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

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):

$$x, y, \{z\}_{x \oplus KP \oplus KM} \mapsto \{z \oplus y\}_{x \oplus KP \oplus KM}$$
$$x, y, \{z\}_{x \oplus KP \oplus KM} \mapsto \{z \oplus y\}_{x \oplus KM}$$

Formal models of cryptographic systems

- Protocols
- Security APIs

Xor is ubiquitous

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

$$x, y, \{z\}_{x \oplus KP \oplus KM} \mapsto \{z \oplus y\}_{x \oplus KP \oplus KM}$$
$$x, y, \{z\}_{x \oplus KP \oplus KM} \mapsto \{z \oplus y\}_{x \oplus KM}$$

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

General setting: quotiented first order-terms

We are given

- A type of terms $\mathcal{T}$ with constructors C_k:
  Inductive $\mathcal{T}$: Set :=

  $$
  \begin{aligned}
  &\mid C_1 : \mathcal{T} \\
  &\quad \vdots \\
  &\mid C_k : \ldots \to \mathcal{T} \ldots \to \mathcal{T} \ldots \to \mathcal{T} \\
  &\quad \vdots
  \end{aligned}
  $$

General setting: quotiented first order-terms

We are given

- A type of terms $\mathcal{T}$ with constructors C_k:
  Inductive $\mathcal{T}$: *Set* :=
  $$\begin{array}{l} \mid C_1 : \mathcal{T} \\ \quad \vdots \\ \mid C_k : \ldots \to \mathcal{T} \ldots \to \mathcal{T} \ldots \to \mathcal{T} \\ \quad \vdots \end{array}$$
- A congruence $\simeq : \mathcal{T} \to \mathcal{T} \to Prop$

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
    $\forall a, \ldots x_1, y_1, b, \ldots x_2, y_2, \ldots c,$
    $x_1 \simeq y_1 \rightarrow x_2 \simeq y_2 \rightarrow$
    $C_k \; a \ldots x_1 \; b \ldots y_1 \; c \simeq C_k \; a \ldots x_2 \; b \ldots y_2 \; c$

General setting: quotiented first order-terms

We are given

- A type of terms $\mathcal{T}$ with constructors C_k:
  Inductive $\mathcal{T}$: *Set* :=
  $| \ C_1 : \mathcal{T}$
  $\vdots$
  $| \ C_k : \ldots \to \mathcal{T} \ldots \to \mathcal{T} \ldots \to \mathcal{T}$
  $\vdots$
- A congruence $\simeq : \mathcal{T} \to \mathcal{T} \to Prop$
  - For each constructor C_k
    $\forall a, \ldots x_1, y_1, b, \ldots x_2, y_2, \ldots c,$
    $x_1 \simeq y_1 \to x_2 \simeq y_2 \to$
    $C_k\ a \ldots x_1\ b \ldots y_1\ c \simeq C_k\ a \ldots x_2\ b \ldots y_2\ c$
  - specific laws, e.g. $\forall xy, C_2\ x\ C_1\ y \simeq C_2\ y\ x$

General setting: quotiented first order-terms

We are given

- A type of terms $\mathcal{T}$ with constructors C_k:
  Inductive $\mathcal{T}$: *Set* :=
  $| \; C_1 : \mathcal{T}$
  $\vdots$
  $| \; C_k : \ldots \to \mathcal{T} \ldots \to \mathcal{T} \ldots \to \mathcal{T}$
  $\vdots$
- A congruence $\simeq \, : \mathcal{T} \to \mathcal{T} \to Prop$
  - For each constructor C_k
    $\forall a, \ldots x_1, y_1, b, \ldots x_2, y_2, \ldots c,$
    $x_1 \simeq y_1 \to x_2 \simeq y_2 \to$
    $C_k \; a \ldots x_1 \; b \ldots y_1 \; c \simeq C_k \; a \ldots x_2 \; b \ldots y_2 \; c$
  - specific laws, e.g. $\forall xy, C_2 \; x \; C_1 \; y \simeq C_2 \; y \; x$

