Automatisierte Logik und Programmierung

Einheit 16

Anwendungsbeispiele

1. Mathematik:
   Automatisierung von Kategorientheorie

2. Programmierung:
   Analyse und Optimierung verteilter Systeme

3. Aktuelle Fragestellungen:
   Language-based Security
• **Category Theory analyzes structure**

  *What properties of mathematical domains depend only on structure?*

  · Focus on mathematical objects and morphisms on these objects
  · Develop a generic framework for expressing abstract properties

  – Results have a wide impact on mathematics and computer science
Automating Proofs in Category Theory

- **Category Theory analyzes structure**
  
  *What properties of mathematical domains depend only on structure?*
  
  - Focus on mathematical objects and morphisms on these objects
  - Develop a generic framework for expressing abstract properties
  - Results have a wide impact on mathematics and computer science

- **Category Theory is extremely precise**
  
  - Even basic proofs can be very tedious
  - Diagrams illustrate the essential insights but are not considered proofs
**Automating Proofs in Category Theory**

- **Category Theory analyzes structure**
  
  *What properties of mathematical domains depend only on structure?*
  
  - Focus on mathematical objects and morphisms on these objects
  - Develop a generic framework for expressing abstract properties
  - Results have a wide impact on mathematics and computer science

- **Category Theory is extremely precise**
  
  - Even basic proofs can be very tedious
  - Diagrams illustrate the essential insights but are not considered proofs

- **Can category theoretical proofs be automated?**
  
  - Detailed proofs often follow standard patterns of reasoning
  - It should be possible to formalize these as proof rules
Automating Proofs in Category Theory

- **Category Theory analyzes structure**
  
  *What properties of mathematical domains depend only on structure?*
  - Focus on mathematical objects and morphisms on these objects
  - Develop a generic framework for expressing abstract properties
  - Results have a wide impact on mathematics and computer science

- **Category Theory is extremely precise**
  - Even basic proofs can be very tedious
  - Diagrams illustrate the essential insights but are not considered proofs

- **Can category theoretical proofs be automated?**
  - Detailed proofs often follow standard patterns of reasoning
    - It should be possible to formalize these as proof rules
  - Key insights are often considered “the only obvious choice”
    - It should be possible to write tactics that construct proofs
Axiomatization of Elementary Category Theory

Make standard reasoning patterns precise
Make standard reasoning patterns precise

- **Formulate as first-order reasoning system**
  - Rules about products, functors, natural transformations, …
  - Analysis and synthesis of structures
  - Equational reasoning is essential in most proofs
Axiomatization of Elementary Category Theory

Make standard reasoning patterns precise

- **Formulate as first-order reasoning system**
  - Rules about products, functors, natural transformations, ...
  - Analysis and synthesis of structures
  - Equational reasoning is essential in most proofs

- **Example: Analysis rules for functors**
  - Rules essentially explain what functors are and how to use them

\[
\Gamma \vdash F : \text{Fun}[C, D], \quad \Gamma \vdash A : C \\
\quad \quad \Gamma \vdash F^1A : D
\]

\[
\Gamma \vdash F : \text{Fun}[C, D], \quad \Gamma \vdash A, B : C, \quad \Gamma \vdash f : C(A, B) \\
\quad \quad \Gamma \vdash F^2f : D(F^1A, F^1B)
\]

\[
\Gamma \vdash F : \text{Fun}[C, D], \quad \Gamma \vdash A, B, C : C, \quad \Gamma \vdash f : C(A, B), \quad \Gamma \vdash g : C(B, C) \\
\quad \quad \Gamma \vdash F^2(g \circ f) = F^2g \circ F^2f
\]

\[
\Gamma \vdash F : \text{Fun}[C, D], \quad \Gamma \vdash A : C \\
\quad \quad \Gamma \vdash F^21_A = 1_{F^1A}
\]
IMPLEMENTATION OF THE FORMAL THEORY

Non-conservative extension with CTT support
Non-conservative extension with CTT support

- Encode the language of Category Theory
  - Abstractions explain category theoretical concepts in CTT
  - Display forms introduce MacLane’s textbook notation
Non-conservative extension with CTT support

• **Encode the language of Category Theory**
  – Abstractions explain category theoretical concepts in CTT
  – Display forms intruduce MacLane’s textbook notation

• **Implement first-order inference rules**
  – Introduce (top-down) rule objects for each inference rule

\[
\text{Rule: NatTransApply}\]

\[
\begin{align*}
H & \vdash \forall X \in D(F^1 X, G^1 X) \\
\quad & \quad \text{BY NatTransApply } C \\
H & \vdash \forall \in \text{Fun}[C, D](F, G) \\
H & \vdash X \in C
\end{align*}
\]
Non-conservative extension with CTT support

- **Encode the language of Category Theory**
  - Abstractions explain category theoretical concepts in CTT
  - Display forms introduce MacLane’s textbook notation

- **Implement first-order inference rules**
  - Introduce (top-down) rule objects for each inference rule
  - Justify inference rules by proving them correct in CTT
  - Convert primitive inferences into simple tactics
Proof Automation

• Most steps are straightforward decompositions
  – Apply analysis and synthesis rules where possible
  – Block application of analysis rules that create subgoals previously decomposed by a synthesis rule
  – Assert goals that will re-occur several times in subproofs
Proof Automation

• Most steps are straightforward decompositions
  – Apply analysis and synthesis rules where possible
  – Block application of analysis rules that create subgoals previously decomposed by a synthesis rule
  – Assert goals that will re-occur several times in subproofs

• Equality reasoning needs guidance
  – Convert equality rules into directed rewrite rules
  – Use Knuth-Bendix completion to make the rewrite system confluent
Proof Automation

- **Most steps are straightforward decompositions**
  - Apply analysis and synthesis rules where possible
  - Block application of analysis rules that create subgoals previously decomposed by a synthesis rule
  - Assert goals that will re-occur several times in subproofs

- **Equality reasoning needs guidance**
  - Convert equality rules into directed rewrite rules
  - Use Knuth-Bendix completion to make the rewrite system confluent

- **Specify functors component-wise**
  - E.g. use $\vartheta^1 G^1 A^1 X \equiv G^1 <A,X>$ and $\vartheta^1 G^1 A^2 h \equiv G^2 <1_A,h>$ instead of $\vartheta \equiv \lambda G,A \ldots$, which is no category theory expression
  - Method keeps reasoning methods first-order
**Proof Automation**

- **Most steps are straightforward decompositions**
  - Apply analysis and synthesis rules where possible
  - Block application of analysis rules that create subgoals previously decomposed by a synthesis rule
  - Assert goals that will re-occur several times in subproofs

- **Equality reasoning needs guidance**
  - Convert equality rules into directed rewrite rules
  - Use Knuth-Bendix completion to make the rewrite system confluent

- **Specify functors component-wise**
  - E.g. use $\vartheta^1G^1A^1X \equiv G^1<A,X>$ and $\vartheta^1G^1A^2h \equiv G^2<1_A,h>$ instead of $\vartheta \equiv \lambda G,A\ldots$, which is no category theory expression
  - Method keeps reasoning methods first-order

- **Guess witnesses for existential quantifiers**
  - Introduce meta-variables for existentially quantified variables
  - Decompose as long as possible and focus of typing subgoals
  - Introduce the simplest term that satisfies the typing requirements
A

AUTOMATISIERTE LOGIK UND PROGRAMMIERUNG §16:

AN EXAMPLE PROOF

BY Unfold `CatIso` 0 THEN prove.

