Abstract

We supply a library of pure functional terms with the following features: (i) any term can be typed in a type system which implicitly certifies it belongs to the class of terms which evaluate in polynomial time; (ii) they implement all the basic functions required to perform arithmetic on binary finite fields.

The type assignment system is Type Functional Assembly (TFA), an extension of Dual Light Affine Logic (DLAL). The development of the whole library shows we can think of TFA as a domain specific language in which the composition of variants of standard functional programming schemes drives a programmer to think of implementations under non standard patterns.

Keywords: Lambda calculus, Finite fields arithmetic, Type assignments, Implicit computational complexity

1. Introduction

In this paper we address the question if a functional programming approach can be of broader interest when implementing efficient arithmetic. The challenge is posed by a double front of constraints:

1. efficient arithmetic implementation is generally done by programming at architectural level even by keeping in account the running architecture,
2. algorithms are in the feasible range of the complexity bounds (i.e., FPTIME) and even the polynomial degree in the known bounds is subject to full consideration.

The arithmetic over binary extension of finite fields has many important applications in the domains of theory of codes and in cryptography. Finite fields’ arithmetic operations include: addition, subtraction, multiplication, squaring, square root, multiplicative inverse, division and exponentiation.
Declarative programming, by its nature, does not permit a tight control on complexity parameters. The scenario has changed in the last twenty years with the introduction of type systems which implicitly guarantee time complexity bounds on the programs they give a type to. This means that they force restrictions on programming schemes which hardly permit to specify an algorithm in a natural way, even if it belongs to the right complexity class. Therefore a certain number of new type systems have been introduced in the last a few years with the declared objective to capture a broader class of polynomial algorithms with respect to the one which was shown to be in the previous systems, [1, 2, 3, 4, 5, 6].

Our pragmatic workplan is to make fully operational a declarative framework with a variant of a type assignment which seems to balance formal simplicity and expressiveness of the fragment of lambda calculus it gives types to. In this system we program feasible arithmetic ensuring its complexity is polynomial.

We introduce a variant of the system DLAL [7], that we call Typeable Functional Assembly (TFA) having in mind what kind of programming patterns should be used in arithmetic. In fact, we would like to have an even improved control on our system (or even other implicit complexity systems) in order to more precisely certify polynomial computations up to a certain exponent, maybe as a development on the quantitative approach introduced in [8].

We build on our previous paper where we introduced basic materials in order to make arithmetic in binary finite fields by using a declarative language. Principal algorithms are known to be polynomial in complexity. Nevertheless it was not an easy task to show that TFA gives them a type and this is the true obstacle in the use of light systems like TFA as a support to the development of programs with certified running-time complexity. Our experience says that the difficulty arises from the unusual programming patterns that light systems like TFA force to adopt. The main one: it forbids arbitrary nested iterations. Despite this limitation, in this work we show and put in practice several patterns derived from classical Map or MapThread terms. These patterns can be generalised and applied in order to prove that TFA gives type to an algorithm for each basic operation on finite fields.

The most difficult part of this result consists in providing an implementation of multiplicative inversion in binary finite fields arithmetic. We implement it as a term of TFA starting from the Binary Euclidean Algorithm (BEA) as efficiently implemented by Fong in [9]. We recall BEA in Figure 1. It perfectly derives from an imperative programming toolbox: it exploits direct assignments of variables in memory and a control flow in the form of a double nested iteration. A goto-statement creates a loop which includes a while-statement. Obviously, no goto-statement exists in a declarative programming language and while-statements are to be realized by structural iterations on some data type, typically derived from Church numerals. This was a first step in order to write BEA in a declarative style. The second one was to simulate direct access to data structures. This forced us to have and use a reverse of the binary sequence representing the number to invert and then to control the access to the head of the sequence. But the most challenging step was to cope with type constraints on variable duplications which oblige to a parsimonious attitude while programming, in the constant trying to approximate at the best, linear types. The point is to think like if terms would be linear terms (any variable is used exactly once), and then very carefully
INPUT: \( a \in \mathbb{F}_{2^m}, a \neq 0 \).

OUTPUT: \( a^{-1} \mod f \).

1. \( u \leftarrow a, v \leftarrow f, g_1 \leftarrow 1, g_2 \leftarrow 0 \).
2. While \( z \) divides \( u \) do:
   (a) \( u \leftarrow u/z \).
   (b) If \( z \) divides \( g_1 \) then \( g_1 \leftarrow g_1/z \) else \( g_1 \leftarrow (g_1 + f)/z \).
3. If \( u = 1 \) then return(\( g_1 \)).
4. If \( \deg(u) < \deg(v) \) then \( u \leftrightarrow v, g_1 \leftrightarrow g_2 \).
5. \( u \leftarrow u + v, g_1 \leftarrow g_1 + g_2 \).

Figure 1: Binary-Field inversion as in Algorithm 2.2 at page 1048 in [9].
<table>
<thead>
<tr>
<th>( \Delta \models \Gamma \vdash M : A )</th>
<th>( \Delta, x : A \vdash \Gamma \vdash M : A )</th>
<th>( \Delta, x : A, y : A \vdash \Gamma \vdash M : B )</th>
<th>( \Delta, z : A \vdash \Gamma \vdash M[\gamma_x/\beta] : B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta \models \Gamma, x : A \vdash M : B )</td>
<td>( \Delta, \Delta' \models \Gamma, \Gamma' \vdash M : A )</td>
<td>( w )</td>
<td>( c )</td>
</tr>
</tbody>
</table>

2. Typeable Functional Assembly

We call Typeable Functional Assembly (TFA) the deductive system in Figure 2. Its rules come from Dual Light Affine Logic (DLAL) [7]. “Assembly” as part of the name comes from our programming experience inside TFA. When programming inside TFA the goal is twofold. Writing the correct \( \lambda \)-term and lowering their computational complexity so that the \( \lambda \)-term gets typeable. It generally results in \( \lambda \)-terms that work at a very low level in a style which recalls the one typical of programming Turing machines.

Every judgment \( \Delta \models \Gamma \vdash M : A \) has two different kinds of context \( \Delta \) and \( \Gamma \), a formula \( A \) and a \( \lambda \)-term \( M \). The judgment assigns \( A \) to \( M \) with hypothesis from the polynomial context \( \Delta \) and the linear context \( \Gamma \). “Assembly” should make it apparent that \( \lambda \)-terms provide the basic programming constructs that we exploit to define every single ground data type from scratch, booleans included, for example.

Formulas belongs to the language of the following grammar:

\[
\mathcal{F} ::= \mathcal{G} \mid \mathcal{T} \rightarrow \mathcal{F} \mid !\mathcal{T} \rightarrow \mathcal{F} \mid \forall \mathcal{G} \cdot \mathcal{F} \mid \$\mathcal{F}.
\]

The countable set \( \mathcal{G} \) contains variables we range over by lowercase Greek letters. Uppercase Latin letters \( A, B, C, D \) will range over \( \mathcal{T} \). Modal formulas \( !A \) can occur in negative positions only. The notation \( A[\beta/\alpha] \) is the clash free substitution of \( B \) for every free occurrence of \( \alpha \) in \( A \). As usual, clash-free means that occurrences of free variables of \( B \) are not bound in \( A[\beta/\alpha] \).

The \( \lambda \)-term \( M \) belongs to \( A \), the \( \lambda \)-calculus given by:

\[
\Lambda ::= \mathcal{V} \mid (\forall \mathcal{V} \cdot \Lambda) \mid (\Lambda \Lambda).
\]  

(1)

The set \( \mathcal{V} \) contains variables. We range over it by any lowercase Teletype Latin letter. Uppercase Teletype Latin letters \( \mathcal{M}, \mathcal{N}, \mathcal{P}, \mathcal{Q}, \mathcal{R} \) will range over \( \Lambda \). We shall tend to write \( \langle x, M \rangle \) in place of \( (x : M) \) and \( M_1 \ M_2 \ldots M_n \) in place of \( ((M_1) \ M_2) \ldots M_n \). We denote \( \text{fv}(M) \) the set of free variables of any \( \lambda \)-term \( M \). The computation mechanism on \( \lambda \)-terms is the beta-reduction:

\[
(\langle x, M \rangle) N \rightarrow M[x'/x]
\]  

(2)
Finally, we show that the following rules, which give type to tuples, are derivable:

\[ \text{TFA-calculus with tuples as part of } \lambda\text{-terms.} \]

Then, we extend \( \alpha = \ldots \)

A

Consideration

\[ \text{β extends } \text{TFA-calculus with } \lambda\text{-terms.} \]

In fact, we shall assign \( \text{types, not mere formulas, to } \lambda\text{-terms.} \) Introducing the notion of types requires some preliminary definitions.

**Projections.** They are sets of functions that project one argument out of many:

\[ \mathbb{B}_n \equiv \forall \alpha. \mathbb{B}_n[\alpha] \text{ with } \mathbb{B}_n[\alpha] = \alpha \to \cdots \to \alpha \to \alpha . \]

Setting \( n = 2 \) we get “lifted” booleans \( \mathbb{B}_2 \) with canonical representatives:

\[ 1 \equiv \lambda xy. \mathbb{B}_2 ; \quad 0 \equiv \lambda xy. \mathbb{B}_2 ; \quad \bot \equiv \lambda xyz. \mathbb{B}_2 \]

The bottom \( \bot \) simplifies the programming of functions, for example, when combining lists of different lengths.

**Tuples.** They are functions that store a predetermined number of \( \lambda\)-terms:

\[ (A_1 \otimes \ldots \otimes A_n) = \forall \alpha. (A_1 \otimes \ldots \otimes A_n)[\alpha] \to \alpha \text{ with } (A_1 \otimes \ldots \otimes A_n)[\alpha] = A_1 \to \cdots \to A_n \to \alpha \]

The definition of the type \( (A_1 \otimes \ldots \otimes A_n) \), which we shorten as \( \otimes^n A \) whenever \( A_1 = \ldots = A_n \), justifies the adoption of a \( \lambda\)-calculus with tuples as part of TFA. This means we extend TFA in three phases. First, to Definition (1) we add:

\[ \mathbb{M} := \ldots | <\mathbb{M}, \ldots, \mathbb{M}> | (\lambda<\mathbb{V}, \ldots, \mathbb{V}>. \mathbb{M}) . \]

Then, we extend \( \beta\)-reduction with:

\[ (\lambda<\mathbb{x}_1, \ldots , \mathbb{x}_n>. \mathbb{M}) <\mathbb{M}_1, \ldots , \mathbb{M}_n> \to \mathbb{M}[\mathbb{x}_1/\mathbb{M}_1, \ldots , \mathbb{x}_n/\mathbb{M}_n] . \]

Finally, we show that the following rules, which give type to tuples, are derivable:

\[ \frac{\Delta_1 | \Gamma_1 \vdash \mathbb{M}_1 : A_1 \quad \ldots \quad \Delta_n | \Gamma_n \vdash \mathbb{M}_n : A_n}{\Delta_1 \ldots \Delta_n | \Gamma_1 \ldots \Gamma_n \vdash \lambda<\mathbb{x}_1, \ldots , \mathbb{x}_n>. : (A_1 \otimes \ldots \otimes A_n) \otimes 1}{\Delta | \Gamma \vdash \lambda<\mathbb{x}_1, \ldots , \mathbb{x}_n>. : (A_1 \otimes \ldots \otimes A_n) \to B \to 1} \]

This implies that we, in fact, use tuples as abbreviations:

\[ <\mathbb{M}_1, \ldots , \mathbb{M}_n> \text{ stands for } \lambda \mathbb{x}. \mathbb{M}_1 \ldots \mathbb{M}_n \]

\[ \lambda<\mathbb{x}_1, \ldots , \mathbb{x}_n>. \mathbb{M} \text{ stands for } \lambda \mathbb{p}. \mathbb{p} (\lambda \mathbb{x}_1, \ldots , (\lambda \mathbb{x}_n. \mathbb{M})) . \]
Sequences of booleans, or simply Sequences. We denote them by \( S \) and recursively define as:

\[
S \equiv \forall \alpha. S[\alpha] \text{ with } S[\alpha] \equiv (B \rightarrow \alpha) \rightarrow ((B \otimes S) \rightarrow \alpha) \rightarrow \alpha.
\] (3)

The equation (3) induces an obvious congruence \( \approx \) on the set \( F \) of formulas. The congruence identifies equivalence classes of formulas that we effectively use as types of \( \lambda \)-terms.

