Proving termination using dependent types: the case of xor-terms

J.-F. Monin  J. Courant

VERIMAG
Grenoble, France

GDR LAC, Chambey, 2007
Outline

Motivation
- The case of cryptographic systems
- State of the art
- Back to cryptographic systems
- Solving strategies

Solution (intuitive)
- Basic idea
- Analyse of $\mathcal{T}$
- Decomposing $\mathcal{T}$
- Stratifying and normalizing a term

Issues
- Lifting
- Alternation
- Forbid fake inclusions
- Fixpoints
- Conversion rule

Conclusion
Formal models of cryptographic systems

XOR is ubiquitous

Examples from a security API called CCA (Common Cryptographic Architecture):

\[ x \oplus (K_P \oplus K_M) \mapsto \{ z \oplus y \} \]

\[ x \oplus K_P \oplus K_M \mapsto \{ z \oplus y \} \]

Reasoning involves:

- **Commutativity:** \( x \oplus y \approx y \oplus x \)
- **Associativity:** \((x \oplus y) \oplus z \approx x \oplus (y \oplus z)\)
- **Neutral element:** \(x \oplus 0 \approx x\)
- **Involutivity:** \(x \oplus x \approx 0\)
Formal models of cryptographic systems

- Protocols
- Security APIs
Formal models of cryptographic systems

- Protocols
- Security APIs

Xor is ubiquitous
Formal models of cryptographic systems

- Protocols
- Security APIs

Xor is ubiquitous

Examples from a security API called CCA (Common Cryptographic Architecture):

\[ x, y, \{ z \} \times \oplus \rightarrow \{ z \oplus y \} \times \oplus \]

Reasoning involves:

- **Commutativity:** \[ x \oplus y \simeq y \oplus x \]
- **Associativity:** \[ (x \oplus y) \oplus z \simeq x \oplus (y \oplus z) \]
- **Neutral element:** \[ x \oplus 0 \simeq x \]
- **Involutivity:** \[ x \oplus x \simeq 0 \]
Formal models of cryptographic systems

- Protocols
- Security APIs

Xor is ubiquitous

Examples from a security API called CCA (Common Cryptographic Architecture):

\[
\begin{align*}
  x, y, \{z\}_x \oplus K_P \oplus K_M & \mapsto \{z \oplus y\}_x \oplus K_P \oplus K_M \\
  x, y, \{z\}_x \oplus K_P \oplus K_M & \mapsto \{z \oplus y\}_x \oplus K_M
\end{align*}
\]

Reasoning involves:

- **Commutativity:** \( x \oplus y \simeq y \oplus x \)
- **Associativity:** \((x \oplus y) \oplus z \simeq x \oplus (y \oplus z)\)
- **Neutral element:** \( x \oplus 0 \simeq x \)
- **Involutivity:** \( x \oplus x \simeq 0 \)
Outline

Motivation
- The case of cryptographic systems

State of the art
- Back to cryptographic systems
- Solving strategies

Solution (intuitive)
- Basic idea
- Analyse of $\mathcal{T}$
- Decomposing $\mathcal{T}$
- Stratifying and normalizing a term

Issues
- Lifting
- Alternation
- Forbid fake inclusions
- Fixpoints
- Conversion rule

Conclusion
General setting: quotiented first order-terms

We are given

- A type of terms $T$ with constructors $C_k$:
  
  Inductive $T : Set$ :=
  
  | $C_1 : T$
  
  \vdots
  
  | $C_k : \ldots \rightarrow T \ldots \rightarrow T \ldots \rightarrow T$
  
  \vdots

- A congruence $\equiv$:

- For each constructor $C_k$
  
  $\forall a, \ldots, x_1, y_1, b, \ldots, x_2, y_2, \ldots, c$
  
  $x_1 \equiv y_1 \rightarrow x_2 \equiv y_2 \rightarrow C_k a \ldots x_1 b \ldots y_1 c \equiv C_k a \ldots x_2 b \ldots y_2 c$

- specific laws, e.g.
  
  $\forall xy, C_2 x C_1 y \equiv C_2 y x$
General setting: quotiented first order-terms

We are given

- A type of terms $\mathcal{T}$ with constructors $C_k$:
  
  Inductive $\mathcal{T} : \text{Set} :=$
  
  \[
  \begin{align*}
  & \mid C_1 : \mathcal{T} \\
  & \vdots \\
  & \mid C_k : \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T} \\
  & \vdots \\
  \end{align*}
  \]

- A congruence $\simeq : \mathcal{T} \rightarrow \mathcal{T} \rightarrow \text{Prop}$

We want to reason on $\mathcal{T}$ up to $\simeq$. 

General setting: quotiented first order-terms

We are given

- A type of terms $\mathcal{T}$ with constructors $C_k$:
  Inductive $\mathcal{T}$: $\text{Set} :=$
  $| \quad C_1 : \mathcal{T}$
  $| \quad \vdots$
  $| \quad C_k : \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T}$
  $| \quad \vdots$

- A congruence $\simeq : \mathcal{T} \rightarrow \mathcal{T} \rightarrow \text{Prop}$

  For each constructor $C_k$
  \[
  \forall a, \ldots x_1, y_1, b, \ldots x_2, y_2, \ldots c,
  \ x_1 \simeq y_1 \rightarrow x_2 \simeq y_2 \rightarrow
  C_k a \ldots x_1 b \ldots y_1 c \simeq C_k a \ldots x_2 b \ldots y_2 c
  \]
General setting: quotiented first order-terms

We are given

- A type of terms $\mathcal{T}$ with constructors $C_k$:
  
  Inductive $\mathcal{T} : Set :=$
  
  | $C_1 : \mathcal{T}$
  | $\vdots$
  | $C_k : \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T}$
  | $\vdots$