* 1

1. C : Categories
2. D : Categories
3. E : Categories
   ⊢ 3φ:Fun[Fun[C×D,E],Fun[C,Fun[D,E]]],
   3η:Fun[Fun[C,Fun[D,E]],Fun[C×D,E]]. φ and η are inverse

BY GUESS `φ` THEN GUESS `η`.

* 1 1

4. theta : Top
5. ∀phi,k,X1,X,f,G:Top.
   ⟨φ⟩1G2fX ≡ G2f1X
   ⊢ φG1×1X ≡ G1fX1
   ⊢ φG1×k ≡ G2f1k
   ⊢ φηX1 ≡ ηX1
6. eta : Top
7. ∀X1,phi,g,f,X,A,G:Top.
   ⟨η⟩1G2fX ≡ G1fX
   ⊢ ηG2f cod(g) ≡ G1 dom(f)2g
   ⊢ ηfX1 ≡ ηA X1
   ⊢ θ and η are inverse

BY Unfold `FunInverse` 0 THEN AutoCAT2
Towards Reliable, High-Performance Networks

Apply Formal Reasoning to a real-world system

Secure software infrastructure
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)
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 more than 60 micro-protocols for specific tasks
– Modules can be stacked arbitrarily
– Modeled as state/event machines
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 more than 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
Link the Ensemble and NuPRL systems
Link the Enseble and NuPrl systems
– Embed Enseble’s code into NuPrl’s language
**Formal Reasoning about a Real-World System**

**Programming Environment**
- OCaml

**Deductive System**
- NuPRL / Type Theory

**Steps in the Process**
- **IMPORT**
- **SIMULATED ENSEMBLE**
- **VERIFY**
- **PROOF**
- **OPTIMIZE**
- **RECONFIGURED ENSEMBLE**
- **EXPORT**
- **ENGEMBLE**
- **FAST & SECURE**

**Link the **ENSEMBLE** and Nuprl systems**
- Embed *ENSEMBLE*’s code into *Nuprl*’s language
- Verify protocol components and system configurations
Link the **ENSEMBLE** and **Nuprl** systems

- **Embed** ENSEMBLE’s code into Nuprl’s language
- **Verify** protocol components and system configurations
- **Optimize** performance of configured systems
Link the Ensemble and NuPrl systems
- Embed Ensemble’s code into NuPrl’s language
- Verify protocol components and system configurations
- Optimize performance of configured systems
- Formally design and verify new protocols
• Develop **type-theoretical semantics of OCaml**
  – Functional core, pattern matching, exceptions, references, modules, . . .
Embedding Ensemble’s code into Nuprl

- **Develop** *type-theoretical semantics of OCaml*
  - Functional core, pattern matching, exceptions, references, modules, . . .

- **Implement** *using Nuprl’s definition mechanism*
  - Represent OCaml’s *semantics* via abstraction objects
  - Represent OCaml’s *syntax* using associated display objects
Embedding Ensemble’s code into Nuprl

- **Develop type-theoretical semantics of OCaml**
  - Functional core, pattern matching, exceptions, references, modules,…

- **Implement using Nuprl’s definition mechanism**
  - Represent OCaml’s semantics via abstraction objects
  - Represent OCaml’s syntax using associated display objects

- **Develop programming logic for OCaml**
  - Implement as rules derived from the abstract representation
  - Raises the level of formal reasoning from Type Theory to OCaml
**Embedding Ensemble’s code into NuPRL**

- **Develop type-theoretical semantics of OCaml**
  - Functional core, pattern matching, exceptions, references, modules, \ldots

- **Implement using NuPrl’s definition mechanism**
  - Represent OCaml’s **semantics** via abstraction objects
  - Represent OCaml’s **syntax** using associated display objects

- **Develop programming logic for OCaml**
  - Implement as **rules** derived from the abstract representation
  - Raises the level of formal reasoning from Type Theory to OCaml

- **Develop tools for importing and exporting code**
  - Translators between OCaml program text and NuPrl terms
• **Basic OCaml expressions similar to CTT terms**
  – Numbers, tuples, lists etc. can be mapped directly onto CTT terms
OCaml Semantics: The functional core

- **Basic OCaml expressions similar to CTT terms**
  - Numbers, tuples, lists etc. can be mapped directly onto CTT terms

- **Complex data structures have to be simulated**
  - Records \{f_1=e_1; \ldots; f_n=e_n\} are functions in f:FIELDS→T[f]
  
  - **Abstraction** for representing the semantics of record expressions
    \[
    \text{RecordExpr}(\text{field}; e; \text{next}) ≡ \lambda f. \text{if } f=\text{field} \text{ then } e \text{ else } \text{next} (f)
    \]

  - **Display form** for representing the flexible syntax of record expressions
    \[
    \begin{align*}
    \{\text{field}=e; & \phantom{=} \text{next}\} & ≡ & \text{RecordExpr}(\text{field}; e; \text{next}) \\
    \{\text{field}=e\} & ≡ & \text{RecordExpr}(\text{field}; e; \lambda f.()) \\
    \text{HD}:: & \{\text{field}=e; & \phantom{=} \#\} & ≡ & \text{RecordExpr}(\text{field}; e; \#) \\
    \text{TL}:: & \text{field}=e; & \phantom{=} \# & ≡ \text{RecordExpr}(\text{field}; e; \#) \\
    \text{TL}:: & \text{field}=e & \phantom{=} \} & ≡ \text{RecordExpr}(\text{field}; e; \lambda f.())
    \end{align*}
    \]
• **Basic OCaml expressions similar to CTT terms**
  – Numbers, tuples, lists etc. can be mapped directly onto CTT terms

