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

Jean-François Monin[1] and Judicaël Courant[1]

VERIMAG - Centre quation, 2 avenue de Vignate, F-38610 Gières, France
{jean-francois.monin|judicael.courant}@imag.fr
http://www-verimag.imag.fr/~monin|courant/

Abstract

We study a normalization function in an algebra of terms quotiented by an associative, commutative and involutive operator (logical xor). This study is motivated by the formal verification of cryptographic systems, where a normalization function for xor-terms turns out to play a key role. Such a function is easy to define using general recursion. However, as it is to be used in a type theoretic proof assistant, we also need a proof of its termination. Instead of using a clever mixture of various rewriting orderings, we follow an approach involving the power of Type Theory with dependent types. The results are to be applied in the proof of the security API described in [CM06].

1 INTRODUCTION

In the course of the formal verification of cryptographic systems using symbolic approaches, one deals with algebras of terms whose constructors include $\oplus$, denoting the binary bitwise exclusive or and O, denoting a bitstring consisting only of zeros. Bitwise exclusive or is often used in cryptographic systems and many (potential or effective) attacks are based on its algebraic properties [YAB$^+$05, Bon04, CKRT05, CLC03].

Dealing with the congruence generated by the usual arithmetic laws on $\oplus$ and O is therefore necessary in order to successfully verify these systems: in the following we consider an algebra of terms $\mathcal{T}$ built up using a number of constructors, where two of them, denoted by $\oplus$ and O, enjoy the following algebraic properties.

Commutativity:	$x \oplus y \simeq y \oplus x$	(1)
Associativity:	$(x \oplus y) \oplus z \simeq x \oplus (y \oplus z)$	(2)
Neutral element:	$x \oplus \mathsf{O} \simeq x$	(3)
Involutivity:	$x \oplus x \simeq \mathsf{O}$	(4)

Formally, $\simeq$ denotes the least congruence generated by equations (1) to (4). In order to reason about terms of $\mathcal{T}$ up to $\simeq$, a standard technique is to define a canonicalization function over $\mathcal{T}$. One also actually needs such a function to give minimal terms with respect to simplification as one also needs a subterm relation

$\preceq$ which takes into account equalities such as $u \simeq u \oplus x \oplus x$:

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

Turning equations (1) to (4) into a convergent and strongly normalizing AC-rewriting system is quite straightforward. Therefore, the existence of a normalization function can be proven easily on the paper. Moreover, in any decent programming language, writing the corresponding normalization function on $\mathcal{T}$ is quite easy, using general recursion.

However, formally giving such a normalization function in Type Theory and formally proving its correctness is much more challenging. The first problem is to ensure termination. In Type Theory, all functions are total by construction. It means that general fixpoints are not allowed for defining functions. The prominent ingredient provided by Type Theory, in order to achieve this fundamental property while keeping a good expressing power, is the limitation of recursion to higher-order primitive recursion, extended to structural recursion over all inductively defined types. In simple cases, one can use the combinators automatically provided by the theory and support tools for natural numbers, lists and all user-defined inductive types. In the modern version of Coq, which implements a version of Type Theory called the Calculus of Inductive Constructions (CIC), the user can equivalently provide, for each recursively defined function, an argument which structurally strictly decreases at each recursive call. A special and very important case is well-founded recursion: the decreasing argument is a proof, formalized as an inductive object, witnessing that a given value is accessible for some binary relation. It complicated cases, the latter approach is by far more convenient. But it can be seen that there is no miracle: while totality in ensured by type-checking (and the fact that well-typed functions are strongly normalizing), designing and providing the right types and arguments are in the responsibility of the user. Standard libraries on well-founded relations are helpful here. Besides, one can consider additional tools or methodology such as those developed by Bertot and Balaa, or Bove and Capretta [BB00, BC05].

In the case of the canonicalization function we have in mind, using standard rewriting arguments is surprisingly difficult in a proof assistant such as Coq [The05, BC04].

- Although some theoretical works address the addition of rewriting to the Calculus of Constructions [Bla01], these works are yet to be implemented.

- Some works provide ways to define tactics for reasonning over associative-commutative theories [AN00], but they only provide ways to normalize given terms, not to define a normalization function.