- A congruence $\simeq : \mathcal{T} \rightarrow \mathcal{T} \rightarrow Prop$
  
  - For each constructor $C_k$
    
    $\forall a, \ldots x_1, y_1, b, \ldots x_2, y_2, \ldots c,$
    
    $x_1 \simeq y_1 \rightarrow x_2 \simeq y_2 \rightarrow$
    
    $C_k a \ldots x_1 b \ldots y_1 c \simeq C_k a \ldots x_2 b \ldots y_2 c$

  - specific laws, e.g. $\forall xy, C_2 \times C_1 y \simeq C_2 y x$
General setting: quotiented first order-terms

We are given

- A type of terms $T$ with constructors $C_k$:

  Inductive $T$: $Set :=$
  
  \[ C_1 : T \]
  
  \[ \vdots \]
  
  \[ C_k : \ldots \rightarrow T \ldots \rightarrow T \ldots \rightarrow T \]
  
  \[ \vdots \]

- A congruence $\simeq : T \rightarrow T \rightarrow Prop$

  - For each constructor $C_k$
    
    $\forall a, \ldots x_1, y_1, b, \ldots x_2, y_2, \ldots c,$
    
    $x_1 \simeq y_1 \rightarrow x_2 \simeq y_2 \rightarrow$
    
    $C_k a \ldots x_1 b \ldots y_1 c \simeq C_k a \ldots x_2 b \ldots y_2 c$

  - specific laws, e.g. $\forall xy, C_2 \times C_1 y \simeq C_2 y x$
General setting: quotiented first order-terms

We are given

- A type of terms $\mathcal{T}$ with constructors $C_k$:
  
  Inductive $\mathcal{T} : \text{Set}$ :=
  
  \[ | C_1 : \mathcal{T} | \ldots | C_k : \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T} \]

- A congruence $\simeq : \mathcal{T} \rightarrow \mathcal{T} \rightarrow \text{Prop}$

  - For each constructor $C_k$
    
    $\forall a, \ldots x_1, y_1, b, \ldots x_2, y_2, \ldots c,$
    
    $x_1 \simeq y_1 \rightarrow x_2 \simeq y_2 \rightarrow$
    
    $C_k a \ldots x_1 b \ldots y_1 c \simeq C_k a \ldots x_2 b \ldots y_2 c$

  - specific laws, e.g. $\forall xy, C_2 x C_1 y \simeq C_2 y x$

We want to reason on $\mathcal{T}$ up to $\simeq$
Already well-known examples

- finite bags represented by finite lists
Already well-known examples

- finite bags represented by finite lists
- algebra of formal arithmetic expressions
Already well-known examples

- finite bags represented by finite lists

- algebra of formal arithmetic expressions

- (mobile) process calculi, chemical abstract machines
Already well-known examples

- finite bags represented by finite lists
- algebra of formal arithmetic expressions
  - $+\$ is associative, commutative, 0 is neutral
  - $\times\$ is associative, commutative, 1 is neutral
  - $\times\$ distributes over $+$
- (mobile) process calculi, chemical abstract machines
  parallel composition and choice operators are AC
Quotients in type theory

- High level approach: setoids
Quotients in type theory

• High level approach: setoids

• Explicit approach:
Quotients in type theory

High level approach: setoids

Explicit approach:
- Define a normalization function $N$ on $T$
Quotients in type theory

- High level approach: setoids

- Explicit approach:
  - Define a normalization function $N$ on $\mathcal{T}$
  - Compare terms using syntactic equality on their norms: $x \simeq y$ iff $N x = N y$
Outline

Motivation
  The case of cryptographic systems
  State of the art
  Back to cryptographic systems
  Solving strategies

Solution (intuitive)
  Basic idea
  Analyse of $\mathcal{T}$
  Decomposing $\mathcal{T}$
  Stratifying and normalizing a term

Issues
  Lifting
  Alternation
  Forbid fake inclusions
  Fixpoints
  Conversion rule

Conclusion
Cryptographic systems need more
Reasoning on such systems involves
▶ comparing terms up to $\text{AC} + \text{involutivity of } \oplus$:

- **Commutativity:** $x \oplus y \simeq y \oplus x$
- **Associativity:** $(x \oplus y) \oplus z \simeq x \oplus (y \oplus z)$
- **Neutral element:** $x \oplus 0 \simeq x$
- **Involutivity:** $x \oplus x \simeq 0$
Cryptographic systems need more
Reasoning on such systems involves

- comparing terms up to AC + involutivity of $\oplus$:

  Commutativity: $x \oplus y \simeq y \oplus x$
  Associativity: $(x \oplus y) \oplus z \simeq x \oplus (y \oplus z)$
  Neutral element: $x \oplus 0 \simeq x$
  Involutivity: $x \oplus x \simeq 0$

- a relation $\preceq$ for occurrence:
  if $x$, $y$ and $z$ are different terms,
Cryptographic systems need more

Reasoning on such systems involves

▶ comparing terms up to AC + involutivity of ⊕:

- **Commutativity:** \( x \oplus y \simeq y \oplus x \)
- **Associativity:** \( (x \oplus y) \oplus z \simeq x \oplus (y \oplus z) \)
- **Neutral element:** \( x \oplus 0 \simeq x \)
- **Involutivity:** \( x \oplus x \simeq 0 \)

▶ a relation ≤ for occurrence:
  if \( x, y \) and \( z \) are different terms,
  ▶ \( y \) occurs in \( x \oplus y \oplus z \)
Cryptographic systems need more

Reasoning on such systems involves

- comparing terms up to AC + involutivity of \( \oplus \):
  
  **Commutativity:** \( x \oplus y \simeq y \oplus x \)
  
  **Associativity:** \((x \oplus y) \oplus z \simeq x \oplus (y \oplus z)\)
  
  **Neutral element:** \(x \oplus 0 \simeq x\)
  
  **Involutivity:** \(x \oplus x \simeq 0\)

- a relation \( \preceq \) for occurrence:
  
  if \( x, y \) and \( z \) are different terms,
  
  - \( y \) occurs in \( x \oplus y \oplus z \)
  - but \( y \) does **not** occur in \( x \oplus y \oplus z \oplus y \)
Cryptographic systems need more
Reasoning on such systems involves