• **Complex data structures have to be simulated**
  Records \( \{ f_1 = e_1; \ldots; f_n = e_n \} \) are functions in \( f : \text{FIELDS} \rightarrow T[f] \)
  – Abstraction for representing the semantics of record expressions
    \[
    \text{RecordExpr}(\text{field};e;\text{next}) \equiv \lambda f. \text{if } f = \text{field} \text{ then } e \text{ else } \text{next}(f)
    \]
  – Display form for representing the flexible syntax of record expressions
    \[
    \{ \text{field}=e; \text{next} \} \equiv \text{RecordExpr}(\text{field};e;\text{next})
    \]
    \[
    \{ \text{field}=e \} \equiv \text{RecordExpr}(\text{field};e;\lambda f.())
    \]
    \[
    \text{HD:: } \{ \text{field}=e; \# \} \equiv \text{RecordExpr}(\text{field};e;\#)
    \]
    \[
    \text{TL:: } \text{field}=e; \# \equiv \text{RecordExpr}(\text{field};e;\#)
    \]
    \[
    \text{TL:: } \text{field}=e \} \equiv \text{RecordExpr}(\text{field};e;\lambda f.())
    \]

• **Sufficient for representing micro protocols**
  – Simple state-event machines, encoded via updates to certain records
  – Transport module and protocol composition require imperative model
• **Type Theory is purely functional**
  
  – Terms are evaluated solely by **reduction**
  – OCaml has **pattern matching**, **reference cells**, **exceptions**, **modules**, ...
• Type Theory is purely functional
  – Terms are evaluated solely by reduction
  – OCaml has pattern matching, reference cells, exceptions, modules, ...

• Modelling Pattern Matching: let \( \text{pat}=e \) in \( t \)
  “Variables of \( \text{pat} \) in \( t \) are bound to corresponding values of \( e \)”
  – Evaluation of OCaml-expressions uses an environment of bindings
  – Patterns are functions that modify the environment of expressions

\[
\begin{align*}
x & \equiv \lambda \text{val}, t. \lambda \text{env}. t \left( \text{env}\{x \mapsto \text{val}\} \right) \\
p_1, p_2 & \equiv \lambda \text{val}, t. \lambda \text{env}. \text{let } <v_1, v_2> = \text{val} \text{ in } (p_1 v_1 (p_2 v_2 t)) \text{ env} \\
\{f_1=p_1; \ldots; f_n=p_n\} & \equiv \lambda \text{val}, t. \lambda \text{env}. p_1 (\text{val } f_1) (\ldots(p_n (\text{val } f_n \ t) \ldots) \text{ env} \\
\vdots & \quad \vdots \\
\text{Local bindings are represented as applications of these functions} \\
\text{let } p=e \text{ in } t & \equiv \lambda \text{env}. (p (e \text{ env} \ t) \text{ env}
\end{align*}
\]
• **Modelling Reference cells**

  – Evaluation of OCaml-expressions may lookup/modify a global store
  – The global store is represented as table with addresses and values

    \[
    \text{ref}(e) \equiv \lambda s,\text{env}. \text{let } <v,s_1> = e \ s \ \text{env} \ \text{in} \\
    \text{let } \text{addr} = \text{NEW}(s_1) \ \text{in} \ <\text{addr}, s_1[\text{addr} \leftarrow v]>
    \]

    \[
    !e \equiv \lambda s,\text{env}. \text{let } <\text{addr},s_1> = e \ s \ \text{env} \ \text{in} \ <s_1[\text{addr}], s_1>
    \]

    \[
    e_1 := e_2 \equiv \lambda s,\text{env}. \text{let } <v,s_1> = e_2 \ s \ \text{env} \ \text{in} \\
    \text{let } <\text{addr},s_2> = e_1 \ s_1 \ \text{env} \ \text{in} \ <(), s_2[\text{addr} \leftarrow v]>
    \]
Extensions of the semantical model (2)

- Modelling **Reference cells**
  - Evaluation of OCaml-expressions may lookup/modify a global store
  - The global store is represented as table with addresses and values
    \[
    \text{ref}(e) \equiv \lambda s,\text{env}. \ \text{let } \langle v, s_1 \rangle = e \ s \ \text{env} \ \text{in} \\
    \quad \text{let } \text{addr} = \text{NEW}(s_1) \ \text{in} \ \langle \text{addr}, s_1[\text{addr} \leftarrow v] \rangle \\
    \]
    \[
    !e \equiv \lambda s,\text{env}. \ \text{let } \langle \text{addr}, s_1 \rangle = e \ s \ \text{env} \ \text{in} \ \langle s_1[\text{addr}], s_1 \rangle \\
    \]
    \[
    e_1 := e_2 \equiv \lambda s,\text{env}. \ \text{let } \langle v, s_1 \rangle = e_2 \ s \ \text{env} \ \text{in} \\
    \quad \text{let } \langle \text{addr}, s_2 \rangle = e_1 \ s_1 \ \text{env} \ \text{in} \ \langle(), s_2[\text{addr} \leftarrow v] \rangle \\
    \]

- Modelling **Exceptions**
  - Expressions like \(x/y\) may raise exceptions, which can be caught
  - Exceptions must have the same type as the expression that raises them
  - An OCaml type \(T\) must be represented as **EXCEPTION + \(T\)**
• Modelling **Reference cells**
  – Evaluation of OCaml-expressions may lookup/modify a global store
  – The global store is represented as a table with addresses and values
    \[
    \text{ref}(e) \equiv \lambda s,\text{env. let } <v,s_1> = e \ s \ \text{env in}
    \]
    \[
    \text{let addr = NEW}(s_1) \ \text{in } <\text{addr}, s_1[\text{addr} \leftarrow v]>.
    \]
  – OCaml expressions may lookup/modify a global store
    \[
    !e \equiv \lambda s,\text{env. let } <\text{addr},s_1> = e \ s \ \text{env in } <s_1[\text{addr}], s_1>.
    \]
  – OCaml expressions may lookup/modify a global store
    \[
    e_1 := e_2 \equiv \lambda s,\text{env. let } <v,s_1> = e_2 \ s \ \text{env in}
    \]
    \[
    \text{let } <\text{addr},s_2> = e_1 \ s_1 \ \text{env in } <(), s_2[\text{addr} \leftarrow v]>.
    \]

• Modelling **Exceptions**
  – Expressions like \(x/y\) may raise exceptions, which can be caught
  – Exceptions must have the same type as the expression that raises them
  – An OCaml type \(T\) must be represented as \(\text{EXCEPTION + } T\)

• **Modules**
  – Modules are second class objects that structure the name space
  – Modules are represented by operations on a global environment
Summary of the formal model

- OCaml expressions are represented as functions
  - Evaluation depends on environment and store
  - Evaluation results in value or exception and an updated store
  - Nuprl type is $\text{STORE} \rightarrow \text{ENV} \rightarrow (\text{EXCEPTION} + T) \times \text{STORE}$
Summary of the formal model

- OCaml expressions are represented as functions
  - Evaluation depends on environment and store
  - Evaluation results in value or exception and an updated store
  - Nuprl type is $\text{STORE} \rightarrow \text{ENV} \rightarrow (\text{EXCEPTION} + T) \times \text{STORE}$

- Equivalent to Wright/Felleisen model
  - The standard model for building ML compilers
  - Model combines several mechanisms for evaluating ML programs
  - Nuprl representation simulates these models functionally
Summary of the formal model

