Light combinators for finite fields arithmetic

Light combinators for finite fields arithmetic

D. Canavese[1], E. Cesena[2], R. Ouchary[1], M. Pedicini[3], L. Roversi[4]

Abstract

This work completes the definition of a library which provides the basic arithmetic operations in binary finite fields as a set of functional terms with very specific features. Such a functional terms have type in Typeable Functional Assembly (TFA). TFA is an extension of Dual Light Affine Logic (DLAL). DLAL is a type assignment designed under the prescriptions of Implicit Computational Complexity (ICC), which characterises polynomial time costing computations.

We plan to exploit the functional programming patterns of the terms in the library to implement cryptographic primitives whose running-time efficiency can be obtained by means of the least hand-made tuning as possible.

We propose the library as a benchmark. It fixes a kind of lower bound on the difficulty of writing potentially interesting low cost programs inside languages that can express only computations with predetermined complexity. In principle, every known and future ICC compliant programming language for polynomially costing computations should supply a simplification over the encoding of the library we present, or some set of combinators of comparable interest and difficulty.

We finally report on the applicative outcome that our library has and which is a reward we get by programming in the very restrictive scenario that TFA provides. The term of TFA which encodes the inversion in binary fields suggested us a variant of a known and efficient imperative implementation of the inversion itself given by Fong. Our variant, can outperform Fong's implementation of inversion on specific hardware architectures.

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

Email addresses: daniele.canavese@polito.it (D. Canavese), ec@theneeds.com (E. Cesena), rachid.ouchary@polito.it (R. Ouchary), pedicini@mat.uniroma3.it (M. Pedicini), roversi@di.unito.it (L. Roversi)

[1] Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy

[2] Theneeds Inc., San Francisco, CA

[3] Università degli Studi Roma Tre, Dipartimento di Matematica e Fisica, Roma, Italy

