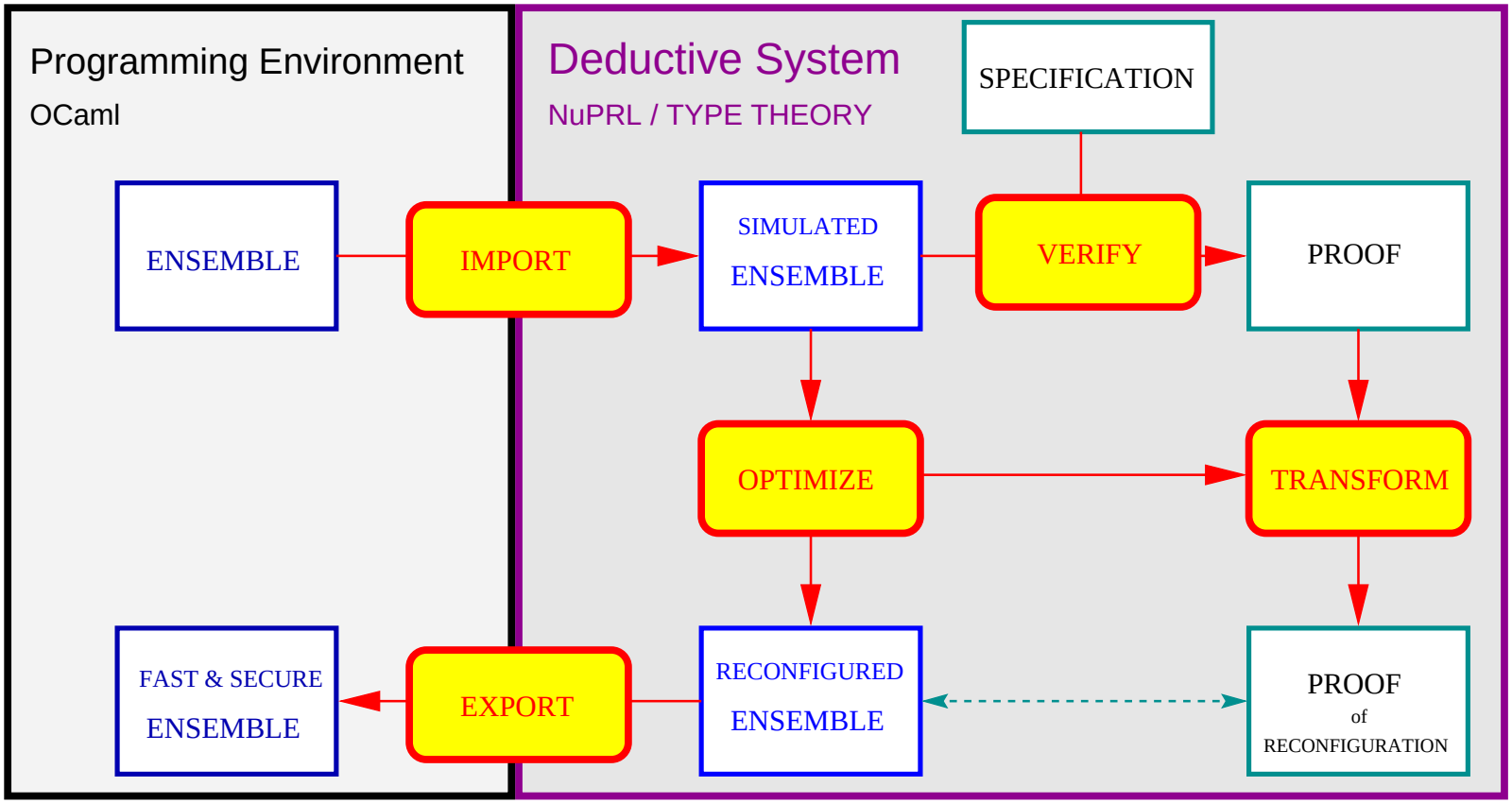
– Easy maintenance (small code, good data structures)

– Mathematical semantics, strict data type concepts

– Efficient compilers and type checkers



Formal Reasoning about Ensemble in Nuprl

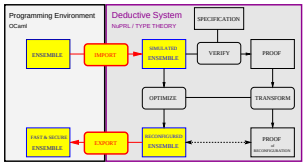


– Formalize semantics of OCaml (CADE 1998, . . .)