We therefore tried to define our own specific rewriting relation corresponding to the defining equations of $\simeq$, but found this approach really costly:

- A well-founded ordering had to be given. As no rpo or lpo ordering library was available in Coq, we used the lexicographic combination of a partial ordering $\leq_1$ with a total ordering $\leq_2$, where $\leq_1$ is a polynomial ordering, and $\leq_2$ is a lexicographic ordering. Although $\leq_2$ is not well-founded, the set of terms having a given weight for the polynomial defining $\leq_1$ is finite, therefore we could prove in Coq the lexicographic combination of $\leq_1$ and $\leq_2$ to be finite.

- Then we defined a rewriting relation $\triangleright$. The difficult part here is to take into account commutativity and associativity. In order to avoid AC-matching issues, we decided to throw in associativity and to add commutativity as a conditional rule ($x \oplus y$ would rewrite to $y \oplus x$ if and only if x is smaller than y). Moreover, we had to complete our rewriting system in order to close critical pairs such as $x \oplus x \oplus y$, which could be rewritten to y or to $x \oplus (x \oplus y)$.

- A normalization function has to be given. As mentionned above, the definition of such a function using well-founded induction in Coq is uneasy. Therefore we stopped there and used an other approach instead.

- Once this would be done, we would still have to prove that the transitive closure of our rewriting relation is irreflexive, that our normalization function is sound with respect to it, and that is computes normal forms. Essentially, the main results to prove here would be $\forall t\ t \not\triangleright^+ t$, $\forall t\ t \triangleright^\star norm(t)$ and $\forall t_1 \forall t_2\ \ t_1 \triangleright t_2 \Rightarrow norm(t_1) = norm(t_2)$.

Instead we experimented a new approach involving typical features of Type Theory. The intuition behind our approach is very simple. In a first stage, the term to be normalized is layered in such a way that each level is built up from terms belonging to the previous level. These levels alternate between layers built up using only $\oplus$ constructors and layers built up using only other constructors, as lasagnas alternate between pasta-only layers and sauce layers (mixed up to your taste of tomato, meat, and cheese – in fact anything but pasta). In a second stage, layers are normalized bottom-up. Normalizing a $\oplus$-layer roughly boils down to sorting, while normalization of a non-$\oplus$-layer is just identity.

Basically, the second stage is not very difficult, though some pitfalls have to be avoided. Surprisingly, the first stage, which is essentially nothing else than type translation and renaming, requires more work than expected. In the whole development, we need the full power of programming with dependent types: each layer owns its specific ordering relation, which depends on the ordered structure of the previous layer.

The approach we describe in this paper was designed and implemented using the Coq proof assistant. Its results are to be applied in the proof of security properties of an API described in [CM06].

2 FORMALIZATION

2.1 Splitting the type of terms

Let $\{\oplus, \mathsf{O}\} \uplus \mathcal{C}$ be the set of constructors of $\mathcal{T}$. For instance, in our case, we have $\mathcal{C} = \{\mathtt{PC}, \mathtt{SC}, \mathtt{E}, \mathtt{Hash}\}$ with

$$\mathtt{PC} : \mathtt{public\_const} \to \mathcal{T} \qquad \mathtt{E} : \mathcal{T} \to \mathcal{T} \to \mathcal{T}$$
$$\mathtt{SC} : \mathtt{secret\_const} \to \mathcal{T} \qquad \mathtt{Hash} : \mathcal{T} \to \mathcal{T} \to \mathcal{T}$$

where `public_const` and `secret_const` are suitable enumerated types.

As explained in the introduction, we want to split a $\mathcal{T}$-term into layers. Moreover, a number of functions and lemmas have to be stated and proved on each layer. For obvious modularity reasons, it is better to handle each layer separately. Each layer provides a datatype, comparison and sorting functions on this type as well as correctness lemmas. Intuitively, it could be seen as a module in the sense of Harper, Lillibridge and Leroy [HL94, Ler00], or better: a functor, because each layer relies on the interface of the previous layer. HLL modules have been adapted to the Calculus of Indiuctive Constructions and are implemented in Coq. But our case is out of their scope, because here the number of layers is a dynamic notion which depends on a piece of data, namely a term to be normalized. Therefore we stick to the features of basic CIC, which are dependent, polymorphic and inductive types.