[4] Università degli Studi di Torino, Dipartimento di Informatica, Torino, Italy

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. $u \leftarrow u + v$, $g_1 \leftarrow g_1 + g_2$.
6. Goto Step 2.
```

where z is the standard name of the independent variable of the polynomial basis representation of the finite filed $\mathbb{F}_{2^m}$ of order 2^m and a, u, v, g_1 and g_2 are polynomials.

Figure 1: Binary-Field inversion as in **Algorithm 2.2** at page 1048 in [3].

1. Introduction

This work completes a first step of a project which started in [1]. The long term goal was, and still is, to exploit functional programming patterns which can express only algorithms with predetermined complexity — typically polynomial time one — to implement cryptographic libraries whose running-time efficiency can be obtained by means of the least hand-made tuning as possible. We recall that hand-crafted tuning can be quite onerous because, for example, it must be tailored on the length of the word in the given running architecture.

Since we express the above polynomial time costing algorithms in a language whose computational complexity is controlled by means of implicit features, this work mainly contributes to the area of implicit computational complexity.

One contribution is pretty technical. This paper extends the set of functional programs, as given in [1]. In there we implement the arithmetic operations subtraction, multiplication, squaring and square root on binary finite fields. The novelty of this work is multiplicative inverse.

Considered that the operations on binary finite fields constitute the core of cryptographic primitives, this work supplies a library with potential real applicative interest inside a so called light complexity programming language, something not quite usual. The language we adopt is a fragment of pure λ-calculus whose terms we can type by means of the type assignment system TFA (Typed Functional Assembly). TFA, defined in [1], is a slight extension of Dual Light Affine Logic [2]. The multiplicative inverse we define here is a λ-term we call `wInv` and which encodes the algorithm BEA in Figure 1.

When trying to give a type to non obvious combinators inside TFA, like the above operations are, the main obstacle is to apply the standard *divide-et-impera* paradigm because of computational complexity limitations. Once a problem that a combinator must solve has been successively split into simpler ones until they become trivial, the composition of the partial results cannot always proceed in the obvious way; the λ-terms with a type in TFA incorporate mechanisms that force to preserve bounds on their computational complexity. For example, if we supply the output of a sub-problem that an iteration produces as the input of another iteration, then we may get a computational complexity blowup. For example, this is why the naive manipulation of lists, for example, that we represent as λ-terms in TFA can rapidly "degrade" to situations where composition, which would be natural in standard functional programming, simply gets forbidden.

Due to the above limitations the pure λ-terms typeable with TFA and implementing finite field operations are not always the natural ones we could write. We mean that we followed as much as we could common ideas like those ones in [4] which advocate the use of standard functional programming patterns like *map*, *map thread*, *fold* to make functional programs more readable and reliable.

However, those patterns cannot always naturally apply inside TFA and they only partially mitigated our programming difficulties.

In particular, the coding of BEA as the λ-term `wInv` is quite involved. It requires to generalise the functional programming pattern that leads to the definition of the predecessor of Church numerals, or similar structures, in Light Affine Logic [5, 6], an ancestor of DLAL, hence of TFA. Let us call it light predecessor pattern.

Our second contribution comes exactly from the need of using the non standard light predecessor pattern to implement BEA in `wInv`. The contribution is somewhat of philosophical nature. It keeps nourishing the debate about how and if intuitionistic deductive systems similar to TFA identify interesting functional programming languages inside pure λ-calculus or alike.

The structural complexity of `wInv` doubtlessly argues against any possibility of exploiting TFA-like systems for every day programming even for specialists.

However, we have arguments that can support the other perspective as well. Writing programs with current light programming languages, even with the most "primitive" ones, may have rewards whose relevance still requires full assessment.

We told that the encoding of BEA as `wInv` relies on the light predecessor patterns which is specific of type assignments that come from Light Affine Logic. The relevance of a new programming pattern, or abstraction, may not be immediately evident. For example, the `MapReduce` paradigm have been exploited as in [7] far after its introduction which, morally, occurs in [8]. Of course, we are not supporting the idea that light predecessor pattern is, or will be, as relevant as `MapReduce`! However, the work [9], which we see as a natural companion of this one, helps pursuing the idea that something interesting in connection with light predecessor pattern exists. In [9] we show that the design of `wInv` in fact suggests to rewrite BEA in Figure 1 in a new imperative algorithm DCEA. We do not recall it here. Suffice it to say that DCEA rearranges the statements in BEA. On standard architectures, under the same optimisations, the speed of `C` implementations of BEA and DCEA are comparable with a slight prevalence of BEA. Instead, on `ARM` architectures, under the same optimisations, DCEA can be

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 2: Library for binary-field arithmetic

up to 20% faster than BEA. Fully investigation of why this happens is on-going work.

However, on one side, reporting on non obvious programming examples, like the one we develop with `wInv`, is a contribution that may renew the interest about the search of improvements on what we know on functional programming and on their implementations. On the other, rephrasing an anonymous referee, the library we supply becomes a first linguistic benchmark which future light programming languages should refer to when the intensional completeness of a light language to program with is among the design goals.

Structure of this work. Section 2 recalls TFA from [1]. Section 3 supplies the two bottommost layers in Figure 2 recalling them from [1]. Section 4 supplies the second topmost layer in Figure 2. One part comes from [1]. The content of Subsection 4.5, namely the description of `wInv`, is new.

Appendix A details out the definition of the combinators in Section 3, of which a very prototypical implementation is available for public download[5].

Also, we have manually checked that all terms have types in DLAL. Some type inference can be found in [2, 10]. Appendix B has some further typing examples.

Finally, Appendix B gives pseudo-code details of `wInv`.

2. Typeable Functional Assembly

We call Typeable Functional Assembly (TFA) the deductive system in Figure 3. Its rules come from Dual Light Affine Logic (DLAL) [2]. “Assembly” as part of the name comes from our programming experience inside TFA. When programming inside TFA the goal is twofold. Writing the correct λ-term and

[5] https://github.com/pis147879/TFA-wInv. It is necessary to have Wolfram Mathematica or an interpreter for its language.

$$\frac{}{\emptyset \mid \mathtt{x}\!:\!A \vdash \mathtt{x}\!:\!A}\,\text{a} \qquad \frac{\Delta \mid \Gamma \vdash \mathtt{M}\!:\!A}{\Delta, \Delta' \mid \Gamma, \Gamma' \vdash \mathtt{M}\!:\!A}\,\text{w} \qquad \frac{\Delta, \mathtt{x}\!:\!A, \mathtt{y}\!:\!A \mid \Gamma \vdash \mathtt{M}\!:\!B}{\Delta, \mathtt{z}\!:\!A \mid \Gamma \vdash \mathtt{M}\{{}^{\mathtt{z}}/_{\mathtt{x}}\,{}^{\mathtt{z}}/_{\mathtt{y}}\}\!:\!B}\,\text{c}$$

$$\frac{\Delta \mid \Gamma, \mathtt{x}\!:\!A \vdash \mathtt{M}\!:\!B}{\Delta \mid \Gamma \vdash \backslash\mathtt{x.M}\!:\!A \multimap B} \multimap\text{I} \qquad \frac{\Delta \mid \Gamma \vdash \mathtt{M}\!:\!A \multimap B \quad \Delta' \mid \Gamma' \vdash \mathtt{N}\!:\!A}{\Delta, \Delta' \mid \Gamma, \Gamma' \vdash \mathtt{M}\,\mathtt{N}\!:\!B} \multimap\text{E}$$

$$\frac{\Delta, \mathtt{x}\!:\!A \mid \Gamma \vdash \mathtt{M}\!:\!B}{\Delta \mid \Gamma \vdash \backslash\mathtt{x.M}\!:\,!A \multimap B} \Rightarrow\text{I} \qquad \frac{\Delta \mid \Gamma \vdash \mathtt{M}\!:\,!A \multimap B \quad \emptyset \mid \Delta' \vdash \mathtt{N}\!:\!A \quad |\Delta'| \leq 1}{\Delta, \Delta' \mid \Gamma \vdash \mathtt{M}\,\mathtt{N}\!:\!B} \Rightarrow\text{E}$$

$$\frac{\emptyset \mid \Delta, \Gamma \vdash \mathtt{M}\!:\!A}{\Delta \mid \S\Gamma \vdash \mathtt{M}\!:\!\S A} \S\text{I} \qquad \frac{\Delta \mid \Gamma \vdash \mathtt{N}\!:\!\S A \quad \Delta' \mid \mathtt{x}\!:\!\S A, \Gamma' \vdash \mathtt{M}\!:\!B}{\Delta, \Delta' \mid \Gamma, \Gamma' \vdash \mathtt{M}\{{}^{\mathtt{N}}/_{\mathtt{x}}\}\!:\!B} \S\text{E}$$

$$\frac{\Delta \mid \Gamma \vdash \mathtt{M}\!:\!A \quad \alpha \notin \mathrm{fv}(\Delta, \Gamma)}{\Delta \mid \Gamma \vdash \mathtt{M}\!:\!\forall\alpha.A} \forall\text{I} \qquad \frac{\Delta \mid \Gamma \vdash \mathtt{M}\!:\!\forall\alpha.A}{\Delta \mid \Gamma \vdash \mathtt{M}\!:\!A[{}^{B}/_{\alpha}]} \forall\text{E}$$

where the pairs Δ, Δ' and Γ, Γ' give type to disjoint sets of variables in $\mathcal{V}$.

Figure 3: Type assignment system TFA

lowering their computational complexity so that the λ-term gets typeable. It generally results in λ-terms that work at a very low level in a style which recalls the one typical of programming Turing machines.

Every judgment $\Delta \mid \Gamma \vdash \mathtt{M}\!:\!A$ has two different kinds of context Δ and Γ, a formula A and a λ-term $\mathtt{M}$. The judgment assigns A to $\mathtt{M}$ with hypothesis from the *polynomial context* Δ and the *linear context* Γ. “Assembly” should make it apparent that λ-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:

$$\mathcal{F} ::= \mathcal{G} \mid \mathcal{F} \multimap \mathcal{F} \mid\, !\mathcal{F} \multimap \mathcal{F} \mid \forall\mathcal{G}.\mathcal{F} \mid \S\mathcal{F} \ .$$

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

The λ-term $\mathtt{M}$ belongs to Λ, the λ-calculus given by:

$$\Lambda ::= \mathcal{V} \mid (\backslash\mathcal{V}.\Lambda) \mid (\Lambda\,\Lambda) \ . \tag{1}$$

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

$$(\backslash\mathtt{x.M})\,\mathtt{N} \rightarrow \mathtt{M}\{{}^{\mathtt{N}}/_{\mathtt{x}}\} \ . \tag{2}$$

Its reflexive, transitive, and contextual closure is $\rightarrow^*$. Since $\rightarrow^*$ is Church-Rosser, while considering λ-terms-as-programs, confluence ensures that no ambiguity can arise in the result of any computation.

Both polynomial and linear contexts are maps $\{\mathtt{x_1} : A_1, \ldots, \mathtt{x_n} : A_n\}$ from variables $\mathcal{V}$ to formulas. Variables of any polynomial context may occur an arbitrary number of times in the *subject* $\mathtt{M}$ of the judgment $\Delta \mid \Gamma \vdash \mathtt{M}\!: A$. Every variable in the linear context must occur at most once in $\mathtt{M}$. The notation $\S\Gamma$ is a shorthand for $\{\mathtt{x_1}\!:\!\S A_1, \ldots, \mathtt{x_n}\!:\!\S A_n\}$, if Γ is $\{\mathtt{x_1}\!:\!A_1, \ldots, \mathtt{x_n}\!:\!A_n\}$.

There are *formula schemes* relevant for our purposes.

Let us define the following scheme:

$$\mathbb{B}_n \equiv \forall\alpha.\, \overbrace{\alpha \multimap \cdots \multimap \alpha}^{n+1} \multimap \alpha \ \ .$$

If we set $n = 2$, we get the formula we can assign to the canonical representatives of “lifted” booleans:

$$\mathtt{1} \equiv \backslash\mathtt{xyz.x} : \mathbb{B}_2 \qquad\qquad \mathtt{0} \equiv \backslash\mathtt{xyz.y} : \mathbb{B}_2 \qquad\qquad \bot \equiv \backslash\mathtt{xyz.z} : \mathbb{B}_2 \ \ .$$

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

Another useful scheme is:

$$(A_1 \otimes \ldots \otimes A_n) \equiv \forall\alpha.\, A_1 \multimap \cdots \multimap A_n \multimap \alpha \ \ ,$$

which we shorten as $(\otimes^n A)$ whenever $A_1 = \ldots = A_n$ and which justifies we introduce tuples as part of TFA. This means adding:

$$\Lambda ::= \ldots \mid <\Lambda, \ldots, \Lambda> \mid \backslash<\mathcal{V}, \ldots, \mathcal{V}>.\Lambda$$

to Definition (1), then extending β-reduction with:

$$(\backslash<\mathtt{x_1}, \ldots, \mathtt{x_n}>.\mathtt{M})\ <\mathtt{N_1}, \ldots, \mathtt{N_n}> \rightarrow \mathtt{M}\{^{\mathtt{N_1}}/_{\mathtt{x_1}}, \ldots, {}^{\mathtt{N_n}}/_{\mathtt{x_n}}\}$$

and finally showing that the following rules are derivable:

$$\frac{\Delta_1 \mid \Gamma_1 \vdash \mathtt{M_1}\!: A_1 \quad \ldots \quad \Delta_n \mid \Gamma_n \vdash \mathtt{M_n}\!: A_n}{\Delta_1, \ldots, \Delta_n \mid \Gamma_1, \ldots, \Gamma_n \vdash <\mathtt{M_1}, \ldots, \mathtt{M_n}>\!: (A_1 \otimes \ldots \otimes A_n)} \otimes\mathtt{I}$$

$$\frac{\Delta \mid \Gamma, \mathtt{x_1}\!: A_1, \ldots, \mathtt{x_n}\!: A_n \vdash \mathtt{M}\!: B}{\Delta \mid \Gamma \vdash \backslash<\mathtt{x_1}, \ldots, \mathtt{x_n}>.\mathtt{M}\!: (A_1 \otimes \ldots \otimes A_n) \multimap B} \multimap\mathtt{I}_\otimes \ \ .$$

In fact, the way we derive the here above rules implies that:

$<\mathtt{M_1}, \ldots, \mathtt{M_n}>$ is an abbreviation of $\backslash\mathtt{x.x}\ \mathtt{M_1} \ldots \mathtt{M_n}$ and
$\backslash<\mathtt{x_1}, \ldots, \mathtt{x_n}>.\mathtt{M}$ is an abbreviation of $\backslash\mathtt{p.p}\,(\backslash\mathtt{x_1}. \ldots (\backslash\mathtt{x_n}.\mathtt{M}))$.

The final crucial recursive scheme is:

$$\mathbb{S} \equiv \forall\alpha.(\mathbb{B}_2 \multimap \alpha) \multimap ((\mathbb{B}_2 \otimes \mathbb{S}) \multimap \alpha) \multimap \alpha \ \ . \tag{3}$$

Let the symbol $\approx$ denote the congruence on the set $\mathcal{F}$ of formulas, which is defined as the reflexive, symmetric, transitive and contextual closure of (3).

By definition, $\mathcal{F}/\approx$ is the set of *types* that we denote as $\mathcal{T}$. We shall assign types to λ-terms, and not "only" formulas. This means that, for any $\mathtt{M}$, if $\mathtt{M} : \mathbb{S}$, then, in fact, we can also use $\mathtt{M} : \forall\alpha.(\mathbb{B}_2 \multimap \alpha) \multimap ((\mathbb{B}_2 \otimes \mathbb{S}) \multimap \alpha) \multimap \alpha$ or $\mathtt{M} : \forall\alpha.(\mathbb{B}_2 \multimap \alpha) \multimap ((\mathbb{B}_2 \otimes (\forall\alpha.(\mathbb{B}_2 \multimap \alpha) \multimap ((\mathbb{B}_2 \otimes \mathbb{S}) \multimap \alpha) \multimap \alpha)) \multimap \alpha) \multimap \alpha$ or ....

The scheme (3) is the type of Sequences of booleans, or simply Sequences, with canonical representatives:

$$\begin{aligned} [\varepsilon] &\equiv \backslash\mathtt{tc.t}\,\bot : \mathbb{S} \\ [\mathtt{b_{n-1}} \ldots \mathtt{b_0}] &\equiv \backslash\mathtt{tc.c}\,\mathtt{<}\mathtt{b_{n-1}}, [\mathtt{b_{n-2}} \ldots \mathtt{b_0}]\mathtt{>} : \mathbb{S}\ . \end{aligned} \tag{4}$$

In accordance with (3), the Sequence $[\mathtt{b_{n-1}} \ldots \mathtt{b_0}]$ 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 $\mathtt{b_{n-1}}$, and whose second element is — recursively! — another Sequence $[\mathtt{b_{n-2}} \ldots \mathtt{b_0}]$. The recursive definition of $\mathbb{S}$ should be evidently crucial.

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

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 $\mathtt{x} : A$ of any kind of context as *type assignment for a variable.*

2.1. Summing up

TFA is DLAL [2] 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 [6, 2]. 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 [1], we recall the meaning and the type of the λ-terms that forms the *two* lowermost layers in Figure 2. We also recall their definition in Appendix A.

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

$$\frac{\emptyset \mid \emptyset \vdash \mathtt{M} : A \multimap B}{\emptyset \mid \emptyset \vdash \S[\mathtt{M}] : \S A \multimap \S B}\ \S\mathrm{L}$$

where $\S[\mathtt{M}] \equiv \backslash\mathtt{x.M\,x}$ is the *paragraph lift of* $\mathtt{M}$. An obvious generalisation is that n consecutive applications of the $\S\mathrm{L}$ rule define a lifted term $\S^n[\mathtt{M}] \equiv$

$\backslash\mathtt{x}.\ldots(\backslash\mathtt{x}.\mathtt{M}\,\mathtt{x})\ldots\mathtt{x}$, that contains n nested $\S[\cdot]$. Its type is $\S^n A \multimap \S^n B$. Borrowing terminology from proof nets, the application of n paragraph lift of M *embeds* it in n paragraph boxes, leaving the behavior of $\mathtt{M}$ unchanged:

$$\S^{\mathtt{n}}[\mathtt{M}]\,\mathtt{N} \to^* \mathtt{M}\,\mathtt{N}.$$

3.1. Basic Definitions and Types

Church numerals. They have type:

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

with canonical representatives:

$$\mathtt{u}\varepsilon \equiv \backslash\mathtt{fx.x} : \mathbb{U} \qquad \overline{n} \equiv \backslash\mathtt{fx.f}\,(\ldots(\mathtt{f}\,\mathtt{x})\ldots) : \mathbb{U} \text{ with } n \text{ occurrences of } f$$

They iterate the first argument on the second one. We use $\mathtt{u}\varepsilon$ in place of $\mathtt{0}$ because we like to look at Church numerals as they were degenerate lists, of which $\mathtt{0}$ is the neuter element.

Lists. They have type:

$$\mathbb{L}(A) \equiv \forall\alpha.\mathbb{L}(A)[\alpha] \text{ where } \mathbb{L}(A)[\alpha] \equiv\, !(A \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$$

with canonical representatives:

$$\{\varepsilon\} \equiv \backslash\mathtt{fx.x} : \mathbb{L}(A)$$
$$\{\mathtt{M}_{\mathtt{n-1}}\ldots\mathtt{M}_{\mathtt{0}}\} \equiv \backslash\mathtt{fx.f}\ \mathtt{M}_{\mathtt{n-1}}\,(\ldots(\mathtt{f}\ \mathtt{M}_{\mathtt{0}}\,\mathtt{x})\ldots) : \mathbb{L}(A) \text{ with } n \text{ occurrences of } f$$

that generalise the iterative structures of Church numerals.

Church words. A Church word is a list $\{\mathtt{b}_{\mathtt{n-1}}\ldots\mathtt{b}_{\mathtt{0}}\}$ whose elements $\mathtt{b}_{\mathtt{i}}$s are booleans, i.e. of type $\mathbb{L}_2 \equiv \mathbb{L}(\mathbb{B}_2)$. By convention, in every Church word $\{\mathtt{b}_{\mathtt{n-1}}\ldots\mathtt{b}_{\mathtt{0}}\}$, or simply *word*, the *least significant bit* (lsb) is $\mathtt{b}_{\mathtt{0}}$, while the *most significant bit* (msb) is $\mathtt{b}_{\mathtt{n-1}}$. The same convention holds for every Sequence $[\mathtt{b}_{\mathtt{n-1}}\ldots\mathtt{b}_{\mathtt{0}}]$.

The combinator $\mathtt{bCast}^m : \mathbb{B}_2 \multimap \S^{m+1}\mathbb{B}_2$. It casts a boolean inside $m+1$ paragraph boxes, without altering the boolean:

$$\mathtt{bCast}^{\mathtt{m}}\,\mathtt{b} \to^* \mathtt{b}.$$

The combinator $\mathtt{b}\nabla_t : \mathbb{B}_2 \multimap \otimes^t\,\mathbb{B}_2$*, for every* $t \geq 2$. It produces t copies of a boolean:

$$\mathtt{b}\nabla_{\mathtt{t}}\,\mathtt{b} \to^* <\overbrace{\mathtt{b},\ldots,\mathtt{b}}^{\mathtt{t}}>.$$

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

The combinator $\mathtt{tCast}^m : (\mathbb{B}_2 \otimes \mathbb{B}_2) \multimap \S^{m+1}(\mathbb{B}_2 \otimes \mathbb{B}_2)$*, for every* $m \geq 0$*.* It casts a pair of bits into $m+1$ paragraph boxes, without altering the structure of the pair:

$$\mathtt{tCast}^{\mathtt{m}} \, \mathtt{<b_0, b_1>} \rightarrow^* \mathtt{<b_0, b_1>}.$$

The combinator $\mathtt{wSuc} : \mathbb{B}_2 \multimap \mathbb{L}_2 \multimap \mathbb{L}_2$*.* It implements the *successor* on Church words:

$$\mathtt{wSuc}\ \mathtt{b}\ \{\mathtt{b_{n-1}} \ldots \mathtt{b_0}\} \rightarrow^* \{\mathtt{b}\ \mathtt{b_{n-1}} \ldots \mathtt{b_0}\}\,.$$

The combinator $\mathtt{wCast}^m : \mathbb{L}_2 \multimap \S^{m+1}\mathbb{L}_2$*, for every* $m \geq 0$*.* It embeds a word into $m+1$ paragraph boxes, without altering the structure of the word:

$$\mathtt{wCast}^{\mathtt{m}}\ \{\mathtt{b_{n-1}} \ldots \mathtt{b_0}\} \rightarrow^* \{\mathtt{b_{n-1}} \ldots \mathtt{b_0}\}\,.$$

The combinator $\mathtt{w}\nabla^m_t : \mathbb{L}_2 \multimap \S^{m+1}(\otimes^t \mathbb{L}_2)$*, for every* $t \geq 2, m \geq 0$*.* It produces t copies of a word embedding the result into $m+1$ paragraph boxes:

$$\mathtt{w}\nabla^{\mathtt{m}}_{\mathtt{t}}\ \{\mathtt{b_{n-1}} \ldots \mathtt{b_0}\} \rightarrow^* \mathtt{<}\overbrace{\{\mathtt{b_{n-1}} \ldots \mathtt{b_0}\}, \ldots, \{\mathtt{b_{n-1}} \ldots \mathtt{b_0}\}}^{\mathtt{t}}\mathtt{>}.$$

3.2. Core Library

The combinator $\mathtt{Xor} : \mathbb{B}_2 \multimap \mathbb{B}_2 \multimap \mathbb{B}_2$*.* It extends the *exclusive or* as follows:

$$\begin{array}{lll}
\mathtt{Xor}\,\mathtt{0}\,\mathtt{0} \rightarrow^* \mathtt{0} & \mathtt{Xor}\,\mathtt{1}\ \mathtt{1} \rightarrow^* \mathtt{0} & \\
\mathtt{Xor}\,\mathtt{0}\,\mathtt{1} \rightarrow^* \mathtt{1} & \mathtt{Xor}\,\mathtt{1}\ \mathtt{0} \rightarrow^* \mathtt{1} & \\
\mathtt{Xor}\bot\,\mathtt{b} \rightarrow^* \mathtt{b} & \mathtt{Xor}\,\mathtt{b}\,\bot \rightarrow^* \mathtt{b} & \text{(where } \mathtt{b} : \mathbb{B}_2\text{).}
\end{array}$$

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 $\mathtt{And} : \mathbb{B}_2 \multimap \mathbb{B}_2 \multimap \mathbb{B}_2$*.* It extends the combinator *and* as follows:

$$\begin{array}{lll}
\mathtt{And}\,\mathtt{0}\,\mathtt{0} \rightarrow^* \mathtt{0} & \mathtt{And}\,\mathtt{1}\ \mathtt{1} \rightarrow^* \mathtt{1} & \\
\mathtt{And}\,\mathtt{0}\,\mathtt{1} \rightarrow^* \mathtt{0} & \mathtt{And}\,\mathtt{1}\ \mathtt{0} \rightarrow^* \mathtt{0} & \\
\mathtt{And}\bot\,\mathtt{b} \rightarrow^* \bot & \mathtt{And}\,\mathtt{b}\,\bot \rightarrow^* \bot & \text{(where } \mathtt{b} : \mathbb{B}_2\text{).}
\end{array}$$

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

The combinator $\texttt{sSpl} : \mathbb{S} \multimap (\mathbb{B}_2 \otimes \mathbb{S})$. It *splits* the sequence it takes as input in a pair with the m.s.b. and the corresponding tail:

$$\texttt{sSpl}\,[\mathtt{b_{n-1}} \ldots \mathtt{b_0}] \to^* \mathtt{<}\mathtt{b_{n-1}}, [\mathtt{b_{n-2}} \ldots \mathtt{b_0}]\mathtt{>}.$$

The combinator $\texttt{wRev} : \mathbb{L}_2 \multimap \mathbb{L}_2$. It *reverses the bits* of a word:

$$\texttt{wRev}\ \{\mathtt{b_{n-1}} \ldots \mathtt{b_0}\} \to^* \{\mathtt{b_0} \ldots \mathtt{b_{n-1}}\}\,.$$

The combinator $\texttt{wDrop}\bot : \mathbb{L}_2 \multimap \mathbb{L}_2$. It *drops* all the (initial) occurrences[6] of $\bot$ in a word:

$$\texttt{wDrop}\bot\ \{\bot \ldots \bot\ \mathtt{b_{n-1}} \ldots \mathtt{b_0}\} \to^* \{\mathtt{b_{n-1}} \ldots \mathtt{b_0}\}\,.$$

The combinator $\texttt{w2s} : \mathbb{L}_2 \multimap \S\mathbb{S}$. It translates a word into a sequence:

$$\texttt{w2s}\ \{\mathtt{b_{n-1}} \ldots \mathtt{b_0}\} \to^* [\mathtt{b_{n-1}} \ldots \mathtt{b_0}].$$

Its type inference is in Appendix B.

The combinator $\texttt{wProj}_1 : \mathbb{L}(\mathbb{B}_2^2) \multimap \mathbb{L}_2$. It *projects* the first component of a list of pairs:

$$\texttt{wProj}_1\ \{\mathtt{<}\mathtt{a_{n-1}}, \mathtt{b_{n-1}}\mathtt{>} \ldots \mathtt{<}\mathtt{a_0}, \mathtt{b_0}\mathtt{>}\} \to^* \{\mathtt{a_{n-1}} \ldots \mathtt{a_0}\}\,.$$

Similarly, $\texttt{wProj}_2 : \mathbb{L}(\mathbb{B}_2^2) \multimap \mathbb{L}_2$ projects the second component.

3.2.1. Meta-combinators

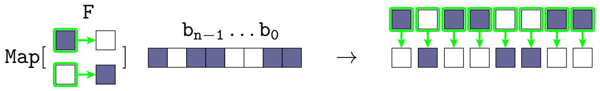
First we recall the meta-combinators from [1]. We used them to implement addition, modular reduction, square and multiplication in layer three of Figure 2.

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

Meta-combinators are λ-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 λ-terms.

The meta-combinator $\texttt{Map}[\cdot]$.. Let $\texttt{F} : A \multimap B$ be a closed term. Then, $\texttt{Map}[\texttt{F}] : \mathbb{L}(A) \multimap \mathbb{L}(B)$ applies F to every element of the list that Map[F] takes as argument, and yields the final list, assuming $\texttt{F}\,\mathtt{b_i} \to^* \mathtt{b'_i}$, for every $0 \leq i \leq n-1$:

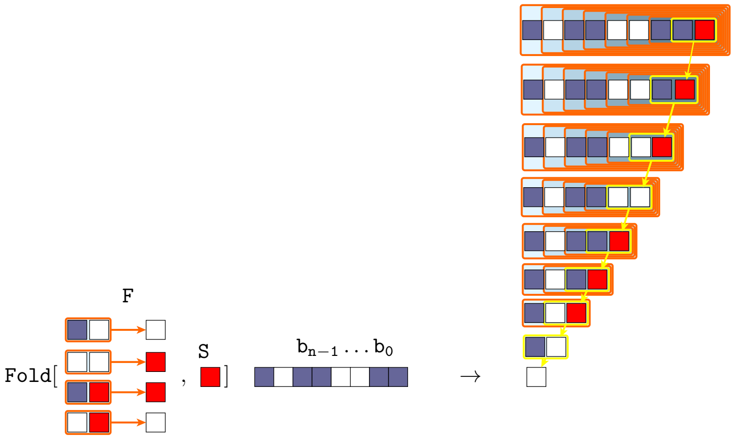
$$\texttt{Map}[\texttt{F}]\ \{\mathtt{b_{n-1}} \ldots \mathtt{b_0}\} \to^* \{\mathtt{b'_{n-1}} \ldots \mathtt{b'_0}\}\,.$$



[6] The current definition actually drops all the occurrences of $\bot$ in a Church word, however we shall only apply $\texttt{wDrop}\bot$ to words that contain $\bot$ in the most significant bits.

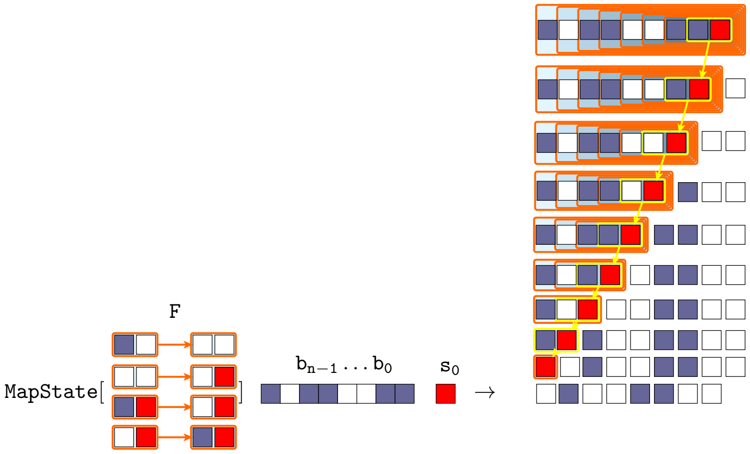
The meta-combinator $\mathtt{Fold}[\cdot,\cdot]$. Let $\mathtt{F} : A \multimap B \multimap B$ and $\mathtt{S} : B$ be closed terms. Then, $\mathtt{Fold}[\mathtt{F},\mathtt{S}] : \mathbb{L}(A) \multimap \S B$, starting from the initial value $\mathtt{S}$, iterates $\mathtt{F}$ over the input list and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

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



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

$$\mathtt{MapState}[\mathtt{F}]\ \{\mathtt{b_{n-1}}\dots\mathtt{b_0}\}\,\mathtt{s_0} \rightarrow^* \left\{\mathtt{b'_{n-1}}\dots\mathtt{b'_0}\right\}\ .$$

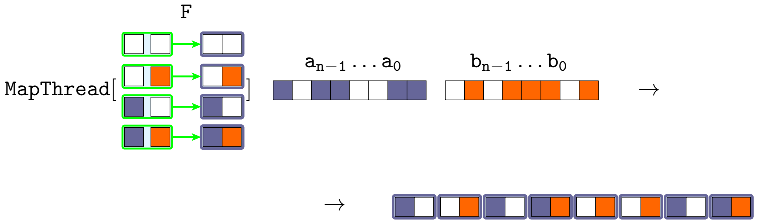


The meta-combinator $\mathtt{MapThread}[\cdot]$. Let $\mathtt{F} : \mathbb{B}_2 \multimap \mathbb{B}_2 \multimap A$ be a closed term. Then, $\mathtt{MapThread}[\mathtt{F}] : \mathbb{L}_2 \multimap \mathbb{L}_2 \multimap \mathbb{L}(A)$ applies $\mathtt{F}$ to the elements of the two input lists which must have equal lengths. Specifically, if $\mathtt{F}\,\mathtt{a_i}\,\mathtt{b_i} \rightarrow^* \mathtt{c_i}$, for every $0 \leq i \leq n-1$:

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

In particular, $\mathtt{MapThread}[\backslash\mathtt{ab}.\texttt{<}\mathtt{a},\mathtt{b}\texttt{>}] : \mathbb{L}_2 \multimap \mathbb{L}_2 \multimap \mathbb{L}(\mathbb{B}_2^2)$ is such that:

$$\mathtt{MapThread}[\backslash\mathtt{ab}.\texttt{<}\mathtt{a},\mathtt{b}\texttt{>}]\ \{\mathtt{a_{n-1}} \ldots \mathtt{a_0}\}\ \{\mathtt{b_{n-1}} \ldots \mathtt{b_0}\} \rightarrow^* \{\texttt{<}\mathtt{a_{n-1}},\mathtt{b_{n-1}}\texttt{>} \ldots \texttt{<}\mathtt{a_0},\mathtt{b_0}\texttt{>}\} \quad .$$



The meta-combinator $\mathtt{wHeadTail[L,B]}$. It has two parameters $\mathtt{L}$ and $\mathtt{B}$ and builds on the core mechanism of the predecessor for Church numerals [5, 6] inside typing systems like TFA. For any types A, α, let $X \equiv (A \multimap \alpha \multimap \alpha) \otimes A \otimes \alpha$. By definition, $\mathtt{wHeadTail[L,B]}$ is as follows:

$$\begin{aligned}
\mathtt{wHeadTail[L,B]} &\equiv \backslash\mathtt{w\ f\ x.L\ (w\ (wHTStep[B]\ f)\ (wHTBase\ x))} \\
\mathtt{wHTStep[B]} &\equiv \backslash\mathtt{f\ e.}\backslash\texttt{<}\mathtt{ft,et,t}\texttt{>}.\mathtt{B} \qquad\qquad (5)\\
\mathtt{wHTBase} &\equiv \backslash\mathtt{x.}\texttt{<}\backslash\mathtt{e\ l.l,ErsblEl,x}\texttt{>} \quad ,
\end{aligned}$$

where:

- $\mathtt{L}$ must be a closed λ-term with type $X \multimap \alpha$. It is the last step we apply after the iteration driven by $\mathtt{w}$ concludes.
- $\mathtt{wHTStep[B]}$ is a step function with type $(A \multimap \alpha \multimap \alpha) \multimap A \multimap X \multimap X$ and body $\mathtt{B}$.
- The body $\mathtt{B}$ of the step function $\mathtt{wHTStep[B]}$ is such that:
  - $\mathtt{B}$ has type X and
  - every of $\mathtt{f}$, $\mathtt{e}$, $\mathtt{ft}$, $\mathtt{et}$ and $\mathtt{t}$ must occur at most once free in $\mathtt{B}$.
- $\mathtt{wHTBase}$ is the base function with type $\alpha \multimap X$.

Appendix B gives type to $\mathtt{wHeadTail[L,B]}$. We now focus on the behavior of $\mathtt{wHeadTail[L,B]}$: $\mathbb{L}(A) \multimap \mathbb{L}(A)$ once applied to the list $\backslash\mathtt{g\,y.g\,b\,(g\,a\,y)}$ with $\mathtt{a}$ and $\mathtt{b}$ closed λ-terms that play the role of elements of the list of type A:

$$\begin{aligned}
&\mathtt{wHeadTail[L,B]}\ \ (\backslash\mathtt{g\,y.g\ b\ (g\ a\ y)}) \\
&\rightarrow^* \backslash\mathtt{f\,x.L\ (wHTStep[B]\ f\ b\ (wHTStep[B]\ f\ a\ (wHTBase\ x)))} \qquad\qquad (6)\\
&\rightarrow^* \backslash\mathtt{f\,x.L}\ ((\backslash\texttt{<}\mathtt{ft,et,t}\texttt{>}.\mathtt{B}\{^{\mathtt{f}}/_{\mathtt{f}}\}\{^{\mathtt{b}}/_{\mathtt{e}}\}) \\
&\qquad\qquad ((\backslash\texttt{<}\mathtt{ft,et,t}\texttt{>}.\mathtt{B}\{^{\mathtt{f}}/_{\mathtt{f}}\}\{^{\mathtt{a}}/_{\mathtt{e}}\})\ \texttt{<}\backslash\mathtt{e\ l.l,ErsblEl,x}\texttt{>})) \quad .
\end{aligned}$$

It iterates of `wHTStep[B]` from (`wHTBase x`). The term `ErsblEl`, which stands for "erasable element", can always be different from any other possible list element for the set we can choose the list of elements from is finite. By letting `ErsblEl` distinguishable from any other element, the rightmost occurrence of `B` in (6) knows that the iteration is performing its initial stepand it can operate on `a` as consequence of this fact. More generally, with `B` we can identify an initial sequence of iteration steps with predetermined length, say n. Then, `B` can operate on the first n elements of the list in a specific way. Moreover, the computation pattern that `wHeadTail[L,B]` develops is that `B` can have simultaneous stepwise access to two consecutive elements in the list. For example, `B` in (6) can use `a` and `ErsblEl` at step zero. At step one it has access to `b` and `et` and the latter may contain `a` or some element derived from it. This invariant is crucial to implement a bitwise forwarding mechanism of the state in the term of TFA that implements the multiplication inverse. For example, if we assume:

$$\begin{aligned} \texttt{L} &\equiv \backslash\texttt{<\_,\_,l>.l} \\ \texttt{B} &\equiv \texttt{<f,e,ft et t>} \quad , \end{aligned} \tag{7}$$

then we can implement a λ-term that pops the last element out of the input list. We can check this by assuming (7) in the λ-terms of (6) which yields `\f x.f a x`.

BEA, implemented as a term of TFA, relies on some variants of the meta-combinator `wHeadTail[L,B]`.

4. TFA Combinators for Binary-Fields Arithmetic

In this section we introduce those λ-terms of TFA which implement basic operations of the third layer in Figure 2; amongst them, inversion yields the most elaborated construction built as a variant of the meta-combinator `wHeadTail`.

Let us recall some essentials on binary-fields arithmetic (See [11, Section 11.2] for wider details). Let $p(X) \in \mathbb{F}_2[X]$ be an irreducible polynomial of degree n over $\mathbb{F}_2$, and let β be a root of $p(X)$ in the algebraic closure of $\mathbb{F}_2$. Then, the finite-field $\mathbb{F}_{2^n} \simeq \mathbb{F}_2[X]/(p(X)) \simeq \mathbb{F}_2(\beta)$.

The set of elements $\{1, \beta, \ldots, \beta^{n-1}\}$ is a basis of $\mathbb{F}_{2^n}$ as a vector space over $\mathbb{F}_2$ and we can represent a generic element of $\mathbb{F}_{2^n}$ as a polynomial in β of degree lower than n:

$$\mathbb{F}_{2^n} \ni a = \sum_{i=0}^{n-1} a_i \beta^i = a_{n-1}\beta^{n-1} + \cdots + a_1\beta + a_0 \ , \qquad a_i \in \mathbb{F}_2 \ .$$

Moreover, the isomorphism $\mathbb{F}_{2^n} \simeq \mathbb{F}_2[X]/(p(X))$ allows us to implement the arithmetic of $\mathbb{F}_{2^n}$ relying on the arithmetic of $\mathbb{F}_2[X]$ and reduction modulo $p(X)$.

Since every $a_i \in \mathbb{F}_2$ can be encoded as a bit, we can represent each element of length n in $\mathbb{F}_{2^n}$ 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 $\mathbb{F}_{2^n}$, instead than $\mathbb{L}_2$, as type. So, $\mathbb{L}_2$, and $\mathbb{F}_{2^n}$ becomes essentially interchangeable.

A first basic notation is $\boldsymbol{n}$. It denotes the Church numeral that stands for $n = \deg p(X)$. A second notation is $\boldsymbol{p}$. It is the Church word $\boldsymbol{p} \equiv \left\{ \mathtt{p_n} \dots \mathtt{p_0} \underbrace{\bot \dots \bot}_{n-1} \right\}$. Every $\mathtt{p_i}$ is the boolean term that encodes the coefficient p_i of $p(X) = \sum p_i X^i$.

A first final remark is that $\boldsymbol{p}$ has length $2n$. The occurrences of $\bot$ in the least significant positions serve for technical reasons.

A second final remark is about the way we must think of using the combinators we are going to introduce in the coming subsections. They build the arithmetic operations of a given binary finite field which we must identify by fixing the characterising parameters $\boldsymbol{n}$ and $\boldsymbol{p}$. Once given the two parameters, the combinators for addition, multiplication, etc. behave consequently.

4.1. Addition

Let $a, b \in \mathbb{F}_{2^n}$. The addition $a + b$ is computed component-wise, namely setting $a = \sum_{i=0}^{n-1} a_i \beta^i$ and $b = \sum_{i=0}^{n-1} b_i \beta^i$, then $a + b = \sum_{i=0}^{n-1} (a_i + b_i) \beta^i$. The sum $(a_i + b_i)$ 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 $\mathtt{Add} : \mathbb{F}_{2^n} \multimap \mathbb{F}_{2^n} \multimap \mathbb{F}_{2^n}$ is:

$$\mathtt{Add} \equiv \mathtt{MapThread}[\mathtt{Xor}] \ . \tag{8}$$

4.2. Modular Reduction

Reduction modulo $p(X)$ is a fundamental building block to keep the size of the operands constrained. Once fixed $\boldsymbol{n}$ and $\boldsymbol{p}$, Modular Reduction is applied to the result of the multiplication we shall define in Subsection 4.4. Multiplication always yields $2n$ bits as result. Modular Reduction transforms it in the correct n-long sequence of bits.

We implemented a naïve left-to-right Modular Reduction under the following two mandatory assumptions: (1) both $p(X)$ and $n = \deg p(X)$, which are fixed, are parameters and (2) the length of the input is $2n$ which can be rearranged by using n repetitions of a basic iteration.
The combinator $\mathtt{wMod}[\boldsymbol{n}, \boldsymbol{p}] : \mathbb{L}_2 \multimap \S\mathbb{F}_{2^n}$ is:

$$\mathtt{wMod}[\boldsymbol{n}, \boldsymbol{p}] \equiv$$

$$\backslash\mathtt{d}.\S\,[\mathtt{wModEnd}]\,(\boldsymbol{n}\,(\backslash\mathtt{l}.\mathtt{MapState}[\mathtt{wModFun}]\,\mathtt{l}\,{<}\bot, \mathtt{0}{>})\,(\mathtt{wModBase}[\boldsymbol{p}]\,(\mathtt{wCast}^{0}\,\mathtt{d})))$$

where:

$$\mathtt{wModEnd} \equiv \backslash\mathtt{l}.\mathtt{wDrop}\bot\,(\mathtt{wRev}\,(\mathtt{wProj}_1\,\mathtt{l}))$$

$$\mathtt{wModFun} \equiv \backslash{<}\mathtt{e}, \mathtt{s}{>}.(\backslash{<}\mathtt{d}, \mathtt{p}{>}.((\backslash{<}\mathtt{s}_0, \mathtt{s}_1{>}.\mathtt{s}_0\ \mathtt{S0is1}\ \ \mathtt{S0is0}\ \mathtt{S0isB}\ \mathtt{d}\ \mathtt{p}\ \mathtt{s}_1)\,\mathtt{s}))\,\mathtt{e}$$

$$\mathtt{S0is1} \equiv \backslash\mathtt{d\,p\,s}.(\backslash{<}\mathtt{p}', \mathtt{p}''{>}.{<}{<}\mathtt{Xor}\ \mathtt{d}\ \mathtt{p}', \mathtt{s}{>}, {<}\mathtt{1}, \mathtt{p}''{>}{>})\,(\mathtt{b}\nabla_2\,\mathtt{p})$$

$$\mathtt{S0is0} \equiv \backslash\mathtt{d\,p\,s}.{<}{<}\mathtt{d}, \mathtt{s}{>}, {<}\mathtt{0}, \mathtt{p}{>}{>}$$

$$\mathtt{S0isB} \equiv \backslash\mathtt{d\,p\,s}.{<}{<}\bot, \mathtt{s}{>}, {<}\mathtt{d}, \mathtt{p}{>}{>}$$

$$\mathtt{wModBase}[\boldsymbol{p}] \equiv \backslash\mathtt{d}.\mathtt{MapThread}[\backslash\mathtt{ab}.{<}\mathtt{a}, \mathtt{b}{>}]\,(\mathtt{wRev}\,\mathtt{d})\,(\mathtt{wRev}\,\boldsymbol{p}) \ .$$