- comparing terms up to AC + involutivity of $\oplus$:

  Commutativity: $x \oplus y \simeq y \oplus x$

  Associativity: $(x \oplus y) \oplus z \simeq x \oplus (y \oplus z)$

  Neutral element: $x \oplus 0 \simeq x$

  Involutivity: $x \oplus x \simeq 0$

- a relation $\preceq$ for occurrence:
  if $x$, $y$ and $z$ are different terms,
  - $y$ occurs in $x \oplus y \oplus z$
  - but $y$ does not occur in $x \oplus y \oplus z \oplus y$
Cryptographic systems need more reasoning on such systems involves

- comparing terms up to AC + involutivity of ⊕:

  - **Commutativity:** \( x \oplus y \simeq y \oplus x \)
  - **Associativity:** \((x \oplus y) \oplus z \simeq x \oplus (y \oplus z)\)
  - **Neutral element:** \(x \oplus 0 \simeq x\)
  - **Involutivity:** \(x \oplus x \simeq 0\)

- a relation \(\preceq\) for occurrence:
  - if \(x\), \(y\) and \(z\) are different terms,
    - \(y\) occurs in \(x \oplus y \oplus z\)
    - but \(y\) does not occur in \(x \oplus y \oplus z \oplus y\)

\[x \preceq y\] if \(x \simeq y\)
\[x \preceq t\] if \(t \simeq x \oplus y_0 \ldots \oplus y_n\)

and \(x \npreceq y_i\) for all \(i, 0 \leq i \leq n\)
Cryptographic systems need more

Reasoning on such systems involves

▶ comparing terms up to AC + involutivity of ⊕:

Commutativity: \[ x \oplus y \simeq y \oplus x \]
Associativity: \[ (x \oplus y) \oplus z \simeq x \oplus (y \oplus z) \]
Neutral element: \[ x \oplus 0 \simeq x \]
Involutivity: \[ x \oplus x \simeq 0 \]

▶ a relation \( \preceq \) for occurrence:
  if \( x, y \) and \( z \) are different terms,
  ▶ \( y \) occurs in \( x \oplus y \oplus z \)
  ▶ but \( y \) does not occur in \( x \oplus y \oplus z \oplus y \)

\[ x \preceq y \text{ if } x \simeq y \]
\[ x \preceq t \text{ if } t \simeq x \oplus y_0 \ldots \oplus y_n \]
and \( x \not\preceq y_i \) for all \( i, 0 \leq i \leq n \)

→ normalization is needed!
Outline

Motivation

The case of cryptographic systems
State of the art
Back to cryptographic systems

Solving strategies

Solution (intuitive)

Basic idea
Analyse of $\mathcal{T}$
Decomposing $\mathcal{T}$
Stratifying and normalizing a term

Issues

Lifting
Alternation
Forbid fake inclusions
Fixpoints
Conversion rule

Conclusion
First attempt: rewrite, rewrite, rewrite...
First attempt: rewrite, rewrite, rewrite...  

Replace equations with rewrite rules
First attempt: rewrite, rewrite, rewrite. . .

Replace equations with rewrite rules

Commutativity: find an suitable well ordering on terms
First attempt: rewrite, rewrite, rewrite... 

Replace equations with rewrite rules
Commutativity: find an suitable well ordering on terms

Functional programming approach:
  ▶ Not very difficult – use general recursion
First attempt: rewrite, rewrite, rewrite... 

Replace equations with rewrite rules

Commutativity: find an suitable well ordering on terms

Functional programming approach:
- Not very difficult – use general recursion
- Just boring
First attempt: rewrite, rewrite, rewrite. . .

Replace equations with rewrite rules

Commutativity: find an suitable well ordering on terms

Functional programming approach:
  ▶ Not very difficult – use \textit{general recursion}
  ▶ Just boring

In a type theoretic framework, termination proof mandatory and non-trivial:
  ▶ combination of polynomial and lexicographic ordering
  ▶ other approaches (lpo, rpo, . . .): overkill?
  ▶ AC matching: a non trivial matter
First attempt: rewrite, rewrite, rewrite... 

Replace equations with rewrite rules 

Commutativity: find an suitable well ordering on terms 

Functional programming approach: 
  ▶ Not very difficult – use general recursion 
  ▶ Just boring 

In a type theoretic framework, termination proof mandatory and non-trivial: 
    ▶ combination of polynomial and lexicographic ordering
First attempt: rewrite, rewrite, rewrite...

Replace equations with rewrite rules

Commutativity: find a suitable well ordering on terms

Functional programming approach:
  ▶ Not very difficult – use general recursion
  ▶ Just boring

In a type theoretic framework, termination proof mandatory and non-trivial:
  ▶ combination of polynomial and lexicographic ordering
  ▶ other approaches (lpo, rpo,...): overkill?
First attempt: rewrite, rewrite, rewrite, rewrite...

Replace equations with rewrite rules

Commutativity: find an suitable well ordering on terms

Functional programming approach:
  ▶ Not very difficult – use general recursion
  ▶ Just boring

In a type theoretic framework, termination proof mandatory and non-trivial:
  ▶ combination of polynomial and lexicographic ordering
  ▶ other approaches (lpo, rpo, ...): overkill?
  ▶ AC matching: a non trivial matter
(Dependent) type theoretic approach

Step 1
Consider a more structured version of $t$

Step 2
Normalize by structural induction on the newly typed version of $t$

Step 1 makes step 2 easy.

Better formulation:
$t:T$ transformed into $t':T'$.

Enriched version of $T$, trivial forgetful morphism $T' \rightarrow T$.