- **OCaml expressions are represented as functions**
  - Evaluation depends on environment and store
  - Evaluation results in value or exception and an updated store
  - Nuprl type is $\text{STORE} \rightarrow \text{ENV} \rightarrow (\text{EXCEPTION} + T) \times \text{STORE}$

- **Equivalent to Wright/Felleisen model**
  - The standard model for building ML compilers
  - Model combines several mechanisms for evaluating ML programs
  - Nuprl representation simulates these models functionally

Genuine OCaml code may occur in formal theorems
Importing and Exporting System Code

**Programming Environment**
- OCaml
  - Camlp4 Parser Preprocessor
  - Abstract Syntax Tree
  - Conversion module (modified Pretty printer)

**Deductive System**
- NuPRL / TYPE THEORY / Meta-Language ML
  - Term- + Object Generators
  - Intermediate Code (NuPRL-ML)
  - NuPRL Library + Representations of basic Ocaml-constructs
  - Print Representation
  - Simulated Ocaml-Code
    - Abstractions Display Forms
    - Type Information

**Ocaml-Code Text file**
- IMPORT
- EXPORT
**Importing and Exporting System Code**

**Import:**
- Parse with **Camlp4** parser-preprocessor
- Convert **abstract syntax tree** into term- & object generators
- Generators perform second pass and create **Nuprl library objects**
**Import:** – Parse with Camlp4 parser-preprocessor
  – Convert abstract syntax tree into term- & object generators
  – Generators perform second pass and create NuPRL library objects

**Export:** – Print-representation is genuine OCaml-code
**Importing and Exporting System Code**

**Import:** – Parse with Camlp4 parser-preprocessor

– Convert abstract syntax tree into term- & object generators

– Generators perform second pass and create NuPRL library objects

**Export:** – Print-representation is genuine OCaml-code

**Actual Ensemble code available for formal reasoning**
OPTIMIZATION OF PROTOCOL STACKS

 Optimization of Protocol Stacks

 ENSEMBLE SIMULATED Programming Environment

 OCaml Deductive System

 NuPRL / TYPE THEORY

 ENSEMBLE RECONFIGURED

 FAST & SECURE

 OPTIMIZE TRANSFORM PROOF SPECIFICATION

 PROOF RECONFIGURATION ENSEMBLE IMPORT VERIFY EXPORT

 FIFO Queues

 MESSAGE EVENT NET

 Protocol Stack

 BOTTOM LAYER

 MESSAGE EVENT NET

 Protocol Stack

 BOTTOM LAYER
Optimization of Protocol Stacks

Performance loss: **redundancies, internal communication, large message headers**
Performance loss: redundancies, internal communication, large message headers

Optimizations: bypass-code for common execution sequences, header compression
Optimization of Protocol Stacks

Performance loss: redundancies, internal communication, large message headers

Optimizations: bypass-code for common execution sequences, header compression

Need formal methods to do this correctly

Automatisierte Logik und Programmierung §16: 15 Anwendungsbeispiele
### Automatisierte Logik und Programmierung §16:

**Example Protocol Stack**

#### Bottom::Mnak::Pt2pt

**Trace downgoing Send events and upgoing Cast events**

**Bottom (200 lines)**

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

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

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

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

let hdrls s (ls,vs)
{up_out=up;upnm_out=upnm;
 dn_out=dn;dnlm_out=dlm;dnmm_out=dnmm}

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 dnmm_hdlr ev = ...
    in {up_in=up_hdlr;uplm_in=uplm_hdlr;upnm_in=upnm_hdlr;
        dn_in=dn_hdlr;dnmm_in=dnmm_hdlr}

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

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

**Mnak (350 lines)**

```plaintext
let init ack_rate (ls,vs) = {...........}

let hdrls s (ls,vs) { ........... }
  = ...
  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)**

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

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

| ...
```
- **Identify Common Case**
  - Events and protocol states of regular communication
  - Formalize as **Common Case** Predicate
Fast-path Optimization with Nuprl

- **Identify Common Case**
  - Events and protocol states of regular communication
  - Formalize as Common Case Predicate

- **Analyze path of events through stack**
- **Identify Common Case**
  - Events and protocol states of regular communication
  - Formalize as Common Case Predicate
- **Analyze path of events through stack**
- **Isolate code for fast-path**
Fast-path Optimization with Nuprl

- **Identify Common Case**
  - Events and protocol states of regular communication
  - Formalize as Common Case Predicate
- **Analyze path of events through stack**
- **Isolate code for fast-path**
- **Integrate code for compressing headers of common messages**
**Fast-path Optimization with Nuprl**

- **Identify Common Case**
  - Events and protocol states of regular communication
  - Formalize as *Common Case* Predicate

- **Analyze path of events through stack**

- **Isolate code for fast-path**

- **Integrate code for compressing headers of common messages**

- **Generate bypass-code**
  - Insert CCP as runtime switch
Fast-path Optimization with Nuprl

- **Identify Common Case**
  - Events and protocol states of regular communication
  - Formalize as Common Case Predicate

- **Analyze path of events through stack**

- **Isolate code for fast-path**

- **Integrate code for compressing headers of common messages**

- **Generate bypass-code**
  - Insert CCP as runtime switch

**Methodology:** compose formal optimization theorems
Fast, error-free, independent of programming language, **speedup factor 3-10**
**Methodology: Compose Optimization Theorems**

1. Use known optimizations of micro-protocols
2. Compose into optimizations of protocol stacks
3. Integrate message header compression
4. Generate code from optimization theorems and reconfigure system

- A priori: Ensemble + Nuprl experts
- Automatic: application designer

---

**OCaml Environment**

**Code**

- Optimize Common Case
- Verify Simple Compositions

**NuPRL**

- Layer Optimization Theorems
- Composition Theorems
- Stack Optimization Theorems
- Join & Generate Code

**Layers**

**Composition**

**Stack**

---

**Automatisierte Logik und Programmierung §16:**

18

**Anwendungsbeispiele**
Static optimization of micro protocols

- **A-priori analysis of common execution sequences**
  - Generate local CCP from conditionals in a layer’s code
• **A-priori analysis of common execution sequences**
  – Generate local CCP from conditionals in a layer’s code

• **Assuming the CCP, apply code transformations**
  – Controlled function inlining and symbolic evaluation (rewrite tactics)
  – Directed equality substitutions (lemma application)
  – Context-dependent simplifications (substitute part of CCP and rewrite)
**Static optimization of micro protocols**

- **A-priori analysis of common execution sequences**
  - Generate local CCP from conditionals in a layer’s code

- **Assuming the CCP, apply code transformations**
  - Controlled function inlining and symbolic evaluation (rewrite tactics)
  - Directed equality substitutions (lemma application)
  - Context-dependent simplifications (substitute part of CCP and rewrite)