– Verify protocol components and system configurations (TACAS 1999)

– Formally design and verify new protocols (DISCEX 2001, TPHOLS 2001)

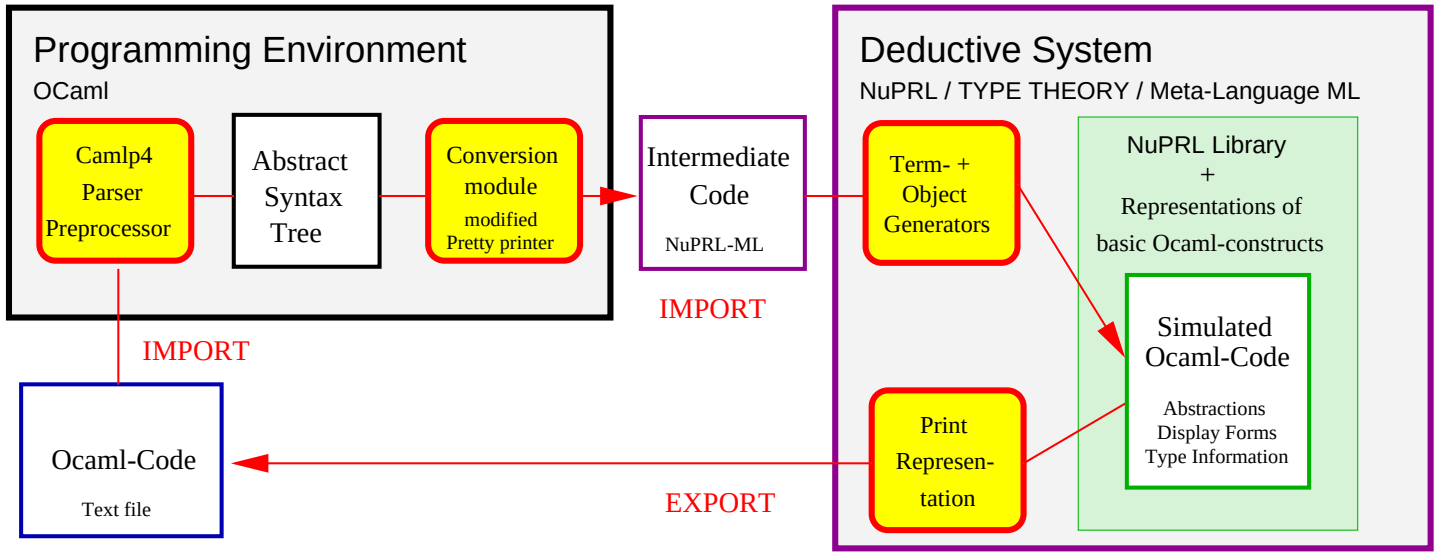
– Optimize performance of configured systems (TACAS 1999, SOSP 1999)

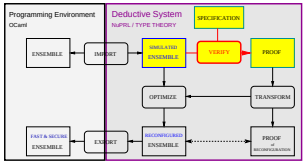


Embedding Ensemble's code into Nuprl

Enable formal reasoning on OCaml level

- **Type-theoretical semantics of OCaml fragment**
- **Nuprl implementation captures syntax & semantics**
- **Programming logic for OCaml**
- **Import and export mechanisms**