Interesting part = $T \rightarrow T'$
(Dependent) type theoretic approach

Step 1
- Consider a more structured version of $t$
(Dependent) type theoretic approach

Step 1

- Consider a more structured version of $t$
(Dependent) type theoretic approach

Step 1
- Consider a more structured version of $t$
  \[ = \text{provide an accurate and informative typing to } t \]
(Dependent) type theoretic approach

Step 1
▶ Consider a more structured version of \( t \)
   \[ = \text{provide an accurate and informative typing to } t \]

Step 2
▶ Normalize by structural induction on the newly typed version of \( t \)
(Dependent) type theoretic approach

Step 1

- Consider a more structured version of $t$
  
  $=\, \text{provide an accurate and informative typing to } t$

Step 2

- Normalize by structural induction on the newly typed version of $t$
(Dependent) type theoretic approach

Step 1
- Consider a more structured version of $t$
  $\Rightarrow$ provide an accurate and informative typing to $t$

Step 2
- Normalize by structural induction on the newly typed version of $t$

Step 1 makes step 2 easy.
(Dependent) type theoretic approach

Step 1
▶ Consider a more structured version of $t$
   $=$ provide an accurate and informative typing to $t$

Step 2
▶ Normalize by structural induction on the newly typed version of $t$

Step 1 makes step 2 easy.

Better formulation: $t : \mathcal{I}$ transformed into $t' : \mathcal{I}'$
   $\mathcal{I}'$ enriched version of $\mathcal{I}$,
   trivial forgetful morphism $\mathcal{I}' \to \mathcal{I}$. 
(Dependent) type theoretic approach

Step 1

▶ Consider a more structured version of \( t \)
  
  \( = \) provide an accurate and informative typing to \( t \)

Step 2

▶ Normalize by structural induction on the newly typed version of \( t \)

Step 1 makes step 2 easy.

Better formulation: \( t : T \) transformed into \( t' : T' \)
  
  \( T' \) enriched version of \( T \),
  
  trivial forgetful morphism \( T' \to T \).

Interesting part = \( T \to T' \)
Outline

Motivation
- The case of cryptographic systems
- State of the art
- Back to cryptographic systems
- Solving strategies

Solution (intuitive)
- Basic idea
- Analyse of \( \mathcal{T} \)
- Decomposing \( \mathcal{T} \)
- Stratifying and normalizing a term

Issues
- Lifting
- Alternation
- Forbid fake inclusions
- Fixpoints
- Conversion rule

Conclusion
Lasagnas reveal the truth

Layering a term:

Each layer possesses its own normalization function:

Normalizing pasta = identity

Normalizing sauce = rearranging + removing duplicates
Lasagnas reveal the truth

Layering a term

Layers do not communicate: each layer possesses its own normalization function

In our case: need 2 layers, pasta and sauce

Normalizing pasta = identity

Normalizing sauce = rearranging + removing duplicates
Lasagnas reveal the truth

- layering a term
Lasagnas reveal the truth

- layering a term
- layers do not communicate:
  each layer possesses its own normalization function
Lasagnas reveal the truth

- layering a term
- layers do not communicate:
  - each layer possesses its own normalization function
- in our case: need 2 layers, pasta and sauce
Lasagnas reveal the truth

- layering a term
- layers do not communicate:
  - each layer possesses its own normalization function
- in our case: need 2 layers, pasta and sauce
- normalizing pasta = identity
Lasagnas reveal the truth

- layering a term
- layers do not communicate: each layer possesses its own normalization function
- in our case: need 2 layers, pasta and sauce
- normalizing pasta = identity
- normalizing sauce = rearranging + removing duplicates
Outline

Motivation
- The case of cryptographic systems
- State of the art
- Back to cryptographic systems
- Solving strategies

Solution (intuitive)
- Basic idea
  - Analyse of $\mathcal{T}$
  - Decomposing $\mathcal{T}$
  - Stratifying and normalizing a term

Issues
- Lifting
- Alternation
- Forbid fake inclusions
- Fixpoints
- Conversion rule

Conclusion
$T$ as a lasagna
\( \mathcal{T} \) as a lasagna

Inductive \( \mathcal{T} : \) Set :=

\[
\begin{align*}
\text{Zero: } & \; \mathcal{T} \\
\text{PC: } & \; \text{public\_const} \to \mathcal{T} \\
\text{E: } & \; \mathcal{T} \to \mathcal{T} \to \mathcal{T} \\
\text{Xor: } & \; \mathcal{T} \to \mathcal{T} \to \mathcal{T} \\
\text{Hash: } & \; \mathcal{T} \to \mathcal{T} \to \mathcal{T}.
\end{align*}
\]
\( \mathcal{T} \) as a lasagna

Inductive \( \mathcal{T} : \text{Set} := \)

- \( \text{Zero} : \mathcal{T} \)
- \( \text{PC} : \text{public\_const} \to \mathcal{T} \)
- \( \text{SC} : \text{secret\_const} \to \mathcal{T} \)
- \( \text{E} : \mathcal{T} \to \mathcal{T} \to \mathcal{T} \)
- \( \text{Xor} : \mathcal{T} \to \mathcal{T} \to \mathcal{T} \)
- \( \text{Hash} : \mathcal{T} \to \mathcal{T} \to \mathcal{T} \).
$\mathcal{T}$ as a lasagna

Inductive $\mathcal{T}$: Set :=
\[
\begin{align*}
&\quad | \text{Zero: } \mathcal{T} \\
&\quad | \text{PC: } \text{public\_const} \to \mathcal{T} | \text{SC: } \text{secret\_const} \to \mathcal{T} \\
&\quad | \text{E: } \mathcal{T} \to \mathcal{T} \to \mathcal{T} \\
&\quad | \text{Xor: } \mathcal{T} \to \mathcal{T} \to \mathcal{T} \\
&\quad | \text{Hash: } \mathcal{T} \to \mathcal{T} \to \mathcal{T}.
\end{align*}
\]
Outline

Motivation
The case of cryptographic systems
State of the art
Back to cryptographic systems
Solving strategies