2.1. The set \( T \) of types

The set \( T \) of types is the quotient \( F / \approx \). We mean that if \( M \) has type \( S \), then we can equivalently use any of the unfolded forms of \( S \) as type of \( M \). The canonical values of type \( S \) are:

\[
\begin{align*}
[\varepsilon] & \equiv \\text{tc}\. t \perp : S \\
[b_{n-1} \ldots b_0] & \equiv \\text{tc}\. c <b_{n-1},[b_{n-2} \ldots b_0]> : S.
\end{align*}
\] (4)

In accordance with (3), the Sequence \( [b_{n-1} \ldots b_0] \) in (4) is a function that takes two constructors as inputs and yields a Sequence. Only the second constructor is used in (4) to build a Sequence out of a pair whose first element is \( b_{n-1} \), and whose second element is — recursively! — another Sequence \( [b_{n-2} \ldots b_0] \). The recursive definition of \( S \) should be evidently crucial.

By convention, in every Sequence \( [b_{n-1} \ldots b_0] \), the least significant bit (lsb) is \( b_0 \) and the most significant bit (msb) is \( b_{n-1} \).

Notations we introduced on formulas, simply adapt to types, i.e. to equivalence classes of formulas which, generally, we identify by means of the obvious representative. Moreover, it is useful to call every pair \( x : A \) of any kind of context as type assignment for a variable.

2.2. Summing up

TFA is DLAL [7] whose set of formulas is quotiented by a specific recursive equation. We recall it is well known that, adding recursive equations among the formulas of DLAL, is harmless as far as polynomial time soundness is concerned. The reason is that the proof of polynomial time soundness of DLAL only depends on its structural properties [12, 7]. It never relies on measures related to the formulas. So, recursive types, whose structure is not well-founded, cannot create concerns on complexity.

3. Basic Definitions, Types and the Core Library

From [13], we recall the meaning and the type of the \( \lambda \)-terms that forms the two lowermost layers in Figure 3. We also recall their definition in Appendix A.
Cryptographic primitives: elliptic curves cryptography, linear feedback shift register cryptography, . . .

Binary-field arithmetic: addition, (modular reduction), square, multiplication, inversion.

Core library: operations on bits (xor, and), operations on sequences (head-tail splitting), operations on words (reverse, drop, conversion to sequence, projections); meta-combinators: fold, map, mapthread, map with state, head-tail scheme.

Basic definitions and types: booleans, tuples, numerals, words, sequences, basic type management and duplication.

Figure 3: Library for binary-field arithmetic

Paragraph lift. We can derive the following rule in TFA:

\[ \emptyset \mid \emptyset \vdash \mathbf{K} : \alpha \rightarrow \beta \]

\[ \emptyset \mid \emptyset \vdash \$[\mathbf{M}] : \alpha \rightarrow \$[\beta] \]

where \$[\mathbf{M}] \equiv \lambda x . \mathbf{M} x \text{ is the paragraph lift of } \mathbf{M}. \text{ As obvious generalization, } n \text{ consecutive applications of the } \$[\cdot] \text{ rule define a lifted term } \$^n[\mathbf{M}] \equiv \lambda x . . . (\lambda x . \mathbf{M} x) . . . x, \text{ that contains } n \text{ nested } \$[\cdot]. \text{ Its type is } \$^n\alpha \rightarrow \$^n\beta. \text{ Borrowing terminology from proof nets, the application of } n \text{ paragraph lift of } \mathbf{M} \text{ embeds it in } n \text{ paragraph boxes, leaving the behavior of } \mathbf{M} \text{ unchanged:}

\[ \$^n[\mathbf{M}] N \rightarrow^* \mathbf{M} N. \]

3.1. Basic Definitions and Types

Church numerals. They have type:

\[ \mathbb{U} \equiv \forall \alpha . \mathbb{U}[\alpha] \text{ where } \mathbb{U}[\alpha] \equiv ! (\alpha \rightarrow \alpha) \rightarrow \$ (\alpha \rightarrow \alpha) \]

with canonical representatives:

\[ \mathbb{U} \equiv \lambda f x . \mathbb{U} \quad \mathbb{N} \equiv \lambda f x . f (\ldots (f (x) \ldots)) : \mathbb{U} \text{ with } n \text{ occurrences of } f \]

They iterate the first argument on the second one.

Lists. They have type:

\[ \mathbb{L}(\alpha) \equiv \forall \alpha . \mathbb{L}(\alpha)[\alpha] \text{ where } \mathbb{L}(\alpha)[\alpha] \equiv !(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \$ (\alpha \rightarrow \alpha) \]

with canonical representatives:

\[ \{ \varepsilon \} \equiv \lambda f x . \mathbb{L}(\alpha) \]

\[ \{ \mathbb{M}_n \ldots \mathbb{M}_0 \} \equiv \lambda f x . \mathbb{M}_n \ldots (f \mathbb{M}_0 x) \ldots : \mathbb{L}(\alpha) \text{ with } n \text{ occurrences of } f \]

that generalize the iterative structures of Church numerals.
**Church words.** A Church word is a list \([b_{n-1} \ldots b_0]\) whose elements \(b_i\)s are booleans, i.e. of type \(\mathbb{L}_2 \equiv \mathbb{L}(\mathbb{B}_2)\). By convention, in every Church word \([b_{n-1} \ldots b_0]\), or simply word, the *least significant bit* (lsb) is \(b_0\), while the *most significant bit* (msb) is \(b_{n-1}\). The same convention holds for every Sequence \([b_{n-1} \ldots b_0]\), or simply word, the least significant bit (lsb) is \(b_0\), while the most significant bit (msb) is \(b_{n-1}\).

The combinator \(b\text{Cast}^m : \mathbb{B}_2 \rightarrow \mathbb{B}^{m+1}_2\). It casts a boolean inside \(m + 1\) paragraph boxes, without altering the boolean:

\[
b\text{Cast}^m b \rightarrow^* b.
\]

The combinator \(b\text{Work} : \mathbb{B}_t \rightarrow \mathbb{B}_2\), for every \(t \geq 2\). It produces \(t\) copies of a boolean:

\[
b\text{Work} b \rightarrow^* \langle b, \ldots, b \rangle.
\]

Despite \(b\text{Work}\) replicates its argument it has a linear type. The reason is that \(t\) is fixed as one can appreciate from the definition of \(b\text{Work}\) in Appendix A.

The combinator \(t\text{Cast}^m : (\mathbb{B}_2 \otimes \mathbb{B}_2) \rightarrow \mathbb{B}^{m+1}_2\), for every \(m \geq 0\). It casts a pair of bits into \(m + 1\) paragraph boxes, without altering the structure of the pair:

\[
t\text{Cast}^m \langle b_0, b_1 \rangle \rightarrow^* \langle b_0, b_1 \rangle.
\]

The combinator \(w\text{Suc} : \mathbb{B}_2 \rightarrow \mathbb{L}_2 \rightarrow \mathbb{L}_2\). It implements the successor on Church words:

\[
w\text{Suc} b \langle b_{n-1} \ldots b_n \rangle \rightarrow^* \langle b \ b_{n-1} \ldots b_n \rangle.
\]

The combinator \(w\text{Cast}^m : \mathbb{L}_2 \rightarrow \mathbb{L}^{m+1}_2\), for every \(m \geq 0\). It embeds a word into \(m + 1\) paragraph boxes, without altering the structure of the word:

\[
w\text{Cast}^m \langle b_{n-1} \ldots b_n \rangle \rightarrow^* \langle b_{n-1} \ldots b_n \rangle.
\]

The combinator \(w\text{Work}^m : \mathbb{L}_t \rightarrow \mathbb{L}^{m+1}_2\), for every \(t \geq 2, m \geq 0\). It produces \(t\) copies of a word embedding the result into \(m + 1\) paragraph boxes:

\[
w\text{Work}^m \langle b_{n-1} \ldots b_n \rangle \rightarrow^* \langle b_{n-1} \ldots b_n \rangle, \ldots, \langle b_{n-1} \ldots b_n \rangle.
\]

### 3.2. Core Library

The combinator \(\text{Xor} : \mathbb{B}_2 \rightarrow \mathbb{B}_2 \rightarrow \mathbb{B}_2\). It extends the exclusive or as follows:

<table>
<thead>
<tr>
<th>(\text{Xor} )</th>
<th>(\rightarrow^*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0 \ 0)</td>
<td>(0)</td>
</tr>
<tr>
<td>(1 \ 1)</td>
<td>(0)</td>
</tr>
<tr>
<td>(1 \ 0)</td>
<td>(1)</td>
</tr>
<tr>
<td>(b \ \bot)</td>
<td>(\rightarrow^*) b</td>
</tr>
</tbody>
</table>

Whenever one argument is \(\bot\), then it gives back the other argument. This is an application oriented choice. Later we shall see why.
The combinator \textbf{And} : \( B_2 \rightarrow B_2 \rightarrow B_2 \). It extends the combinator \textit{and} as follows:
\[
\begin{align*}
\text{And} \; 0 \; 0 & \rightarrow^* 0 \quad & \text{And} \; 1 \; 1 & \rightarrow^* 1 \\
\text{And} \; 0 \; 1 & \rightarrow^* 0 \quad & \text{And} \; 1 \; 0 & \rightarrow^* 0 \\
\text{And} \; \bot \; b & \rightarrow^* \bot \quad & \text{And} \; b \; \bot & \rightarrow^* \bot \\
\end{align*}
\] (where \( b : B_2 \)).

Whenever one argument is \( \bot \) then the result is \( \bot \). Again, this is an application oriented choice.

The combinator \textbf{sSpl} : \( S \rightarrow (B_2 \otimes S) \). It \textit{splits} the sequence it takes as input in a pair with the m.s.b. and the corresponding tail:
\[
s\text{Spl} \; [b_{n-1} \ldots b_0] \rightarrow^* <b_{n-1}, [b_{n-2} \ldots b_0]>.
\]

The combinator \textbf{wRev} : \( L_2 \rightarrow L_2 \). It \textit{reverses} the bits of a word:
\[
w\text{Rev} \; \{b_{n-1} \ldots b_0\} \rightarrow^* \{b_0 \ldots b_{n-1}\}.
\]

The combinator \textbf{wDrop} : \( L_2 \rightarrow \bot \rightarrow L_2 \). It \textit{drops} all the (initial) occurrences\footnote{The current definition actually drops all the occurrences of \( \bot \) in a Church word, however we shall only apply \( \text{wDrop} \) to words that contain \( \bot \) in the most significant bits.} of \( \bot \) in a word:
\[
w\text{Drop} \; \{\bot \ldots \bot b_{n-1} \ldots b_0\} \rightarrow^* \{b_{n-1} \ldots b_0\}.
\]

The combinator \textbf{w2s} : \( L_2 \rightarrow S \). It \textit{translates} a word into a sequence:
\[
w\text{2s} \; \{b_{n-1} \ldots b_0\} \rightarrow^* \{b_{n-1} \ldots b_0\}.
\]

Its type inference is in Appendix B.

The combinator \textbf{wProj} : \( L(\mathbb{B}_2) \rightarrow L_2 \). It \textit{projects} the first component of a list of pairs:
\[
w\text{Proj}_1 \; \{<a_{n-1}, b_{n-1}> \ldots <a_0, b_0>\} \rightarrow^* \{a_{n-1} \ldots a_0\}.
\]

Similarly, \( w\text{Proj}_2 : L(\mathbb{B}_2) \rightarrow L_2 \) projects the second component.

### 3.2.1. Meta-combinators

First we recall the meta-combinators from \cite{13}. We used them to implement addition, modular reduction, square and multiplication in layer three of Figure 3.

Then, we introduce a new meta-combinator that supplies the main programming pattern to implement BEA as a \( \lambda \)-term of TFA.