General setting: quotiented first order-terms

We are given

- A type of terms $\mathcal{T}$ with constructors C_k:
  Inductive $\mathcal{T}$: *Set* :=
  $$\begin{array}{l} \mid C_1 : \mathcal{T} \\ \quad \vdots \\ \mid C_k : \ldots \to \mathcal{T} \ldots \to \mathcal{T} \ldots \to \mathcal{T} \\ \quad \vdots \end{array}$$
- A congruence $\simeq : \mathcal{T} \to \mathcal{T} \to Prop$
  - For each constructor C_k
    $\forall a, \ldots x_1, y_1, b, \ldots x_2, y_2, \ldots c,$
    $x_1 \simeq y_1 \to x_2 \simeq y_2 \to$
    $C_k\, a \ldots x_1\, b \ldots y_1\, c \simeq C_k\, a \ldots x_2\, b \ldots y_2\, c$
  - 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 $\mathcal{T}$

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

Cryptographic systems need more

Reasoning on such systems involves

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

$$
\begin{array}{lrcl}
\text{Commutativity:} & x \oplus y & \simeq & y \oplus x \\
\text{Associativity:} & (x \oplus y) \oplus z & \simeq & x \oplus (y \oplus z) \\
\text{Neutral element:} & x \oplus 0 & \simeq & x \\
\text{Involutivity:} & x \oplus x & \simeq & 0
\end{array}
$$

Cryptographic systems need more

Reasoning on such systems involves

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

$$\begin{array}{lrcl}
\text{Commutativity:} & x \oplus y & \simeq & y \oplus x \\
\text{Associativity:} & (x \oplus y) \oplus z & \simeq & x \oplus (y \oplus z) \\
\text{Neutral element:} & x \oplus 0 & \simeq & x \\
\text{Involutivity:} & x \oplus x & \simeq & 0
\end{array}$$

- 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 $\oplus$:

$$\begin{array}{lrcl}
\text{Commutativity:} & x \oplus y & \simeq & y \oplus x \\
\text{Associativity:} & (x \oplus y) \oplus z & \simeq & x \oplus (y \oplus z) \\
\text{Neutral element:} & x \oplus 0 & \simeq & x \\
\text{Involutivity:} & x \oplus x & \simeq & 0
\end{array}$$

- 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 $\oplus$:

$$\begin{array}{lrcl}
\text{Commutativity:} & x \oplus y & \simeq & y \oplus x \\
\text{Associativity:} & (x \oplus y) \oplus z & \simeq & x \oplus (y \oplus z) \\
\text{Neutral element:} & x \oplus 0 & \simeq & x \\
\text{Involutivity:} & x \oplus x & \simeq & 0
\end{array}$$

- 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 $\oplus$:

$$\begin{array}{lrcl}
\text{Commutativity:} & x \oplus y & \simeq & y \oplus x \\
\text{Associativity:} & (x \oplus y) \oplus z & \simeq & x \oplus (y \oplus z) \\
\text{Neutral element:} & x \oplus 0 & \simeq & x \\
\text{Involutivity:} & x \oplus x & \simeq & 0
\end{array}$$

- 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 $\oplus$:

$$\begin{array}{lrcl}
\text{Commutativity:} & x \oplus y & \simeq & y \oplus x \\
\text{Associativity:} & (x \oplus y) \oplus z & \simeq & x \oplus (y \oplus z) \\
\text{Neutral element:} & x \oplus 0 & \simeq & x \\
\text{Involutivity:} & x \oplus x & \simeq & 0
\end{array}$$

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

$$\begin{array}{ll}
x \preceq y & \text{if } x \simeq y \\
x \preceq t & \text{if } t \simeq x \oplus y_0 \ldots \oplus y_n \\
 & \text{and } x \not\preceq y_i \text{ for all } i,\ 0 \leq i \leq n
\end{array}$$

Cryptographic systems need more

Reasoning on such systems involves

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

$$\begin{array}{lrcl} \text{Commutativity:} & x \oplus y & \simeq & y \oplus x \\ \text{Associativity:} & (x \oplus y) \oplus z & \simeq & x \oplus (y \oplus z) \\ \text{Neutral element:} & x \oplus 0 & \simeq & x \\ \text{Involutivity:} & x \oplus x & \simeq & 0 \end{array}$$

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

$$\begin{array}{ll} x \preceq y & \text{if } x \simeq y \\ x \preceq t & \text{if } t \simeq x \oplus y_0 \ldots \oplus y_n \\ & \text{and } x \not\preceq y_i \text{ for all } i,\ 0 \leq i \leq n \end{array}$$

$\longrightarrow$ 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 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

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 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?

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

(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
  $=$ provide an accurate and informative typing to 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

(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: $t : \mathcal{T}$ transformed into $t' : \mathcal{T}'$
$\mathcal{T}'$ enriched version of $\mathcal{T}$,
trivial forgetful morphism $\mathcal{T}' \to \mathcal{T}$.

