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

J.-F. Monin  J. Courant

VERIMAG
Grenoble, France

Trends in Functional Programming, Nottingham, 2006
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
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):

\[ x, y, \{z\} x \oplus KP \oplus KM \leftrightarrow \{z \oplus y\} x \oplus KP \oplus KM \]

\[ x, y, \{z\} x \oplus KP \oplus KM \leftrightarrow \{z \oplus y\} x \oplus KM \]
Formal models of cryptographic systems

- Protocols
- Security APIs

Xor is ubiquitous

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

\[ x, y, \{ z \} \xoplus KP \oplus KM \leftrightarrow \{ z \oplus y \} \xoplus KP \oplus KM \]
\[ x, y, \{ z \} \xoplus KP \oplus KM \leftrightarrow \{ z \oplus y \} \xoplus KM \]

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 $T$
Decomposing $T$
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 $\mathcal{T}$ with constructors $C_k$:
  
  Inductive $\mathcal{T}$: $\text{Set} :=$
  
  $| C_1 : \mathcal{T}$
  
  $| C_2 : \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T}$
  
  $| \ldots$
General setting: quotiented first order-terms

We are given

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

- A congruence $\simeq : \mathcal{T} \to \mathcal{T} \to \text{Prop}$
General setting: quotiented first order-terms

We are given

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

- 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 $\mathcal{T}$: $\text{Set} :=$
  $| C_1 : \mathcal{T}$
  $| C_2 : \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T} \ldots \rightarrow \mathcal{T}$
  $| \ldots$

- 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. $\forall x y, C_2 x C_1 y \simeq C_2 y x$
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} $
  
  $| C_2 : \ldots \to \mathcal{T} \ldots \to \mathcal{T} \ldots \to \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. $\forall x y, C_2 x C_1 y \simeq C_2 y x$
General setting: quotiented first order-terms

We are given

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

- 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 $T$
Proving termination using dependent types

J.-F. Monin, J. Courant

Motivation
Crypto. syst.
State of the art
Back to crypto
Solving strategies

Solution
(intuitive)
Basic idea
Analyse of $T$
Decomposing $T$
Stratification

Issues
Lifting
Alternation
Fake incl
Fixpoints
Conv rule

Conclusion

Quotients in type theory

- High level approach: setoids

- Explicit approach:
  - Define a normalization function $N$ on $T$
  - 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
Analyze 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 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: $x \oplus x \simeq 0$
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 \( \oplus \):

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 \( x \oplus y \oplus z \)
Cryptographic systems need more

Reasoning on such systems involves

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

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

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

  \[ x \preceq y \text{ if } x \simeq y \]
  \[ x \preceq t \text{ if } t \simeq x \oplus y_0 \ldots \oplus y_n \]
  and $x \npreceq y_i$ for all $i$, $0 \leq i \leq n$
Cryptographic systems need more

Reasoning on such systems involves

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

  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 $x \oplus y \oplus z \oplus y$

  $x \preceq y$ if $x \simeq y$
  $x \preceq t$ if $t \simeq x \oplus y_0 \ldots \oplus y_n$

  and $x \not\preceq y_i$ for all $i$, $0 \leq i \leq n$

$\rightarrow$ 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 a suitable well ordering on terms
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
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
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
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...

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

Motivation
- Crypto. syst.
- State of the art
- Back to crypto

Solving strategies

Solution
(intuitive)
- Basic idea
- Analyse of $T$
- Decomposing $T$
- Stratification

Issues
- Lifting
- Alternation
- Fake incl
- Fixpoints
- Conv rule

Conclusion
(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 $t$
  
  $\Rightarrow$ 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 $t$
  - = 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 $t$
  
  $=$ 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: $t : T$ transformed into $t' : T'$
$T'$ enriched version of $T$,
trivial forgetful morphism $T' \rightarrow T$. 
(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.

Better formulation: $t : \mathcal{T}$ transformed into $t' : \mathcal{T}'$
  $\mathcal{T}'$ enriched version of $\mathcal{T}$,
  trivial forgetful morphism $\mathcal{T}' \rightarrow \mathcal{T}$.