```
        wMultStep ≡
          \s l f x.wBMult[f] (l MSStep[f,wFMult] (MSBase[x] (tCast⁰ s)))
         wBMult[f] ≡ \<w,s>.(\<M,m‴>.f <m‴,0> w) s
MSStep[f,wFMult] ≡ \e.\<w,s>.(\<e′,s′>.<f e′ w,s′>) (wFMult e s)
        MSBase[x] ≡ \s.<x,s>
           wFMult ≡ \<m,r>.\<M,m‴>.wFMultBody[m,r,M,m‴]
wFMultBody[m,r,M,m‴] ≡
          (\ <m′,m″> .
            (\<M′,M″>.<<m‴,bMult[m′,M′,r]>,<M″,m″>>)(b∇₂ m))(b∇₂ M)
    bMult[m′,M′,r] ≡ Xor (And m′ M′) r
       wMultBase ≡ \m.MapThread[\a b.<a,b>] m {ε}
```

Figure 4: Combinators that compose the definition of `wMult`

The combinator `MapState`$[\cdot]$ implements the basic iteration operating on a list $\{\ldots\texttt{<d}_\texttt{i}\texttt{,p}_\texttt{i}\texttt{>}\ldots\}$ of pairs of bits, where $\texttt{d}_\texttt{i}$ are the bits of the input and $\texttt{p}_\texttt{i}$ the bits of $\boldsymbol{p}$. The core of the algorithm is the combinator `wModFun` : $(\mathbb{B}_2^2 \otimes \mathbb{B}_2^2) \multimap (\mathbb{B}_2^2 \otimes \mathbb{B}_2^2)$, that behaves as follows:

$$\texttt{wModFun}\,\texttt{<}\underbrace{\texttt{<d}_\texttt{i},\texttt{p}_\texttt{i}\texttt{>}}_{\text{elem. }\texttt{e}},\underbrace{\texttt{<s}_\texttt{0},\texttt{p}_{\texttt{i+1}}\texttt{>}}_{\text{status }\texttt{s}}\texttt{>} \rightarrow^* \texttt{<}\underbrace{\texttt{<d}_\texttt{i}{}',\texttt{p}_{\texttt{i+1}}\texttt{>}}_{\texttt{e}'},\underbrace{\texttt{<s}_\texttt{0}{}',\texttt{p}_\texttt{i}\texttt{>}}_{\texttt{s}'}\texttt{>}\ ,$$

where $\texttt{s}_\texttt{0}$ keeps the m.s.b. of $\{\ldots\texttt{d}_\texttt{i}\ldots\}$ and it is used to decide whether to reduce or not at this iteration. Thus, $\texttt{d}_\texttt{i}{}' = \texttt{d}_\texttt{i} + \texttt{p}_\texttt{i}$ if $\texttt{s}_\texttt{0} = \texttt{1}$; $\texttt{d}_\texttt{i}{}' = \texttt{d}_\texttt{i}$ if $\texttt{s}_\texttt{0} = \texttt{0}$; and $\texttt{d}_\texttt{i}{}' = \perp$ when $\texttt{s}_\texttt{0} = \perp$ (that represents the initial state, when $\texttt{s}_\texttt{0}$ still needs to be set).

Note that the second component of the status is used to shift $\boldsymbol{p}$ (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 $a^2 = \sum a_i\beta^{2i}$. 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: `wSqr` : $\mathbb{L}_2 \multimap \S\mathbb{L}_2$ that performs the bit expansion, and `Sqr` : $\mathbb{F}_{2^n} \multimap \S^2\mathbb{F}_{2^n}$ that is the actual square in $\mathbb{F}_{2^n}$. We have:

$$\texttt{Sqr} \equiv \backslash\texttt{a}.\S[\texttt{wMod}[\boldsymbol{n},\boldsymbol{p}]]\,(\texttt{wSqr a}) \tag{9}$$

and `wSqr` ≡ `\l f x.l wSqrStep[f] x`, where `wSqrStep[f]` ≡ `\e t.f 0 (f e t)` has type $\mathbb{B}_2 \multimap \alpha \multimap \alpha$ if `f` is a non linear variable with type $\mathbb{B}_2 \multimap \alpha \multimap \alpha$.

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, $ab = \sum_{j+k=i}(a_j + b_k)\beta^i$.

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 `Sqr`, we have to distinguish between multiplication of two arbitrary degree polynomials represented as binary lists, `wMult` : $\mathbb{L}_2 \multimap \mathbb{L}_2 \multimap \S\mathbb{L}_2$ and the field operation `Mult` : $\mathbb{F}_{2^n} \multimap \mathbb{F}_{2^n} \multimap \S^2\mathbb{F}_{2^n}$, obtained by composing with the modular reduction. We have:

$$\begin{aligned} \texttt{Mult} &\equiv \backslash\texttt{a}\,\texttt{b}.\S[\texttt{wMod}[\boldsymbol{n}, \boldsymbol{p}]]\,(\texttt{wMult a b}) \\ \texttt{wMult} &\equiv \backslash\texttt{a}\,\texttt{b}.\S[\texttt{wProj}_2]\,(\texttt{b}\ (\backslash\texttt{M}\,\texttt{l}.\texttt{wMultStep}\ \texttt{<M}, \bot\texttt{>}\ \texttt{l})\,(\texttt{wMultBase}\ (\texttt{wCast}^0\ \texttt{a})))\ . \end{aligned}$$

The internals of `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 `wMultStep` : $\mathbb{B}^2_2 \multimap \mathbb{L}(\mathbb{B}^2_2) \multimap \mathbb{L}(\mathbb{B}^2_2)$ behaves as follows:

$$\texttt{wMultStep}\ \texttt{<M}, \bot\texttt{>}\ \ \{\ldots \texttt{<m}_\texttt{i}, \texttt{r}_\texttt{i}\texttt{>} \ldots\} \rightarrow^* \{\ldots \texttt{<m}_{\texttt{i}-\texttt{1}}, \texttt{r}'_\texttt{i}\texttt{>} \ldots\}$$

where `M` is the current bit of the multiplier b, and every $\texttt{m}_\texttt{i}$ is a bit of the multiplicand a, and every $\texttt{r}_\texttt{i}$ is a bit in the current result. The iteration is enabled by the combinator `wMultBase` : $\mathbb{L}_2 \multimap \mathbb{L}(\mathbb{B}^2_2)$, that, on input a, creates $\{\texttt{<m}_{\texttt{n}-\texttt{1}}, \bot\texttt{>} \ldots \texttt{<m}_\texttt{0}, \bot\texttt{>}\}$, setting the initial bits of the result to $\bot$. The projection $\texttt{wProj}_2$ 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 `wFMult` : $\mathbb{B}^2_2 \multimap \mathbb{B}^2_2 \multimap (\mathbb{B}^2_2 \otimes \mathbb{B}^2_2)$, that behaves as follows:

$$\texttt{wFMult}\ \ \underbrace{\texttt{<m}_\texttt{i}, \texttt{r}_\texttt{i}\texttt{>}}_{\text{elem. }\texttt{e}}\ \ \underbrace{\texttt{<M}, \texttt{m}_{\texttt{i}-\texttt{1}}\texttt{>}}_{\text{status }\texttt{s}} \rightarrow^* \texttt{<}\underbrace{\texttt{<m}_{\texttt{i}-\texttt{1}}, \texttt{M}\cdot\texttt{m}_\texttt{i} + \texttt{r}_\texttt{i}\texttt{>}}_{\texttt{e}'}, \underbrace{\texttt{<M}, \texttt{m}_\texttt{i}\texttt{>}}_{\texttt{s}'}\texttt{>}\ \ .$$

For completeness, we list the type of the other combinators: $\texttt{MSStep}[f, \texttt{wFMult}]$: $\mathbb{B}^2_2 \multimap (\alpha \otimes \mathbb{B}^2_2) \multimap (\alpha \otimes \mathbb{B}^2_2)$, $\texttt{MSBase}[x] : \mathbb{B}^2_2 \multimap (\alpha \otimes \mathbb{B}^2_2)$, $\texttt{wBMult}[f] : (\alpha \otimes \mathbb{B}^2_2) \multimap \alpha$.

4.5. Multiplicative Inversion

We reformulate BEA in Figure 1 as a λ-term `wInv` of TFA as in Figure 5. `wInv` starts building a list which it obtains by means of `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 BEA. We represent the polynomials as words:

```
U  = \f.\x.f 0 (f 1 (f 0 (f 0 x)))
V  = \f.\x.f 1 (f 0 (f 1 (f 1 x)))
G1 = \f.\x.f 0 (f 0 (f 0 (f 1 x)))                    (10)
G2 = \f.\x.f 0 (f 0 (f 0 (f 0 x))) .
```

```
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] # FwfV       "  "      "
                [      0...  0 0] # FwdG2      "  "      "
                [      0...  0 0] # FwdM       "  "      "
    ) # Base function of D.
)
#                         LEGENDA
# Meaning          | Text abbreviation | Name of variable
# -------------------------------------------------------
# Step number      | StpNmbr           | sn
# Right shift      | RghtShft          | rs
# Forwarding of V  | FwdV              | fwdv
# Forwarding of G2 | FwdG2             | fwdg2
# Forwarding of M  | FwdM              | fwdm
```

Figure 5: Definition of wInv.

`wInv` builds an initial list by applying `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:

```
\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>          (11)
     (f <0,1, 0, 0,1,   0,     B,       B,   0,    0,   0>
     (f <0,1, 1, 0,1,   0      B,       B,   0,    0,   0> # lsb
                                                            x))) .
```

We call *threaded words vector* the list (11) that `wInv` builds in its first step. We shall call threaded words vector every list whose tuples have eleven boolean elements with the same position meaning as the comments in (11) fix. The `i`th element of column `U` is `U[i]`. We adopt analogous notation on `V`, `G1`, etc.. We write `<V,..,M>[i]`, or `<V[i],..,M[i]>` 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`, ..., `FwdM`. The most significant bit (msb) of any threaded words vector is on top; its least 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 `\tw.wRevInit (BkwVst (wRev (FwdVst tw)))` is the step function which is iterated starting from a threaded words vector built like (11) was. The step function implements steps from 2 through 5 of BEA in Figure 1. The iteration that `D` implements is the outermost loop which starts at step 2 and stops at step 6. `FwdVst` shortens *forward visit.* `wRev` reverses the threaded words vector it takes as input. `BkwdVst` stands for *backward visit.* `wRevInit` reverses the threaded words vector it gets in input while reinitialising the bits in positions `StpNmb`, `RghtShft`, `FwdV`, `FwdG2` and `FwdM`.

`FwdVst` builds on the pattern of the meta-combinator `wHeadTail[L,B]`. Its input is a threaded words vector which we call `wFwdVstInput`. Its output is again a threaded words vector `wFwdVstOutput`. `FwdVst` can distinguish its step zero, and its last step. Yet, for every `0<i<=msb`, `FwdVst` builds the `i`th element of `wFwdVstOutput` on the base of `<U,V,..,FwdM>[i]` which it takes from `wFwdVstInput` and moreover `<U,V,..,FwdM>[i-1]` taken from `wFwdVstOutput`.

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

$$\text{“Is } \texttt{Stop[0]=1}\text{?”} \tag{12}$$

$$\text{“Does } z \text{ divide both } u \text{ and } g_1\text{?”} \tag{13}$$

$$\text{“Does } z \text{ divide } u \text{ but not } g_1\text{?”} \tag{14}$$

$$\text{“Neither of the previous questions has positive answer?”.} \tag{15}$$

If (12) holds, `FwdVst` must behave as the identity. Such a situation is equivalent to saying that bits in position `G1` contain the result.

Let us assume instead that (13) or (14) holds. Answering the first question requires to verify U[0]=0 and G1[0]=0 in wFwdVstInput. Answering the second one needs to check both U[0]=0 and G1[0]=1 in wFwdVstInput. Under our conditions, just after reading wFwdVstInput, the combinator FwdVst generates the following first element, i.e. the lsb, of wFwdVstOutput:

```
<U[0],B,g1,B,B,B,0,rs,V[0],G2[0],M[0]> .                    (16)
```

If (13) holds, then g1 is G1[0] and rs is 1. If (14) holds, then g1 is Xor G1[0] M[0] and rs is 0. For building (16) we first record V[0], G2[0] and M[0], which wFwdVstInput supplies, in position FwdV[0], FwdG2[0] and FwdM[0], respectively, of wFwdVstOutput. Then we set V[0]=G2[0]=M[0]=B in wFwdVstOutput.

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

```
<U,V,G1,G2,M,Stop,StpNmbr,RghtShft,FwdV,FwdG2,FwdM>[i]
<U,V,G1,G2,M,Stop,StpNmbr,RghtShft,FwdV,FwdG2,FwdM>[i-1] .       (17)
```

The tuple with index i belongs to wFwdVstInput. The one with index i-1 is the i-1th element of wFwdVstOutput. So, FwdVst generates the new ith element of wFwdVstOutput from them which will become the i-1th element of wFwdVstOutput in the succeeding step:

```
<U[i],FwdV[i-1],g1,FwdG2[i-1],FwdM[i-1],B,0,rs,V[i],G2[i],M[i]> .   (18)
```

Yet, g1 and rs depend on u and g_1 being divisible by z.

Finally, under the above condition that (13) or (14) holds, the last step of FwdVst adds two elements to wFwdVstOutput. Let msb be the length of wFwdVstInput. The two last elements of wFwdVstOutput are:

```
<0,V[msb],0,G2[msb],M[msb],B,0,rs,B,B,B> # msb of wFwdVstOutput
<U[msb],FwdV[msb-1],g1,FwdG2[msb-1],FwdM[msb-1],B,0,rs,B,B,B> .     (19)
```

As before, g1 and rs keeps depending on which between (13) or (14) holds. The elements FwdV[msb-1], FwdG2[msb-1] and FwdM[msb-1] come from the term wFwdVstOutput. The elements U[msb], V[msb], G2[msb] and M[msb] belong to the last element of wFwdVstInput.

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 wFwdVstInput and wFwdVstOutput and which adds two last elements to wFwdVstOutput as specified in (19) — amounts to shifting the bits in positions V, G2 and M of wFwdVstInput one step to their *left*. Instead, it leaves the bits of position U and G1 as they were in wFwdVstInput so that they, in fact, shift one step to their right if we are able to erase the lsb of wFwdVstOutput. We shall erase such a lsb by means of BkwdVst. Roughly, Let `l` be the position of the last element of `wFwdVstOutput`.

1. If `<Stop,StpNmbr,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 `<Stop,StpNmbr,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 `<Stop,StpNmbr,RghtShft>[l]=<B,_,0>` or `<Stop,StpNmbr,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 g_1. In case of a negative answer `FwdVst` bitwise added `G1` and `M` in the course of its whole iteration.

Figure 6: Relevant combinations of `<Stop,StpNmb,RghtShft>` as given by `FwdVst`.

only a correct concatenation of both `FwdVst` and `BkwdVst` shifts to the right every `U[i]` and `G1[i]`, or `Xor G1[i] M[i]`, while preserving the position of every other element.

The description of how `FwdVst` works concludes by the assumption that neither Condition (13) nor Condition (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 vector 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 `Stop[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 `wFwdVstOutput` that contains `<Stop,StpNmb>[msb]=<0,1>`. This 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 the term `wFwdVstOutput` contain `<Stop,StpNmbr,RghtShft>` in one of the three configurations of Figure 6.

Then, `wRev` reverses the result of `FwdVst` exchanging lsb and msb. Let us call `wBkwdVstInput` the threaded words vector `wFwdVstOutput` that `wBkwdVst` takes in input.

`BkwdVst` behaves in accordance with the lsb of `wBkwdVstInput`.

Let `wBkwdVstInput` be such that `<Stop,StpNmb,RghtShft>[lsb]=<1,_,_>` which, in accordance with Figure 6, implies that u is 1. So, `G1[lsb]`, ..., `G1[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 `wBkwdVstInput` with the value `1`. This implements Step 3 of BEA.

Let `wBkwdVstInput` be such that `<Stop,StpNmb,RghtShft>[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 `wBkwdVstInput`, it sets `U[i]` with `Xor U[i] V[i]` and `G1[i]` with `Xor G1[i] G2[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 `Xor V[i] U[i]` and `G2[i]` with `Xor G2[i] G1[i]`.

The last case is with `<Stop,StpNmbr,RghtShft>[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 `wBkwdVstInput`. This is possible exactly because `BkwdVst` builds on the programming pattern of the meta-combinator `wHeadTail[L,B]`. Erasing the msb is equivalent to erase the lsb of `wFwdVstOutput`. I.e., we realize the one-step shift to the right of `U` and of one between `G1` or `G1 + F`. Instead, while `V`, `G2` and `M` which were shifted *one place to the left* survive the erasure.

4.6. A simple 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:

```
\f.\x.
 #  U V      G1 G2 M Stop StpNmb RghtShft FwdV FwdG2 FwdM
 f <0,1,      0, 0,1,   B,     B,       0,   B,    B,   B># msb
(f <0,0,Xor 0 1, 0,0,   B,     0,       0,   1,    0,   1>
(f <1,1,Xor 0 0, 0,1,   B,     0,       0,   0,    0,   0>
(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># org lsb
                                                              x))))
```

(20)

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

```
\f.\x.
 #  U V      G1 G2 M Stop StpNmb RghtShft FwdV FwdG2 FwdM
 f <0,B,Xor 1 1, 0,1,   B,     0,       0,   1,    0,   1># org lsb
(f <0,1,Xor 0 1, 0,1,   B,     0,       0,   1,    0,   1># new lsb
(f <1,1,Xor 0 0, 0,1,   B,     0,       0,   0,    0,   0>
(f <0,0,Xor 0 1, 0,0,   B,     0,       0,   1,    0,   1>
(f <0,1,      0, 0,1,   B,     B,       0,   B,    B,   B># msb
                                                              x))))
```

(21)

`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:

```
\f.\x.
 #  U V      G1 G2 M Stop StpNmb RghtShft FwdV FwdG2 FwdM
 f <0,1,Xor 0 1, 0,1,   B,     0,       0,   B,    B,   B>
(f <1,1,Xor 0 0, 0,1,   B,     0,       0,   B,    B,   B>
(f <0,0,Xor 0 1, 0,0,   B,     0,       0,   B,    B,   B>
(f <0,1,      0, 0,1,   B,     B,       0,   B,    B,   B> x)))
```

(22)

Finally, wRevInit reverses (22), yielding:

```
\f.\x.
 #  U V      G1 G2 M Stop StpNmb RghtShft FwdV FwdG2 FwdM
 f <0,1,      0, 0,1,   B,     B,       B,   0,    0,   0>
(f <0,0,Xor 0 1, 0,0,   B,     B,       B,   0,    0,   0>
(f <1,1,Xor 0 0, 0,1,   B,     B,       B,   0,    0,   0>
(f <0,1,Xor 0 1, 0,1,   B,     B,       B,   0,    0,   0> x)))
```
(23)

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 reinitialised 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 realises the shift to the right.

4.7. The code of FwdVst *and of* BkwdVst

Appendix C contains the detailed definitions of FwdVst and BkwdVst, the two main components of wInv. This paragraph is to help those readers who want to get some more catch on the structure of FwdVst and BkwdVst without looking directly at the code in Appendix C.

Both FwdVst and BkwdVst follow the pattern, namely the metacombinator wHeadTail[L,B]. Both of them have step functions and a "last step functions", the latter useful to correctly manipulate the final tuple. Their step functions as well as their last step functions are branching functions. Every choice among the branch to follow depends on the values of the bits that belong to the state or on the values of some bits of U or G1.

Trying to improve readability of the branching structures we use an explicit switch as syntactic sugar:

```
switch (N)    {
   case 1: M1
   case 0: M0
   case B: MB }
```
(24)

Depending on the value of N, which must be of type $\mathbb{B}_2$, the above switch behaves as the application N M1 M0 MB eventually choosing one among M1, M0 and MB.

We take the definition of LastStepFwdVst in Figure 7 as a paradigmatic example of all the terms that contribute to define FwdVst and BkwdVst.

Every variable in Figure 7 recalls its meaning. The name stopt stands for "Stop that comes from step msb-1", the name rst stands for "RghtShft that comes from step msb-1" and snt stands for "StpNmbr that comes from step msb-1".