In a first stage, we introduce two polymorphic inductive types $\mathcal{T}_x(\alpha)$ and $\mathcal{T}_n(\alpha)$ respectively called the pasta layer type and the sauce layer type. The constructors of $\mathcal{T}_x(\alpha)$ are (copies of) $\oplus$ and O while the constructors of $\mathcal{T}_n(\alpha)$ are (copies of) those belonging to $\mathcal{C}$. Moreover, $\mathcal{T}_x(\alpha)$ (respectively $\mathcal{T}_n(\alpha)$) has an additional constructor $I_x : \alpha \to \mathcal{T}_x(\alpha)$ (respectively $I_n : \alpha \to \mathcal{T}_n(\alpha)$).

It is then clear that any term t in $\mathcal{T}$ can be recasted into either the type $\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))$ or the type $\mathcal{T}_n(\mathcal{T}_x(\mathcal{T}_n(\ldots(\emptyset))))$, according to the top constructor of t.

In a second stage, normalizing t can be defined as bottom-up sorting in the following way. We say that a type X is *sortable* if it is equipped with a decidable equality and a decidable total irreflexive and transitive relation $<$. Equivalently, we could take a decidable total ordering but the above choice turns out to be more convenient.

If X is sortable,

- $\mathcal{T}_n(X)$ is sortable;
- the multiset of X-leaves of any inhabitant t of $\mathcal{T}_x(X)$ can be sorted (with deletion of duplicates) into a list $N_X(t)$, such that $t_1 \simeq t_2$ iff $N_X(t_1)$ is syntactically equal to $N_X(t_2)$;
- $\mathtt{list}(X)$ is sortable (*i.e.* can be equipped with suitable equality and comparison relation).

Formally, we define the type *sortable* as a dependent record made of a *Set* X, two binary relations $=$ and $<$ on X, and of proofs that $=$ and $<$ have the properties just mentionned. If A is a *sortable*, the underlying *Set* is denoted by $|A|$.

Let A be a *sortable* structure, then by the above remarks we can construct two sortable structures:

- $\mathcal{R}_n(A)$, such that $|\mathcal{R}_n(A)| = \mathcal{T}_n(|A|)$ and
- $\mathcal{R}_x(A)$, such that $|\mathcal{R}_x(A)| = \texttt{list}(|A|)$,

and moreover, we have a local normalizing function from $|\mathcal{T}_x(A)|$ to $|\mathcal{R}_x(A)|$.

We actually go one step further. It is easy to define a function which takes as input a *sortable* A, a decidable property P on $|A|$, and yields as its output another *sortable* B such that $|B| = \{x : |A| \mid P\,x\}$. We apply this mechanism in order to get sorted-by-construction lists. That is, we can replace $\mathcal{R}_x$ with $\mathcal{S}_x$, such that

- $|\mathcal{S}_x(A)| = \{l : \texttt{list}(|A|) \mid \textit{sorted}\, l\})$.

We are then ready for the third stage: normalizing any term of type $\ldots\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\emptyset))))$ to a term of type $\ldots\mathcal{R}_x(\mathcal{R}_n(\mathcal{R}_x(\emptyset))))$, or better $\ldots\mathcal{S}_x(\mathcal{R}_n(\mathcal{S}_x(\emptyset))))$, by induction on the number of layers. Note that thanks to polymorphism, we deal with each layer in a pleasant modular way.

We now have to handle types such as $\ldots\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\emptyset))))$ in a formal way.

2.2 Formalized stratified types

2.2.1 *Defining pasta and sauce layers*

A layer is said trivial when it consists only of a term $I_x(a)$ or $I_n(a)$. In order to unfold sequences of $\oplus$, we want to avoid artificial separation of $\oplus$ layers like $x \oplus I_x(I_n(y \oplus z))$. Therefore, we want to be able to forbid constructions like $I_x(I_n(a))$. Hence we distinguish between potentially trivial layers and non-trivial layers, by adding to the pasta layer type $\mathcal{T}_x$ a boolean parameter telling us whether trivial layers are included:

Section *sec_x*.
Variable A : *Set*.
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*
.

Definition I_x := *X_ns true I*.

Likewise the inductive sauce layer type $\mathcal{T}_n$ (non-xor terms) is parameterized by a boolean telling whether trivial layers are included.

Section *sec_nx*.

Variable A : *Set*.
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*
.
Definition I_n := *NX_sum true I*.

2.2.2 Lasagnas

The types of lasagnas are defined using mutual recursion on the height.