Solution (intuitive)
Basic idea
Analyse of $\mathcal{T}$
Decomposing $\mathcal{T}$
Stratifying and normalizing a term

Issues
Lifting
Alternation
Forbid fake inclusions
Fixpoints
Conversion rule

Conclusion
Decomposing $\mathcal{T}$

Inductive $\mathcal{T}_x : Set :=$
- $X_{-}\text{Zero} : \mathcal{T}_x$
- $X_{-}\text{Xor} : \mathcal{T}_x \rightarrow \mathcal{T}_x \rightarrow \mathcal{T}_x$

Inductive $\mathcal{T}_n : Set :=$
- $NX_{-}\text{PC} : public\_const \rightarrow \mathcal{T}_n$
- $NX_{-}\text{SC} : secret\_const \rightarrow \mathcal{T}_n$
- $NX_{-}\text{E} : \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n$
- $NX_{-}\text{Hash} : \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n$
Decomposing $\mathcal{T}$

Variable $A : \text{Set}$.

Inductive $\mathcal{T}_x : \text{Set} :=$

\[ \begin{align*}
& \mid X\_Zero : \mathcal{T}_x \\
& \mid X\_Xor : \mathcal{T}_x \to \mathcal{T}_x \to \mathcal{T}_x \\
& \mid X\_ns : A \to \mathcal{T}_x
\end{align*} \]

Inductive $\mathcal{T}_n : \text{Set} :=$

\[ \begin{align*}
& \mid NX\_PC : \text{public\_const} \to \mathcal{T}_n \\
& \mid NX\_SC : \text{secret\_const} \to \mathcal{T}_n \\
& \mid NX\_E : \mathcal{T}_n \to \mathcal{T}_n \to \mathcal{T}_n \\
& \mid NX\_Hash : \mathcal{T}_n \to \mathcal{T}_n \to \mathcal{T}_n \\
& \mid NX\_sum : A \to \mathcal{T}_n
\end{align*} \]
Outline

Motivation
- The case of cryptographic systems
- State of the art
- Back to cryptographic systems
- Solving strategies

Solution (intuitive)
- Basic idea
- Analyse of $\mathcal{T}$
- Decomposing $\mathcal{T}$
- Stratifying and normalizing a term

Issues
- Lifting
- Alternation
- Forbid fake inclusions
- Fixpoints
- Conversion rule

Conclusion
Stratifying and normalizing a term

Step 1
Translate a term $t$ into $t'$ according to the mapping $0 \mapsto X$, $\text{Xor} \mapsto X$, $\text{PC} \mapsto NX$, etc.

The typing of $t'$ is $T_x(T_n(T_x(\ldots(\emptyset)^k\ldots)))$ for $k$ large enough.

Step 2
A type is sortable if it is equipped with a decidable equality and a decidable total ordering. If $A$ is sortable, then $\nabla T_n(A)$ is sortable as well; $\nabla$ the multiset of $A$-leaves of a $T_x(A)$-term can be sorted (and removed when possible) into a list; $\nabla$ $\text{list}(A)$ is sortable.
Stratifying and normalizing a term

**Step 1** Translate a term $t$ into $t'$ according to the mapping

$0 \mapsto X_{\text{Zero}}, \ Xor \mapsto X_{\text{Xor}}, \ PC \mapsto NX_{\text{PC}},$ etc.
Stratifying and normalizing a term

Step 1 Translate a term $t$ into $t'$ according to the mapping $0 \mapsto X\_Zero$, $Xor \mapsto X\_Xor$, $PC \mapsto NX\_PC$, etc.

Step 2 A type is sortable if it is equipped with a decidable equality and a decidable total ordering. If $A$ is sortable, then
Stratifying and normalizing a term

**Step 1** Translate a term $t$ into $t'$ according to the mapping
$0 \mapsto X_{\text{Zero}}, Xor \mapsto X_{\text{Xor}}, PC \mapsto NX_{\text{PC}},$ etc.

**Step 2** A type is **sortable** if it is equipped with a decidable equality and a decidable total ordering. If $A$ is sortable, then
- $\mathcal{I}_n(A)$ is sortable as well;
Stratifying and normalizing a term

**Step 1** Translate a term $t$ into $t'$ according to the mapping
$0 \mapsto X_{\text{Zero}}, \ Xor \mapsto X_{\text{Xor}}, \ PC \mapsto NX_{\text{PC}},$ etc.

**Step 2** A type is **sortable** if it is equipped with a decidable equality and a decidable total ordering. If $A$ is sortable, then
- $\mathcal{T}_n(A)$ is sortable as well;
- the multiset of $A$-leaves of a $\mathcal{T}_x(A)$-term can be sorted (and removed when possible) into a list;
Step 1 Translate a term $t$ into $t'$ according to the mapping $0 \mapsto \text{X_Zero}$, $\text{Xor} \mapsto \text{X_Xor}$, $\text{PC} \mapsto \text{NX_PPC}$, etc.

Step 2 A type is sortable if it is equipped with a decidable equality and a decidable total ordering. If $A$ is sortable, then

- $\mathcal{T}_n(A)$ is sortable as well;
- the multiset of $A$-leaves of a $\mathcal{T}_x(A)$-term can be sorted (and removed when possible) into a list;
- $\text{list}(A)$ is sortable.
Stratifying and normalizing a term

**Step 1** Translate a term $t$ into $t'$ according to the mapping
$0 \mapsto X\_Zero, \ Xor \mapsto X\_Xor, \ PC \mapsto NX\_PC$, etc.

The typing of $t'$ is $\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset)))))$ for $k$ large enough.

**Step 2** A type is **sortable** if it is equipped with a decidable equality and a decidable total ordering. If $A$ is sortable, then
- $\mathcal{T}_n(A)$ is sortable as well;
- the multiset of $A$-leaves of a $\mathcal{T}_x(A)$-term can be sorted (and removed when possible) into a list;
- $\text{list}(A)$ is sortable.
Stratifying and normalizing a term

**Step 1** Translate a term $t$ into $t'$ according to the mapping
$0 \mapsto X_{\text{Zero}}, \ Xor \mapsto X_{\text{Xor}}, \ PC \mapsto NX_{\text{PC}},$ etc.