Meta-combinators are \( \lambda \)-terms with one or two “holes” that allow to use standard higher-order programming patterns to extend the API. Holes must be filled with type constrained \( \lambda \)-terms.
The meta-combinator \texttt{Map}[-]. Let $F : A \rightarrow B$ be a closed term. Then, $\texttt{Map}[F] : \mathbb{L}(A) \rightarrow \mathbb{L}(B)$ applies $F$ to every element of the list that $\texttt{Map}[F]$ takes as argument, and yields the final list, assuming $F \ b_i \rightarrow^* b'_i$, for every $0 \leq i \leq n - 1$:

\[
\texttt{Map}[F] \{b_{n-1} \ldots b_0\} \rightarrow^* \{b'_{n-1} \ldots b'_0\}.
\]

The meta-combinator \texttt{Fold}[-,-]. Let $F : A \rightarrow B \rightarrow B$ and $S : B$ be closed terms. Let also $\texttt{Cast}^0 : B \rightarrow \mathbb{S}B$. Then, $\texttt{Fold}[F,S] : \mathbb{L}(A) \rightarrow \mathbb{S}B$, starting from the initial value $S$, iterates $F$ over the input list and builds up a value, assuming $((F \ b_1) \ b'_1) \rightarrow^* b'_{i+1}$, for every $0 \leq i \leq n - 1$, and setting $b'_0 \equiv S$ and $b'_n \equiv b'$:

\[
\texttt{Fold}[F,S] \{b_{n-1} \ldots b_0\} \rightarrow^* b'.
\]

The meta-combinator \texttt{MapState}[-]. Let $F : (A \otimes S) \rightarrow (B \otimes S)$ be a closed term. Then, $\texttt{MapState}[F] : \mathbb{L}(A) \rightarrow \mathbb{L}(B)$ applies $F$ to the elements of the input list, keeping track of a state of type $S$ during the iteration. Specifically, if $F \langle b_1, s_i \rangle \rightarrow^* \langle b'_1, s_{i+1} \rangle$, for every $0 \leq i \leq n - 1$:

\[
\texttt{MapState}[F] \{b_{n-1} \ldots b_0\} s_0 \rightarrow^* \{b'_{n-1} \ldots b'_0\}.
\]
The meta-combinator $\text{MapThread}$. Let $F : B_2 \rightarrow B_2 \rightarrow A$ be a closed term. Then, $\text{MapThread}[F] : L_2 \rightarrow L_2 \rightarrow \mathbb{L}(A)$ applies $F$ to the elements of the input list. Specifically, if $F_{a_i} b_i \rightarrow^* c_i$, for every $0 \leq i \leq n - 1$:

$$\text{MapThread}[F] \{a_{n-1} \ldots a_0\} \{b_{n-1} \ldots b_0\} \rightarrow^* \{c_{n-1} \ldots c_0\}.$$

In particular, $\text{MapThread}[\lambda a.<a\ b>] : L_2 \rightarrow L_2 \rightarrow \mathbb{L}(B_2^2)$ is such that:

$$\text{MapThread}[\lambda a.<a\ b>] \{a_{n-1} \ldots a_0\} \{b_{n-1} \ldots b_0\} \rightarrow^* \{<a_{n-1}, b_{n-1}> \ldots <a_0, b_0>\}.$$

The meta-combinator $\text{wHeadTail}[L, B]$. It has two parameters $L$ and $B$ and builds on the core mechanism of the predecessor for Church numerals [14, 12] inside typing systems like TFA. For any types $A, \alpha$, let $X \equiv (A \rightarrow \alpha \rightarrow \alpha) \otimes A \otimes \alpha$. By definition, $\text{wHeadTail}[L, B]$ is as follows:

$$\text{wHeadTail}[L, B] \equiv \lambda f x. L \left( \lambda (\text{wHTStep}[B] \ f) (\text{wHTBase} \ x) \right)$$

$$\text{wHTStep}[B] \equiv \lambda f e. <f t, e t, t> . B[f, e, f t, e t, t] \quad (5)$$

$$\text{wHTBase} \equiv \lambda x. <\lambda e l. \text{DummyElement}, x>$$

where:

- $L$ stands for “last (step)”. It denotes a closed $\lambda$-term with type $X \rightarrow \alpha$.
- $B[f, e, f t, e t, t]$ stands for “body (of the step function)”. It denotes a closed $\lambda$-term with the following two features. It must have type $X$ and the variables $f$, $e$, $f t$, $e t$ and $t$ must be sub-terms of $B$ that must occur linearly in it.
4. TFA Combinators for Binary-Fields Arithmetic

In this section we introduce those $\lambda$-terms of TFA which implement basic operations of the third layer in Figure 3; amongst them, inversion yields the most elaborate construction built as a variant of the meta-combinator $\text{wHeadTail}$.

Let us recall some essentials on binary-fields arithmetic (See [15, Section 11.2] for wider details). Let $p(X) \in F_2[X]$ be an irreducible polynomial of degree $n$ over $F_2$, and let $\beta \in F_2$ be a root of $p(X)$ in the algebraic closure of $F_2$. Then, the finite-field $F_{2^n} \cong F_2[X]/(p(X)) \cong F_2(\beta)$.

The set of elements $\{1, \beta, \ldots, \beta^{n-1}\}$ is a basis of $F_{2^n}$ as a vector space over $F_2$, and we can represent a generic element of $F_{2^n}$ as a polynomial in $\beta$ of degree lower than $n$:

$$F_{2^n} \ni a = \sum_{i=0}^{n-1} a_i \beta^i = a_{n-1} \beta^{n-1} + \cdots + a_1 \beta + a_0$$

Moreover, the isomorphism $F_{2^n} \cong F_2[X]/(p(X))$ allows us to implement the arithmetic of $F_{2^n}$ relying on the arithmetic of $F_2[X]$ and reduction modulo $p(X)$.

Since every $a_i \in F_2$ can be encoded as a bit, we can represent each element of length $n$ in $F_{2^n}$ as a Church word of bits of type $\mathbb{L}_2$. For this reason, when useful, we remark that a Church word is, in fact, a finite-field instance by replacing the notation $F_{2^n}$, instead than $\mathbb{L}_2$, as type. So, $\mathbb{L}_2$ and $F_{2^n}$ becomes essentially interchangeable.
In what follows, we denote by \( n \) the Church numeral \( \bar{n} \), representing the integer \( n = \deg p(X) \), and, by \( p \), the Church word \( p = \left\{ p_0 \ldots p_n \right\} \), where \( p_i \) are the boolean terms associated to the corresponding coefficient \( p_i \) of the polynomial \( p(X) = \sum p_i X^i \). Note that \( p \) has length \( 2n \). The \( \bot \) in the least significant part are included for technical reasons, to simplify the discussion later.

4.1. Addition

Let \( a, b \in \mathbb{F}_2 \). The addition \( a + b \) is computed component-wise, i.e., setting \( a = \sum a_i \beta^i \) and \( b = \sum b_i \beta^i \), then \( a + b = \sum (a_i + b_i) \beta^i \). The sum \( (a + b) \) is done in \( \mathbb{F}_2 \) and corresponds to the bitwise exclusive or. This led us to the following definition: The combinator acting on lists \( \text{Add} : \mathbb{F}_2 \rightarrow \mathbb{F}_2 \rightarrow \mathbb{F}_2 \) is:

\[
\text{Add} \equiv \text{MapThread}[\text{Xor}] .
\] (8)

4.2. Modular Reduction

Reduction modulo \( p(X) \) is a fundamental building block to keep the size of the operands constrained. We implemented a naïf left-to-right method, assuming that: (1) both \( p(X) \) and \( n = \deg p(X) \) are fixed (thus given as parameters); (2) the length of the input is \( 2n \), i.e., we need exactly \( n \) repetitions of a basic iteration. The combinator \( \text{wMod} \) is:

\[
\text{wMod} : \mathbb{F}_2 \rightarrow \mathbb{S}_2 \rightarrow \mathbb{F}_2
\]

where:

\[
\text{wMod} \equiv \text{l} \cdot \text{MapThread}[\text{wModFun}] \cdot \text{l} \cdot \text{MapState}[\text{wModBase}] \cdot (\text{wCast}^* \cdot \text{d})
\]

The combinator \( \text{MapState}[-] \) implements the basic iteration operating on a list \([\ldots, <d_1, p_1>, \ldots] \) of pairs of bits, where \( d_1 \) are the bits of the input and \( p_1 \) the bits of \( p \). The core of the algorithm is the combinator \( \text{wModFun} : (\mathbb{B}_2 \otimes \mathbb{B}_2) \rightarrow (\mathbb{B}_2 \otimes \mathbb{B}_2) \), that behaves as follows:

\[
\text{wModFun} \left( \langle d_1, p_1 \rangle, \langle s_0, p_{1+i} \rangle \right) \rightarrow \langle \langle d_1', p_{1+i}' \rangle, \langle \langle s_0', \perp \rangle \rangle \rangle
\]

where \( s_0 \) keeps the m.s.b. of \([\ldots, d_1, \ldots] \) and it is used to decide whether to reduce or not at this iteration. Thus, \( d_1' = d_1 + p_1 \) if \( s_0 = 1 \); \( d_1' = d_1 \) if \( s_0 = \emptyset \); and \( d_1' = \bot \) when \( s_0 = \bot \) (that represents the initial state, when \( s_0 \) still needs to be set).

Note that the second component of the status is used to shift \( p \) (right shift as the words have been reverted).
4.3. Square

Square in binary-fields is a linear map (it is the absolute Frobenius automorphism). If \( a \in \mathbb{F}_2^n \), \( a = \sum a_i\beta^i \), then \( a^2 = \sum a_i a_j \beta^{i+j} \). This operation is obtained by inserting zeros between the bits that represent \( a \) and leads to a polynomial of degree \( 2n - 2 \), that needs to be reduced modulo \( p(X) \).

Therefore, we introduce two combinators: \( \text{wSqr} : \mathbb{L}_2 \rightarrow \mathbb{L}_2 \) that performs the bit expansion, and \( \text{Sqr} : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n \), that is the actual square in \( \mathbb{F}_2^n \). We have:

\[
\text{Sqr} = \lambda a. \text{wMod}[n, p] (\text{wSqr} a)
\]

and \( \text{wSqr} = \lambda f. l \ \text{wSqrStep}[f] \) \( x \), where \( \text{wSqrStep}[f] = \lambda e. t. f (f e t) \) has type \( \mathbb{B}_2 \rightarrow \mathbb{B}_2 \rightarrow \mathbb{B}_2 \rightarrow \mathbb{B}_2 \).

4.4. Multiplication

Let \( a, b \in \mathbb{F}_2^n \). The multiplication \( ab \) is computed as polynomial multiplication, i.e., with the usual definition, \( ab = \sum_{r \in \mathbb{L}_2} (a_i + b_i) \beta^r \).

We currently implemented the naïve schoolbook method. A possible extension to the comb method is left as future straightforward work. On the contrary, it is not clear how to implement the Karatsuba algorithm, which reduces the multiplication of \( n \)-bit words to operations on \( n/2 \)-bit words. The difficulty is to represent the splitting of a word in its half upper and lower parts.