Fixpoint $\mathcal{L}_{\mathcal{T}x}$ (e:*alte*) : *Set* :=
  match e with
  | $0_e \Rightarrow$ *empty*
  | $S_{o\rightarrow e}$ $o \Rightarrow \mathcal{T}_x$ ($\mathcal{L}_{\mathcal{T}n}$ o) *false*
  end
with $\mathcal{L}_{\mathcal{T}n}$ (o:*alto*) : *Set* :=
  match o with
  | $S_{e\rightarrow o}$ $e \Rightarrow \mathcal{T}_n$ ($\mathcal{L}_{\mathcal{T}x}$ e) *false*
  end.

The pair of types for normalized terms is similarly defined, but their realm is *sortable* instead of *Set*.

Fixpoint $\mathcal{L}_{\mathcal{S}x}$ (e:*alte*) : *sortable* :=
  match e with
  | $0_e \Rightarrow \mathcal{R}_{\emptyset}$
  | $S_{o\rightarrow e}$ $o \Rightarrow \mathcal{S}_x$ ($\mathcal{L}_{\mathcal{S}n}$ o)
  end
with $\mathcal{L}_{\mathcal{S}n}$ (o:*alto*) : *sortable* :=
  match o with
  | $S_{e\rightarrow o}$ $e \Rightarrow \mathcal{R}_{n}$ ($\mathcal{L}_{\mathcal{S}x}$ e) *false*
  end.

2.2.3 Maps over lasagnas

Given a function f from $A : Set$ to $B : Set$, one can easily define a map_x (resp. map_n) function lifting f to functions from $\mathcal{T}_x(A)$ to $\mathcal{T}_x(B)$ (resp. from $\mathcal{T}_n(A)$ to $\mathcal{T}_n(B)$).

Moreover, given an evaluation function $f : A \rightarrow \mathcal{T}$, one can extend it to the domain $\mathcal{T}_x(A)$ (resp. $\mathcal{T}_n(A)$) by interpreting copies of $\oplus$ and O (resp. of constructors belonging to $\mathcal{C}$) as the corresponding constructors of $\mathcal{T}$ and I_x (resp. I_n) as the identity over $\mathcal{T}$.

2.2.4 Stacking layers

Building a stack of k layers now essentially amounts to building the type $(\mathcal{T}_x \circ \mathcal{T}_n)^{k/2}(\emptyset)$ or $\mathcal{T}_n(\mathcal{T}_x \circ \mathcal{T}_n)^{k/2}(\emptyset)$, depending on the parity of k. In a more type-theoretic fashion, we define two mutually inductive types alt_{even} and alt_{odd}, denoting even and odd natural numbers respectively: the constructors of alt_{even} are 0_e and $S_{o\to e}$, the successor function from odd to even numbers, whereas alt_{odd} has only one constructor, $S_{e\to o}$, the successor function from even to odd numbers. We also define *parity* as either P_e or P_o. One can then build the function

$$\begin{array}{rrcl} alt\_of\_parity: & parity & \to & Set \\ & P_e & \mapsto & alt_{even} \\ & P_o & \mapsto & alt_{odd} \end{array}$$

2.3 Stratifying a term

2.3.1 Lifting a lasagna

The intuitive idea we have about lasagnas is somewhat misleading, because the number of pasta and sauce layers is uniform in a whole lasagna dish, while the number of layers of subterms which are rooted at the same depth of a given term are different in the general case. However, any lasagna of height n can be lifted to a lasagna of height $n+e$, where e is even, because the empty type at the bottom of types such as $\mathcal{T}_x(\mathcal{T}_n(\mathcal{T}_x(\ldots(\emptyset))))$ can be replaced with any type. Formally, the lifting is defined by structural mutual induction as follows, thanks to map combinators.

Fixpoint $lift\_lasagna\_x\ e_1\ e_2\ \{struct\ e_1\} : \mathcal{L}_{\mathcal{T}x}\ e_1 \to \mathcal{L}_{\mathcal{T}x}\ (e_1 + e_2)$:=
  match e_1 return $\mathcal{L}_{\mathcal{T}x}\ e_1 \to \mathcal{L}_{\mathcal{T}x}\ (e_1 + e_2)$ with
  $\mid 0_e \Rightarrow \lambda\ emp \Rightarrow$ match emp with end
  $\mid S_{o\to e}\ o_1 \Rightarrow map_x\ (lift\_lasagna\_n\ o_1\ e_2)\ false$
  end
with $lift\_lasagna\_n\ o_1\ e_2\ \{struct\ o_1\} : \mathcal{L}_{\mathcal{T}n}\ o_1 \to \mathcal{L}_{\mathcal{T}n}\ (o_1 + e_2)$:=
  match o_1 return $\mathcal{L}_{\mathcal{T}n}\ o_1 \to \mathcal{L}_{\mathcal{T}n}\ (o_1 + e_2)$ with
  $\mid S_{e\to o}\ e_1 \Rightarrow map_n\ (lift\_lasagna\_x\ e_1\ e_2)\ false$
  end.