(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.

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

Lunch time!

Lasagnas reveal the truth

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

$\mathcal{T}$ as a lasagna

$\mathcal{T}$ as a lasagna

Inductive $\mathcal{T}$: Set :=
$| \; Zero \colon \mathcal{T}$
$| \; PC \colon public\_const \to \mathcal{T} \qquad | \; SC \colon secret\_const \to \mathcal{T}$
$| \; E \colon \mathcal{T} \to \mathcal{T} \to \mathcal{T}$
$| \; Xor \colon \mathcal{T} \to \mathcal{T} \to \mathcal{T}$
$| \; Hash \colon \mathcal{T} \to \mathcal{T} \to \mathcal{T}.$

$\mathcal{T}$ as a lasagna

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

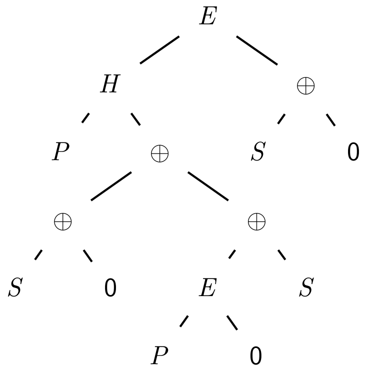
| *Zero*: $\mathcal{T}$

| *PC*: *public_const* $\rightarrow \mathcal{T}$ | *SC*: *secret_const* $\rightarrow \mathcal{T}$

| *E*: $\mathcal{T} \rightarrow \mathcal{T} \rightarrow \mathcal{T}$

| *Xor*: $\mathcal{T} \rightarrow \mathcal{T} \rightarrow \mathcal{T}$

| *Hash*: $\mathcal{T} \rightarrow \mathcal{T} \rightarrow \mathcal{T}$.



$\mathcal{T}$ as a lasagna

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

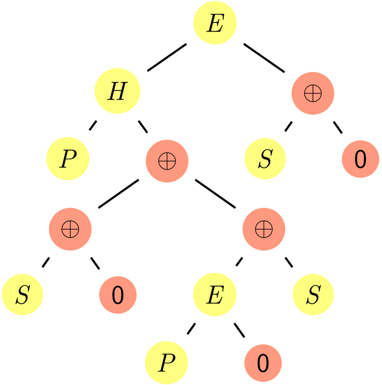
| *Zero*: $\mathcal{T}$

| *PC*: *public_const* $\rightarrow \mathcal{T}$ | *SC*: *secret_const* $\rightarrow \mathcal{T}$

| *E*: $\mathcal{T} \rightarrow \mathcal{T} \rightarrow \mathcal{T}$

| *Xor*: $\mathcal{T} \rightarrow \mathcal{T} \rightarrow \mathcal{T}$

| *Hash*: $\mathcal{T} \rightarrow \mathcal{T} \rightarrow \mathcal{T}$.



Outline

Decomposing $\mathcal{T}$

Inductive $\mathcal{T}_x$:Set :=
| $X\_Zero : \mathcal{T}_x$
| $X\_Xor : \mathcal{T}_x \rightarrow \mathcal{T}_x \rightarrow \mathcal{T}_x$

Inductive $\mathcal{T}_n$: Set :=
| $NX\_PC : public\_const \rightarrow \mathcal{T}_n$
| $NX\_SC : secret\_const \rightarrow \mathcal{T}_n$
| $NX\_E : \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n$
| $NX\_Hash : \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n$

Decomposing $\mathcal{T}$

Variable A : Set.

Inductive $\mathcal{T}_x$:Set :=
| X_Zero : $\mathcal{T}_x$
| X_Xor : $\mathcal{T}_x \to \mathcal{T}_x \to \mathcal{T}_x$
| X_ns : $A \to \mathcal{T}_x$

Inductive $\mathcal{T}_n$: Set :=
| NX_PC : $public\_const \to \mathcal{T}_n$
| NX_SC : $secret\_const \to \mathcal{T}_n$
| NX_E : $\mathcal{T}_n \to \mathcal{T}_n \to \mathcal{T}_n$
| NX_Hash : $\mathcal{T}_n \to \mathcal{T}_n \to \mathcal{T}_n$
| NX_sum : $A \to \mathcal{T}_n$

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

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.