The typing of $t'$ is $\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots (\emptyset))))$ for $k$ large enough.

**Step 2** A type is **sortable** if it is equipped with a decidable equality and a decidable total ordering. If $A$ is sortable, then
- $\mathcal{T}_n(A)$ is sortable as well;
- the multiset of $A$-leaves of a $\mathcal{T}_x(A)$-term can be sorted (and removed when possible) into a list;
- $\text{list}(A)$ is sortable.
Outline

Motivation
- The case of cryptographic systems
- State of the art
- Back to cryptographic systems
- Solving strategies

Solution (intuitive)
- Basic idea
- Analyse of $\mathcal{T}$
- Decomposing $\mathcal{T}$
- Stratifying and normalizing a term

Issues
- Lifting
- Alternation
- Forbid fake inclusions
- Fixpoints
- Conversion rule

Conclusion
Lifting lasagna

\[ L_x k \overset{\text{def}}{=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\ldots(\emptyset))))}_{k \text{ layers}} \text{ for } k \text{ large enough.} \]
Lifting lasagna

\[ L_x k \overset{\text{def}}{=} \mathcal{I}_x(\mathcal{I}_n(\mathcal{I}_x(\ldots (\emptyset)))) \text{ for } k \text{ large enough.} \]

- What is \( k \)?
Lifting lasagna

\[ \mathcal{L}_x k \overset{\text{def}}{=} \mathcal{I}_x(\mathcal{I}_n(\mathcal{I}_x(\ldots(\emptyset))))_{\text{k layers}} \] for \( k \) large enough.

- What is \( k \)?
- The number of layers on the left subterm and on the right subterm are different in general.
Lifting lasagna

\[ \mathcal{L}_x k \overset{\text{def}}{=} \mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset)))) \] for \( k \) large enough.

- What is \( k \)?
- The number of layers on the left subterm and on the right subterm are different in general.
Lifting lasagna

\[ L_x k \overset{\text{def}}{=} \underbrace{T_x(T_n(T_x(\ldots(\emptyset))))}_{k \text{ layers}} \text{ for } k \text{ large enough.} \]

- What is \( k \)?
- The number of layers on the left subterm and on the right subterm are different in general.

Take the max
Lifting lasagna

\[ \mathcal{L}_x k \overset{\text{def}}{=} \underbrace{\mathcal{I}_x(\mathcal{I}_n(\mathcal{I}_x(\ldots(\emptyset))))}_{k \text{ layers}} \] for \( k \) large enough.

- What is \( k \)?
- The number of layers on the left subterm and on the right subterm are different in general.

Take the max

- Standard solution: \( \{\text{le n m}\} + \{\text{le m n}\} \)
Lifting lasagna

\[ \mathcal{L}_x k \overset{\text{def}}{=} \mathcal{I}_x(\mathcal{I}_n(\mathcal{I}_x(\ldots(\emptyset)))) \text{ for } k \text{ large enough.} \]

▶ What is \( k \)?
▶ The number of layers on the left subterm and on the right subterm are different in general.

Take the max

▶ Standard solution: \{le n m\} + \{le m n\}
  ▶ interactive definition, large proof term
Lifting lasagna

\[ \mathcal{L}_x k \overset{\text{def}}{=} \{0\}^k \text{ for } k \text{ large enough.} \]

- What is \( k \)?
- The number of layers on the left subterm and on the right subterm are different in general.

Take the max

- Standard solution: \( \{\text{le } n\ m\} + \{\text{le } m\ n\} \)
  - interactive definition, large proof term
  - heavy encoding of \( m - n \) or \( n - m \)
Lifting lasagna

\[ \mathcal{L}_x k \overset{\text{def}}{=} \underbrace{\mathcal{I}_x(\mathcal{I}_n(\mathcal{I}_x(\ldots(\emptyset))))}_{k \text{ layers}} \] for \( k \) large enough.

▷ What is \( k \)?

▷ The number of layers on the left subterm and on the right subterm are different in general.

Take the max

▷ Standard solution: \( \{\text{le } n \ m\} + \{\text{le } m \ n\} \)
  ▷ interactive definition, large proof term
  ▷ heavy encoding of \( m - n \) or \( n - m \)
  ▷ need to lift \( \mathcal{L}_x n \) and \( \mathcal{L}_x m \) to \( \mathcal{L}_x (\text{max } n \ m) \)
Lifting lasagna

\[ \mathcal{L}_x k \overset{\text{def}}{=} \mathcal{I}_x(\mathcal{I}_n(\mathcal{I}_x(\ldots (\emptyset)))) \]  
for \( k \) large enough.

- What is \( k \)?
- The number of layers on the left subterm and on the right subterm are different in general.

Take the max

- Standard solution: \( \{\text{le } n \ m\} + \{\text{le } m \ n\} \)
  - interactive definition, large proof term
  - heavy encoding of \( m - n \) or \( n - m \)
  - need to lift \( \mathcal{L}_x \ n \) and \( \mathcal{L}_x \ m \) to \( \mathcal{L}_x (\max n \ m) \)
- Lightweight approach: \( \max n \ m \overset{\text{def}}{=} m + (n - m) \)
Lifting lasagna

\[ \mathcal{L}_x k \overset{\text{def}}{=} \mathcal{I}_x(\mathcal{I}_n(\mathcal{I}_x(\ldots (\emptyset)))) \] for \( k \) large enough.

- What is \( k \)?
- The number of layers on the left subterm and on the right subterm are different in general.