2.3.2 Counting layers of a $\mathcal{T}$-term

Given a $\mathcal{T}$-term t, the type of the corresponding lasagna depends on the number $l(t)$ of its layers, which has to be computed first.

At first sight, we may try to escape the problem by computing a number $u(t)$ which is known to be greater, or equal to, $l(t)$ (a suitable u is the height). However we would then have to handle proofs that the proposed number $u(t)$ does provide an upper bound on $l(t)$. Such proofs have to be constructive, because they provide a bound on the number of recursive calls in the computation of the layering of a

$\mathcal{T}$-term. Then they embark the difference between $u(t)$ and $l(t)$, in a more or less hidden way. So it is unclear that $u(t)$ would really help us to simplify definitions, and we chose to stick to an accurate computation of $l(t)$ as follows.

The lifting functions explained in section 2.3.1 are basically used in the following way. We define the maximum of two natural numbers n and m as $n-m+m$. It is easy to check that this operation is commutative, hence the lasagnas of two immediate subterms of a $\mathcal{T}$-term can be lifted to lasagnas of the same height.

A further difficulty is that the arguments of a constructor occurrence in t are heterogeneous, i.e. some of them can be $\oplus$ and the others can be in $\mathcal{C}$. We then may use appropriate injections I_x or I_n. However, recall that their use is controlled (see section 2.2.1): they can be used only at the borderline between two different layers.

The trick is that, in general, we do not compute the lasagna of height n of a given term, that is, a $\mathcal{T}_x(X,\texttt{false})$ or a $\mathcal{T}_n(X,\texttt{false})$, where X is a lasagna of the opposite kind and of height $n-1$ but only a *lasagna candidate of height* $n-1$, that is, a function which yields a $\mathcal{T}_x(X,b)$ or a $\mathcal{T}_n(X,b)$ for any Boolean b.

Similarly, the definition of the height for a lasagna candidate (called *alt_allpar_of_term*) depends on a given parity p.

Definition *inj_odd_parity p* : *alto* $\rightarrow$ *alt_of_parity p* :=
  match p return *alto* $\rightarrow$ *alt_of_parity p* with
  $\mid P_e \Rightarrow S_{o\rightarrow e}$
  $\mid P_o \Rightarrow \lambda\, o \Rightarrow o$
  end.

Similarly for inj_even_parity

Fixpoint *alt_allpar_of_term* (t:$\mathcal{T}$) : $\forall$ p, *alt_of_parity p* :=
 match t return $\forall$ p, *alt_of_parity p* with
 $\mid$ *Zero* $\Rightarrow \lambda\, p \Rightarrow$ *inj_odd_parity p* ($S_{e\rightarrow o}$ 0_e)
 $\mid$ *Xor x y* $\Rightarrow$
  let o_1 := *alt_allpar_of_term x* P_o in
  let o_2 := *alt_allpar_of_term y* P_o in
  $\lambda\, p \Rightarrow$ *inj_odd_parity p* (*max_oo* o_1 o_2)

 $\mid$ *PC x* $\Rightarrow \lambda\, p \Rightarrow$ *inj_even_parity p* 0_e
 $\mid$ *E x y* $\Rightarrow$
  let e_1 := *alt_allpar_of_term x* P_e in
  let e_2 := *alt_allpar_of_term y* P_e in
  $\lambda\, p \Rightarrow$ *inj_even_parity p* (*max_ee* e_1 e_2)
 [Similarly for other constructors]
 end.

The lifting functions of section 2.3.1 are easily generalized to lasagna candidates.

2.3.3 Computing the lasagna

The main recursive function computes a true lasagna candidate. In other words, the type of its result depends on the desired parity.