As for \( \text{Sqr} \), we have to distinguish between multiplication of two arbitrary degree polynomials represented as binary lists, \( \text{wMult} : \mathbb{L}_2 \rightarrow \mathbb{L}_2 \rightarrow \mathbb{L}_2 \) and the field operation \( \text{Mult} : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n \), obtained by composing with the modular reduction. We have:

\[
\text{Mult} = \lambda a. \text{wMod}[n, p] (\text{wMult} a b)\\
\text{wMult} = \lambda a. \text{wMod}[n, p] (\text{wProj}[b] (\lambda (m. \text{wMultStep} <M, \perp> 1)(\text{wMultBase} (\text{wCast}^k a)))) .
\]

The internals of \( \text{wMult} \) are in Figure 4. It implements two nested iterations. The parameter \( b \) controls the external, and \( a \) the internal one. The external iteration (controlled by \( b \)) works on words of bit pairs. The combinator \( \text{wMultStep} : \mathbb{E}_2^2 \rightarrow \mathbb{L}(\mathbb{E}_2^2) \rightarrow \mathbb{L}(\mathbb{E}_2^2) \) behaves as follows:

\[
\text{wMultStep} <M, \perp> [...<m_i, r_i>...] \rightarrow^* [...<m_{i-1}, r_{i-1}>, ...]
\]
$$\text{wInv} =$$

```
\U. # Word in input.
(wProj # Extract the bits of G1 from the threaded word.
 (D # Parameter of wInv. It is a Church numeral. Its value is # the square of the degree n of the binary field.
   (\tw.\wRevInit (BkwVst (wRev (FwdVst tw)))) # Step funct. of D.
 ) (MapThread[\u.\v.\g1.\g2.
 \m.\stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
 <u,v,g1,g2,m,stop,sn,rs,fwdv,fwdg2,fwdm> ]
 U
 [m_{n-1}...m_1 1] # V is a copy of the modulus.
 [ 0... 0 1] # G1 with n components.
 [ 0... 0 0] # G2 " " "
 [m_{n-1}...m_1 1] # M is a copy of the modulus.
 [ 0... 0 0] # Stop with n components.
 [ B... B B] # StpNmbr " " "
 [ B... B B] # RghtShft " " "
 [ 0... 0 0] # FwdV " " "
 [ 0... 0 0] # FwdG2 " " "
 [ 0... 0 0] # FwdM " " "
 ) # Base function of D.
 )
```

Figure 5: Definition of \text{wInv}.

where $M$ is the current bit of the multiplier $b$, and every $m_i$ is a bit of the multiplicand $a$, and every $r_i$ is a bit in the current result. The iteration is enabled by the combinator $\text{wMultBase}: B_2 \rightarrow L(B_2^2)$, that, on input $a$, creates $\langle m_{n-1}, \perp \rangle \ldots \langle m_0, \perp \rangle$, setting the initial bits of the result to $\perp$. The projection $\text{wProj}$ returns the result when the iteration stops.

The internal iteration is used to update the above list of bit pairs. The core of this iteration is the combinator $\text{wFMult}: B_2^2 \rightarrow (B_2 \otimes B_2^2)$, that behaves as follows:

$$\text{wFMult} <m_i, r_i> \langle M, m_{i-1} \rangle \rightarrow' \langle <m_{i-1}, M \cdot m_i + r_i>, \langle M, m_i \rangle >.$$ 

For completeness, we list the type of the other combinators: $\text{MSStep}[f, \text{wFMult}]: B_2 \rightarrow (\alpha \otimes B_2^2) \rightarrow (\alpha \otimes B_2^2)$, $\text{MSBase}[x]: B_2^2 \rightarrow (\alpha \otimes B_2^2)$, $\text{wBMult}[f]: (\alpha \otimes B_2^2) \rightarrow \alpha$.

4.5. Multiplicative Inversion

We reformulate BEA in Figure 1 as a $\lambda$-term wInv of TFA as in Figure 5. wInv starts building
a list which it obtains by means of \texttt{MapThread} applied to eleven lists. For example, let $u = z^2$ and $v = z^3 + z + 1$ and $g_1 = 1$ and $g_2 = 0$ be an input of \texttt{BEA}. We represent the polynomials as words:

\begin{align*}
U &= f \cdot x \cdot f \ 0 \ (f \ 1 \ (f \ 0 \ x))) \\
V &= f \cdot x \cdot f \ 1 \ (f \ 0 \ (f \ 1 \ x))) \\
G_1 &= f \cdot x \cdot f \ 0 \ (f \ 0 \ (f \ 1 \ x))) \\
G_2 &= f \cdot x \cdot f \ 0 \ (f \ 0 \ (f \ 0 \ x))) .
\end{align*}

\texttt{wInv} builds an initial list by applying \texttt{MapThread} to the four words in (10) and to further seven words which build the state of the computation. In our running example, the whole initial list is:

\begin{verbatim}
\f \. \x \ . # |------- This is a state ----------|
#   v   v
# U V G1 G2 M Stop StpNmb RghtShft FwdV FwdG2 FwdM
f <0,1,0,0,1,0,B,B,0,0,0> # msb
(f <1,0,0,0,0,0,B,B,0,0,0>
(f <0,1,0,0,0,0,B,B,0,0,0>
(f <0,1,0,0,0,0,B,B,0,0,0>
(f <0,1,1,0,1,0,B,B,0,0,0> # lsb
x))) .
\end{verbatim}

We call \textit{threaded words} the list (11) that \texttt{wInv} builds in its first step. We adopt the same name for every list whose tuples have eleven boolean elements with the position meaning that (11) highlights. The 4th element of column U is U[i]. We adopt analogous notation on V, G1, etc.. We write \langle V, \ldots, M \rangle[i] or \langle V[i], \ldots, M[i] \rangle to denote the projection of the bits in column V, G1, G2 and M out of the i-th element. Analogous notation holds for arbitrary sub-sequences we need to project out of U, \ldots, FwdM. The most significant bit msb of any threaded words is on top; its less significant bit lsb is at the bottom.

The variable D which appears in Figure 5 takes the type of a Church numeral and the term which follows \texttt{tw.wRevInit (BkwVst (wRev (FwdVst tw))))} is the step function which is iterated starting from a threaded words built like (11) was. The step function implements steps from 2 through 5 of \texttt{BEA} in Figure 1. The iteration that D implements is the outermost loop which starts at step 2 and stops at step 6. \texttt{FwdVst} shortens \textit{forward visit}. \texttt{wRev} reverses the threaded words it takes as input. \texttt{BkwdVst} stands for \textit{backward visit}. \texttt{wRevInit} reverses the threaded words it gets in input while reinitializing the bits in positions StpNmb, RghtShft, FwdV, FwdG2 and FwdM.

\texttt{FwdVst} builds on the pattern of the meta-combinator \texttt{wHeadTail}[L, B]. Its input is a threaded words which we call \texttt{wFwdVstInput}. Its output is again a threaded words \texttt{wFwdVstOutput}. \texttt{FwdVst} can distinguish its step zero, and its last step. Yet, for every \(0<i<=\text{msb}\), \texttt{FwdVst} builds the i-th element of \texttt{wFwdVstOutput} on the base of \langle 0, V, \ldots, FwdM \rangle[i] which it takes from \texttt{wFwdVstInput} and \langle 0, V, \ldots, FwdM \rangle[i-1] taken from \texttt{wFwdVstOutput}.

The identification of step zero allows \texttt{FwdVst} to simultaneously check which of the following mutually exclusive questions has a positive answer:

\begin{quote}
"Is Stop[0]=1?" (12)
"Does z divide both u and g_1?" (13)
"Does z divide u but not g_1?" (14)
"Neither of the previous questions has positive answer?" (15)
\end{quote}

If (12) holds, \texttt{FwdVst} must behave as the identity. Such a situation is equivalent to saying that all the bits in position G1 contain the result.
Let us assume instead that (13) or (14) hold. Answering the first question requires to verify \( U[0] = 0 \) and \( G_1[0] = 0 \) in \( w_{FwdVstInput} \). Answering the second one needs to check both \( U[0] = 0 \) and \( G_1[0] = 1 \) in \( w_{FwdVstInput} \). Under our conditions, just after reading \( w_{FwdVstInput} \), the combinator \( FwdVst \) generates the following first element, i.e. the lsb, of \( w_{FwdVstOutput} \):

\[
<U[0], B, g_1, B, B, 0, rs, V[0], G_2[0], M[0]>
\]  

(16)

If (13) holds, then \( g_1 \) is \( G_1[0] \) and \( rs \) is 1. If (14) holds, then \( g_1 \) is \( \text{Xor } G_1[0] M[0] \) and \( rs \) is 0. For building (16) we first record \( V[0], G_2[0] \) and \( M[0] \), which \( w_{FwdVstInput} \) supplies, in position \( FwdV[0], FwdG2[0] \) and \( FwdM[0] \), respectively, of \( w_{FwdVstOutput} \). Then we set \( V[0] = G_2[0] = M[0] = B \) in \( w_{FwdVstOutput} \).

After the generation of the first element (16), for every \( 0 < i \leq \text{msb} \), the iteration that \( FwdVst \) implements proceeds as follows. It focuses on two elements at step \( i \):

\[
<U, V, G_1, G_2, M, \text{Stop}, \text{StpNmb}, \text{RghtShft}, FwdV, FwdG2, FwdM>[i]
\]

\[
<U, V, G_1, G_2, M, \text{Stop}, \text{StpNmb}, \text{RghtShft}, FwdV, FwdG2, FwdM>[i-1]
\]  

(17)

The tuple with index \( i \) belongs to \( w_{FwdVstInput} \). The one with index \( i-1 \) is the \( i-1 \)th element of \( w_{FwdVstOutput} \). So, \( FwdVst \) generates the new \( i \)th element of \( w_{FwdVstOutput} \) from them which will become the \( i-1 \)th element of \( w_{FwdVstOutput} \) in the succeeding step:

\[
<U[i], FwdV[i-1], g_1, FwdG2[i-1], FwdM[i-1], B, rs, V[i], G_2[i], M[i]>
\]  

(18)

Yet, \( g_1 \) and \( rs \) depend on \( u \) and \( g_1 \) being divisible by \( z \).

Finally, under the above condition that (13) or (14) hold, the last step of \( FwdVst \) adds two elements to \( w_{FwdVstOutput} \). Let \( \text{msb} \) be the length of \( w_{FwdVstInput} \). The two last elements of \( w_{FwdVstOutput} \) are:

\[
<0, V[\text{msb}], 0, G_2[\text{msb}], M[\text{msb}], B, 0, rs, B, B, B> \ # \text{msb of } w_{FwdVstOutput}
\]

\[
<U[\text{msb}], FwdV[\text{msb}-1], g_1, FwdG2[\text{msb}-1], FwdM[\text{msb}-1], B, 0, rs, B, B, B>
\]  

(19)

As before, \( g_1 \) and \( rs \) keeps depending on which between (13) or (14) hold. The elements \( FwdV[\text{msb}-1], FwdG2[\text{msb}-1] \) and \( FwdM[\text{msb}-1] \) come from \( w_{FwdVstOutput} \). The elements \( U[\text{msb}], V[\text{msb}], G_2[\text{msb}] \) and \( M[\text{msb}] \) belong to the last element of \( w_{FwdVstInput} \).

Even though this might sound a bit paradoxically, the overall effect of iterating the process we have just described — the one which exploits the simultaneous access to an element of both \( w_{FwdVstInput} \) and \( w_{FwdVstOutput} \) and which adds two last elements to \( w_{FwdVstOutput} \) as specified in (19) — amounts to shifting the bits in positions \( V \), \( G_2 \) and \( M \) of \( w_{FwdVstInput} \) one step to their left. Instead, it leaves the bits of position \( U \) and \( G_1 \) as they were in \( w_{FwdVstInput} \) so that they, in fact, shift one step to their right if we are able to erase the lsb of \( w_{FwdVstOutput} \). We shall erase such a lsb by means of \( BkwdVst \). Roughly, only a correct concatenation of both \( FwdVst \) and \( BkwdVst \) shifts to the right every \( U[i] \) and \( G_1[i] \), or \( \text{Xor } G_1[i] M[i] \), while preserving the position of every other element.

The description of how \( FwdVst \) works concludes by assuming that neither (13) nor (14) hold. This occurs when \( U[0] = 1 \). \( FwdVst \) must forcefully answer to: "Is \( u \) different from 1?". Answering the question requires a complete visit of the threaded words that \( FwdVst \) takes in input. The visit serves to verify whether some \( j > 0 \) exists such that \( U[j] = 1 \). The non existence of \( j \) implies that \( FwdVst \) sets \( \text{Stop}[\text{msb}] = 1 \). This will impede any further change of any bit in any position of the threaded words generated so far. If, instead, \( j \) such that \( U[j] = 1 \) exists, then the last step of \( FwdVst \) adds a tuple to \( w_{FwdVstOutput} \) that contains \( <\text{Stop}, \text{StpNmb}>[\text{msb}] = <0, 1> \). This
Let $l$ be the position of the last element of $w_{FwdVstOutput}$.

1. If $<\text{Stop}, \text{StpNmbr}, \text{RghtShft}>[l]=<1, -, ->$, then $FwdVst$ has verified that $u$ is 1. I.e., $U[0]=1$ and $U[i]=0$ for every $i>0$.

2. If $<\text{Stop}, \text{StpNmbr}, \text{RghtShft}>[l]=<0, 1, ->$, then $FwdVst$ has verified that $z$ does not divide $u$ and that $u$ is different from 1. I.e., there are two distinct indexes $i$ and $j$ such that $U[i]=1$ and $U[j]=1$.