Take the max

- Standard solution: \( \{\text{le } n \text{ m}\} + \{\text{le } m \text{ n}\} \)
  - interactive definition, large proof term
  - heavy encoding of \( m - n \) or \( n - m \)
  - need to lift \( \mathcal{L}_x n \) and \( \mathcal{L}_x m \) to \( \mathcal{L}_x (\max n m) \)

- Lightweight approach: \( \max n m \overset{\text{def}}{=} m + (n - m) \)
  - \( \text{lift}_x : \mathcal{L}_x k \to \mathcal{L}_x (k + d) \), \( \text{lift}_n : \mathcal{L}_n k \to \mathcal{L}_n (k + d) \)
Lifting lasagna

\[ \mathcal{L}_x k \overset{\text{def}}{=} \mathcal{I}_x(\mathcal{I}_n(\mathcal{I}_x(\ldots(\emptyset)))) \]  for \( k \) large enough.

- What is \( k \)?
- The number of layers on the left subterm and on the right subterm are different in general.

Take the max

- Standard solution: \( \{\text{le } n \text{ m}\} + \{\text{le } m \text{ n}\} \)
  - interactive definition, large proof term
  - heavy encoding of \( m - n \) or \( n - m \)
  - need to lift \( \mathcal{L}_x n \) and \( \mathcal{L}_x m \) to \( \mathcal{L}_x (\max n m) \)

- Lightweight approach: \( \max n m \overset{\text{def}}{=} m + (n - m) \)
  - \( \text{lift}_x : \mathcal{L}_x k \to \mathcal{L}_x (k + d), \text{lift}_n : \mathcal{L}_n k \to \mathcal{L}_n (k + d) \)
  - No need to proof that \( \max \) is the max.
Outline

Motivation
The case of cryptographic systems
State of the art
Back to cryptographic systems
Solving strategies

Solution (intuitive)
Basic idea
Analyse of $\mathcal{T}$
Decomposing $\mathcal{T}$
Stratifying and normalizing a term

Issues
Lifting
**Alternation**
Forbid fake inclusions
Fixpoints
Conversion rule

Conclusion
Internalizing alternation

Well designed types help us to design programs. Many functions are defined by mutual induction, e.g. lift\(_x\) and lift\(_n\). Control them using alternating natural numbers:

**Inductive** \(\text{alt\_even} : \text{Set} = 0 \triangleright \text{alt\_even} \lor S \triangleright \text{alt\_odd} \rightarrow \text{alt\_even} \)**

with **\(\text{alt\_odd} : \text{Set} = S \triangleright \text{alt\_even} \lor 0 \rightarrow \text{alt\_odd} \)**
Internalizing alternation

Well designed types help us to design programs
Internalizing alternation

Well designed types help us to design programs

Many functions are defined by mutual induction, e.g. $lift_x$ and $lift_n$
Internalizing alternation

Well designed types help us to design programs

Many functions are defined by mutual induction, e.g. $\text{lift}_x$ and $\text{lift}_n$

Control them using alternating natural numbers
Internalizing alternation

Well designed types help us to design programs.

Many functions are defined by mutual induction, e.g. \( \text{lift}_x \) and \( \text{lift}_n \).

Control them using alternating natural numbers.

Inductive \( \text{alt}_{\text{even}} \): \( \text{Set} := \)

- \( 0_e : \text{alt}_{\text{even}} \)
- \( S_{o\rightarrow e} : \text{alt}_{\text{odd}} \rightarrow \text{alt}_{\text{even}} \)

with \( \text{alt}_{\text{odd}} \): \( \text{Set} := \)

- \( S_{e\rightarrow o} : \text{alt}_{\text{even}} \rightarrow \text{alt}_{\text{odd}} \)
Outline

Motivation
The case of cryptographic systems
State of the art
Back to cryptographic systems
Solving strategies

Solution (intuitive)
Basic idea
Analyse of $\mathcal{T}$
Decomposing $\mathcal{T}$
Stratifying and normalizing a term

Issues
Lifting
Alternation
Forbid fake inclusions
Fixpoints
Conversion rule

Conclusion
Forbid fake inclusions
Forbid fake inclusions

Inductive $\mathcal{T}_x$: \[
\text{Set} := \\
| X\_Zero : \mathcal{T}_x \\
| X\_ns : A \rightarrow \mathcal{T}_x \\
| X\_Xor : \mathcal{T}_x \rightarrow \mathcal{T}_x \rightarrow \mathcal{T}_x
\]

Inductive $\mathcal{T}_n$: \[
\text{Set} := \\
| NX\_PC : public\_const \rightarrow \mathcal{T}_n \\
| NX\_SC : secret\_const \rightarrow \mathcal{T}_n \\
| NX\_sum : A \rightarrow \mathcal{T}_n \\
| NX\_E : \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n \\
| NX\_Hash : \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n
\]
Forbid fake inclusions

Inductive $\mathcal{T}_x$: $\text{Set} :=$

- $\text{X_Zero}: \mathcal{T}_x$
- $\text{X_ns}: A \rightarrow \mathcal{T}_x$
- $\text{X_Xor}: \mathcal{T}_x \rightarrow \mathcal{T}_x \rightarrow \mathcal{T}_x$

Inductive $\mathcal{T}_n$: $\text{Set} :=$

- $\text{NX_PC}: \text{public\_const} \rightarrow \mathcal{T}_n$
- $\text{NX_SC}: \text{secret\_const} \rightarrow \mathcal{T}_n$
- $\text{NX_sum}: A \rightarrow \mathcal{T}_n$
- $\text{NX_E}: \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n$
- $\text{NX_Hash}: \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n$

$\text{X_ns} (\text{NX_sum} (\text{X_ns} (\text{NX_sum} (\ldots))))$
Forbid fake inclusions

Inductive $\mathcal{T}_x$: $bool \rightarrow Set :=$

| $X_{\text{Zero}} : \forall b, \mathcal{T}_x b$
| $X_{\text{ns}} : \forall b, \mathsf{ls\_true\ b} \rightarrow A \rightarrow \mathcal{T}_x b$
| $X_{\text{Xor}} : \forall b, \mathcal{T}_x \mathsf{true} \rightarrow \mathcal{T}_x \mathsf{true} \rightarrow \mathcal{T}_x b$