```
LastStepFwdVst =
\f.
\<ft,et,t>. # Element from step i-1.
(\<ut,vt,g1t,g2t,mt,stopt,snt,rst,fwdvt,fwdg2t,fwdmt>.
 (switch (stopt) {
   case 1: # of stopt. We checked U=1. The whole wInv must be
           # the identity.
    \f.\ft.\ut.\vt.\g1t.\g2t.\mt.\snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
    (ft <ut,vt,g1t,g2t,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.
      \f.\ft.\ut.\vt.\g1t.\g2t.\mt.\snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
      (ft <ut,vt,g1t,g2t,mt,0,1,B,B,B,B> t )
     case 0: # of snt. Here we detect that U=1 and we set Stop=1 !!!!
      \f.\ft.\ut.\vt.\g1t.\g2t.\mt.\snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
      (ft <ut,vt,g1t,g2t,mt,1,B,B,B,B,B> t )
     case B: # of snt. Can never occur.
      \f.\ft.\ut.\vt.\g1t.\g2t.\mt.\snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
      (ft <ut,vt,g1t,g2t,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.
      \f.\ft.\ut.\vt.\g1t.\g2t.\mt.\snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
      (\<fwdvt1,fwdvt2>. (\<fwdmt1,fwdmt2>. (\<fwdg2t1,fwdg2t2,fwdg2t3>.
              (f <0,fwdvt1,0,fwdg2t1,fwdmt1,B,B,1,B,B,B >
        (ft <ut,vt,g1t,fwdg2t2,mt,B,snt,1,fwdvt2,fwdg2t3,fwdmt2> t ))
       (fwdg2t1 <1,1,1> <0,0,0> <B,B,B>)) (fwdmt <1,1> <0,0> <B,B>))
       (fwdvt <1,1> <0,0> <B,B>))
     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 vector.
      \f.\ft.\ut.\vt.\g1t.\g2t.\mt.\snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
      (\<fwdvt1,fwdvt2>. (\<fwdmt1,fwdmt2>. (\<fwdg2t1,fwdg2t2,fwdg2t3>.
       (f <0,fwdvt,0,fwdg2t,fwdmt,B,B,0,B,B,B >
         (ft <ut,vt,g1t,fwdg2t,mt,B,snt,0,fwdvt,fwdg2t,fwdmt> t))
          (fwdg2t1 <1,1,1> <0,0,0> <B,B,B>)) (fwdmt <1,1> <0,0> <B,B>))
       (fwdvt <1,1> <0,0> <B,B>))
     case B: # of rst. Can never occur.
      \f.\ft.\ut.\vt.\g1t.\g2t.\mt.\snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
      (ft <ut,vt,g1t,g2t,mt,B,B,B,B,B,B> t )
    }
   }
 ) f ft ut vt g1t g2t mt snt rst fwdvt fwdg2t fwdmt t
) et
```

Figure 7: Definition of `LastStepFwdVst`.

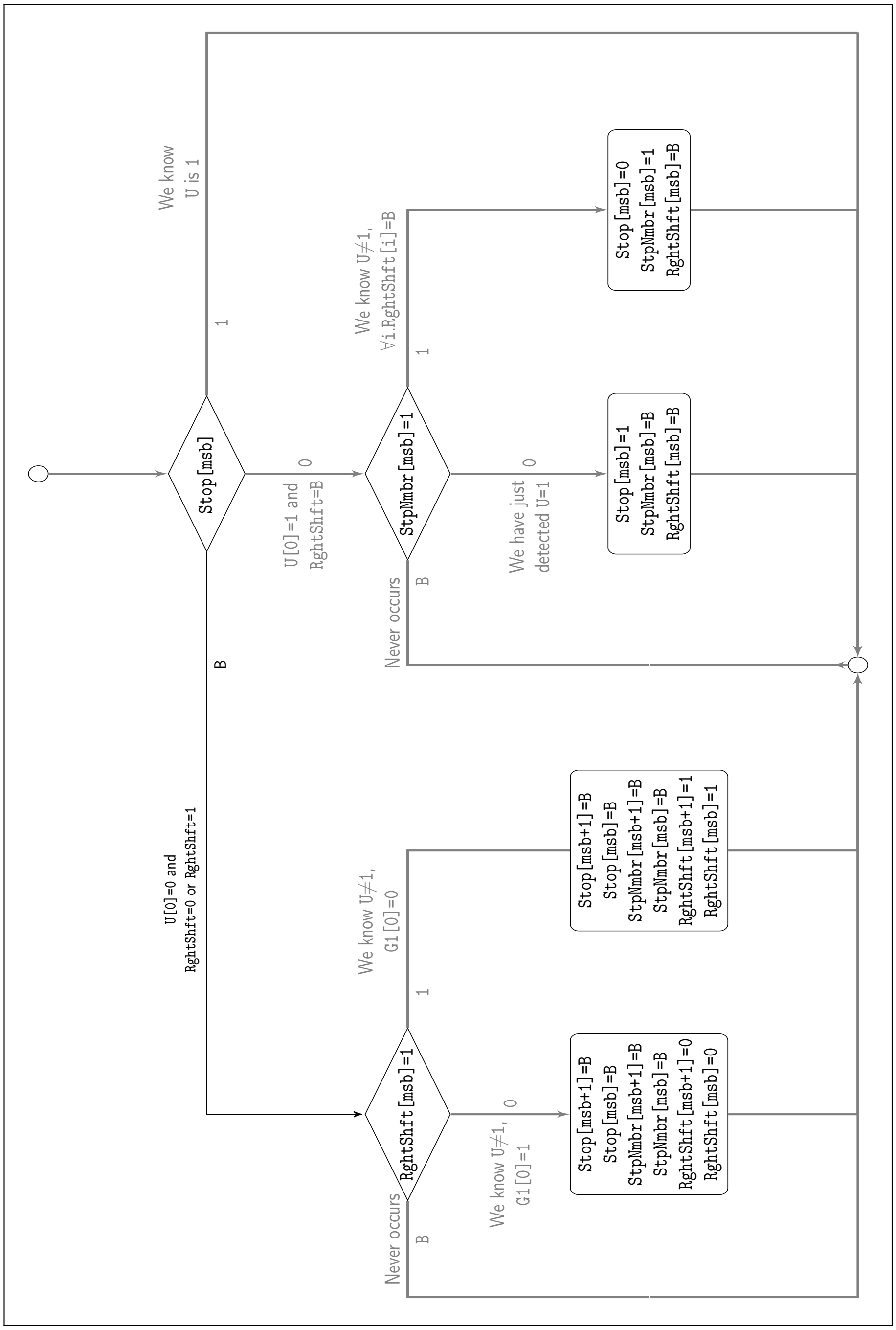


Figure 8: Flow-chart of the decision network that `LastStepFwdVst` implements.

Figure 9 depicts the essence LastStepFwdVst. Its rightmost path from the topmost decision diamond corresponds to the first branch in Figure 7. In this case nothing has to be done apart from propagating the current content of the threaded words vector. This is why, eventually, the chosen branch of the λ-term gives a λ-function which behaves as the identity. The result of the remaining paths in Figure 9 depends in one case from the value of snt and in the other on the one of rst. Globally, they give a λ-abstraction as a result which correctly sets the bits in the state in accordance with points 2 and 3 in Figure 6.

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

4.8. Typeability of wInv

Let us recall that $\mathbb{B}_2^{11} \equiv \overbrace{\mathbb{B}_2 \otimes \ldots \otimes \mathbb{B}_2}^{11}$ and $\mathbb{L}(\mathbb{B}_2^{11}) \equiv \forall\alpha.!(\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. Let us take $\mathtt{F} \equiv \backslash\mathtt{a}_1 \ \ldots \mathtt{a}_{11}.\texttt{<}\mathtt{a}_1, \ldots, \mathtt{a}_{11}\texttt{>} : \mathbb{B}_2^{11}$. Figure 13 lists the types of the main components of wInv. We remark that FwdVst, BkwdVst, LastStepFwdVst and wRevInit map a threaded words vector to another threaded words vector. 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 MapThread[F] and wRevInit are iterations that work at the lowest possible level of their syntactic components. Ideally, we can view MapThread[F] and wRevInit as adaptations and generalisations of the same programming pattern that uSuc relies on and whose type derivation is in Appendix B.

We already underlined that both FwdVst and BkwdVst adjust the programming pattern of wHeadTail[L,B] to our purposes. Appendix B recalls the type inference of 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 Figure 8. The main point to assure we can give a type to SFwdVst, BFwdVst, ... is to organise 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 introduce a library that implements basic arithmetic on binary finite fields as a set of λ-terms which have type in TFA, a type assignment system that certifies the polynomial time complexity of the λ-terms it gives types to.

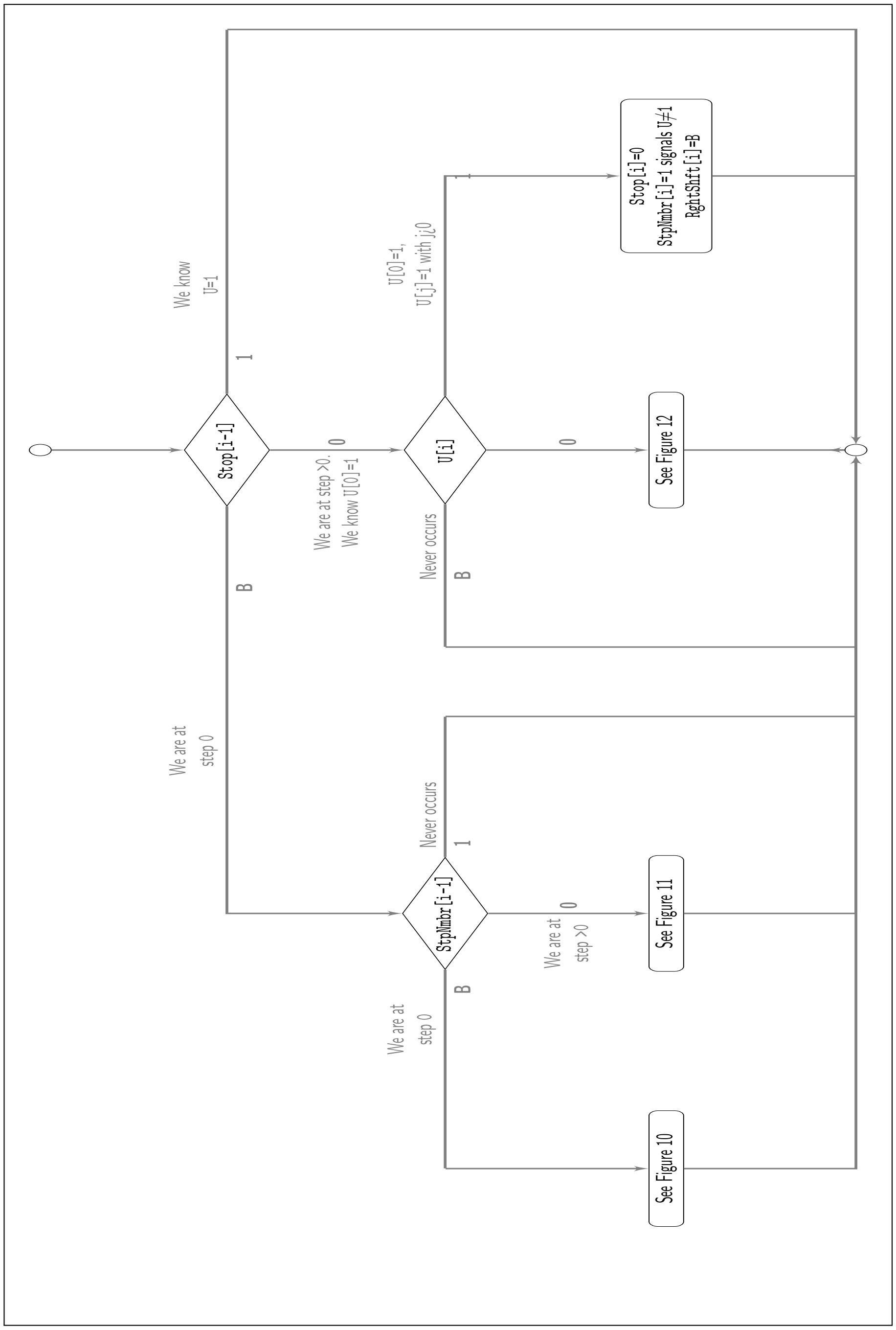


Figure 9: Flow-chart of the decision network that the step function **SFwdVst** of **FwdVst** implements.

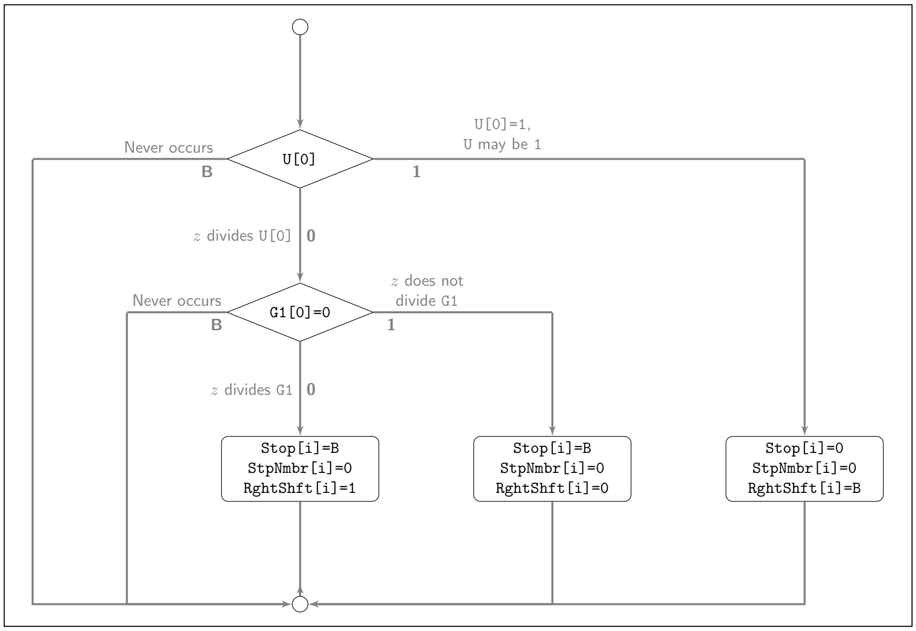


Figure 10: First component of the decision network that the step function `SFwdVst` of `FwdVst` implements.

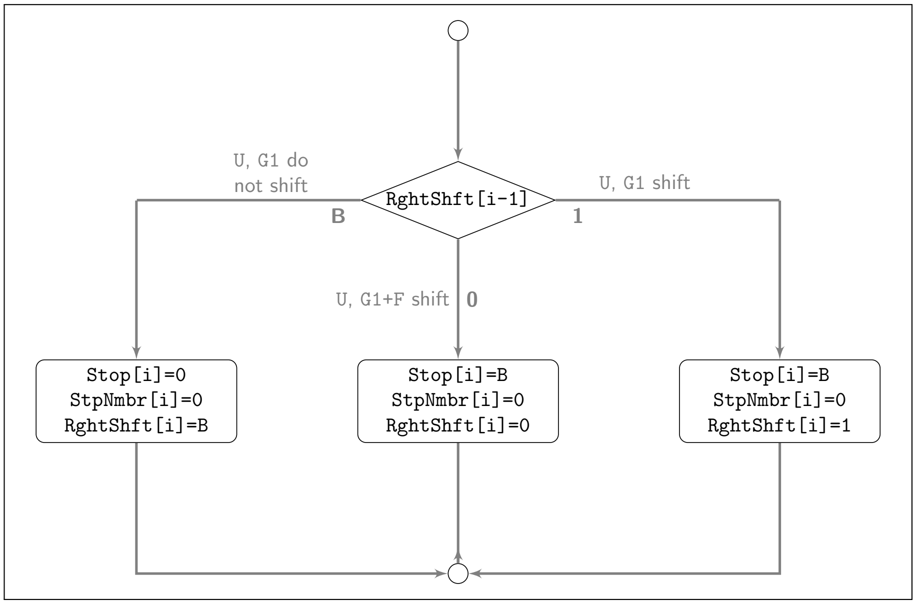


Figure 11: Second component of the decision network that the step function `SFwdVst` of `FwdVst` implements.

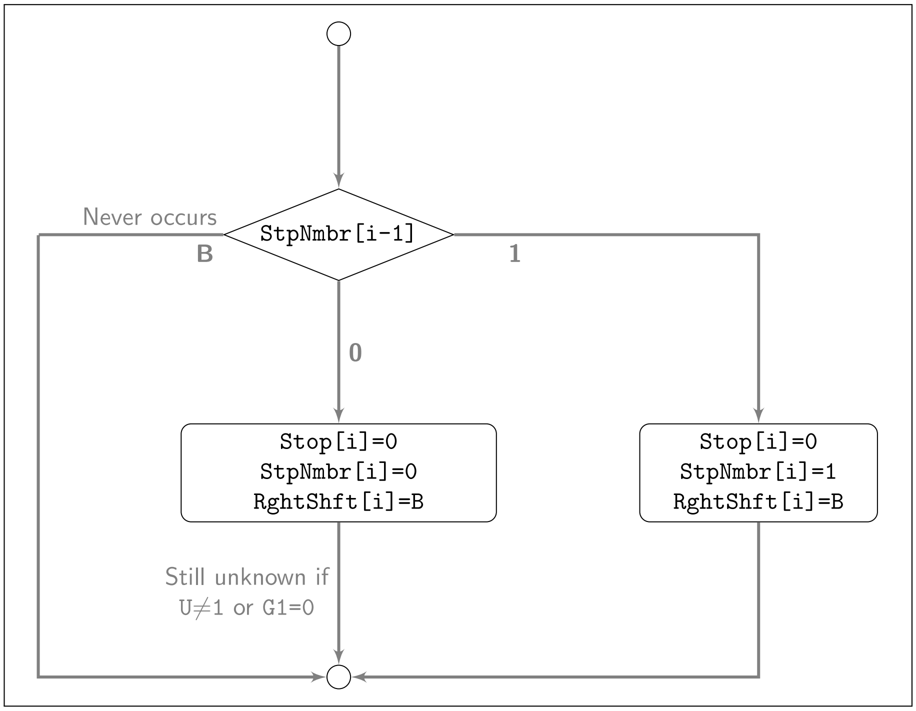


Figure 12: Third component of the decision network that the step function `SFwdVst` of `FwdVst` implements.

$$
\begin{aligned}
\texttt{MapThread}[\texttt{F}] &: \underbrace{\mathbb{L}_2 \multimap \ldots \multimap \mathbb{L}_2}_{11} \multimap \mathbb{L}(\mathbb{B}_2^{11})\\
\texttt{FwdVst} &: \mathbb{L}(\mathbb{B}_2^{11}) \multimap \mathbb{L}(\mathbb{B}_2^{11})\\
\texttt{SFwdVst} &:\\
&\quad (\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \multimap \mathbb{B}_2^{11} \multimap\\
&\qquad\quad ((\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \otimes \mathbb{B}_2^{11} \otimes \alpha) \multimap ((\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \otimes \mathbb{B}_2^{11} \otimes \alpha)\\
\texttt{BFwdVst} &: (\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \otimes \mathbb{B}_2^{11} \otimes \alpha\\
\texttt{LastStepFwdVst} &: (\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \multimap ((\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \otimes \mathbb{B}_2^{11} \otimes \alpha) \multimap \alpha\\
\texttt{BkwdVst} &: \mathbb{L}(\mathbb{B}_2^{11}) \multimap \mathbb{L}(\mathbb{B}_2^{11})\\
\texttt{SBkwdVst} &:\\
&\quad (\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \multimap \mathbb{B}_2^{11} \multimap\\
&\qquad\quad ((\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \otimes \mathbb{B}_2^{11} \otimes \alpha) \multimap ((\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \otimes \mathbb{B}_2^{11} \otimes \alpha)\\
\texttt{BBkwdVst} &: (\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \otimes \mathbb{B}_2^{11} \otimes \alpha\\
\texttt{LastStepBkwdVst} &: ((\mathbb{B}_2^{11} \multimap \alpha \multimap \alpha) \otimes \mathbb{B}_2^{11} \otimes \alpha) \multimap \alpha\\
\texttt{wRevInit} &: \mathbb{L}(\mathbb{B}_2^{11}) \multimap \mathbb{L}(\mathbb{B}_2^{11})
\end{aligned}
$$

Figure 13: The types of the main sub-terms of `wInv`.

In the course of the design of all the λ-terms, but the multiplicative inverse `wInv`, we have been able to apply standard functional programming patterns to a certain extent. Instead, `wInv` requires to adopt what we called predecessor functional pattern which generalises the pattern one has to use for writing the predecessor on Church numerals-like terms inside type assignments similar to TFA.

The set of λ-terms we write can work as a benchmark to assess the extensional expressiveness of those languages proposed to become a reference for programming with predetermined computational cost. Such languages should, in fact, simplify programming of truly interesting libraries like the one we supply.

Clearly, our library does not candidate TFA as an every-day light programming language, potentially tampering the usefulness of any language derived from light logical system similar to TFA for widespread use.

However, the programming solution we have been forced to adopt suggest research direction we think are worth exploring.

`wInv` suggests how to rearrange BEA in Figure 1 into another imperative algorithm with improved running time on specific architectures [9]. This suggests to look at the predecessor programming pattern as the potential source for the design of a domain specific language whose computational time complexity can be certified and which is expressive enough to encode interesting algorithms. We plan a bottom-up synthesis of such a domain specific language so going through the opposite top-down path that, generally speaking, proposers of languages with predetermined computational complexity followed so far when suggesting a new programming language with limited complexity.

Moreover, being the λ-calculus our programming language of reference, any of its known interpreters can be used to evaluate the implementation performance of the library we supply. Since interpreters differ in the way they evaluates terms, we plan to compare their performance without getting back to the imperative paradigm like we do in [9]. We plan to assess performance experiments on PELCR [12] which looks at λ-terms as they were algorithms whose components we can interpret in parallel on a cluster. Once more this might suggest domain specific primitives that may become as relevant as the `MapReduce` paradigm [7].

References

[1] E. Cesena, M. Pedicini, L. Roversi, Typing a Core Binary-Field Arithmetic in a Light Logic, in: R. Peña, M. van Eekelen, O. Shkaravska (Eds.), Foundational and Practical Aspects of Resource Analysis (subtitle: 2nd International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA 2011), Vol. 7177 of Lecture Notes in Computer Science, Springer, 2012, pp. 19 – 35.

[2] P. Baillot, K. Terui, Light types for polynomial time computation in lambda calculus, Information and Computation 207 (1) (2009) 41–62.
URL `http://dx.doi.org/10.1016/j.ic.2008.08.005`

[3] K. Fong, D. Hankerson, J. Lopez, A. Menezes, Field inversion and point halving revisited, IEEE Trans. Comput. 53 (8) (2004) 1047–1059.

[4] G. Hutton, A tutorial on the universality and expressiveness of Fold, Journal of Functional Programming 9 (4) (1999) 355–372.

[5] L. Roversi, A P-Time Completeness Proof for Light Logics, in: Ninth Annual Conference of the EACSL (CSL'99), Vol. 1683 of Lecture Notes in Computer Science, Springer-Verlag, Madrid (Spain), 1999, pp. 469 – 483.

[6] A. Asperti, L. Roversi, Intuitionistic light affine logic, ACM Transactions on Computational Logic 3 (1) (2002) 1–39.

[7] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM 51 (2008) 107–113.
URL `http://dx.doi.org/10.1145/1327452.1327492`

[8] J. Backus, Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, Communications of the Association for Computing Machinery 21 (8) (1978) 613–641.

[9] D. Canavese, E. Cesena, R. Ouchary, M. Pedicini, L. Roversi, Can a light typing discipline be compatible with an efficient implementation of finite

fields inversion?, in: U. Dal Lago, R. Peña (Eds.), Foundational and Practical Aspects of Resource Analysis (subtitle: 3rd International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA 2013), Vol. 8552 of LNCS, Springer, 2014, pp. 38 – 57.

[10] V. Atassi, P. Baillot, K. Terui, Verification of PTIME reducibility for System F terms: Type inference in dual light affine logic, Logical Methods in Computer Science 3 (4).

[11] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, F. Vercauteren, Handbook of Elliptic and Hyperelliptic Curve Cryptography, CRC Press, 2005.

[12] M. Pedicini, F. Quaglia, PELCR: parallel environment for optimal lambda-calculus reduction, ACM Trans. Comput. Log. 8 (3).
URL http://dx.doi.org/10.1145/1243996.1243997

Appendix A. Definition of Basic Combinators

We recall the following definitions from [1].

$\mathtt{bCast}^m$ is $\backslash\mathtt{b.b\ 1\ 0}\ \bot$.

$\mathtt{b}\nabla_t$ is $\backslash\mathtt{b.b}\ {<}\overbrace{\mathtt{1}\ldots\mathtt{1}}^{\mathtt{t}}{>}\,{<}\overbrace{\mathtt{0}\ldots\mathtt{0}}^{\mathtt{t}}{>}{<}\overbrace{\bot\ldots\bot}^{\mathtt{t}}{>}$, for every $t \geq 2$.

$\mathtt{tCast}^m$ is, for every $m \geq 0$:

$$\begin{aligned}
\mathtt{tCast}^0 &\equiv \backslash{<}\mathtt{a,b}{>}.\mathtt{a\ aIsOne\ aIsZero\ aIsBottom\ b}\\
\mathtt{aIsOne} &\equiv \backslash\mathtt{x.x}\ {<}\mathtt{1,1}{>}\ {<}\mathtt{1,0}{>}\ {<}\mathtt{1},\bot{>}\\
\mathtt{aIsZero} &\equiv \backslash\mathtt{x.x}\ {<}\mathtt{0,1}{>}\ {<}\mathtt{0,0}{>}\ {<}\mathtt{0},\bot{>}\\
\mathtt{aIsBottom} &\equiv \backslash\mathtt{x.x}\ {<}\bot,\mathtt{1}{>}\ {<}\bot,\mathtt{0}{>}\ {<}\bot,\bot{>}\\
\mathtt{tCast}^{m+1} &\equiv \backslash\mathtt{p}.\S[\mathtt{tCast^m}]\ (\mathtt{tCast^0\ p})\ .
\end{aligned}$$

$\mathtt{wSuc}$ is $\backslash\mathtt{b\ p}.\backslash\mathtt{f\ x.f}\ (\mathtt{bCast^0\ b})(\mathtt{p\ f\ x})$.

$\mathtt{wCast}^m$ is, for every $m \geq 0$:

$$\begin{aligned}
\mathtt{wCast}^0 &\equiv \backslash\mathtt{l.l}\ (\mathtt{wSuc\ 0})\ (\mathtt{wSuc\ 1})\ (\mathtt{wSuc}\ \bot)\ \{\varepsilon\}\\
\mathtt{wCast}^{m+1} &\equiv \backslash\mathtt{l}.\S[\mathtt{wCast^m}]\ (\mathtt{wCast^0\ l})\ .
\end{aligned}$$

$\mathtt{w}\nabla_t^m$**,** for every $t \geq 2$, and $m \geq 0$ is:

$$\begin{aligned}
\mathtt{w}\nabla_t^0 &\equiv \backslash\mathtt{l.l}\ (\mathtt{w}\nabla\mathtt{Step\ 0})\ (\mathtt{w}\nabla\mathtt{Step\ 1})\ \mathtt{w}\nabla\mathtt{Base}\\
\mathtt{w}\nabla_t^{m+1} &\equiv \backslash\mathtt{l}.\S[\mathtt{w}\nabla^{\mathtt{m}}_{\mathtt{t}}]\ (\mathtt{w}\nabla^{\mathtt{0}}_{\mathtt{t}}\ \mathtt{l})\\
\mathtt{w}\nabla\mathtt{Step} &\equiv \backslash\mathtt{b}.\backslash{<}\mathtt{x_1}\ldots\mathtt{x_t}{>}.{<}\overbrace{\mathtt{wSuc\ b\ x_1}\ldots\mathtt{wSuc\ b\ x_t}}^{\mathtt{t}}{>}\\
\mathtt{w}\nabla\mathtt{Base} &\equiv {<}\overbrace{\{\varepsilon\}\ldots\{\varepsilon\}}^{\mathtt{t}}{>}\ .
\end{aligned}$$

Xor is \b c.b (\x.x 0 1 1) (\x.x 1 0 0) (\x.x) c.

And is \b c.b (\x.x) (\x.x 0 0 $\perp$) $\perp$ c.

sSpl is \s.s (\t.<$\perp$, [ε]>) (\x.x).

wRev is \l f x.l wRevStep[f] (\x.x) x with:
wRevStep[f] $\equiv$ \e r x.r (f e x) : $\mathbb{B}_2 \multimap (\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$, when f : $\mathbb{B}_2 \multimap \alpha \multimap \alpha$.

wDrop$\perp$ is \l f x.l (\e.e (\f.f 1) (\f.f 0) (\f z.z) f) x.

w2s is \l.l (\e s t c.c <e, s>) [ε].

wProj$_1$ is \l f x.l (\<a, b>.f a) x.

wProj$_2$ is \l f x.l (\<a, b>.f b) x.

Map[F] is \l f x.l (\e.f (F e)) x, with F : $A \multimap B$ closed.

Fold[F, S] is \l.l (\e z.F e z) (Cast^0 S), with F : $A \multimap B \multimap B$ and S : B closed.

MapState[F] is \l s f x.(\<w, s'>.w) (l MSStep[F, f] (MSBase[x] (Cast^0 s))) with F : $(A \otimes S) \multimap (B \otimes S)$ closed, and:

$$\begin{aligned} \texttt{MSStep[F,f]} &\equiv \texttt{\\e.\\<w,s>.(\\<e',s'>.<f e' w,s'>) (F <e,s>)} \\ \texttt{MSBase[x]} &\equiv \texttt{\\s.<x,s>} \ . \end{aligned}$$

In particular MSStep[F, f] : $(A \otimes S) \multimap (\alpha \otimes S) \multimap (\alpha \otimes S)$ and MSBase[x] : $S \multimap (\alpha \otimes S)$.

MapThread[F] is
\l m f x.(\<w, s>.w) (l MTStep[F, f] (MTBase (w2s (wRev m)))) with F : $\mathbb{B}_2 \multimap \mathbb{B}_2 \multimap A$ closed and w2s (wRev m) : $\S\mathbb{S}$ whenever $m : \mathbb{L}_2$ and:

$$\begin{aligned} \texttt{MTStep[F,f]} &\equiv \texttt{\\a.\\<w,s>.(\\<b,s'>.<f (F a b) w,s'>) (sSpl s)} \\ \texttt{MTBase} &\equiv \texttt{\\x.<x,m>} \ . \end{aligned}$$

In particular MTStep[F, f] : $\mathbb{B}_2 \multimap (\mathbb{S} \otimes \mathbb{S}) \multimap (\mathbb{S} \otimes \mathbb{S})$ and MTBase : $\mathbb{S} \multimap \mathbb{S} \otimes \mathbb{S}$.

Appendix B. Some examples of type inference

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

$$\texttt{uSuc} \equiv \texttt{\\n.\\f x.f ((n f) x)}$$

of Church numerals. The type of uSuc is $\mathbb{U} \multimap \mathbb{U}$, in accordance with the type inference in Figure B.14.

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

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{\emptyset \mid \texttt{n}:\mathbb{U} \vdash \texttt{n}:\mathbb{U}}\,\mathrm{a}
    }{\emptyset \mid \texttt{n}:\mathbb{U} \vdash \texttt{n}:!(\alpha\multimap\alpha)\multimap\S(\alpha\multimap\alpha)}\,\forall\mathrm{E}
    \quad
    \dfrac{}{\emptyset \mid \texttt{g}:\alpha\multimap\alpha \vdash \texttt{g}:\alpha\multimap\alpha}\,\mathrm{a}
  }{\texttt{g}:\alpha\multimap\alpha \mid \texttt{n}:\mathbb{U} \vdash \texttt{n g}:\S(\alpha\multimap\alpha)}\,\Rightarrow\mathrm{E}
  \quad
  \dfrac{
    \dfrac{
      \dfrac{\vdots}{\emptyset \mid \texttt{h},\texttt{w}:\alpha\multimap\alpha,\texttt{x}:\alpha \vdash \texttt{h (w x)}:\alpha}\,\multimap\mathrm{E}
    }{\emptyset \mid \texttt{h},\texttt{w}:\alpha\multimap\alpha \vdash \texttt{\textbackslash x.h (w x)}:\alpha\multimap\alpha}\,\multimap\mathrm{I}
  }{\emptyset \mid \texttt{h},\texttt{w}:\S(\alpha\multimap\alpha) \vdash \texttt{\textbackslash x.h (w x)}:\S(\alpha\multimap\alpha)}\,\S\mathrm{I}
}{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{\texttt{g}:\alpha\multimap\alpha \mid \texttt{n}:\mathbb{U},\texttt{h}:\S(\alpha\multimap\alpha) \vdash \texttt{\textbackslash x.h ((n g) x)}:\S(\alpha\multimap\alpha)}{\texttt{f}:\alpha\multimap\alpha \mid \texttt{n}:\mathbb{U} \vdash \texttt{\textbackslash x.f ((n f) x)}:\S(\alpha\multimap\alpha)}\,\mathrm{c}_i
      }{\emptyset \mid \texttt{n}:\mathbb{U} \vdash \texttt{\textbackslash f x.f ((n f) x)}:!(\alpha\multimap\alpha)\multimap\S(\alpha\multimap\alpha)}\,\Rightarrow\mathrm{I}
    }{\emptyset \mid \texttt{n}:\mathbb{U} \vdash \texttt{\textbackslash f x.f ((n f) x)}:\mathbb{U}}\,\forall\mathrm{I}
  }{\emptyset \mid \emptyset \vdash \texttt{\textbackslash n.\textbackslash f x.f ((n f) x)}:\mathbb{U}\multimap\mathbb{U}}\,\multimap\mathrm{I}
}\,\S\mathrm{E}
$$

Figure B.14: The type inference of `uSuc`.

Typing `uSuc` is interesting because it is a simple term that keeps the dimension of the derivation acceptable, and shows how using the rule §E, whose application is not apparent from the structure of `uSuc` itself. Similar use of $\multimap$E occurs in typing $\mathtt{tCast}^m$, `wSuc`, $\mathtt{wCast}^m$, $\mathtt{w}\nabla_t^m$, `wRev`, for example, and, more generally, whenever a λ-terms that results from an iteration becomes the argument of a function.

Typing a predecessor built on `wHeadTail[L,B]`. Let $X \equiv (A \multimap \alpha \multimap \alpha) \otimes A \otimes \alpha$ and $\mathbb{L}(A) \equiv \forall\alpha.\,!(A \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. Let `L` and `B` be defined as in (7). This means that $\mathtt{L} : X \multimap \alpha$ and $\mathtt{B} : X$. The type assignment of `wHeadTail[L,B]` follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1 \qquad \Pi_2}{\mathtt{f}:A\multimap\alpha\multimap\alpha \mid \mathtt{w}:\mathbb{L}(A) \vdash \mathtt{w\ \ (wHTStep[B]\ \ f)}:\S(X\multimap X)}\ \Rightarrow\mathrm{E} \qquad \Pi_3}{\mathtt{f}:A\multimap\alpha\multimap\alpha \mid \mathtt{w}:\mathbb{L}(A) \vdash \mathtt{\backslash x.L\ \ (w\ \ (wHTStep[B]\ \ f)\ \ (wHTBase\ \ x))}:\S(\alpha\multimap\alpha)}\ \S\mathrm{E}}{\emptyset \mid \mathtt{w}:\mathbb{L}(A) \vdash \mathtt{\backslash f\ x.L\ \ (w\ \ (wHTStep[B]\ \ f)\ \ (wHTBase\ \ x))}:!(A\multimap\alpha\multimap\alpha)\multimap\S(\alpha\multimap\alpha)}\ \Rightarrow\mathrm{I}}{\emptyset \mid \mathtt{w}:\mathbb{L}(A) \vdash \mathtt{\backslash f\ x.L\ \ (w\ \ (wHTStep[B]\ \ f)\ \ (wHTBase\ \ x))}:\mathbb{L}(A)}\ \forall\mathrm{I}}{\emptyset \mid \emptyset \vdash \mathtt{\backslash w\ f\ x.L\ \ (w\ \ (wHTStep[B][B]\ \ f)\,(wHTBase\ \ x))}:\mathbb{L}(A)\multimap\mathbb{L}(A)}\ \multimap\mathrm{I}$$

where Π_1 is:

$$\dfrac{\dfrac{}{\emptyset \mid \mathtt{w}:\mathbb{L}(A) \vdash \mathtt{w}:\mathbb{L}(A)}\ \mathrm{a}}{\emptyset \mid \mathtt{w}:\mathbb{L}(A) \vdash \mathtt{w}:!(A\multimap X\multimap X)\multimap\S(X\multimap X)}\ \forall\mathrm{E}$$

and Π_2 is:

$$\dfrac{\dfrac{}{\emptyset \mid \emptyset \vdash \mathtt{wHTStep[B]}:(A\multimap\alpha\multimap\alpha)\multimap(A\multimap X\multimap X)} \qquad \dfrac{}{\emptyset \mid \mathtt{f}:A\multimap\alpha\multimap\alpha \vdash \mathtt{f}:A\multimap\alpha\multimap\alpha}\ \mathrm{a}}{\emptyset \mid \mathtt{f}:A\multimap\alpha\multimap\alpha \vdash \mathtt{wHTStep[B]\ \ f}:A\multimap X\multimap X}\ \multimap\mathrm{E}$$

and Π_3 is:

$$\dfrac{\dfrac{\dfrac{}{\emptyset \mid \emptyset \vdash \mathtt{L}:X\multimap\alpha} \qquad \dfrac{\dfrac{}{\emptyset \mid \mathtt{y}:X\multimap X \vdash \mathtt{y}:X\multimap X}\ \mathrm{a} \qquad \dfrac{\dfrac{}{\emptyset \mid \emptyset \vdash \mathtt{wHTBase}:\alpha\multimap X} \qquad \dfrac{}{\emptyset \mid \mathtt{x}:\alpha \vdash \mathtt{x}:\alpha}\ \mathrm{a}}{\emptyset \mid \mathtt{x}:\alpha \vdash \mathtt{wHTBase\ \ x}:X}\ \multimap\mathrm{E}}{\emptyset \mid \mathtt{y}:X\multimap X, \mathtt{x}:\alpha \vdash \mathtt{y\ \ (wHTBase\ \ x)}:X}\ \multimap\mathrm{E}}{\dfrac{\emptyset \mid \mathtt{y}:X\multimap X, \mathtt{x}:\alpha \vdash \mathtt{L\ \ (y\ \ (wHTBase\ \ x))}:\alpha\multimap\alpha}{\emptyset \mid \mathtt{y}:X\multimap X \vdash \mathtt{\backslash x.L\ \ (y\ \ (wHTBase\ \ x))}:\alpha\multimap\alpha}\ \multimap\mathrm{I}}}{\emptyset \mid \mathtt{y}:\S(X\multimap X) \vdash \mathtt{\backslash x.L\ \ (y\ \ (wHTBase\ \ x))}:\S(\alpha\multimap\alpha)}\ \S\mathrm{I} \qquad .$$

Appendix C. Pseudocode of the main components of `wInv`