- **Store result in library as optimization theorem**

```plaintext
OPTIMIZING LAYER Pt2pt
FOR EVENT DnM (ev, msg)
AND STATE s_pt2pt
ASSUMING (getType ev) = ESend ∧ not (getPeer ev = ls.rank)
YIELDS HANDLERS dn ev (Full (Data (Iq.hi
(Arraye.get s_pt2pt.sends (getPeer ev))), msg))
AND UPDATES Iq.add (Arraye.get s_pt2pt.sends (getPeer ev))
(getIov ev) msg
```

- Theorem proves correctness of the local optimization
- Optimizations of micro protocols part of ENSEMBLE’s distribution
Dynamic Optimization of Application Stacks

- **Compose Optimization Theorems**
  - Consult optimization theorems for individual layers
  - Apply *composition theorems* to generate *stack optimization theorems* (Linear, simple split, bouncing – send/receive)

```
OPTIMIZING LAYER Upper
  FOR EVENT DnM(ev, hdr) AND STATE s_up
  YIELDS HANDLERS dn ev msg1 AND UPDATES stmt1
∧ OPTIMIZING LAYER Lower
  FOR EVENT DnM(ev, hdr1) AND STATE s_low
  YIELDS HANDLERS dn ev msg2 AND UPDATES stmt2
⇒ OPTIMIZING LAYER Upper || Lower
  FOR EVENT DnM(ev, hdr) AND STATE (s_up, s_low)
  YIELDS HANDLERS dn ev msg2 AND UPDATES stmt2; stmt1
```
**Dynamic Optimization of Application Stacks**

- **Compose Optimization Theorems**
  - Consult optimization theorems for *individual layers*
  - Apply *composition theorems* to generate *stack optimization theorems*  
    (Linear, simple split, bouncing – send/receive)

\[
\text{OPTIMIZING LAYER Lower} \\
\text{FOR EVENT DnM(ev, hdr1) AND STATE s\_low} \\
\text{YIELDS HANDLERS dn ev msg2 AND UPDATES stmt2}
\]

\[
\Rightarrow \text{OPTIMIZING LAYER Upper \|\| Lower} \\
\text{FOR EVENT DnM(ev, hdr) AND STATE \(s\_up, s\_low\)} \\
\text{YIELDS HANDLERS dn ev msg2 AND UPDATES stmt2; stmt1}
\]

- Formal proof complex because of complex code for composition
Dynamic Optimization of Application Stacks

- **Compose Optimization Theorems**
  - Consult optimization theorems for *individual layers*
  - Apply *composition theorems* to generate *stack optimization theorems*
  
  (Linear, simple split, bouncing – send/receive)

  - **Formal proof complex because of complex code for composition**

- **Optimization of Protocol Stacks in Linear Time**
  - Use of optimization theorems reduces proof burden for optimizer
  - **Pushbutton Technology**: requires only configuration of stack
Header compression for fast-path code

Integrate compression into optimization process
Integrate compression into optimization process
– Generate code for compression and expansion from fast-path headers
Integrate compression into optimization process
– Generate code for compression and expansion from fast-path headers
– Combine optimization theorem for stack with compression theorems
Integrate compression into optimization process
– Generate code for compression and expansion from fast-path headers
– Combine optimization theorem for stack with compression theorems
– Optimized stack uses compressed headers directly
Example Optimization of Bottom::Mnak::Pt2pt

- **Generated optimization theorem for application stack**

  OPTIMIZING LAYER Pt2pt::Mnak::Bottom
  FOR EVENT DnM(ev, msg)
  AND STATE (s_pt2pt, s_mnak, s_bottom)
  ASSUMING getType ev = ESend ∧ getPeer ev ≠ ls.rank ∧ s_bottom.enabled
  YIELDS HANDLERS dn ev (Full(NoHdr, Full(NoHdr,
    Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),msg))))
  AND UPDATES Iq.add (Arraye.get s_pt2pt.sends (getPeer ev))(getIov ev) msg

- **Generated code for header compression**

  let compress hdr = match hdr with
    Full(NoHdr, Full(NoHdr, Full(Data(seqno), hdr))) -> OptSend(seqno, hdr)
  | Full(NoHdr, Full(Data(seqno), Full(NoHdr, hdr))) -> OptCast(seqno, hdr)
  | hdr -> Normal(hdr)

- **Optimization theorem including header compression**

  OPTIMIZING LAYER Pt2pt::Mnak::Bottom
  FOR EVENT DnM(ev, msg)
  AND STATE (s_pt2pt, s_mnak, s_bottom)
  ASSUMING getType ev = ESend ∧ getPeer ev ≠ ls.rank ∧ s_bottom.enabled
  YIELDS HANDLERS dn ev (OptSend(Iq.hi s_pt2pt.sends.(getPeer ev), msg))
  AND UPDATES Iq.add (Arraye.get s_pt2pt.sends (getPeer ev))(getIov ev) msg
1. **Convert Theorems into Code Pieces**
   - handlers + updates $\mapsto$ command sequence
   - CCP $\mapsto$ conditional / case-expression
   - Call original event handler if CCP fails
1. **Convert Theorems into Code Pieces**
   - handlers + updates $\mapsto$ command sequence
   - CCP $\mapsto$ conditional / case-expression
   - Call original event handler if CCP fails

2. **Assemble Code Pieces**
   - Case expression for 4 common cases
     (send/receive, broadcast/point-to-point)
   - Call original event handler in final case
1. **Convert Theorems into Code Pieces**
   - handlers + updates $\mapsto$ command sequence
   - CCP $\mapsto$ conditional / case-expression
   - Call original event handler if CCP fails

2. **Assemble Code Pieces**
   - Case expression for 4 common cases
     (send/receive, broadcast/point-to-point)
   - Call original event handler in final case

3. **Export into OCaml environment**
1. **Convert Theorems into Code Pieces**
   - handlers + updates $\mapsto$ command sequence
   - CCP $\mapsto$ conditional / case-expression
   - Call original event handler if CCP fails

2. **Assemble Code Pieces**
   - Case expression for 4 common cases
     (send/receive, broadcast/point-to-point)
   - Call original event handler in final case

3. **Export into OCaml environment**

   **Fully automated,**
   generated code 3–10 times faster
Specifications and Correctness

- **System properties**
  
  "Messages are received in the same order in which they were sent"
  
  – Represented in **formal mathematics**
Specifications and Correctness

- **System properties**
  
  “Messages are received in the same order in which they were sent”
  
  – Represented in formal mathematics

- **Abstract (global) behavioral specification**
  
  “Messages may be appended to global event queue and removed from its beginning”
  
  – Represented as formal nondeterministic I/O Automaton
Specifications and Correctness

- **System properties**
  “Messages are received in the same order in which they were sent”
  – Represented in formal mathematics