Inductive $\mathcal{T}_n$: $bool \rightarrow Set :=$

| $\mathsf{NX\_PC} : \forall b, \mathsf{public\_const} \rightarrow \mathcal{T}_n b$
| $\mathsf{NX\_SC} : \forall b, \mathsf{secret\_const} \rightarrow \mathcal{T}_n b$
| $\mathsf{NX\_sum} : \forall b, \mathsf{ls\_true\ b} \rightarrow A \rightarrow \mathcal{T}_n b$
| $\mathsf{NX\_E} : \forall b, \mathcal{T}_n \mathsf{true} \rightarrow \mathcal{T}_n \mathsf{true} \rightarrow \mathcal{T}_n b$
| $\mathsf{NX\_Hash} : \forall b, \mathcal{T}_n \mathsf{true} \rightarrow \mathcal{T}_n \mathsf{true} \rightarrow \mathcal{T}_n b$

$X_{\text{ns}} (\mathsf{NX\_sum} (X_{\text{ns}} (\mathsf{NX\_sum} (\ldots))))$
Outline

Motivation
   The case of cryptographic systems
   State of the art
   Back to cryptographic systems
   Solving strategies

Solution (intuitive)
   Basic idea
   Analyse of $\mathcal{T}$
   Decomposing $\mathcal{T}$
   Stratifying and normalizing a term

Issues
   Lifting
   Alternation
   Forbid fake inclusions
   Fixpoints
   Conversion rule

Conclusion
Mutual induction

- Prefer fixpoints: built-in computation, no inversion
Mutual induction

- Prefer fixpoints: built-in computation, no inversion
- Use map combinators
Mutual induction

- Prefer fixpoints: built-in computation, no inversion
- Use map combinators
Mutual induction

- Prefer fixpoints: built-in computation, no inversion
- Use map combinators

Many 10 lines definitions, almost no theorem
Mutual induction

- Prefer fixpoints: built-in computation, no inversion
- Use map combinators

Many 10 lines definitions, almost no theorem

Fixpoint $\text{lift\_lasagna\_x\ e}_1\ e_2$ \{\text{struct } e_1\} :

$\mathcal{L}_x\ e_1 \rightarrow \mathcal{L}_x\ (e_1 + e_2) :=$

match $e_1$ return $\mathcal{L}_x\ e_1 \rightarrow \mathcal{L}_x\ (e_1 + e_2)$ with

| $0_e$ ⇒ fun $\text{emp}$ ⇒ match $\text{emp}$ with end
| $S_{o \rightarrow e}\ o_1$ ⇒ $\text{map}_x\ (\text{lift\_lasagna\_n\ o}_1\ e_2)\ false$ end

with $\text{lift\_lasagna\_n\ o}_1\ e_2$ \{\text{struct } o_1\} :

$\mathcal{L}_n\ o_1 \rightarrow \mathcal{L}_n\ (o_1 + e_2) :=$

match $o_1$ return $\mathcal{L}_n\ o_1 \rightarrow \mathcal{L}_n\ (o_1 + e_2)$ with

| $S_{e \rightarrow o}\ e_1$ ⇒ $\text{map}_n\ (\text{lift\_lasagna\_x\ e}_1\ e_2)\ false$ end.
Outline

Motivation
The case of cryptographic systems
State of the art
Back to cryptographic systems
Solving strategies

Solution (intuitive)
Basic idea
Analyse of $\mathcal{T}$
Decomposing $\mathcal{T}$
Stratifying and normalizing a term

Issues
Lifting
Alternation
Forbid fake inclusions
Fixpoints
Conversion rule

Conclusion
Conversion rule
Conversion rule

Used everywhere
Conversion rule

Used everywhere

Definition \textit{bin\_xor}
\[
(bin : \forall A b, \mathcal{T}_x A \, \text{true} \to \mathcal{T}_x A \, \text{true} \to \mathcal{T}_x A \, b) \ o_1 \ o_2 \ b
(l_1 : \text{lasagna\_cand\_x} \ o_1 \ \text{true})
(l_2 : \text{lasagna\_cand\_x} \ o_2 \ \text{true}) : \text{lasagna\_cand\_x} \ (\text{max\_oo} \ o_1 \ o_2) \ b :=
\]
\[
b \ (\mathcal{L}_n \ (\text{max\_oo} \ o_1 \ o_2)) \ b
\quad (\text{lift\_lasagna\_cand\_x} \ \text{true} \ o_1 \ (o_2 - o_1) \ l_1)
\quad (\text{coerce\_max\_comm}
\quad \quad (\text{lift\_lasagna\_cand\_x} \ \text{true} \ o_2 \ (o_1 - o_2) \ l_2)).
\]
Conclusion

Type theory is flexible
Conclusion

Type theory is flexible

- Polymorphism

Conversion rule

JMEQ not used (until now)
Conclusion

Type theory is flexible

- Polymorphism
- Mutually inductive types
Conclusion

Type theory is flexible

- Polymorphism
- Mutually inductive types
- Dependent types
Conclusion

Type theory is flexible

- Polymorphism
- Mutually inductive types
- Dependent types
- Conversion rule
Conclusion

Type theory is flexible

- Polymorphism
- Mutually inductive types
- Dependent types
- Conversion rule
- JMEQ not used
Conclusion

Type theory is flexible

- Polymorphism
- Mutually inductive types
- Dependent types
- Conversion rule
- JMEQ *not used*
Conclusion

Type theory is flexible

- Polymorphism
- Mutually inductive types
- Dependent types
- Conversion rule
- JMEQ *not used* (until now)
Conclusion

Type theory is flexible

- Polymorphism
- Mutually inductive types
- Dependent types
- Conversion rule
- JMEQ *not used* (until now)