Interesting part $= \mathcal{T} \rightarrow \mathcal{T}'$
Outline

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

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

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

Conclusion
Lunch time!
Lasagnas reveal the truth

Motivation
Crypto. syst.
State of the art
Back to crypto
Solving strategies

Solution (intuitive)
Basic idea
Analyse of $T$
Decomposing $T$
Stratification

Issues
Lifting
Alternation
Fake incl
Fixpoints
Conv rule

Conclusion
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 $T$
- Decomposing $T$
- Stratifying and normalizing a term

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

Conclusion
\[ \mathcal{T} \] as a lasagna

Motivation
- Crypto. syst.
- State of the art
- Back to crypto

Solving strategies

Solution (intuitive)
- Basic idea
- Analyse of \( \mathcal{T} \)
- Decomposing \( \mathcal{T} \)
- Stratification

Issues
- Lifting
- Alternation
- Fake incl
- Fixpoints
- Conv rule

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

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

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

- Zero: \( \mathcal{T} \)
- \( PC: public\_const \to \mathcal{T} \)
- \( E: \mathcal{T} \to \mathcal{T} \to \mathcal{T} \)
- \( Xor: \mathcal{T} \to \mathcal{T} \to \mathcal{T} \)
- \( Hash: \mathcal{T} \to \mathcal{T} \to \mathcal{T} \).

\[
\begin{aligned}
E \\
H \\
P \oplus S \oplus 0 \\
P \oplus S \\
P \oplus 0
\end{aligned}
\]
\[ T \text{ as a lasagna} \]

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

- \( \text{Zero}: \ T \)
- \( PC: \ public\_const \rightarrow T \)
- \( SC: \ secret\_const \rightarrow T \)
- \( E: \ T \rightarrow T \rightarrow T \)
- \( \text{Xor}: \ T \rightarrow T \rightarrow T \)
- \( \text{Hash}: \ T \rightarrow T \rightarrow T \).

\[ E \]

\[ H \]

\[ P \]

\[ S \]

\[ 0 \]
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
Decomposing $\mathcal{T}$

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

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

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

- $NX_{\text{PC}} : \text{public\_const} \rightarrow \mathcal{T}_n$
- $NX_{\text{SC}} : \text{secret\_const} \rightarrow \mathcal{T}_n$
- $NX_{\text{E}} : \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n$
- $NX_{\text{Hash}} : \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n$
Proving termination using dependent types

J.-F. Monin, J. Courant

Motivation
Crypto. syst.
State of the art
Back to crypto
Solving strategies

Solution (intuitive)
Basic idea
Analyse of \( \mathcal{T} \)
Decomposing \( \mathcal{T} \)
Stratification

Issues
Lifting
Alternation
Fake incl
Fixpoints
Conv rule

Conclusion

Decomposing \( \mathcal{T} \)

Variable \( A : \text{Set} \).

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

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

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

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

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

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

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

Conclusion
Stratifying and normalizing a term

Proving termination using dependent types

J.-F. Monin, J. Courant

Motivation
Crypto. syst.
State of the art
Back to crypto
Solving strategies

Solution (intuitive)
Basic idea
Analyse of $T$
Decomposing $T$
Stratification

Issues
Lifting
Alternation
Fake incl
Fixpoints
Conv rule

Conclusion
Stratifying and normalizing a term

**Step 1** Translate a term $t$ into $t'$ according to the mapping $0 \mapsto X_{\text{Zero}}, \text{Xor} \mapsto X_{\text{Xor}}, \text{PC} \mapsto NX_{\text{PC}}$, etc.
Step 1 Translate a term $t$ into $t'$ according to the mapping $0 \mapsto X_{\text{Zero}}, \text{Xor} \mapsto X_{\text{Xor}}, \text{PC} \mapsto NX_{\text{PC}},$ 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
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
- $\mathcal{T}_n(A)$ is sortable as well;
Stratifying and normalizing a term

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

- $\mathcal{T}_n(A)$ 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;
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
- $T_n(A)$ is sortable as well;
- the multiset of $A$-leaves of a $T_x(A)$-term can be sorted (and removed when possible) into a list;
- $list(A)$ 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 $T_x(T_n(T_x(\ldots(\emptyset))))$ 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

- $T_n(A)$ is sortable as well;
- the multiset of $A$-leaves of a $T_x(A)$-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.
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

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

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

▶ What is \( k \)?