- **Abstract (global) behavioral specification**
  “Messages may be appended to global event queue and removed from its beginning”
  – Represented as formal nondeterministic I/O Automaton

- **Concrete (local) behavioral specification**
  “Messages whose sequence number is too big will be buffered”
  – Represented as deterministic I/O Automaton
Specifications and Correctness

• **System properties**
  “Messages are received in the same order in which they were sent”
  – Represented in **formal mathematics**

• **Abstract (global) behavioral specification**
  “Messages may be appended to global event queue and removed from its beginning”
  – Represented as formal **nondeterministic I/O Automaton**

• **Concrete (local) behavioral specification**
  “Messages whose sequence number is too big will be buffered”
  – Represented as **deterministic I/O Automaton**

• **Implementation**
  – **ENSEMBLE** module Pt2pt.ml: **250 lines of OCaml code**
Specifications and Correctness

- **System properties**
  
  “*Messages are received in the same order in which they were sent*”

  - Represented in *formal mathematics*

- **Abstract (global) behavioral specification**

  “*Messages may be appended to global event queue and removed from its beginning*”

  - Represented as *formal nondeterministic I/O Automaton*

- **Concrete (local) behavioral specification**

  “*Messages whose sequence number is too big will be buffered*”

  - Represented as *deterministic I/O Automaton*

- **Implementation**

  - *ENSEMBLE* module `Pt2pt.ml`: 250 lines of OCaml code

  All formalisms can be represented in Nuprl’s type theory
Example specifications of a FIFO network

**FIFO property**

\[ \forall i, j, k, l < |\text{tr}| . (i < j \land \text{tr}[i] \downarrow \text{tr}[k] \land \text{tr}[j] \downarrow \text{tr}[l]) \Rightarrow k < l \]
Example specifications of a FIFO network

FIFO property

\[ \forall i, j, k, l < |\text{tr}| . (i < j \land \text{tr}[i] \downarrow \text{tr}[k] \land \text{tr}[j] \downarrow \text{tr}[l]) \Rightarrow k < l \]

Abstract behavioral specification as formal I/O-automaton

<table>
<thead>
<tr>
<th>Specification</th>
<th>FifoNetwork()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>in-transit: queue of ( \langle \text{Address}, \text{Message} \rangle )</td>
</tr>
<tr>
<td>Actions</td>
<td>Send(( dst : \text{Address}; \text{msg} : \text{Message} ))</td>
</tr>
<tr>
<td></td>
<td>condition: true {in-transit.append(( \langle dst, msg \rangle ))}</td>
</tr>
<tr>
<td></td>
<td>Deliver(( dst : \text{Address}; \text{msg} : \text{Message} ))</td>
</tr>
<tr>
<td></td>
<td>condition: in-transit.head() = ( \langle dst, msg \rangle ) {in-transit.dequeue()}</td>
</tr>
</tbody>
</table>
Example specifications of a FIFO network

FIFO property

∀i,j,k,l<|tr|. (i<j ∧ tr[i]↓tr[k] ∧ tr[j]↓tr[l]) ⇒ k<l

Abstract behavioral specification as formal I/O-automaton

Specification  
FifoNetwork()

Variables  
in-transit: queue of \langle Address, Message \rangle

Actions  
Send(dst : Address; msg : Message)
condition: true  
{in-transit.append(\langle dst, msg \rangle)}

Deliver(dst : Address; msg : Message)
condition: in-transit.head() = \langle dst, msg \rangle  
{in-transit.dequeue()}

Concrete behavioral specification as formal I/O-automaton

Specification  
FifoProtocol(p : Address)

Variables  
send-window, recv-window, ...

Actions  
Above.Send(dst : Address; msg : Message)
{ ... list of individual sub-actions ... }

Below.Send(dst : Address; \langle hdr, msg \rangle : \langle Header, Message \rangle)
Below.Deliver(dst : Address; \langle hdr, msg \rangle : \langle Header, Message \rangle)
Above.Deliver(dst : Address; msg : Message)
Timer()
• Verify **IOA-specifications of micro-protocols**
  – Concrete specification ↔ abstract specification → system properties
  – Easy for benign networks

\[\sim \text{subtle bug discovered}\]
• Verify **IOA-specifications of micro-protocols**
  – Concrete specification ↔ abstract specification → system properties
  – Easy for benign networks

• Verify **protocol stacks by IOA-composition**
  – IOA-Composition preserves all safety properties

**Verification Methodology**

- Verify IOA-specifications of micro-protocols
  - Concrete specification ↔ abstract specification → system properties
  - Easy for benign networks
    - subtlet bug discovered

- Verify protocol stacks by IOA-composition
  - IOA-Composition preserves all safety properties

**Abstract Network Model**

**Concrete Behavioral Specification**

**Implementation**

**Properties**

**Abstract Behavioral Specification**

**Refinement**

**Scheduling**
• **Verify IOA-specifications of micro-protocols**
  – Concrete specification $\leftrightarrow$ abstract specification $\rightarrow$ system properties
  – Easy for benign networks $\sim$ subtle bug discovered

• **Verify protocol stacks by IOA-composition**
  – IOA-Composition preserves all safety properties

• **Weave Aspects** (subtle, still open)
  – Transformations add tolerance against network failures or security attacks
• **Verify IOA-specifications of micro-protocols**
  – Concrete specification ↔ abstract specification → system properties
  – Easy for benign networks

• **Verify protocol stacks by IOA-composition**
  – IOA-Composition preserves all safety properties

• **Weave Aspects** (subtle, still open)
  – Transformations add tolerance against network failures or security attacks

• **Verify code**
  – Micro-protocols ↔ IOA-specifications
  – Layer composition ↔ IOA-composition
**Lessons Learned**

- **Results**
  - Type theory *expressive enough* to formalize today’s software systems
  - Nuprl capable of supporting *real design* at reasonable pace
  - Formal optimization can significantly improve *practical performance*
  - Formal verification *reveals errors* even in well-investigated designs
  - Formal design *reveals hidden assumptions / limitations* of specifications
LESSONS LEARNED

• **Results**
  – Type theory **expressive enough** to formalize today’s software systems
  – Nuprl capable of supporting **real design** at reasonable pace
  – Formal optimization can significantly improve **practical performance**
  – Formal verification **reveals errors** even in well-investigated designs
  – Formal design **reveals hidden assumptions / limitations** of specifications

• **Ingredients for success . . .**
  – **Collaboration** between systems and formal reasoning groups
  – Small and **simple components**, well-defined module composition
  – Implementation language with **precise semantics**
Lessons learned

• Results
  – Type theory expressive enough to formalize today’s software systems
  – Nuprl capable of supporting real design at reasonable pace
  – Formal optimization can significantly improve practical performance
  – Formal verification reveals errors even in well-investigated designs
  – Formal design reveals hidden assumptions / limitations of specifications