Verifying System Properties
Link four levels of abstraction

Formalize system specification and code

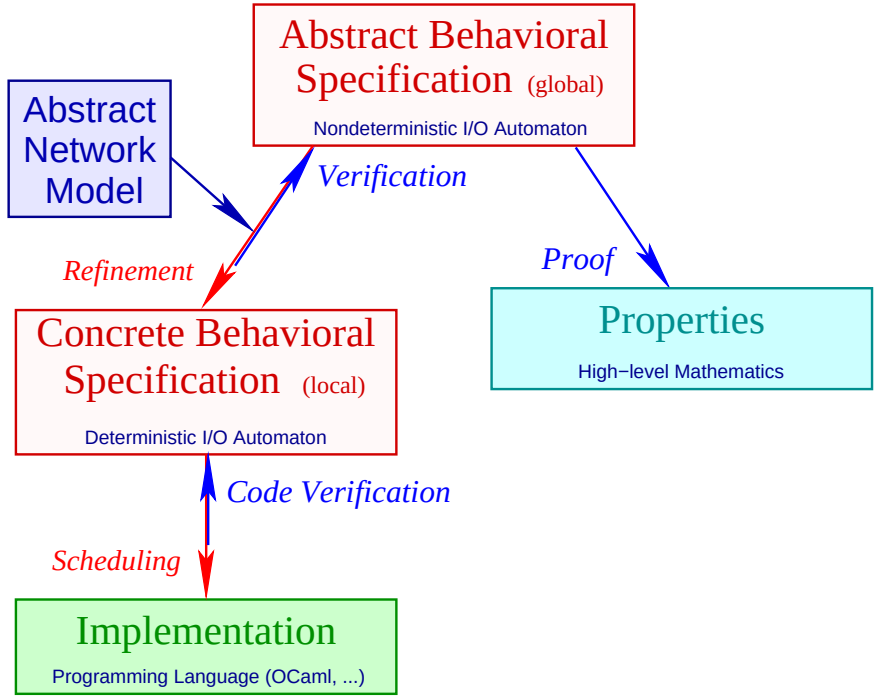
e.g. *"Messages are received in the same order in which they were sent"*

- *"Messages may be appended to global event queue and removed from its beginning"*
- *"Messages whose sequence number is too big will be buffered"*
- Ensemble *module* `Pt2pt.ml`*: 250 lines of* OCaml *code*

All levels represented in type theory

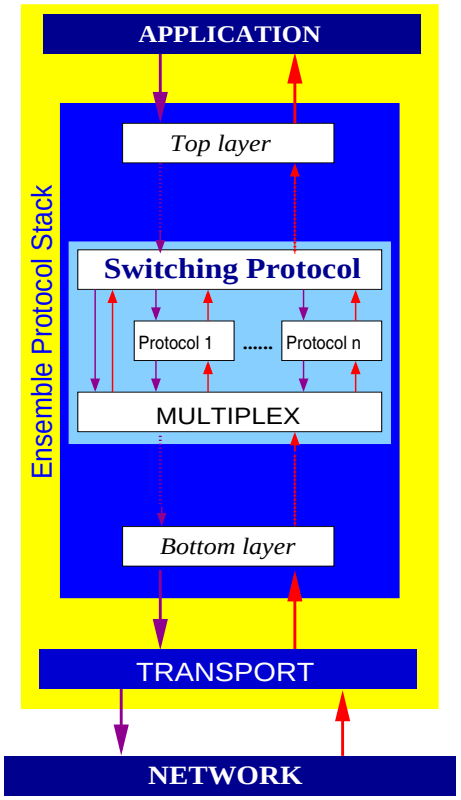
Verification methodology

- Verify component specifications
  (benign assumptions — subtle bug detected)
- Verify systems by composition
  (IOA-composition preserves safety properties)
- Weave aspects
- Verify code



Formal Design of Adaptive Systems

- **Make systems adapt safely to run-time dynamics**
  - On-line upgrading, security, performance
  - Difficult to design correctly *(distributed migration?)*



- **Generic switching protocol**
  - Construct hybrid protocols from simpler ones
  - Normal mode: interact with one protocol
  - Switching mode: deliver old messages, buffer new ones

- **Correctness issues**
  - *What kind of protocols are switchable at all?*
    - Reliability? Integrity? Confidentiality? Total Order? ...
  - *What code invariant guarantees that switchable properties are preserved?*

LPE verification answers both questions

A Formal Model of Communication

- **Communication property *P***
  - Predicate on traces, i.e. lists of Send(p,m) and Deliver(p,m) events

    e.g. $\texttt{Reliable}(\boldsymbol{tr}) \equiv \forall \texttt{p,q:PID}.\forall \texttt{m:Msg}.\ \texttt{Send(p,m)} \in \boldsymbol{tr} \Rightarrow \texttt{Deliver(q,m)} \in \boldsymbol{tr}$

- **Characterize switchable properties by meta-properties**
  - Predicates on communication properties
  - Expressed by relation R between traces tr_u, tr_l above/below a protocol