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

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

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

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

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

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

Take the max
Lifting lasagna

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

- 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: \( \{\leq n \; m\} + \{\leq m \; n\} \)
Lifting lasagna

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

- 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: \( \{\text{le } n \text{ m}\} + \{\text{le } m \text{ n}\} \)
  - interactive definition, large proof term
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.

Take the max

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

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

- 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: \( \{\text{le n m}\} + \{\text{le m n}\} \)
  - 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

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

- 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
  - heavy encoding of \( m - n \) or \( n - m \)
  - need to lift \( L_x n \) and \( L_x m \) to \( L_x (\text{max } n m) \)
- Lightweight approach: \( \text{max } n m \overset{\text{def}}{=} m + (n - m) \)
Lifting lasagna

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

- 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:** \( \{\text{le } n \text{ m}\} + \{\text{le } m \text{ n}\} \)
  - 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

\[ \mathcal{L}_x k \overset{\text{def}}{=} \underbrace{\mathcal{I}_x(\mathcal{I}_n(\mathcal{I}_x(\ldots(\emptyset))))}_{k \text{ layers}} \] 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: \( \{\text{le } n \text{ m}\} + \{\text{le } m \text{ n}\} \)
  - 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 \( \mathcal{L}_x (\text{max } n \text{ m}) \)
- Lightweight approach: \( \text{max } n \text{ m} \overset{\text{def}}{=} m + (n - m) \)
  - \( \text{lift}_x : \mathcal{L}_x k \rightarrow \mathcal{L}_x (k + d), \text{lift}_n : \mathcal{L}_n k \rightarrow \mathcal{L}_n (k + d) \)
  - No need to proof that \( \text{max} \) 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 \( T \)
- Decomposing \( T \)
- 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 and lift\(^n\). Control them using alternating natural numbers:

\[
\text{Inductive alt even:}\ \\
\text{Set:}\ \\
\ \\
\text{\textbar\ } 0 \text{ e:}\text{alt even}\ \\
\text{\textbar\ } S \rightarrow e: \text{alt odd}\rightarrow \text{alt even}
\]

with alt odd:

\[
\text{Set:}\ \\
\ \\
\text{\textbar\ } S \rightarrow o: \text{alt even}\rightarrow \text{alt odd}
\]
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. $lift_x$ and $lift_n$

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

\[
| 0_{\text{e}}: \text{alt}_{\text{even}} \\
| S_{0 \rightarrow e}: \text{alt}_{\text{odd}} \rightarrow \text{alt}_{\text{even}}
\]

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

\[
| S_{e \rightarrow 0}: \text{alt}_{\text{even}} \rightarrow \text{alt}_{\text{odd}}
\]
Outline

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

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

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

Conclusion
Forbid fake inclusions
Forbid fake inclusions

Inductive $\mathcal{T}_x$: \hfill Set :=
\begin{align*}
| \quad & X_{\text{Zero}} : \quad \mathcal{T}_x \\
| \quad & X_{\text{ns}} : \quad A \rightarrow \mathcal{T}_x \\
| \quad & X_{\text{Xor}} : \quad \mathcal{T}_x \rightarrow \mathcal{T}_x \rightarrow \mathcal{T}_x
\end{align*}

Inductive $\mathcal{T}_n$: \hfill Set :=
\begin{align*}
| \quad & NX_{\text{PC}} : \quad \text{public\_const} \rightarrow \mathcal{T}_n \\
| \quad & NX_{\text{SC}} : \quad \text{secret\_const} \rightarrow \mathcal{T}_n \\
| \quad & NX_{\text{sum}} : \quad A \rightarrow \mathcal{T}_n \\
| \quad & NX_{\text{E}} : \quad \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n \\
| \quad & NX_{\text{Hash}} : \quad \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n
\end{align*}
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 $\mathcal{T}_n$: \[\text{Set} :=
| \text{NX\_PC} : \text{public\_const} \rightarrow \mathcal{T}_n
| \text{NX\_SC} : \text{secret\_const} \rightarrow \mathcal{T}_n
| \text{NX\_sum} : A \rightarrow \mathcal{T}_n
| \text{NX\_E} : \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n
| \text{NX\_Hash} : \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n\]