3. If $<\text{Stop}, \text{StpNmbr}, \text{RghtShft}>[l]=<B, -, 0>$ or $<\text{Stop}, \text{StpNmbr}, \text{RghtShft}>[l]=<B, -, 1>$, then $FwdVst$ has verified that $z$ divides at least $u$ at step zero, i.e. that $U[0]=0$. Simultaneously, $FwdVst$ also has checked if $z$ divides at least $u$. In case of positive answer $FwdVst$ bitwise added $G_1$ and $M$ in the course of its whole iteration.

Figure 6: Relevant combinations of $<\text{Stop}, \text{StpNmbr}, \text{RghtShft}>$ as given by $FwdVst$.

records that the result of $FwdVst$ must be subject to the implementation in TFA of Step 4 and 5 of BEA in Figure1.

To sum up, one of the goal of $FwdVst$ is to let the last element of $w_{FwdVstOutput}$ contain $<\text{Stop}, \text{StpNmbr}, \text{RghtShft}>$ in one of the three configurations of Figure 6.

Then, $w_{Rev}$ reverses the result of $FwdVst$ exchanging lsb and msb. Let us call $w_{BkwdVstInput}$ the threaded words $w_{FwdVstOutput}$ that $w_{BkwdVst}$ takes in input.

$BkwdVst$ behaves in accordance with the lsb of $w_{BkwdVstInput}$.

Let $w_{BkwdVstInput}$ be such that $<\text{Stop}, \text{StpNmbr}, \text{RghtShft}>[\text{lsb}]=<1, -, ->$ which, in accordance with Figure 6, implies that $u$ is 1. So, $G_1[\text{lsb}], \ldots, G_1[\text{msb}]$ contain the result of the inversion of $u$ and we must avoid any change on them. $BkwdVst$ reacts by filling every $Stop[i]$ of $w_{BkwdVstInput}$ with the value 1. This implements Step 3 of BEA.

Let $w_{BkwdVstInput}$ be such that $<\text{Stop}, \text{StpNmbr}, \text{RghtShft}>[\text{lsb}]=<0, 1, ->$. In accordance with Figure 6, we know that $z$ does not divide $u$ and that $u$ is different from 1. In this case $BkwdVst$ implements Step 4 and 5 of BEA in Figure 1. For every element $i$ of $w_{BkwdVstInput}$, it sets $U[i]$ with $\text{Xor} U[i]$ $V[i]$ and $G_1[i]$ with $\text{Xor} G_1[i]$ $G_2[i]$ until it eventually finds the least $j>=0$ such that $V[j]=1$ and $U[j]=0$. If $j$ exists, then $BkwdVst$ sets $V[i]$ with $\text{Xor} V[i]$ $U[i]$ and $G_2[i]$ with $\text{Xor} G_2[i]$ $G_1[i]$.

The last case is with $<\text{Stop}, \text{StpNmbr}, \text{RghtShft}>[\text{msb}]=<B, -, rs>$ with $rs$ different from B. We are in this case only when $FwdVst$ verified that one between (13) and (14) holds. Then, $BkwdVst$ erases the msb of $w_{BkwdVstInput}$. This is possible exactly because $BkwdVst$ builds on the programming pattern of the meta-combinator $w_{HeadTail[L,B]}$. Erasing the msb is equivalent to erase the lsb of $w_{FwdVstOutput}$. I.e., we realize the one-step shift to the right of $U$ and of one between $G_1$ or $G_1 + F$. Instead, while $V$, $G_2$ and $M$ which were shifted one place to the left survive the erasure.
Running example. Let us focus on (11) which we apply FwdVst to. FwdVst can check $U[0]=0$ and $G1[0]=1$ and determines that (14) holds. The result is:

\[
\begin{array}{cccccccccc}
\text{f}. \ x. \\
\text{# U V G1 G2 M Stop Stpbm RghtShft FwdV FwdG2 FwdM} \\
f <0,1, 0, 0, 1, B, B, 0, B, B, B> # msb \\
(f <0,0,Xor 0 0, 0, 1, B, 0, 0, 1, 0, 1> \\
(f <1,1,Xor 0 0, 0, 0, B, 0, 0, 0, 0, 0> (20) \\
(f <0,1,Xor 0 1, 0,1, B, 0, 0, 1, 0, 1> # new lsb \\
(f <0,B,Xor 1 1, 0,1, . B, 0, 0, 1, 0, 1> # orig. lsb \\
(x)))) \\
\end{array}
\]

The threaded words (20) is the input of $wRev$ giving the following instance of $wBkwdVstInput$:

\[
\begin{array}{cccccccccc}
\text{f}. \ x. \\
\text{# U V G1 G2 M Stop Stpbm RghtShft FwdV FwdG2 FwdM} \\
f <0,1,Xor 0 1, 0,1, B, B, 0, B, B, B> # orig. lsb \\
(f <0,1,Xor 0 1, 0,1, B, 0, 0, 1, 0, 1> # new lsb \\
(f <1,1,Xor 0 0, 0, 0, B, 0, 0, 0, 0, 0> (21) \\
(f <0,0,Xor 0 0, 0, 1, B, 0, 0, 1, 0, 1> \\
(f <0,1, 0, 0,1, B, B, 0, B, B, B> x))) \\
\end{array}
\]

BkwdVst applies to (21). It finds that Stop[0]=B and RghtShft[0]=0 which requires to shift all the bits of U and G1 one position to the their right. BkwdVst commits the requirement by erasing the topmost element of (21). The result is:

\[
\begin{array}{cccccccccc}
\text{f}. \ x. \\
\text{# U V G1 G2 M Stop Stpbm RghtShft FwdV FwdG2 FwdM} \\
f <0,1,Xor 0 1, 0,1, B, B, 0, B, B, B> \\
(f <0,0,Xor 0 0, 0, 1, B, 0, 0, 1, 0, 1> (22) \\
(f <0,1, 0, 0,1, B, B, 0, B, B, B> x))) \\
\end{array}
\]

Finally, $wRevInit$ reverses (22), yielding:

\[
\begin{array}{cccccccccc}
\text{f}. \ x. \\
\text{# U V G1 G2 M Stop Stpbm RghtShft FwdV FwdG2 FwdM} \\
f <0,1, 0, 0, 1, B, B, B, 0, 0, 0> \\
(f <0,0,Xor 0 0, 0, 1, B, B, 0, 0, 0> (23) \\
(f <1,1,Xor 0 0, 0, 0, B, B, 0, 0, 0> \\
(f <1,1,Xor 0 1, 0,1, B, B, 0, 0, 0> x))) \\
\end{array}
\]

Let us compare (23) and (20). All the bits of position U and G1 have been shifted while those ones of position V, G2 and M have not. Moreover, the bits of position Stop, ..., FwdM have been reinitialized so that (23) is a consistent input for FwdVst. We remark that the whole process of shifting the bits of positions U and G1 requires the concatenation of both FwdVst and BkwdVst up to some reverse. The first one shifts the bits of position V, G2 and M to the left while operates on those of position U and G1. The latter erases the correct element and fully realizes the shift to the right.
The code of FwdVst and of BkwdVst. We recall that FwdVst and BkwdVst follow the programming pattern of wHeadTail[L,B]. The step functions they rely on and their “last step functions” implement branching. Choices of the branching structures depend on the values of the bits that belong to the state or on the values of some bits of U or G1. We talk of pseudo-code because Figure 7 adds obvious syntactic sugar to the syntax of \( \lambda \)-terms. Let \( N \) be of type \( B_2 \). Then \( N M_1 \ M_0 \ MB \) is a \( \lambda \)-term which eventually chooses among \( M_1, M_0 \) and \( MB \), depending on the normal form \( N \) evaluates to. The switch-structure:

\[
\text{switch } (N) \quad \{
\text{case 1: } \ldots
\text{case 0: } \ldots
\text{case B: } \ldots
\}
\]

represent \( N M_1 M_0 MB \). The name of variables in the pseudo-code should recall their meaning. In Figure 7, \( \text{stopt} \) recalls “Stop of the tail”, i.e. “Stop that comes from step \( \text{msb} \equiv 1 \)”. Analogously \( \text{rst} \) is “RghtShft that comes from step \( \text{msb} \equiv 1 \)”. Let us focus on the two branches with \( \text{stopt} = B \) and \( \text{rst} = 1 \) or \( \text{rst} = 0 \). They take care of the situations that require the shift to the right of \( U \) and \( G1 \). I.e., if we think in general terms, they generate the two elements in (19). If we prefer to think in terms of our example, they generate the two topmost elements in (20). We remark that LastStepFwdVst is completely linear. Branching after branching it yields a \( \lambda \)-abstraction that correctly builds required elements that complete the threaded words under construction.

Figure 8 is a flow-chart that summarizes the essentials of the decision network that the pseudo-code of LastStepFwdVst in Figure 7 implements. Ellipses contain comments on the meaning of the variables along the possible branches. The names of variables in the flow-chart and in the pseudo-code correspond as follows: \( \text{stopt} \) is \( \text{Stop[msb]} \), \( \text{snt} \) is \( \text{StpNbmr[msb]} \) and \( \text{rst} \) is \( \text{RghtShft[msb]} \).

Decision networks analogous to the one in Figure 8 exist for all the components of \( \text{wInv} \). For example, Figure 9, 10, 11 and 12 summarize the essentials of the decision network that the step function \( \text{SFwdVst} \) (see Appendix C) of FwdVst implements. Again we have to trace how the names of variables in the flow-chart link to the names of variables of the pseudo-code correspond. If we assume we are at step \( i \), then \( \text{stopt} \) is \( \text{Stop[i-1]} \), \( \text{rst} \) is \( \text{RghtShft[i-1]} \), \( \text{uba}, \text{ubb} \) are \( U[i], gb \) is \( G1[i] \) and \( \text{sntb1}, \text{sntb2} \) are \( \text{StpNbmr[i]} \).

Typeability of \( \text{wInv} \). Let us recall that \( B_1^1 \equiv B_2 \otimes \ldots \otimes B_2 \) and \( L(B_1^1) \equiv \forall \alpha \exists \beta \ x : B_1^1 \). Figure 13 lists the types of the main components of \( \text{wInv} \). We remark that FwdVst, BkwdVst, LastStepFwdVst and \( \text{wRevInit} \) map a threaded words to another threaded words. So their composition can be used, as we do, as a step function in a iteration.

We do not detail out all the type derivations because quite impractical. Instead, we highlight the main reasons why the terms in Figure 13 have a type.

Both \( \text{MapThread[F]} \) and \( \text{wRevInit} \) are iterations that work at the lowest possible level of their syntactic components. Ideally, we can view \( \text{MapThread[F]} \) and \( \text{wRevInit} \) as adaptations and generalizations of the same programming pattern that \( \text{ussuc} \) relies on and whose type derivation is in Appendix B.