Definition $kind\_lasagna\_cand\_of\_term\ (t{:}\mathcal{T})\ (p{:}\ parity) : Set :=$
match p with
$|\ P_e \Rightarrow lasagna\_cand\_n\ (alt\_allpar\_of\_term\ t\ P_e)\ true$
$|\ P_o \Rightarrow lasagna\_cand\_x\ (alt\_allpar\_of\_term\ t\ P_o)\ true$
end.

Its body introduces injections as required. Here is its definition.

Fixpoint $lasagna\_cand\_of\_term\ (t{:}\mathcal{T}) :$
$\forall\ p,\ kind\_lasagna\_cand\_of\_term\ t\ p :=$
match t return $\forall\ p,\ kind\_lasagna\_cand\_of\_term\ t\ p$ with
$|\ Zero \Rightarrow$
$\lambda\ p \Rightarrow$ match p return $kind\_lasagna\_cand\_of\_term\ Zero\ p$ with
$|\ P_e \Rightarrow I_n\ \ (X\_Zero\ false)$
$|\ P_o \Rightarrow X\_Zero\ true$
end
$|\ Xor\ t_1\ t_2 \Rightarrow$
let $l_1 := lasagna\_cand\_of\_term\ t_1\ P_o$ in
let $l_2 := lasagna\_cand\_of\_term\ t_2\ P_o$ in
$\lambda\ p \Rightarrow$ match p return $kind\_lasagna\_cand\_of\_term\ (Xor\ t_1\ t_2)\ p$ with
$|\ P_e \Rightarrow I_n\ \ (bin\_xor\ X\_Xor\ l_1\ l_2)$
$|\ P_o \Rightarrow bin\_xor\ X\_Xor\ l_1\ l_2$
end

$|\ PC\ x \Rightarrow$
[similarly for constructors in $\mathcal{C}$].

The above definition requires a function called *bin_xor* which maps a constructor of $\mathcal{T}_x$ to an operation on lasagna candidates of arbitrary height. This is the place where lifting is used. Note the essential use of the conversion rule in its typing.

Definition bin_xor
$(bin : \forall\ A\ b,\ \mathcal{T}_x\ A\ true \to \mathcal{T}_x\ A\ true \to \mathcal{T}_x\ A\ b)\ o_1\ o_2\ b$
$(l_1 : lasagna\_cand\_x\ o_1\ true)\ (l_2 : lasagna\_cand\_x\ o_2\ true) :$
$lasagna\_cand\_x\ (max\_oo\ o_1\ o_2)\ b :=$
$bin\ (\mathcal{L}_{\mathcal{T}n}\ (max\_oo\ o_1\ o_2))\ b$
$(lift\_lasagna\_cand\_x\ true\ o_1\ (o_2 - o_1)\ l_1)$
$(coerce\_max\_comm\ (lift\_lasagna\_cand\_x\ true\ o_2\ (o_1 - o_2)\ l_2)).$

Finally, the function *lasagna_of_term* is defined on top of *lasagna_cand_of_term*. In contrast with the latter, we force the parity to depend on the constructor at the root:

Definition $alt\_of\_term\ t := alt\_allpar\_of\_term\ t\ (parity\_of\_term\ t).$

Definition $lasagna\_of\_parity\ p : alt\_of\_parity\ p \rightarrow Set$:=
  match p return $alt\_of\_parity\ p \rightarrow Set$ with
  | $P_e \Rightarrow \mathcal{L}_{\mathcal{T}x}$
  | $P_o \Rightarrow \mathcal{L}_{\mathcal{T}n}$
  end.

Definition $lasagna\_of\_term\ (t{:}\mathcal{T})$:
  $lasagna\_of\_parity\ (parity\_of\_term\ t)\ (alt\_of\_term\ t)$:=
  match t return $lasagna\_of\_parity\ (parity\_of\_term\ t)\ (alt\_of\_term\ t)$ with
  | $Zero \Rightarrow X\_Zero\ false$
  | $Xor\ t_1\ t_2 \Rightarrow$
    let l_1 := $lasagna\_cand\_of\_term\ t_1\ P_o$ in
    let l_2 := $lasagna\_cand\_of\_term\ t_2\ P_o$ in
    $bin\_xor\ X\_Xor\ l_1\ l_2$
  | $PC\ x \Rightarrow NX\_PC\ false\ x$
  [similarly for constructors in $\mathcal{C}$].