    $P\ \texttt{is}\ R\ \texttt{property} \equiv \forall \texttt{tr}_u\texttt{,tr}_l\texttt{:Trace}.\ (P(\texttt{tr}_l) \wedge \texttt{tr}_u\, R\, \texttt{tr}_l) \Rightarrow P(\texttt{tr}_u)$

Examples of meta-properties:

tr_u safety tr_l $\equiv$ $tr_u \sqsubseteq tr_l$

tr_u asynchrony tr_l $\equiv$ tr_u $\texttt{swap-adjacent}_{[\texttt{loc(e)}\neq\texttt{loc(e')}]}$ tr_l

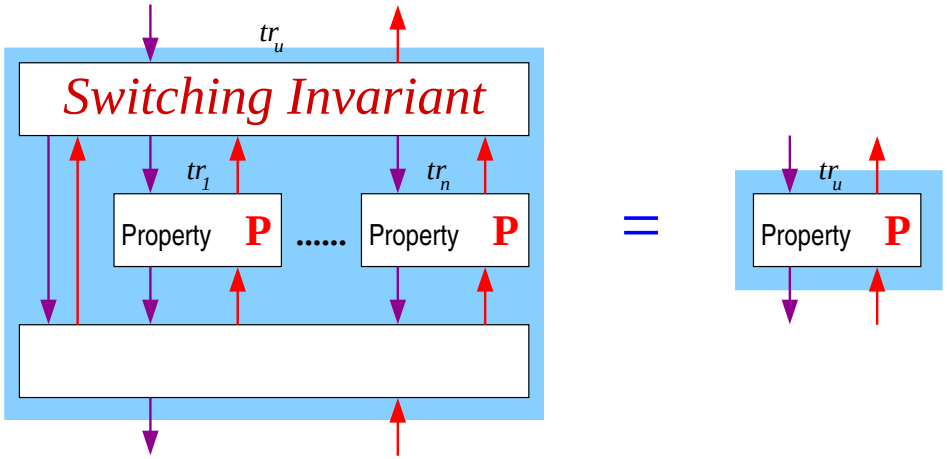
tr_u delayable tr_l $\equiv$ tr_u $\texttt{swap-adjacent}_{[\texttt{msg(e)}\neq\texttt{msg(e')} \wedge \texttt{is-send(e)}\neq\texttt{is-send(e')}]}$ tr_l

tr_u send-enabled tr_l $\equiv$ $\exists \texttt{e:Events}.\ \texttt{is-send(e)} \wedge tr_u = tr_l\texttt{@[e]}$

tr_u memoryless tr_l $\equiv$ $\exists \texttt{e:Events}.\ tr_u = [\,\texttt{e}_1 \in tr_l \mid \texttt{msg(e)} \neq \texttt{msg(e}_1\texttt{)}\,]$

$\texttt{composable}(tr_u, tr_1, tr_2)$ $\equiv$ $tr_u = tr_1\texttt{@}tr_2 \wedge \forall \texttt{e}_1 \in tr_1.\forall \texttt{e}_2 \in tr_2.\ \texttt{msg(e}_1\texttt{)} \neq \texttt{msg(e}_2\texttt{)}$

Verifying the Correctness of Switching



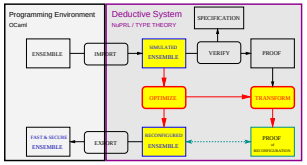
switchable(P)
≡ P refines Causality
∧ P refines No-replay
∧ P is safety property
∧ P is asynchrony property
∧ P is delayable property
∧ P is send-enabled property
∧ P is memoryless property
∧ P is composable property$_3$

- **Characterize switch invariant between tr_u and tr_1,..,tr_n**
  - tr_u results from joint trace by swapping events with different origin
  - Messages sent by different protocols must be delivered in same order
- **Prove that switchable properties will be preserved**