• Ingredients for success ...
  – Collaboration between systems and formal reasoning groups
  – Small and simple components, well-defined module composition
  – Implementation language with precise semantics
  – Formal models of: communication, IO-automata, programming language
  – Employing formal methods at every design stage
  – Knowledge-based approach using large library of algorithmic knowledge
  – Cooperating reasoning tools
Lessons learned

• Results
  – Type theory expressive enough to formalize today’s software systems
  – Nuprl capable of supporting real design at reasonable pace
  – Formal optimization can significantly improve practical performance
  – Formal verification reveals errors even in well-investigated designs
  – Formal design reveals hidden assumptions / limitations of specifications

• Ingredients for success . . .
  – Collaboration between systems and formal reasoning groups
  – Small and simple components, well-defined module composition
  – Implementation language with precise semantics
  – Formal models of: communication, IO-automata, programming language
  – Employing formal methods at every design stage
  – Knowledge-based approach using large library of algorithmic knowledge
  – Cooperating reasoning tools
  – Great colleagues! Stuart Allen, Mark Bickford, Ken Birman, Robert Constable,
    Richard Eaton, Xiuming Liu, Lori Lorigo, Robbert van Renesse
Future Challenges

The Ensemble case study is just a ‘proof of concept’
The Ensemble case study is just a ‘proof of concept’

- Build better reasoning tools
  - Build interactive library of formal algorithmic knowledge
  - Increase performance and application range of proof tools
  - Connect more external systems
  - Improve cooperation between research groups
The Ensemble case study is just a ‘proof of concept’

- **Build better reasoning tools**
  - Build interactive library of formal algorithmic knowledge
  - Increase performance and application range of proof tools
  - Connect more external systems
  - Improve cooperation between research groups

- **Learn more from applications**
  - Support reasoning about real-time & embedded systems
    - reason about probabilistic protocols
    - reason about end-to-end quality of service
  - Support programming languages with less clean semantics
  - Invert reasoning direction from verification to synthesis
NEW RESEARCH ISSUES: LANGUAGE-BASED SECURITY

When can we trust downloaded code?
NEW RESEARCH ISSUES: LANGUAGE-BASED SECURITY

When can we trust downloaded code?

• Application scenarios
  – Programmable mobile devices (cell phones, smart cards, ...)
  – Plug-ins for internet browsers
  – Downloaded code has to be checked for secure behavior
  – Acceptable code does not leak private data to other processes
NEW RESEARCH ISSUES: LANGUAGE-BASED SECURITY

When can we trust downloaded code?

- **Application scenarios**
  - Programmable mobile devices (cell phones, smart cards, ...)
  - Plug-ins for internet browsers
  - Downloaded code has to be checked for secure behavior
  - Acceptable code does not leak private data to other processes

- **This has nothing to do with**
  - Encryption algorithms or cryptographic protocols
  - Malicious behavior beyond revealing secrets
NEW RESEARCH ISSUES: LANGUAGE-BASED SECURITY

When can we trust downloaded code?

- **Application scenarios**
  - Programmable mobile devices (cell phones, smart cards, ...)
  - Plug-ins for internet browsers
  - Downloaded code has to be **checked for secure behavior**
  - Acceptable code does not leak private data to other processes

- **This has nothing to do with**
  - Encryption algorithms or cryptographic protocols
  - Malicious behavior beyond revealing secrets

- **It is about information flow**
  - Simple setting: distinguish high and low confidence levels
  - Secure code will be able to access high-confidence information ($h$)
  - Modifications of low-confidence data ($l$) must not depend on $h$-data

*Can we check secure behavior statically by looking at the code?*
When can code be considered secure?

- **The decision is not always easy**
  - \( l := h \)
When can code be considered secure?

• The decision is not always easy
  – \( l := h \) obviously insecure, data are copied directly
When can code be considered secure?

- The decision is not always easy
  - \( l := h \)  
  - \( \text{if } h \text{ then } l := 0 \text{ else } l := 1 \) obviously insecure, data are copied directly
When can code be considered secure?

- **The decision is not always easy**
  
  - \( l := h \)  
    
    *obviously insecure, data are copied directly*
  
  - \( \text{if } h \text{ then } l := 0 \text{ else } l := 1 \)  
    
    *boolean information is revealed*
When can code be considered secure?

• The decision is not always easy
  – l:=h  
    obviously insecure, data are copied directly
  – if h then l:=0 else l:=1  
    boolean information is revealed
  – l:=h;...; l:=0
When can code be considered secure?

- The decision is not always easy
  - $l := h$  
    obviously insecure, data are copied directly
  - if $h$ then $l := 0$ else $l := 1$  
    boolean information is revealed
  - $l := h; \ldots; l := 0$  
    secure if attacker only sees final result
When can code be considered secure?

• The decision is not always easy
  – l := h  obviously insecure, data are copied directly
  – if h then l := 0 else l := 1  boolean information is revealed
  – l := h; ...; l := 0  secure if attacker only sees final result
  – if h then (skip; l := 1) else l := 1
When can code be considered secure?

• The decision is not always easy
  – $l := h$ obviously insecure, data are copied directly
  – if $h$ then $l := 0$ else $l := 1$ boolean information is revealed
  – $l := h; \ldots; l := 0$ secure if attacker only sees final result
  – if $h$ then (skip; $l := 1$) else $l := 1$ not secure if attacker can measure execution time
When can code be considered secure?

- The decision is not always easy
  - `l:=h` \textit{obviously insecure, data are copied directly}
  - `if h then l:=0` else `l:=1` \textit{boolean information is revealed}
  - `l:=h;...; l:=0` \textit{secure if attacker only sees final result}
  - `if h then (skip; l:=1)` else `l:=1`
    \textit{not secure if attacker can measure execution time}

- It depends on the capabilities of the attacker
  - Can the attacker see intermediate results?
  - Can the attacker analyze hardware performance (heat, time, ...)?
When can code be considered secure?

- **The decision is not always easy**
  - `l:=h` obviously insecure, data are copied directly
  - `if h then l:=0 else l:=1` boolean information is revealed
  - `l:=h;...; l:=0` secure if attacker only sees final result
  - `if h then (skip; l:=1) else l:=1` not secure if attacker can measure execution time

- **It depends on the capabilities of the attacker**
  - Can the attacker see intermediate results?
  - Can the attacker analyze hardware performance (heat, time, ...)?

- **It also depends on the computation environment**
  - How are multi-threaded processes handled by the scheduler?
When can code be considered secure?

- The decision is not always easy
  - \( l := h \) obviously insecure, data are copied directly
  - if \( h \) then \( l := 0 \) else \( l := 1 \) boolean information is revealed
  - \( l := h; \ldots; l := 0 \) secure if attacker only sees final result
  - if \( h \) then (skip; \( l := 1 \)) else \( l := 1 \) not secure if attacker can measure execution time

- It depends on the capabilities of the attacker
  - Can the attacker see intermediate results?
  - Can the attacker analyze hardware performance (heat, time, ...)?