We already underlined that both FwdVst and BkwdVst adjust the programming pattern of \( \text{wHeadTail[L,B]} \) to our purposes. Appendix B recalls the type inference of \( \text{wHeadTail[L,B]} \) with \( L \) and \( B \) as in (7) which can be simply adapted to type FwdVst and BkwdVst. Mainly, FwdVst and BkwdVst use SFwdVst, BFwdVst, ... to find the right branch in decision networks like those ones in Figure 9 and 8. The main point to assure we can give a type to SFwdVst, BFwdVst, ...
LastStepFwdVst =
\f.
\<ft,et,t> . # Element from step i-1.
\(<ut,vt,glt,gt,mt,stopt,snt,rst,fwdvt,fwdgt,ftwmt> .
(switch (stopt) {
  case 1: # of stopt. We checked U=1. The whole wInv must be
  # the identity.
  \(<ft,vt,glt,gt,mt,stopt,snt,rst,fwdvt,fwdg2t,ftwmt,t> .
  (ft <ut,vt,glt,gt,mt,1,B,B,B,B,B> t)
  case 0: # of stopt. So we have also RghtShft=B and U[0]=1.
  switch (snt) {
    case 1: # of snt. U is different from 1.
      \(<ft,vt,glt,gt,mt,stopt,snt,rst,fwdvt,fwdg2t,ftwmt,t> .
      (ft <ut,vt,glt,gt,mt,0,1,B,B,B,B> t)
    case 0: # of snt. Here we detect that U=1 and we set Stop=1 !!!!
      \(<ft,vt,glt,gt,mt,stopt,snt,rst,fwdvt,fwdg2t,ftwmt,t> .
      (ft <ut,vt,glt,gt,mt,1,B,B,B,B,B> t)
    case B: # of snt. Can never occur.
      \(<ft,vt,glt,gt,mt,stopt,snt,rst,fwdvt,fwdg2t,ftwmt,t> .
      (ft <ut,vt,glt,gt,mt,0,B,B,B,B,B> t)
  }
  case B: # of stopt. We have U[0]=0 and RghtShft=0 or RghtShft=1.
  switch (rst) {
    case 1: # of rst. U[0]=0 and G1[0]=0. We are shifting and we
    # have to add a new msb to the threaded words.
      \(<ft,vt,glt,gt,mt,stopt,snt,rst,fwdvt,fwdg2t,ftwmt,t> .
      (ft <ut,0,0,gt,mt,B,B,0,B,B,B> t)
      case 0: # of rst. U[0]=0 and G1[0]=1. We are shifting and we
      # have to add a new msb to the threaded words.
      \(<ft,vt,glt,gt,mt,stopt,snt,rst,fwdvt,fwdg2t,ftwmt,t> .
      (ft <ut,0,0,gt,mt,B,B,0,B,B,B> t)
      case B: # of rst. Can never occur.
      \(<ft,vt,glt,gt,mt,stopt,snt,rst,fwdvt,fwdg2t,ftwmt,t> .
      (ft <ut,vt,glt,gt,mt,B,B,B,B,B,B> t)
  }
}) f ft ut vt glt gt mt stopt snt rst fwdvt fwdg2t ftwmt t
) et

Figure 7: Definition of LastStepFwdVst.
Figure 8: Flow-chart of the decision network that LastStepFwdVst implements.
Figure 9: Flow-chart of the decision network that the step function $S_{\text{FwdVst}}$ of $\text{FwdVst}$.
Figure 10: First component of the decision network that the step function SFwdVst of FwdVst implements.

Figure 11: Second component of the decision network that the step function SFwdVst of FwdVst implements.
Figure 12: Third component of the decision network that the step function $SF_{\text{FwdVst}}$ of $F_{\text{FwdVst}}$ implements.
is to organize them so that every possible choice results in a closed term. This maintains as much linear as we can the whole term, so letting it iterable and simply composable.

5. Conclusions and future work

We complete a project started in [13], whose one goal was to implement a library of potential real interest by using a language conceived in the ambit of Implicit Computational Complexity (ICC). We succeeded in spite of the widespread opinion that the expressivity of languages like the one we used is too weak to program anything interesting.

We introduce several functional programs (Map[·], Fold[·], MapState[·], MapThread[·], Add, wMod[·], wSqr, wMult and wInv). We worked on the programming patterns to show that they have types in TFA and in particular we implement the multiplicative inverse in a quite general way by giving it in a binary field of arbitrary, but fixed, degree. By the way, we remark that the existence of wInv in TFA gives an alternative proof that inversion has a polynomial cost.

In the course of this work we have remarked that programming with a language full of restrictions like TFA may be rewarding. In a follow up of this work, we are about providing evidence of such a statement: it is not at all difficult to port the algorithm of inversion we implemented in TFA, back to an imperative language. The result is a variant of the BEA which we call DCEA (DLAL Certified Euclidean Algorithm) with some structural regularity in the execution flow. In future work we plan to show that DCEA is competitive with BEA and in fact we have that it outperforms current implementations of BEA in some real world application like SSL.

On the other side we missed the development of a complete realistic applicative example, such as elliptic curves cryptography. In the same line, the implementation of symmetric-key cryptographic algorithms (block/stream ciphers, hash functions, . . . ) looks attractive, thanks to the higher-order bitwise operations at the core of the library.
Next, we shall investigate a compilation process targeting parallelization, which, in general follows from functional programming thanks to the reduced data dependency it embodies. This goal should be feasible because the lambda terms we write to implement finite fields arithmetic exploit programming patterns that can be assimilated to the MapReduce paradigm [16].

Finally, we do not exclude that more refined logics than DLAL can be used to realize a similar framework with even better built-in properties. Our choice of DLAL originated as a trade-off between flexibility in programming and constraints imposed by the typing system, but it is at the same time an experiment. Different logics can for instance measure the space complexity, or provide a more fine-grained time complexity.

References


URL http://projects.wizardlike.ca/attachments/2/LCC09_19.pdf


URL http://dx.doi.org/10.1142/S0129054100000077

URL http://dx.doi.org/10.1016/j.ic.2008.08.005

URL http://dx.doi.org/10.1109/LICS.2006.21


URL http://dx.doi.org/10.1145/1243996.1243997
Appendix A. Definition of Basic Combinators

We recall the following definitions from [13].

\(b\text{Cast}^m\) is \(\lambda b.1\ 0\ \bot\).

\(b\text{Var}_t\) is \(\lambda b.\overline{1,\ldots,1}\overline{0,\ldots,0}\overline{\bot,\ldots,\bot}\), for every \(t \geq 2\).

\(t\text{Cast}^m\) is, for every \(m \geq 0\):

\[
\begin{align*}
\text{tCast}^0 & \equiv \lambda a,b.a\ \text{aIsOne}\ a\text{IsZero}\ a\text{IsBottom}\ b \\
\text{aIsOne} & \equiv \lambda x.\overline{1}\ \overline{1}\ \overline{\bot}\ \\
\text{aIsZero} & \equiv \lambda x.\overline{0}\ \overline{1}\ \overline{\bot}\ \\
\text{aIsBottom} & \equiv \lambda x.\overline{\bot}\ \overline{1}\ \overline{\bot}\ \\
\text{tCast}^{m+1} & \equiv \lambda p.\overline{\bot}[\text{tCast}^m](\text{tCast}^m(p)) \\
\text{wSuc} & \equiv \lambda b.p.\lambda f.x.\overline{f}\ (\text{bCast}^m\ b)(p\ x).
\end{align*}
\]
\(\text{wCast}^m\) is, for every \(m \geq 0\):

\[
\begin{align*}
\text{wCast}^0 & \equiv \\ll 1.1\ (\text{wSuc}\ 0)\ (\text{wSuc}\ 1)\ (\text{wSuc}\ \bot)\ [e] \\
\text{wCast}^m+1 & \equiv \\ll 1.\$[\text{wCast}^m] (\text{wCast}^m 1).
\end{align*}
\]

\(\text{w\nabla}^m\), for every \(t \geq 2\), and \(m \geq 0\) is:

\[
\begin{align*}
\text{w\nabla}^0_t & \equiv \\ll 1.1\ (\text{w\nabla\nabla}\ 0)\ (\text{w\nabla\nabla}\ 1)\ \text{w\nabla\Base} \\
\text{w\nabla}^{m+1}_t & \equiv \\ll 1.\$[\text{w\nabla}^m_t] (\text{w\nabla}^m 1) \\
\text{w\nabla\Step} & \equiv \\ll b.\ \langle x_1 \ldots x_t \rangle <\text{wSuc}\ b\ x_1 \ldots \text{wSuc}\ b\ x_t > \\
\text{w\nabla\Base} & \equiv \langle [e] \ldots [e] \rangle .
\end{align*}
\]

\(\text{Xor}\ is\ \\ll b\ c\ \ll (x\ x\ 0\ 1)\ (\ll (x\ x\ 1\ 0)\ (\ll x\ x)\ c).
\)

\(\text{And}\ is\ \\ll b\ c\ (\ll (x\ x)\ (\ll (x\ x\ 0\ \bot)\ \bot)\ \bot)\ \bot\ c.
\)

\(\text{ssPl}\ is\ \ll s\ s\ (\ll t.\ (\ll [e])\ )\ )\ (\ll x\ x).
\)

\(\text{wRev}\ is\ \ll 1.1\ \text{wRev\Step}[f]\ (\ll x\ x)\ x\ \text{with}:
\[
\text{wRev\Step}[f] \equiv \ll e\ r\ x\ r\ (f\ e\ x) : B_2 \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha,\ \text{when} \\
 f : B_2 \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha.
\]
\)

\(\text{wDrop}\ is\ \ll 1.1\ (\ll (\ll e\ \ll (f\ f\ 1)\ (\ll (f\ f\ 0)\ (\ll f\ z\ z)\ f)\ x).
\)

\(\text{w2s}\ is\ \ll 1.1\ (\ll (\ll e\ \ll t\ c\ c)\ <e,\ s>)\ [e].
\)

\(\text{wProj}_1\ is\ \ll 1.1\ (\ll (\ll a\ b)\ .\ f\ a)\ x.
\)

\(\text{wProj}_2\ is\ \ll 1.1\ (\ll (\ll a\ b)\ .\ f\ b)\ x.
\)

\(\text{Map}[F]\ is\ \ll 1.1\ (\ll (\ll e\ f)\ F\ e)\ x,\ \text{with}\ F : A \rightarrow B\ \text{closed}.
\)

\(\text{Fold}[F, S]\ is\ \ll 1.1\ (\ll (\ll e\ z\ F\ e\ z)\ (\text{Cast}^* S),\ \text{with}\ F : A \rightarrow B \rightarrow B\ \text{and}\ S : B\ \text{closed}.
\)

\(\text{MapState}[F]\ is\ \ll 1.1\ \ll s\ f\ x, (\langle w, s'\rangle.\ m)(\ll \text{MS\Step}[F, f]\ (\text{MS\Base}[x]\ (\text{Cast}^* s)))\)

\(\text{with}\ F : (A \otimes S) \rightarrow (A \otimes S)\ \text{closed, and}:
\[
\begin{align*}
\text{MSStep}[F, f] & \equiv \ll e, \ll (w, s), (\ll (e', s'). <f, e' w, s'>) (F <e, s>) \\
\text{MSBase}[x] & \equiv \ll s, <x, s> .
\end{align*}
\]

\(\text{In}\ \text{particular}\ \text{MSStep}[F, f] : (A \otimes S) \rightarrow (A \otimes S) \rightarrow (A \otimes S)\ \text{and}\ \text{MSBase}[x] : S \rightarrow (A \otimes S).
\)

\(\text{MapThread}[F]\ is\)

\(\ll m\ f\ x, (\ll (\ll w, s).\ m)(\ll \text{MS\Step}[F, f]\ (\text{MS\Base}[x] (\ll \text{w2s}\ (\ll \text{w\nabla}\ f))))\ \text{with}\ F : B_2 \rightarrow B_2 \rightarrow A\ \text{closed,}\ \text{w2s}\ \ (\ll \text{w\nabla}\ f) : \$S\ \text{whenever}\ m : L_2\ \text{and}:
\[
\begin{align*}
\text{MSStep}[F, f] & \equiv \ll a, \ll (w, s), (\ll (b, s'). <f, (a b) w, s'>) (\text{ssPl} s) \\
\text{MSBase}[x] & \equiv \ll x, <x, m> .
\end{align*}
\]

\(\text{In}\ \text{particular}\ \text{MSStep}[F, f] : B_2 \rightarrow (A \otimes S) \rightarrow (A \otimes S)\ \text{and}\ \text{MSBase}[x] : A \rightarrow (A \otimes S).
\)
Appendix B. Some examples of type inference

Typing uSuc. A first example is the typing of the successor

\[ uSuc \equiv \ \lambda n. \ \lambda f. \ \lambda x. \ (f \ x) \ x \]

of Church numerals. We have \( uSuc : \mathbb{U} \rightarrow \mathbb{U} \), in accordance with the type inference:

\[
\begin{align*}
\emptyset \vdash n : \mathbb{U} & \quad \forall E \\
\emptyset \vdash \lambda n. \ \lambda f. \ \lambda x. \ (f \ x) \ x : \mathbb{U} \rightarrow \mathbb{U} & \quad \forall E \\
\emptyset \vdash \lambda n. \ \lambda f. \ \lambda x. \ (f \ x) \ x : \mathbb{U} \rightarrow \mathbb{U} & \quad \forall E
\end{align*}
\]

Few steps, required to conclude the typing, are missing on top of the rightmost occurrence of \( \neg E \). We leave finding them as a simple exercise.