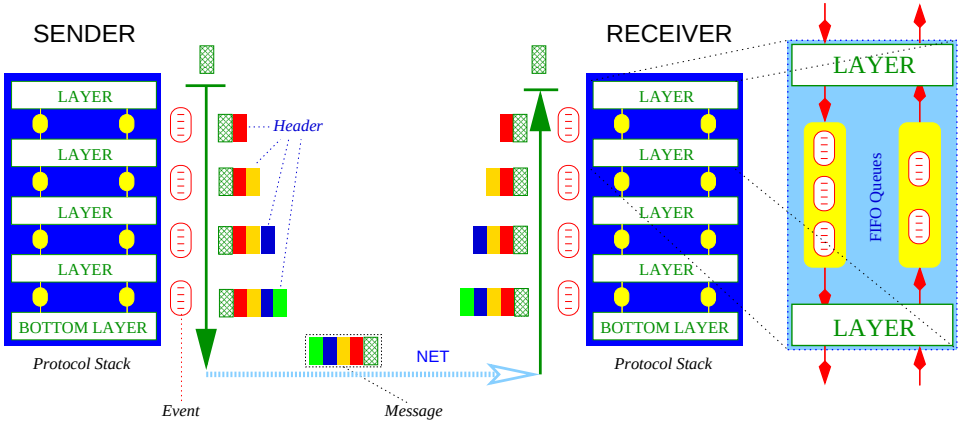
$\vdash \forall P{:}\text{TraceProperty}.\ \forall tr_u, tr_1, \ldots tr_n{:}\text{Trace}.$
$\quad \text{switchable}(P) \wedge \text{switch\_invariant}(tr_u; tr_1, .., tr_n)$
$\quad \Rightarrow\ (\ \forall i \leq n.\ P(tr_i)\ \Rightarrow\ P(tr_u)\ )$

$\Downarrow$

Correct implementation and use of switch



Optimization of Protocol Stacks



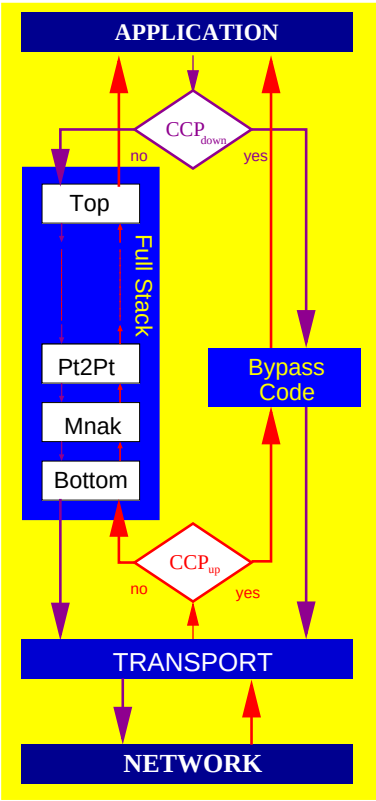
Protocol stacking creates performance loss

– redundancy, internal communication, large message headers

Possible optimizations

- **Fast-path for common execution sequences**
  - Identify Common Case as Predicate
  - Analyze path of events through stack
  - Isolate code for fast-path and generate bypass
  - Insert CCP as runtime switch
- **Header compression for common messages**

Need formal reasoning tools to do this correctly



Example Protocol Stack Bottom::Mnak::Pt2pt

Trace downgoing Send events and upgoing Cast events

Bottom (200 lines)

```
let name = Trace.source_file "BOTTOM"

type header = NoHdr | ... | ...

type state = {mutable all_alive : bool ; ... }

let  init _ (ls,vs) = {.........}

let  hdlrs s (ls,vs)
     {up_out=up;upnm_out=upnm;
      dn_out=dn;dnlm_out=dnlm;dnnm_out=dnnm}
 = ...
 let up_hdlr ev abv hdr =
  match getType ev, hdr with
  | (ECast|ESend), NoHdr ->
    if s.all_alive  or not (s_bottom.failed.(getPeer ev))
       then up ev abv
       else free name ev
  | :
 and uplm_hdlr ev hdr    = ...
 and upnm_hdlr ev        = ...
 and dn_hdlr ev abv      =
  if s.enabled then
    match getType ev with
     | ECast          -> dn ev abv NoHdr
     | ESend          -> dn ev abv NoHdr
     | ECastUnrel     -> dn (set name ev[Type ECast]) abv Unrel
     | ESendUnrel     -> dn (set name ev[Type ESend]) abv Unrel
     | EMergeRequest -> dn ev abv MergeRequest
     | EMergeGranted -> dn ev abv MergeGranted
     | EMergeDenied  -> dn ev abv MergeDenied
     | _ -> failwith "bad down event[1]"
   else (free name ev)
 and dnnm_hdlr ev        = ...
 in {up_in=up_hdlr;uplm_in=uplm_hdlr;upnm_in=upnm_hdlr;
     dn_in=dn_hdlr;dnnm_in=dnnm_hdlr}

let l args vs = Layer.hdr init hdlrs args vs

Layer.install name (Layer.init l)
```