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 $0 \mapsto X\_Zero$, $Xor \mapsto X\_Xor$, $PC \mapsto NX\_PC$, 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

- $\mathcal{T}_n(A)$ 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;
- $list(A)$ 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 $\underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}}$ 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;
- $list(A)$ 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 $\underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}}$ 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;
- $list(A)$ is sortable.

Outline

Lifting lasagna

$\mathcal{L}_x\, k \stackrel{\text{def}}{=\!=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}}$ for k large enough.

Lifting lasagna

$$\mathcal{L}_x\, k \stackrel{\text{def}}{=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}} \text{ for } k \text{ large enough.}$$

- What is k?

Lifting lasagna

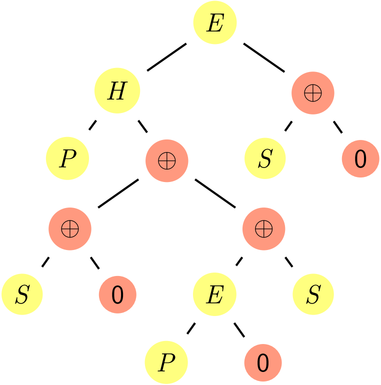
$$\mathcal{L}_x\,k \stackrel{\text{def}}{=\!=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}} \text{ for } k \text{ large enough.}$$

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

Lifting lasagna

$\mathcal{L}_x\, k \stackrel{\text{def}}{=\!=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}}$ for k large enough.

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



Lifting lasagna

$$\mathcal{L}_x\, k \stackrel{\mathrm{def}}{=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}} \text{ for } k \text{ large enough.}$$

- 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

$\mathcal{L}_x\, k \stackrel{\text{def}}{=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}}$ 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: {le n m} + {le m n}

Lifting lasagna

$$\mathcal{L}_x\, k \stackrel{\text{def}}{=\!=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}} \text{ for } k \text{ large enough.}$$

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

Take the max

- Standard solution: {le n m} + {le m n}
  - interactive definition, large proof term

Lifting lasagna

$$\mathcal{L}_x\, k \stackrel{\text{def}}{=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}} \text{ for } k \text{ large enough.}$$

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

Take the max

- Standard solution: {le n m} + {le m n}
  - interactive definition, large proof term
  - heavy encoding of $m - n$ or $n - m$

Lifting lasagna

$$\mathcal{L}_x\, k \stackrel{\text{def}}{=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}} \text{ for } k \text{ large enough.}$$

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

Take the max

- Standard solution: {le n m} + {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 $\mathcal{L}_x\,(\max\, n\, m)$

Lifting lasagna

$$\mathcal{L}_x\, k \stackrel{\text{def}}{=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}} \text{ for } k \text{ large enough.}$$

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

Take the max

- Standard solution: {le n m} + {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: $\max n\, m \stackrel{\text{def}}{=} m + (n - m)$

Lifting lasagna

$\mathcal{L}_x\, k \stackrel{\mathrm{def}}{=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}}$ 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: {le n m} + {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 $n\, m \stackrel{\mathrm{def}}{=} m + (n - m)$
  - $lift_x : \mathcal{L}_x\, k \to \mathcal{L}_x\,(k+d)$, $lift_n : \mathcal{L}_n\, k \to \mathcal{L}_n\,(k+d)$

Lifting lasagna

$$\mathcal{L}_x\, k \stackrel{\text{def}}{=} \underbrace{\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))}_{k \text{ layers}} \text{ for } k \text{ large enough.}$$

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

Take the max

- Standard solution: {le n m} + {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: $\max\, n\, m \stackrel{\text{def}}{=} m + (n - m)$
  - $lift_x : \mathcal{L}_x\, k \to \mathcal{L}_x\,(k + d)$, $lift_n : \mathcal{L}_n\, k \to \mathcal{L}_n\,(k + d)$
  - No need to proof that max is the max.

Outline

Internalizing alternation

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. $lift_x$ and $lift_n$

Control them using alternating natural numbers

Inductive alt_{even}: Set :=
  | 0_e: alt_{even}
  | $S_{o \to e}$: $alt_{odd} \to alt_{even}$
with alt_{odd}: Set :=
  | $S_{e \to o}$: $alt_{even} \to alt_{odd}$