- It also depends on the computation environment
  - How are multi-threaded processes handled by the scheduler?
    What if program 4 runs concurrently with \( \text{skip}; l := 0 \)?
When can code be considered secure?

• The decision is not always easy
  – \( l := h \) \hspace{1cm} \textit{obviously insecure, data are copied directly}
  – if \( h \) then \( l := 0 \) else \( l := 1 \) \hspace{1cm} \textit{boolean information is revealed}
  – \( l := h ; \ldots ; l := 0 \) \hspace{1cm} \textit{secure if attacker only sees final result}
  – if \( h \) then (skip; \( l := 1 \)) else \( l := 1 \) \hspace{1cm} \textit{not secure if attacker can measure execution time}

• It depends on the capabilities of the attacker
  – Can the attacker see intermediate results?
  – Can the attacker analyze hardware performance (heat, time, ...)?

• It also depends on the computation environment
  – How are multi-threaded processes handled by the scheduler?
    What if program 4 runs concurrently with \( \text{skip; } l := 0 \) ?
    • Code is secure in sequential setting
When can code be considered secure?

- The decision is not always easy
  - \( l := h \) \textit{obviously insecure, data are copied directly}
  - \( \text{if } h \text{ then } l := 0 \text{ else } l := 1 \) \textit{boolean information is revealed}
  - \( l := h; \ldots; l := 0 \) \textit{secure if attacker only sees final result}
  - \( \text{if } h \text{ then (skip; } l := 1 \text{) else } l := 1 \) \textit{not secure if attacker can measure execution time}

- It depends on the capabilities of the attacker
  - Can the attacker see intermediate results?
  - Can the attacker analyze hardware performance (heat, time, ...)?

- It also depends on the computation environment
  - How are multi-threaded processes handled by the scheduler?
    - What if program 4 runs concurrently with \texttt{skip; l:=0}?
      - Code is secure in sequential setting
      - Not secure when schedule involves shared memory, round robin:
        Final result of \( l \) is 1, if \( h \) and otherwise 0
When can code be considered secure?

- The decision is not always easy
  - \( l := h \)  
    - obviously insecure, data are copied directly
  - if \( h \) then \( l := 0 \) else \( l := 1 \)  
    - boolean information is revealed
  - \( l := h ; . . . ; l := 0 \)  
    - secure if attacker only sees final result
  - if \( h \) then (skip; \( l := 1 \)) else \( l := 1 \)  
    - not secure if attacker can measure execution time

- It depends on the capabilities of the attacker
  - Can the attacker see intermediate results?
  - Can the attacker analyze hardware performance (heat, time, ...)?

- It also depends on the computation environment
  - How are multi-threaded processes handled by the scheduler?
    - What if program 4 runs concurrently with skip; \( l := 0 \)?
      - Code is secure in sequential setting
      - Not secure when schedule involves shared memory, round robin:
        - Final result of \( l \) is 1, if \( h \) and otherwise 0

Security checkers need precise security models
• **Use an abstract programming language**
  
  – **MWL**: Multi-threaded while language
  
  – Allows clearer formulation of security conditions
  
  – Mechanism can be adapted to, e.g., Java byte code, “automatically”
Security Formalized

• **Use an abstract programming language**
  – **MWL**: Multi-threaded while language
  – Allows clearer formulation of security conditions
  – Mechanism can be adapted to, e.g., Java byte code, “automatically”

• **Formalize security relations**
  – Define program states that are undistinguishable for attackers
    \[ s_1 =_L s_2 \equiv \forall v : \text{Lowvar}. \ s_1(v) = s_2(v) \]
• **Use an abstract programming language**
  - **MWL**: Multi-threaded while language
  - Allows clearer formulation of security conditions
  - Mechanism can be adapted to, e.g., Java byte code, “automatically”

• **Formalize security relations**
  - Define program states that are undistinguishable for attackers
    
    
    $s_1 =_L s_2 \equiv \forall v : \text{Lowvar. } s_1(v) = s_2(v)$

  - Programs are undistinguishable if they are always “low-equal”
    
    $\sim_L \equiv \bigcup \{ \sim \mid P \sim P' \land s =_L s' \land [P_i, s] \mapsto [X, t] \land \exists X' : \text{Com. } \exists t' : \text{St. } [P_i', s'] \mapsto [X', t'] \land t =_L t' \land X \sim X' \}$
Security Formalized

- **Use an abstract programming language**
  - **MWL**: Multi-threaded while language
  - Allows clearer formulation of security conditions
  - Mechanism can be adapted to, e.g., Java byte code, “automatically”

- **Formalize security relations**
  - Define program states that are undistinguishable for attackers
    - \( s_1 =_L s_2 \equiv \forall v:\text{Lowvar. } s_1(v) = s_2(v) \)
  - Programs are undistinguishable if they are always “low-equal”
    - \( \sim_L \equiv \bigcup \{ \sim \mid P \sim P' \land s =_L s' \land [P_i, s] \mapsto [X, t] \}
      \Rightarrow \exists X':\text{Com. } \exists t':\text{St. } [P'_i, s'] \mapsto [X', t'] \land t =_L t' \land X \sim X' \}

- **Introduce type-checking rules for security**
  - 15 inference rules for proving equivalence of programs
  - Prove that the rules are correct and complete
Checking Security of Code

Code is acceptable if it can be made secure
Code is acceptable if it can be made secure

- **Make information flow secure by transformation**
  - Insert and instantiate meta-variables into the code
    - Essentially add appropriate skip-operations
  - 11 transformation rules for modifying programs
  - **Prove that the transformed program is equivalent and has secure information flow**
Checking Security of Code

Code is acceptable if it can be made secure

- **Make information flow secure by transformation**
  - Insert and instantiate meta-variables into the code
    - Essentially add appropriate skip-operations
  - 11 transformation rules for modifying programs
  - **Prove that the transformed program is equivalent and has secure information flow**

- **Automate application of transformation rules**
  - Automate injection and unification of meta-variables
  - Involves inference rules for lifting and AC1-unification
  - **Prove that automated mechanism is complete**
Checking Security of Code

Code is acceptable if it can be made secure

- **Make information flow secure by transformation**
  - Insert and instantiate meta-variables into the code
    - Essentially add appropriate skip-operations
  - 11 transformation rules for modifying programs
  - **Prove that the transformed program is equivalent and has secure information flow**

- **Automate application of transformation rules**
  - Automate injection and unification of meta-variables
  - Involves inference rules for lifting and AC1-unification
  - **Prove that automated mechanism is complete**

- **Research challenge (e.g. thesis work)**
  - Validate correctness and completeness with tactical theorem prover
  - Develop proof tactics that can validate similar security concepts