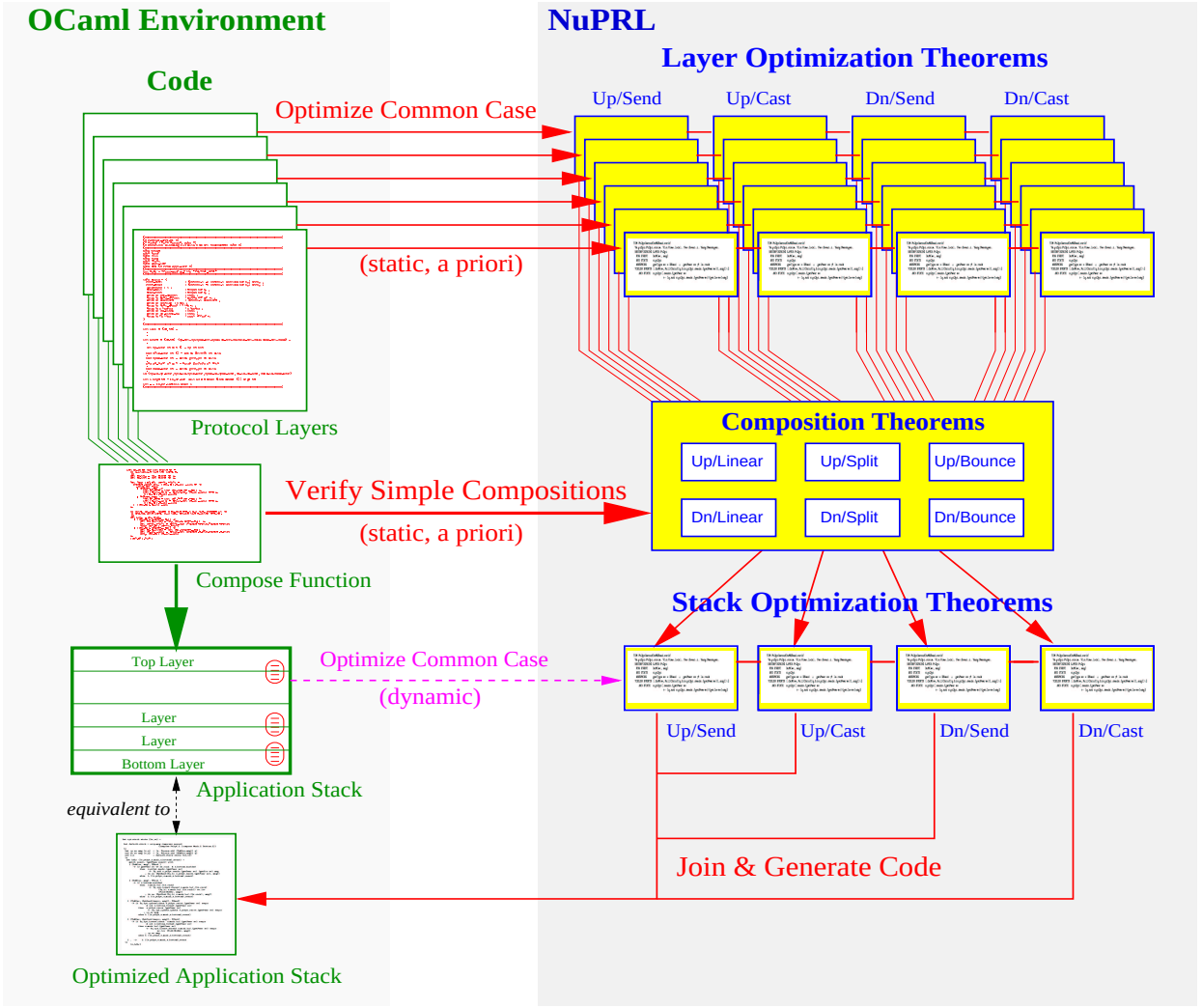
Mnak (350 lines)

```
let init ack_rate (ls,vs) = {.........}
let hdlrs s (ls,vs) { ......... }
 = ...
 let ...
 and dn_hdlr ev abv      =
   match getType ev with
    | ECast ->
       let iov = getIov ev in
       let buf = Arraye.get s.buf ls.rank in
       let seqno = Iq.hi buf in
       assert (Iq.opt_insert_check buf seqno) ;
       Arraye.set s.buf ls.rank
             (Iq.opt_insert_doread buf seqno iov abv) ;
       s.acct_size <- s.acct_size + getIovLen ev ;
       dn ev abv (Data seqno)
    | _ -> dn ev abv NoHdr
  :
```