Outline

Forbid fake inclusions

Forbid fake inclusions

$$
\begin{array}{l}
\text{Inductive } \mathcal{T}_x : \quad Set := \\
\quad \mid X\_Zero : \quad \mathcal{T}_x \\
\quad \mid X\_ns : \quad A \rightarrow \mathcal{T}_x \\
\quad \mid X\_Xor : \quad \mathcal{T}_x \rightarrow \mathcal{T}_x \rightarrow \mathcal{T}_x
\end{array}
$$

$$
\begin{array}{l}
\text{Inductive } \mathcal{T}_n : \quad Set := \\
\quad \mid NX\_PC : \quad public\_const \rightarrow \mathcal{T}_n \\
\quad \mid NX\_SC : \quad secret\_const \rightarrow \mathcal{T}_n \\
\quad \mid NX\_sum : \quad A \rightarrow \mathcal{T}_n \\
\quad \mid NX\_E : \quad \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n \\
\quad \mid NX\_Hash : \quad \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n
\end{array}
$$

Forbid fake inclusions

Inductive $\mathcal{T}_x$: $\;Set$:=
| 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 $\mathcal{T}_n$: $\;Set$:=
| 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 : $\mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n$

$X\_ns\ (NX\_sum\ (\ X\_ns\ (NX\_sum\ (\ldots))))$

Forbid fake inclusions

Inductive $\mathcal{T}_x$: $bool \rightarrow Set$:=
| $X\_Zero : \forall\ b,\ \mathcal{T}_x\ b$
| $X\_ns : \forall\ b,\ Is\_true\ b \rightarrow A \rightarrow \mathcal{T}_x\ b$
| $X\_Xor : \forall\ b,\ \mathcal{T}_x\ true \rightarrow \mathcal{T}_x\ true \rightarrow \mathcal{T}_x\ b$

Inductive $\mathcal{T}_n$: $bool \rightarrow Set$:=
| $NX\_PC : \forall\ b,\ public\_const \rightarrow \mathcal{T}_n\ b$
| $NX\_SC : \forall\ b,\ secret\_const \rightarrow \mathcal{T}_n\ b$
| $NX\_sum : \forall\ b,\ Is\_true\ b \rightarrow A \rightarrow \mathcal{T}_n\ b$
| $NX\_E : \forall\ b,\ \mathcal{T}_n\ true \rightarrow \mathcal{T}_n\ true \rightarrow \mathcal{T}_n\ b$
| $NX\_Hash : \forall\ b,\ \mathcal{T}_n\ true \rightarrow \mathcal{T}_n\ true \rightarrow \mathcal{T}_n\ b$

$X\_ns\ (NX\_sum\ (\ X\_ns\ (NX\_sum\ (\ldots))))$

Outline

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 *lift_lasagna_x* e_1 e_2 $\{struct\ e_1\}$:
$\mathcal{L}_x\ e_1 \to \mathcal{L}_x\ (e_1 + e_2)$:=
match e_1 return $\mathcal{L}_x\ e_1 \to \mathcal{L}_x\ (e_1 + e_2)$ with
| $0_e \Rightarrow$ fun *emp* $\Rightarrow$ match *emp* with end
| $S_{o\to e}\ o_1 \Rightarrow map_x\ (lift\_lasagna\_n\ o_1\ e_2)\ false$
end
with *lift_lasagna_n* o_1 e_2 $\{struct\ o_1\}$:
$\mathcal{L}_n\ o_1 \to \mathcal{L}_n\ (o_1 + e_2)$:=
match o_1 return $\mathcal{L}_n\ o_1 \to \mathcal{L}_n\ (o_1 + e_2)$ with
| $S_{e\to o}\ e_1 \Rightarrow map_n\ (lift\_lasagna\_x\ e_1\ e_2)\ false$
end.

Outline

Conversion rule

Conversion rule

Used everywhere

Conversion rule

Used everywhere

Definition *bin_xor*

$(bin : \forall\ A\ b, \mathcal{T}_x\ A\ true \rightarrow \mathcal{T}_x\ A\ true \rightarrow \mathcal{T}_x\ A\ b)\ o_1\ o_2\ b$
$(l_1 : lasagna\_cand\_x\ o_1\ true)$
$(l_2 : lasagna\_cand\_x\ o_2\ true)$:
$lasagna\_cand\_x\ (max\_oo\ o_1\ o_2)\ b :=$

$bin\ (\mathcal{L}_n\ (max\_oo\ o_1\ o_2))\ b$
$(lift\_lasagna\_cand\_x\ true\ o_1\ (o_2 - o_1)\ l_1)$
$(coerce\_max\_comm$
$(lift\_lasagna\_cand\_x\ true\ o_2\ (o_1 - o_2)\ l_2))$.

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