Typing uSuc is interesting because it is a simple term that keeps dimension of the derivation acceptable, and shows how using the rule \( \exists E \), whose application is not apparent from the structure of uSuc itself. Similar use of \( \neg E \) occurs in typing tCast^m, wSuc, wCast^m, wRev, for example, and, more generally, whenever a \( \lambda \)-terms that results from an iteration becomes the argument of a function.

Typing a predecessor built on wHeadTail[L, B]. Let \( X \equiv (A \rightarrow \alpha \rightarrow \alpha) \otimes A \otimes \alpha \) and \( L(A) \equiv \forall \alpha. \ (A \rightarrow \alpha \rightarrow \alpha) \rightarrow \bot \). Let L and B be defined as in (7). This means that \( L : X \rightarrow \alpha \) and \( B : X \). The type assignment of \( \text{wHeadTail}[L, B] \) follows:

\[
\begin{align*}
\Pi_1 & \\
\Pi_2 & \\
\Pi_3 &
\end{align*}
\]

where \( \Pi_1 \) is:

\[
\emptyset \vdash w : \mathbb{L}(A) \quad \forall E
\]

and \( \Pi_2 \) is:

\[
\emptyset \vdash w : \mathbb{L}(A) \quad \emptyset \vdash \lambda x. \ \lambda y. \ (A \rightarrow \alpha \rightarrow \alpha) \rightarrow (A \rightarrow X \rightarrow X) \quad \emptyset \vdash \lambda x. \ \lambda y. \ (A \rightarrow \alpha \rightarrow \alpha) \rightarrow (A \rightarrow X \rightarrow X) \quad \emptyset \vdash \lambda x. \ \lambda y. \ (A \rightarrow \alpha \rightarrow \alpha) \rightarrow (A \rightarrow X \rightarrow X)
\]

and \( \Pi_3 \) is:

\[
\begin{align*}
\emptyset \vdash x : \alpha \rightarrow \alpha & \\
\emptyset \vdash \lambda x. \ \lambda y. \ (A \rightarrow \alpha \rightarrow \alpha) \rightarrow (A \rightarrow X \rightarrow X) & \\
\emptyset \vdash \lambda x. \ \lambda y. \ (A \rightarrow \alpha \rightarrow \alpha) \rightarrow (A \rightarrow X \rightarrow X) & \\
\emptyset \vdash \lambda x. \ \lambda y. \ (A \rightarrow \alpha \rightarrow \alpha) \rightarrow (A \rightarrow X \rightarrow X)
\end{align*}
\]
Appendix C. Pseudocode of the main components of wInv

FwdVst =
\tw. # Threaded words that FwdVst visits in forward direction.
   # In the main text we call it wFwdVstInput.
\f.\x. (LastStepFwdVst f) (tw (SFwdVst f) (BFwdVst x))

SFwdVst =
\f.
\<u,v,gl,g2,p,stop,sn,rs,fwdv,fwdg2,fwdm>.
\ft,et,t>.
()\<ut,vt,git,g2t,mt
 ,stopt,snt,rst,fwdvt,fwdg2t,fwdmt>. # Get the i-th element
\<u1,u2,ue>. # three copies of u[i]:
   # -) the first two for branching
   # -) one to be inserted in the list
\<gb,ge>. # two copies of u[i]:
   # -) one for branching
   # -) one to be inserted in the list
\<sntb1,sntb2>. # copies of sn[i-1] for branching

(switch (stopt) {
   case 1: # of stopt. We checked U=1. wInv must be
        # the identity
   \f.\u.\v.\gl.\g2.\m.
   \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
   \ut.\vt.\git.\g2t.\mt.
   \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
   \<f,\u,v,gl,g2,m,stop,sn,rs,fwdv,fwdg2,fwdm>
   ,ft <ut,vt,git,g2t,mt,0,B,rst,fwdvt,fwdg2t,fwdmt> t>
   case 0: # of stopt. We are at a step>0 and we know
        # U[0]=1. We do not have to shift anything
   switch (uba) {
      case 1: # of uba. U contains at least two occurrences
        # of 1. I.e. U[0]=1, U[j]=1 and j>0
   \f.\u.\v.\gl.\g2.\m.
   \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
   \ut.\vt.\git.\g2t.\mt.
   \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
   \<f,\u,v,gl,g2,m # Values from this step.
   ,0 # Stop keeps recording that U[0]=1
   ,1 # StepNumber keeps recording we are at step>0
        # It also signals U[0]=1, U[j]=1 and j>0,
        # This means the whole U!=1
   ,B # RightShift keeps recording that
        # neither of U, Gl shift
        # I.e. z does not divide U and Gl
   ,B,B,B > # Dummy values.
   ,ft <ut,vt,git,g2t,mt,0,snt,rst,fwdvt,fwdg2t,fwdmt> t>
   case 0: # of uba.
   switch (sntbi) {

case 1: # of sntb1.
<\text{f},<0,v,g1,g2,m>
,\text{0} # Values from this step.
,\text{0} # Stop keeps recording \(U[0]=1\)
,\text{1} # StepNumber keeps recording we are at step>0
 # It also signals \(U[0]=1\), \(U[j]=1\) and \(j>0\),
,\text{B} # RightShift keeps recording neither
 # of \(U,G_1\) shift
 # I.e. \(z\) does not divide \(U\) ad \(G_1\)
,B,B,B > # Dummy values
,\text{ft} <\text{ut},\text{vt},g1t,g2t,mt,0,1,rst,fwdvt,fwdg2t,fwdmt> t>

case 0: # of sntb1.
<\text{f},<0,v,g1,g2,m>
,\text{0} # Stop keeps recording \(U[0]=1\)
,\text{0} # StepNumber keeps recording we are at step>0
 # We do not know whether \(U!=1\) or \(U=1\) yet
,B # RightShift keeps recording that neither
 # of \(U,G_1\) shift i.e. \(U[0]=1\)
,B,B,B > # Dummy values.
,\text{ft} <\text{ut},\text{vt},g1t,g2t,mt,0,0,rst,fwdvt,fwdg2t,fwdmt> t>

} # switch of sntb1 end

} # switch uba end.

} # switch stopt. We are at step 0

switch (sntb2) {
case 1: # Cannot occur. As soon as one of the
 # previous cases sets StpNmbr[j]=1,
 # for some \(j<i-1\), then Stop[k]=0,
 # for every \(k>j\)
<\text{f},<0,v,g1,g2,m>
,\text{0} # Values from this step.
,\text{0} # Stop keeps recording \(U[0]=1\)
,\text{1} # StepNumber keeps recording we are at step>0
 # It also signals \(U[0]=1\), \(U[j]=1\) and \(j>0\),
,\text{B} # RightShift keeps recording neither
 # of \(U,G_1\) shift
 # I.e. \(z\) does not divide \(U\) ad \(G_1\)
,B,B,B > # Dummy values.
,\text{ft} <\text{ut},\text{vt},g1t,g2t,mt,0,0,rst,fwdvt,fwdg2t,fwdmt> t>

} # switch of sntb2 end
case 0: # of sntb2. We are at step>0
switch (rst) {
  case 0: # of rst. U and G1 shift to the right.
    # I.e. U[0]=0, G1[0]=0
    \snt\rst\fwdvt\fwdg2t\fwdmt\t.
    \f,\u,\v,\g1,\g2,\m,\0,\1,\B,B,B,B >
    ,ft <ut,\v,\g1t,\g2t,\mt,\0,\1,\rst,\fwdvt,\fwdg2t,\fwdmt> t>
  case 1: # of rst. U and G1+F shift to the right
    # I.e. U[0]=0, G1[0]=1
    \snt\rst\fwdvt\fwdg2t\fwdmt\t.
    \f,\u,\v,\g1,\g2,\m,\0,\1,\r1,\fwdvt,\fwdg2t,\fwdmt> t>
  case B: # Neither of U, G1 shift to the right.
\f.\u.\v.\g1.\g2.\m. \\stop.\sn.\rs.\fwdv.\fwdg2.\fwdm. \\ut.\vt.\glt.\g2t.\mt. \\snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
\f,<1,v,g1,g2,p
 ,0 # Stop keeps storing that U[0]=1
 ,0 # StepNumber keeps recording
   # we are at step>0
 ,B # RightShift keeps recording that
   # neither of U, G1 shift
 ,B,B,B > # dummy values
 ,ft <ut,vt,glt,g2t,mt,B,0,B,fwdvt,fwdg2t,fwdmt> t>
} # switch rst end.
case B: # of sntb2. We are at step 0
   # We must check the value of U[lsb], G1[lsb]
switch (ubb) {
case 1: # of ubb. z does not divide U.
   # I.e. U[0]=1. Moreover, U may be 1.
   # I.e. the only bit equal to 1 is U[0]
\f.\u.\v.\g1.\g2.\m. \\stop.\sn.\rs.\fwdv.\fwdg2.\fwdm. \\ut.\vt.\glt.\g2t.\mt. \\snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
\f,<1,v,g1,g2,m
 ,0 # Stop records that U[0]=1
 ,0 # StepNumber 'increases' by 1
 ,B # RightShift records that neither of U, G1 shift
 ,B,B,B > # dummy values
 ,ft <ut,vt,glt,g2t,mt,B,B,rst,fwdvt,fwdg2t,fwdmt> t>
case 0: # of ubb. z divides U i.e. U[0]=0
switch (gb) {
case 1: # of gb. z does not divide G1, i.e. G1[0]=1.
\f.\u.\v.\g1.\g2.\m. \\stop.\sn.\rs.\fwdv.\fwdg2.\fwdm. \\ut.\vt.\glt.\g2t.\mt. \\snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
(\<me1,me2>. # two copies of m to build elements
\f,<0
 ,B # Dummy value. This is the lsb of V.
   # We shall erase it
 ,Xor 1 me1
 ,B # Dummy value. This is G2[lsb]
   # We shall erase it
 ,B # Dummy value. This is the M[lsb].
   # We shall erase it
 ,B # Forward Stop which records that U[0]=0
 ,0 # Forward StepNumber
 ,0 # Forward RightShift which records
   # that U, G1=F must shift
 ,v # Forward the three bits that
# must shift to the left
,g2
,me2 >
,ft <ut,vt,g1t,g2t,mt,B,B,rst,fwdvt,fwdg2t,fwdmt> t>
) (m <1,1> <0,0> <B,B>)
case 0: # of gb. z divides G1, i.e. G1[0]=0
\f.u.v.g1.g2.m.
\stop.sn.rs.fwdv.fwdg2.fwdm.
\ut vt.gl t.g2t.m t.
\snt rst.fwdvt.fwdg2t.fwdmt.t t.
<f,<0
,B # Dummy value. This is V[lsb].
 # We shall erase it
,0
,B # Dummy value. This is G2[lsb].
 # We shall erase it
,B # Dummy value. This is M[lsb].
 # We shall erase it
,B # Forward Stop which records that U[0]=0
,0 # Forward StepNumber
,1 # Forward RightShift which records
 # that U, G1 must shift.
,v # Forwarding the three bits that
 # must shift to the left
,g2
,m >
,ft <ut,vt,g1t,g2t,mt,B,B,rst,fwdvt,fwdg1t,fwdm> t>
case B: # of gb can never happen
\f.u.v.g1.g2.m.
\stop.sn.rs.fwdv.fwdg2.fwdm.
\ut vt.gl t.g2t.m t.
\snt rst.fwdvt.fwdg2t.fwdmt.t t.
<f,<0,v,B,g2,m,B,B,B,B,B,B >
,ft <ut,vt,g1t,g2t,mt,B,B,rst,fwdvt,fwdg2t,fwdmt> t>
)# switch of gb end
case B: # of ubb.
\f.u.v.g1.g2.m.
\stop.sn.rs.fwdv.fwdg2.fwdm.
\ut vt.gl t.g2t.m t.
\snt rst.fwdvt.fwdg2t.fwdmt.t t.
<f,B,v,B,g2,m,B,B,B,B,B >
,ft <ut,vt,g1t,g2t,mt,B,B,rst,fwdvt,fwdg2t,fwdmt> t>
)# switch ubb end
)# switch stntb2 end
)# switch of stopt
)f # is the 'virtual' successor of the threaded words given
# as output. It must be used linearly, after we choose
# what to do on the threaded words. Analogously to f,
# after we choose what to do on the threaded words, we
# use linearly (a copy) ue (of u), v, g1, g2, p, fwdv,
# fwdgb and fwdp.

ue v ge g2 m stop sn rs fwdv fwdg2 fwdm
ut vt glt g2t mt snt rst fwdvt fwdg2t fwdmt t

) (u <1,1,1> <0,0,0> <B,B,B>) # The first copy of u[i] may
# serve for branching. The
# second one serves to build a
# new state. The first copy of
# gl[i] may serve for branching.
# The second one serves to
# build a new state.