$\text{X\_ns (NX\_sum ( X\_ns (NX\_sum (\ldots))))}$
Forbid fake inclusions

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

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

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

| $NX_{\text{PC}} : \forall b, \text{public\_const} \rightarrow \mathcal{T}_n b$
| $NX_{\text{SC}} : \forall b, \text{secret\_const} \rightarrow \mathcal{T}_n b$
| $NX_{\text{sum}} : \forall b, \text{ls\_true} b \rightarrow A \rightarrow \mathcal{T}_n b$
| $NX_{\text{E}} : \forall b, \mathcal{T}_n \text{true} \rightarrow \mathcal{T}_n \text{true} \rightarrow \mathcal{T}_n b$
| $NX_{\text{Hash}} : \forall b, \mathcal{T}_n \text{true} \rightarrow \mathcal{T}_n \text{true} \rightarrow \mathcal{T}_n b$

$X_{\text{ns}} (NX_{\text{sum}} (X_{\text{ns}} (NX_{\text{sum}} (...))))$
Outline

Motivation

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

Solution (intuitive)

Basic idea
Analyse of $T$
Decomposing $T$
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 \( \text{lift\_lasagna\_x} \ e_1 \ e_2 \ {\text{struct} \ e_1} \) :
\[
\mathcal{L}_x \ e_1 \to \mathcal{L}_x \ (e_1 + e_2) :=
\]
match \( e_1 \) return \( \mathcal{L}_x \ e_1 \to \mathcal{L}_x \ (e_1 + e_2) \) with
| \( 0_e \) \Rightarrow \text{fun} \ \text{emp} \Rightarrow \text{match} \ \text{emp} \text{ with end}
| \( S_{o \to e} \ o_1 \Rightarrow \text{map}_x \ (\text{lift\_lasagna\_n} \ o_1 \ e_2) \ false \)
end

with \( \text{lift\_lasagna\_n} \ o_1 \ e_2 \ {\text{struct} \ o_1} \) :
\[
\mathcal{L}_n \ o_1 \to \mathcal{L}_n \ (o_1 + e_2) :=
\]
match \( o_1 \) return \( \mathcal{L}_n \ o_1 \to \mathcal{L}_n \ (o_1 + e_2) \) with
| \( S_{e \to o} \ e_1 \Rightarrow \text{map}_n \ (\text{lift\_lasagna\_x} \ e_1 \ e_2) \ false \)
end.
Outline

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

Solution (intuitive)
- Basic idea
- Analyse of $T$
- Decomposing $T$
- 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}

\begin{align*}
\text{bin} & : \forall A \ b , \text{\old{T}}_x A \text{ true } \rightarrow \text{\old{T}}_x A \text{ true } \rightarrow \text{\old{T}}_x A \ b \\
(\text{l}_1 & : \text{lasagna\_cand\_x} \ o_1 \text{ true}) \\
(\text{l}_2 & : \text{lasagna\_cand\_x} \ o_2 \text{ true}) : \\
\text{lasagna\_cand\_x} (\text{max\_oo} \ o_1 \ o_2) \ b := \\
\text{bin} (\text{\old{L}}_n (\text{max\_oo} \ o_1 \ o_2)) \ b \\
(\text{lift\_lasagna\_cand\_x} \text{ true} \ o_1 (o_2 - o_1) \ l_1) \\
(\text{coerce\_max\_comm} \\
(\text{lift\_lasagna\_cand\_x} \text{ true} \ o_2 (o_1 - o_2) \ l_2)).
\end{align*}
Conclusion

Type theory is flexible

- Polymorphism
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
- No JMEQ
Conclusion

Type theory is flexible

- Polymorphism
- Mutually inductive types
- Dependent types
- Conversion rule
- No JMEQ
Conclusion

Type theory is flexible

- Polymorphism
- Mutually inductive types
- Dependent types
- Conversion rule
- No JMEQ

Future work

- Study variants, compare and choose the best
Conclusion

Type theory is flexible

- Polymorphism
- Mutually inductive types
- Dependent types
- Conversion rule
- No JMEQ

Future work

- Study variants, compare and choose the best
- application to CCA