Pt2pt (250 lines)

```
let init _ (ls,vs) = {.........}
let hdlrs s (ls,vs) { ......... }
 = ...
 let ...
 and dn_hdlr ev abv      =
  match getType ev with
   | ESend ->
      let dest = getPeer ev in
      if dest = ls.rank then (
        eprintf "PT2PT:%s\nPT2PT:%s\n"
         (Event.to_string ev) (View.string_of_full (ls,vs));
        failwith "send to myself" ;
       ) ;
      let sends = Arraye.get s.sends dest in
      let seqno = Iq.hi sends in
      let iov = getIov ev in
      Arraye.set s.sends dest (Iq.add sends iov abv) ;
      dn ev abv (Data seqno)
   | _ -> dn ev abv NoHdr
   :
```

Methodology: Compose Optimization Theorems



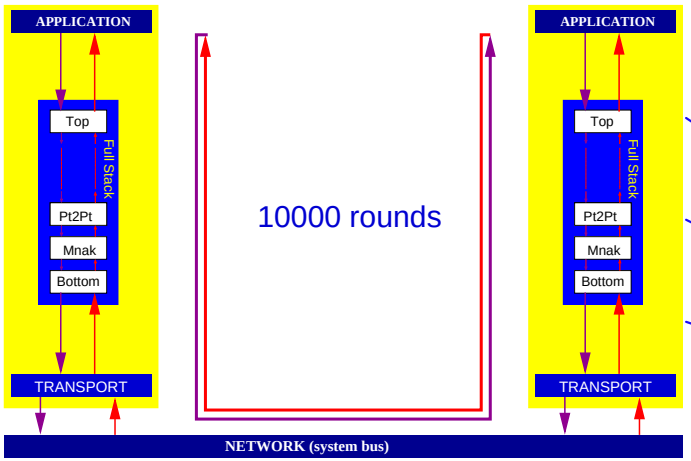
1. Use known optimizations of micro-protocols — A priori: Ensemble + Nuprl experts
2. Compose into optimizations of protocol stacks — automatic: application designer
3. Integrate message header compression — automatic: ⋮
4. Generate code from optimization theorems and reconfigure system — automatic: ⋮

Fast, error-free, independent of programming language **speedup factor 3-10**

DEMO: Optimizing a 24-layer Protocol Stack

Top :: Heal :: Switch :: Migrate :: Leave :: Inter :: Intra :: Elect :: Merge :: Slander :: Sync :: Suspect :: Stable :: Vsync :: Partial_appl :: Total :: Collect :: Local :: Frag :: Pt2ptw :: Mflow :: Pt2pt :: Mnak :: Bottom