2.4 Normalizing

We define a pair of normalization functions $N_x : \forall\, e, \mathcal{L}_{\mathcal{T}x}\, e \rightarrow |\mathcal{L}_{\mathcal{R}x}\, e|$ and $N_n : \forall\, o, \mathcal{L}_{\mathcal{T}n}\, o \rightarrow |\mathcal{L}_{\mathcal{R}n}\, o|$. Basically, the latter does essentially nothing, while the core of the former is
$\lambda x.\ fold\_insert\ (map\_xor\ (N_n\ o)\ false\, x)\ [\,]$.

However, things are a little more complicated: nomalizing a list may produce a one-element list, which behaves exactly as I_x, as explained in 2.2.1. In order to remove such fake layers, we replace the type $\texttt{list}(\alpha)$ in $\mathcal{R}_x$ with a type $\texttt{n1\_list}(\alpha)$ of lists having either no element, or at least two elements. Then, when sorting an inhabitant of $\mathcal{T}_x(|A|)$ the normalization function for lasagnas returns a value in a type with two options: either a $\texttt{n1\_list}(|A|)$, or an element of $|A|$ lifted as an element of $\mathcal{R}_a(\mathcal{S}_x(|A|))$.

Using this lifting requires some work: we have to show that it is monotonic, and that the various map functionnals preserve monotony.

In summary, the whole normalization function typechecks, and syntactic equality in the type of its output corresponds exactly to the equivalence of its input wrt algebraic laws of $\oplus$.

3 CONCLUSION

The Epigram project [AMM05] already advocates the definition of functions using dependent types. They mostly aim at ensuring partial correctness properties (such as a balancing invariant in the case of *mergesort*).

The present paper shows how dependent types can help for ensuring termination too. We showed that an alternate path to termination orderings can be followed in some situations. While our approach is certainly less general, it relies on more elementary arguments. As a consequence, we can get a better insight on the reasons that make the normalization process terminate: they boil down to a (mutual) induction on the implicit structure of terms. Like for approaches advocated by Epigram, the whole game consists in finding dependent types that render this implicit structure explicit.

Our development is available at http://www-verimag.imag.fr/˜monin/.

REFERENCES

[AMM05] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[AN00] C. Alvarado and Q. Nguyen. ELAN for equational reasoning in Coq. In J. Despeyroux, editor, *Proc. of 2nd Workshop on Logical Frameworks and Metalanguages. Institut National de Recherche en Informatique et en Automatique, ISBN 2-7261-1166-1*, June 2000.

[BB00] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In M. Aagaard and J. Harrison, editors, *Proc. of 13th Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLS'00, Portland, OR, USA, 14–18 Aug. 2000*, volume 1689, pages 1–16. Springer-Verlag, Berlin, 2000.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, volume XXV of *Texts in Theoretical Computer Science. An EATCS Series*. Springer, 2004. 469 p., Hardcover. ISBN: 3-540-20854-2.

[BC05] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, August 2005.

[Bla01] Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. In *Logic in Computer Science*, pages 9–18, 2001.

[Bon04] Mike Bond. *Understanding Security APIs*. PhD thesis, University of Cambridge Computer Laboratory, June 2004.

[CKRT05] Yannick Chevalier, Ralf Küsters, Michaël Rusinowitch, and Mathieu Turuani. An np decision procedure for protocol insecurity with xor. *Theor. Comput. Sci.*, 338(1-3):247–274, 2005.

[CLC03] H. Comon-Lundh and V. Cortier. New decidability results for fragments of first-order logic and application to cryptographic protocols. In *Proc. 14th Int. Conf. Rewriting Techniques and Applications (RTA'2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 148–164. Springer, 2003.

[CM06] Judicaël Courant and Jean-François Monin. Defending the bank with a proof assistant. In Dieter Gollmann and Jan Jürjens, editors, *Sixth International IFIP WG 1.7 Workshop on Issues in the Theory of Security*, pages 87 – 98, Vienna, March 2006. European Joint Conferences on Theory And Practice of Software.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–137, New York, NY, USA, 1994. ACM Press.

[Ler00] Xavier Leroy. A modular module system. *J. Funct. Program.*, 10(3):269–303, 2000.

[The05] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.0*. Logical Project, January 2005.

[YAB+05] Paul Youn, Ben Adida, Mike Bond, Jolyon Clulow, Jonathan Herzog, Amerson Lin, Ronald L. Rivest, and Ross Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, Computer Laboratory, August 2005.