(snt <1,1> <0,0> <B,B>) # Both copies of sn[i-1] serve
# for branching.

) et

BFwdVst =
\x.<(\w.(z.z),<B # This is U[0]
,\B # This is V[0]
,\B # This is G1[0]
,\B # This is G2[0]
,\B # This is P[0]
,\B # This is Stop[0]
,\B # This is StpNmbr[0]. We are at step 0
,\B # This is RghtShft[0]
,\B # This is FvdV[0]
,\B # This is FvdG2[0]
,\B # This is FvdF[0]
> ,x>

BkwdVst =
\tw. # Threaded words that BkwdVst visits in backward direction.
   # In the main text we call it wBkwdVstInput.
\f.\x. (LastStepBkwdVst f) (tw (SBkwdVst f) (BBkwdVst x))

BBkwdVst =
\x.<(\w.(z.z),<B # This is U[0].
,\B # This is V[0].
,\B # This is G1[0].
,\B # This is G2[0].
,\B # This is M[0].
,\B # This is Stop[0].
,\B # This is StpNmbr[0].
,\B # This is RghtShft[0].
,\B # This is FvdV[0].
,\B # This is FvdG2[0].
,\B # This is FvdF[0].
> ,x>

SBkwdVst =
\f.
\(\langle u, v, g_1, g_2, m, \text{stop}, sn, rs, \ldots \rangle\).

\(\langle ft, et, t \rangle\).

\(\langle u, t, v, t, g_1t, g_2t, mt, \text{stop}, \text{snt}, \text{rst}, \ldots \rangle\).

\(\text{(switch (stop)}) \{
\text{case 1:} \# \text{ of stop} \text{t} \text{ means } U=1. \text{ Keep propagating Stop}=1
\\langle u, t, v, g_1t, g_2t, m, \text{stop}, \text{snt}, \text{rst}, \ldots \rangle.
\text{<f, u, v, g_1t, g_2t, m,}
\text{,1 \# Propagation of Stop}=1.
\text{,sn, rs, B, B, B> \# Dummy values.}
\}\text{,stop, t, snt, rst, B, B> x>}
\text{case 0:} \# \text{ of stop} \text{t}. \text{ So } U[0]=1, U[1]=1. \text{ Keep executing}
\text{# Step 4, 5 of BEA. StepNumber keeps recording}
\text{# the relation between deg(U), deg(V)}
\text{switch (rst) \{
\text{case 1:} \# \text{ of rst. deg(U)} < deg(V) detected.
\\langle u, v, g_1, g_2, m, \text{stop}, \text{snt}, \text{rst}, \ldots \rangle.
\text{<f, u, v, g_1, g_2, m,}
\text{,0 \# Propagate stop}=0.
\text{,1 \# Propagate deg(U) < deg(V).}
\text{,rs, B, B, B>}
\text{,ft <u, v, t, g_1, g_2, mt, 0, sn, rs, B, B> t>}
\text{)} (u <1,1> <0,0> <B, B>) (g_1 <1,1> <0,0> <B, B>)
\text{case 0:} \# \text{ of rst. deg(U)} > deg(V) detected.
\\langle u, v, g_1, g_2, m, \text{stop}, \text{snt}, \text{rst}, \ldots \rangle.
\text{<f, u, v, g_1, g_2, m,}
\text{,0 \# Propagate deg(U) > deg(V).}
\text{,rs, B, B, B>}
\text{,ft <u, v, t, g_1, g_2, mt, 0, sn, rs, B, B> t>}
\text{)} (v <1,1> <0,0> <B, B>) (g_2 <1,1> <0,0> <B, B>)
\text{case B:} \# \text{ of rst. Relation between deg(U), deg(V)}
\text{# still unknown}
\\langle u, v, g_1, g_2, m, \text{stop}, \text{snt}, \text{rst}, \ldots \rangle.
\text{<f, u, v, g_1, g_2, m,}
\text{,0 \# Propagate Stop}=0.
\text{,B \# Set StepNumber=B to propagate that the)
\text{# relation between deg(U) and deg(V)}
\text{# is unknown}
\text{,rs, B, B, B>}
\text{,ft <u, v, t, g_1, g_2, mt, 0, sn, rs, B, B, B> t>}
\text{)} (v <1,1> <0,0> <B, B>) (g_2 <1,1> <0,0> <B, B>)
\text{)} # \text{ switch rst}
case B: # of stopt
switch (rst) {
case 1: # of rst. So U[0]=0. Keep propagating
    # RightSihft=1. The last step will compute
    # the predecessor of the input threaded words
    # to implement the shift to the right U and one
    # between G1 or G1+F
    \u.\v.\g1.\g2.\m.\stop.\sn.\rs.
    \ut.\vt.\glt.\g2t.\mt.\go.\snt.\rst.
    <f,<u,v,g1,g2,m,
        B # Propagation of Stop=B.
        ,sn # Dummy value.
        ,1 # Keep propagating RightShift=1 which implies
        # we shall calculate the predecessor on the
        # threaded words in input
        ,B,B,B> # Dummy values.
        ,ft <ut,vt,glt,g2t,mt,B,snt
        ,1 # Propagates the previous value of RightShift
        ,B,B,B> x>
case 0: # of rst. Never occurs because the base case, i.e.
    # stopt=B and rst=B and Stop=B, sets RightShift=1
    # which the case here above with rst=1 keeps
    # propagating. This is not a mistake because it is
    # important to calculate the predecessor in the
    # course of the very last step
    \u.\v.\g1.\g2.\m.\stop.\sn.\rs.
    \ut.\vt.\glt.\g2t.\mt.\go.\snt.\rst.
    <f,<u,v,g1,g2,m,
        B # Propagation of Stop=B
        ,sn # Dummy value.
        ,0 # Keep propagating RightShift=0 which implies we
        # shall calculate the predecessor on the threaded
        # words in input
        ,B,B,B> # Dummy values.
        ,ft <ut,vt,glt,g2t,mt,B,snt
        ,0 # Propagates RightShift=0 from the previous step
        ,B,B,B> x>
case B: # of rst.
    # Base case. Start propagating the relevant bits
switch (stop) {
case 1: # of stop. So U=1. The iteration must be
    # an identity. We start propagating Stop=1
    \u.\v.\g1.\g2.\m.\stop.\sn.\rs.
    \ut.\vt.\glt.\g2t.\mt.\go.\snt.\rst.
    <f,<u,v,g1,g2,m,
        ,1 # Propagation of Stop=1.
        ,B,B,B,B> # Dummy values.
        ,ft <ut,vt,glt,g2t,mt,stop,snt,sn,rs,B,B,B> x>
case 0: # of stop. I.e. U[0]=1, U!=1.

38
# Start executing Step 4, 5 of BEA
# Need to compare u and v
switch (u) {
  case 1: # of u
    switch (v) {
      case 1: # of v
        \u \v \g1 \g2 \m \stop \sn \rs.
        \ut \vt \glt \g2t \mt \stopo \snt \rst.
        (\<g2a,g2b>.
          <f,\xor 1 1,1,\xor \g1 \g2a,\g2a,\m
          ,0 # Propagate Stop=0
          ,B # StepNumber=0 says we do not know
          # the relation between deg(U), deg(V)
          ,rs,B,B,B>
          ,ft <ut,vt,glit,g2t,mt,stopo,snt,rst,B,B,B> t>
        ) (g2 <1,1> <0,0> <B,B>)
      case 0: # of v
        \u \v \g1 \g2 \m \stop \sn \rs.
        \ut \vt \glt \g2t \mt \stopo \snt \rst.
        (\<g2a,g2b>.
          <f,\xor 0 0,0,\xor \g1 \g2a,\g2b,\m
          ,0 # Propagate Stop=0
          ,0 # StepNumber=0 records deg(U)>deg(V)
          ,rs,B,B,B>
          ,ft <ut,vt,glit,g2t,mt,stopo,snt,rst,B,B,B> t>
        ) (g2 <1,1> <0,0> <B,B>)
      case B: # of v. Never occurs.
        SBkwVst45NeverOccurs
    } # switch v
  case 0: # of u
    switch (v) {
      case 1: # of v
        \u \v \g1 \g2 \m \stop \sn \rs.
        \ut \vt \glt \g2t \mt \stopo \snt \rst.
        (\<g1a,g1b>.
          <f,\xor 1 0,0,\xor \g2 \g1a,\g1b,\m
          ,0 # Propagate Stop=0
          ,1 # StepNumber=0 records deg(U)<deg(V)
          ,rs,B,B,B>
          ,ft <ut,vt,glit,g2t,mt,stopo,snt,rst,B,B,B> t>
        ) (g1 <1,1> <0,0> <B,B>)
      case 0: # of v. I.e. deg(U)=deg(V)
        \u \v \g1 \g2 \m \stop \sn \rs.
        \ut \vt \glt \g2t \mt \stopo \snt \rst.
        (\<g2a,g2b>.
          <f,\xor 0 0,0,\xor \g1 \g2a,\g2b,\m
          ,0 # Propagate stop=0
          ,B # StepNumber=B propagates we do not know
          # the relation between deg(U),deg(V)
          ,rs,B,B,B>
          ,ft <ut,vt,glit,g2t,mt,stopo,snt,rst,B,B,B> t>
        ) (g2 <1,1> <0,0> <B,B>)
      case B: # of v. Always occurs. (Non-occurrence)
    } # switch v
} # switch u

LastStepBkwVst =
\langle f,e,t \rangle.
(\langle u,v,g1,g2,m,_,_,_,_,_,_,_ \rangle).
( switch (stop) {
  case 1: # of stop says that U=1. Do nothing
   \langle u,v,g1,g2,m,f <u,v,g1,g2,m,B,B,B,B,B,B \rangle t
  case 0: # of stop. Conclude an iteration that
   # implements Step 4 and 5 of BEA
   switch (rs) {
     case 1: # of rs. deg(U)<deg(V) detected
      .} # switch stop
     .} # switch rst
  .} # switch stopt
  .} # switch v
  .} # switch u
  case B: # of stop. So U[0]=0. Start propagating
  # RightShift=1. The last step will compute the
  # predecessor of the input list
  # to implement the shift to the right of U and
  # one between G1 or G1+F.
  \langle u,v,g1,g2,m,stop,sn,rs,u,v,g1,g2,m,stop,sn,rs,B,B,B \rangle t
  .} # switch v
  .} # switch u
  .} # switch B
  .} # Propagation of Stop=B.
  .} # Dummy values.
  .} # Propagation RightShift=1. I.e. we shall calculate
  # the predecessor on the threaded words in input.
  # The predecessor realizes the shift to the right.
  # Propagating 0 in place of 1 would yield the
  # same result
  .} # Propagation of Stop=B.
  .} # Dummy values.
  .} # Propagation of Stop=B.
  .} # Dummy values.
case 0: # of rs. deg(U)>deg(V) detected.

\begin{align*}
\text{case } B: \text{ # of rs. We know deg(U)=deg(V)}
\end{align*}

\text{case B: # of stop. Conclude an iteration that must}\n\text{ implement a shift to the right. Do not insert}\n\text{ the last element of the threaded list. I.e.,}\n\text{ calculate the predecessor}\n\text{ case B: # switch rs}\n\text{ case B: # switch stop}\n\)

\text{e}

\text{wRevInit} = \text{wRevInitS f} \text{ wRevInitB}

\text{wRevInitS} = \text{f. } \text{e. } \text{\langle } \text{u, v, g1, g2, m, stop, \ldots, \ldots, \ldots, \rangle } .
\text{ (\text{e. } \text{r. } \text{z. } \text{r } \text{f } \text{\langle } \text{u, v, g1, g2, m, stop, B, 0, 0, 0, z,\rangle } } \text{ e}

\text{wRevInitB} = \text{x} . \text{x}