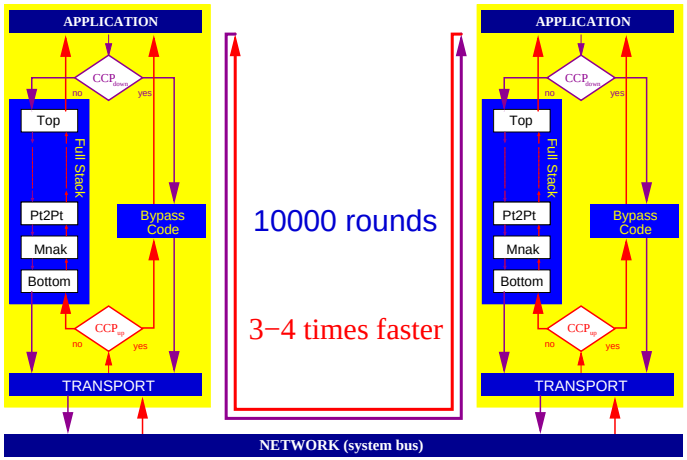
Performance Test

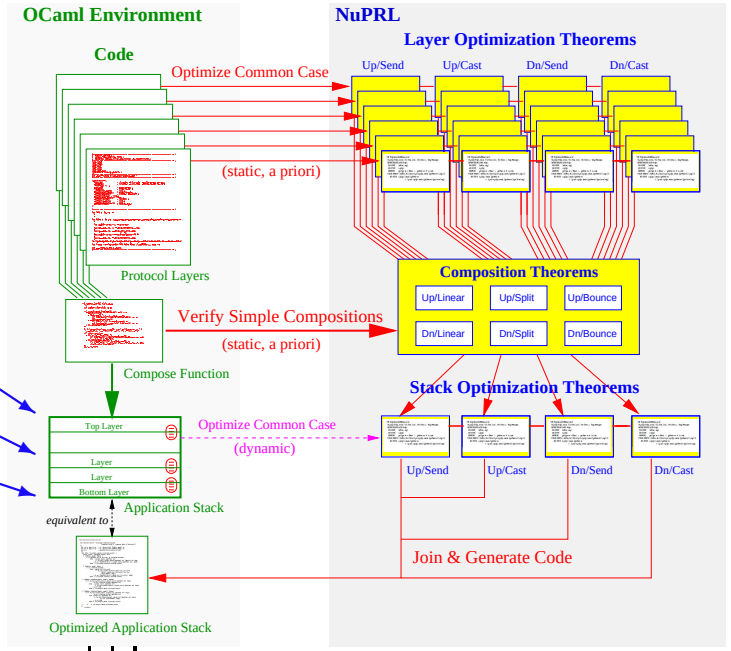


Original Ensemble *System*

After Optimizations

Performance Test





Formal Optimization

System Rebuild

make

Lessons learned

- **Results**
  - Type theory expressive enough to formalize today's software systems
  - Formal optimization can significantly improve practical performance
  - Formal verification reveals errors even in well-explored designs
  - Formal design reveals hidden assumptions and limitations for use of protocols
- **Ingredients for success . . .**
  - Implementation language with precise semantics
  - Collaboration between systems and formal reasoning groups
  - Formal models of: communication, I/O-automata, programming language
  - Knowledge-based formal reasoning tools
  - Great colleagues! Stuart Allen, Mark Bickford, Ken Birman, Robert Constable, Richard Eaton, Xiaming Liu, Lori Lorigo, Robbert van Renesse

Future Challenges

- **Advanced reasoning environment**
  - Interactive library of formal algorithmic knowledge
    - Archival capacities (documentation & certification, version control)
    - A variety of justifications (levels of trust)
    - Creation of formal and textual documents
    - Meta-reasoning and reflection
  - Embed external library contents
  - Connect additional proof engines

  Improve cooperation between research groups

- **Learn more from applications**
  - Reasoning about real-time, embedded systems and stream computations
    - verified self-adaptation to changing resource constraints (MediaNet)
  - Support programming languages with less clean semantics
  - Invert reasoning direction: from verification towards network synthesis

Areas for Study & Research

- **Courses**
  - Applied Logic, Automated Reasoning, PRL Seminar, ...
- **Formal Logics & Type Theory**
  - Classes & inheritance, recursive & partial objects, concurrency, real-time
  - Meta-reasoning, reflection, relating different logics, ...
- **Theorem Proving Environments**
  - Logical accounting, theory modules, interfaces, proof presentation, ...
- **Automated Proof Search Procedures**
  - Matrix methods, inductive theorem proving, rewriting, proof planning
  - Decision procedures, extended type inference, cooperating provers
  - Proof reuse, analogy, distributed proof procedures, ...
- **Applications**
  - Formal CS knowledge: graph theory, automata, trees, arrays, ...
  - Strategies for program synthesis, verification, and optimization
  - Modelling programming languages (OCaml, Java, ..)

  ... Participation in applied research projects