```
FwdVst =
\tw. # Threaded words vector 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,g1,g2,m,stop,sn,rs,fwdv,fwdg2,fwdm>.
\<ft,et,t>.
```

```
(\<ut,vt,g1t,g2t,mt
  ,stopt,snt,rst,fwdvt,fwdg2t,fwdmt>. # Get the i-1th element
 (\<uba, ubb, ue>. # three copies of u[i]:
                   # -) the first two for branching
                   # -) one to be inserted in the list
  \<gb,ge>.      # two copies of G1[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.\g1.\g2.\m.
        \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
        \ut.\vt.\g1t.\g2t.\mt.
              \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
     <f, < u,v,g1,g2,m,1,sn,rs,B,B,B>
     ,ft <ut,vt,g1t,g2t,mt,1,snt,rst,B,B,B> 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.\g1.\g2.\m.
          \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
          \ut.\vt.\g1t.\g2t.\mt.
                \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
       <f,<1,v,g1,g2,m  # Values from this step.
         ,1   # Stop keeps recording that U[0]=1
         ,sn   # 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
         ,rs   # RightShift keeps recording that
              # neither of U, G1 shift
              # I.e. z does not divide U and G1
         ,B,B,B > # Dummy values.
       ,ft <ut,vt,g1t,g2t,mt,0,snt,rst,fwdvt,fwdg2t,fwdmt> t>
      case 0: # of uba.
       switch (sntb1) {
        case 1: # of sntb1.
         \f.\u.\v.\g1.\g2.\m.
            \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
            \ut.\vt.\g1t.\g2t.\mt.
                  \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
         <f,<0,v,g1,g2,m  # Values from this step.
            ,0   # Stop keeps recording U[0]=1
            ,1   # StepNumber keeps recording we are at step>0
                 # It also signals U[0]=1, U[j]=1 and j>0,
            ,B   # RightShift keeps recording neither
                 # of U,G1 shift
```

```
             # I.e. z does not divide U ad G1
        ,B,B,B > # Dummy values
     ,ft <ut,vt,g1t,g2t,mt,0,1,rst,fwdvt,fwdg2t,fwdmt> t>
    case 0: # of sntb1.
     \f.\u.\v.\g1.\g2.\m.
        \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
        \ut.\vt.\g1t.\g2t.\mt.
              \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
     <f,<0,v,g1,g2,m  # Values from this step
        ,0   # Stop keeps recording that U[0]=1
        ,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,G1 shift i.e. U[0]=1
        ,B,B,B > # Dummy values.
     ,ft <ut,vt,g1t,g2t,mt,0,0,rst,fwdvt,fwdg2t,fwdmt> t>
    case B: # of sntb1. Can never happen
     \f.\u.\v.\g1.\g2.\m.
        \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
        \ut.\vt.\g1t.\g2t.\mt.
              \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
     <f,<0,v,g1,g2,m,0,0,B,B,B,B >
       ,ft <ut,vt,g1t,g2t,mt,0,B,rst,fwdvt,fwdg2t,fwdmt> t>
   } # switch of sntb1 end
  case B: # of uba. Can never happen
   \f.\u.\v.\g1.\g2.\m.
      \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
      \ut.\vt.\g1t.\g2t.\mt.
            \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
   <f,<0,v,g1,g2,m,0,0,B,B,B,B >
     ,ft <ut,vt,g1t,g2t,mt,0,B,rst,fwdvt,fwdg2t,fwdmt> t>
 } # switch uba end.
case B: # of 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
   \f.\u.\v.\g1.\g2.\m.
      \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
      \ut.\vt.\g1t.\g2t.\mt.
            \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
   <f,<u,v,g1,g2,m,0,1,B,B,B,B >
   ,ft <ut,vt,g1t,g2t,mt,B,1,rst,fwdvt,fwdg2t,fwdmt> t>
  case 0: # of sntb2. We are at step>0
   switch (rst) {
    case 1: # of rst. U and G1 shift to the right.
            # I.e. U[0]=0, G1[0]=0
     \f.\u.\v.\g1.\g2.\m.
        \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
      \ut.\vt.\g1t.\g2t.\mt.
            \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
   (\<fwdvt1,fwdvt2>. (\<fwdmt1,fwdmt2>. (\<fwdg2t1,fwdg2t2>.
    <f,<u   # Value of this step
      ,fwdvt  # Value from step i-1
      ,g1  # Value of this step
      ,fwdg2t1 # Value from step i-1
      ,fwdmt1  # Value from step i-1
      ,B   # Stop keeps recording that U[0]=0
      ,0   # StepNumber keeps recording we are at step>0
      ,1   # RightShift keeps recording that U, G1 shift
      ,v   # Forwarding the three bits that
           # must shift to the left
      ,g2
      ,m >
   ,ft <ut,vt1,g1t,g2t,mt,B,0,1,vt2,fwdg2t2,fwdmt2> t>)
  (fwdg2t <1,1> <0,0> <B,B>)) (fwdmt <1,1> <0,0> <B,B>))
(fwdvt <1,1> <0,0> <B,B>))
  case 0: # of rst. U and G1+F shift to the right
          # I.e. U[0]=0, G1[0]=1
   \f.\u.\v.\g1.\g2.\m.
      \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
      \ut.\vt.\g1t.\g2t.\mt.
            \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
  (\<fwdmt1,fwdmt2>.    # two copies of mt to build elements
   (\<fwdg2t1,fwdg2t2>. # two copies of fwdg2t to build elements
    (\<me1,me2>.        # two copies of m to build elements
    <f,<u   # Value of this step
       ,fwdvt  # Value from step i-1
       ,Xor g1 me1 # Values of this step
       ,B   # Value from step i-1
       ,fwdmt1  # Value from step i-1
       ,B   # Stop keeps storing that U[0]=0
       ,0   # StepNumber keeps recording
            # we are at step>0
       ,0   # RightShift keeps recording that
            # U, G1+F shift
       ,v   # Forwarding the three bits that
            # must shift to the left
       ,g2
       ,me2 >
    ,ft <ut,vt,g1t,fwdg2t1,mt,B,0,0,fwdvt,fwdg2t2,fwdmt2> t>
    (m <1,1> <0,0> <B,B>) ) (fwdg2t <1,1> <0,0> <B,B>))
    (fwdmt <1,1> <0,0> <B,B>))
  case B: # Neither of U, G1 shift to the right.
   \f.\u.\v.\g1.\g2.\m.
      \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
      \ut.\vt.\g1t.\g2t.\mt.
            \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
    (\<fwdmt1,fwdmt2>.    # two copies of mt to build elements
    (\<fwdg2t1,fwdg2t2>.
     (\<fwdvt1,fwdvt2>.
      <f,<1,fwdvt1,g1,fwdg2t1,fwdmt1
        ,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
        ,v,g2,m >
         ,ft <ut,vt,g1t,g2t,mt,B,0,B,fwdvt2,fwdg2t2,fwdmt2> t>
     (fwdvt <1,1> <0,0> <B,B>) )
    (fwdg2t <1,1> <0,0> <B,B>))
   (fwdmt <1,1> <0,0> <B,B>))
 } # 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.\g1t.\g2t.\mt.
            \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
  (\<g2c,g2d>.
   <f,<1,v,g1,g2c,m
      ,0 # Stop records that U[0]=1
      ,0 # StepNumber 'increases' by 1
      ,B # RightShift records that neither of U, G1 shift
      ,v,g2d,m >
   ,ft <ut,vt,g1t,g2t,mt,B,B,rst,fwdvt,fwdg2t,fwdmt> t>
  (g2 <1,1> <0,0> <B,B>))
  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.\g1t.\g2t.\mt.
              \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
     (\<fwdmt1,fwdmt2,fwdmt3>.
      (\<fwdg2t1,fwdg2t2>.
       (\<fwdvt1,fwdvt2>.
        (\<me1,me2>. # two copies of m to build elements
      <f,<0
         ,fwdvt1  # This is the lsb of V.
                  # We shall erase it
         ,Xor g1 me1
         ,fwdg2t1 # This is G2[lsb]
                  # We shall erase it.
         ,fwdmt1  # 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,fwdmt2,B,B,rst,fwdvt2,fwdg2t2,fwdmt3> t>
    (m <1,1> <0,0> <B,B>)) (fwdvt <1,1> <0,0> <B,B>))
    (fwdg2t <1,1> <0,0> <B,B>)) (fwdmt <1,1,1> <0,0,0> <B,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.\g1t.\g2t.\mt.
            \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
  (\<fwdmt1,fwdmt2,fwdmt3>.
   (\<fwdg2t1,fwdg2t2>.
    (\<fwdvt1,fwdvt2>.
      <f,<0
      ,fwdvt1  # This is V[lsb].
               # We shall erase it
      ,0
      ,fwdg2t1 # This is G2[lsb].
               # We shall erase it
      ,fwdmt1  # 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,fwdmt2,B,B,rst,fwdvt2,fwdg2t2,fwdmt3> t>
   (fwdvt <1,1> <0,0> <B,B>)) (fwdg2t <1,1> <0,0> <B,B>))
    (fwdmt <1,1,1> <0,0,0> <B,B,B>))
  case B: # of gb can never happen
   \f.\u.\v.\g1.\g2.\m.
      \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
      \ut.\vt.\g1t.\g2t.\mt.
            \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\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.\g1t.\g2t.\mt.
                  \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
         <f,<B,v,B,g2,m,B,B,B,B,B,B >
         ,ft <ut,vt,g1t,g2t,mt,B,B,rst,fwdvt,fwdg2t,fwdmt> t>
       } # switch ubb end
     } # switch sntb2 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, m, fwdv,
      # fwdgb and fwdp.
    ue v  ge g2  m  stop sn  rs  fwdv  fwdg2  fwdm
    ut vt g1t 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
   (g1 <1,1>   <0,0>   <B,B>)    # G1[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 M[0]
              ,B  # This is Stop[0]
              ,B  # This is StpNmbr[0]. We are at step 0
              ,B  # This is RghtShft[0]
              ,B  # This is FwdV[0]
              ,B  # This is FwdG2[0]
              ,B  # This is FwdM[0]
             > ,x>
```

```
BkwdVst =
\tw. # Threaded words vector 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 FwdV[0].
              ,B  # This is FwdG2[0].
              ,B  # This is FwdM[0].
             > ,x>
```

```
SBkwdVst =
\f.
\<u,v,g1,g2,m,stop,sn,rs,fwdv,fwdg2,fwdm>.
\<ft,et,t>.
(\<ut,vt,g1t,g2t,mt,stopt,snt,rst,fwdvt,fwdg2t,fwdmt>.
 (switch (stopt) {
   case 1: # of stopt means U=1. Keep propagating Stop=1
    \u.\v.\g1.\g2.\m.\stop.\sn.\rs.
    \ut.\vt.\g1t.\g2t.\mt.\stopt.\snt.\rst.
    <f,<u,v,g1,g2,m,
       ,1 # Propagation of Stop=1.
       ,sn,rs,fwdv,fwdg2,fwdm>
       >
    ,ft <ut,vt,g1t,g2t,mt,1,snt,rst,fwdvt,fwdg2t,fwdmt> x>
   case 0: # of stopt. So U[0]=1, U1=1. Keep executing
           # Step 4, 5 of BEA. StepNumber keeps recording
           # the relation between deg(U), deg(V)
    switch (rs) {
     case 1: # of rs. deg(U)<deg(V) detected.
      \u.\v.\g1.\g2.\m.\stop.\sn.\rs.
      \ut.\vt.\g1t.\g2t.\mt.\stopt.\snt.\rst.
        ((\<ua,ub>.\<g1a,g1b>.
          <f,<Xor v ua,ub,Xor g2 g1a,g1b,m
             ,0 # Propagate stop=0.
             ,1 # Propagate deg(U) < deg(V).
             ,rs,B,B,B>
          ,ft <ut,vt,g1t,g2t,mt,0,snt,1,B,B,B> t>
         ) (u <1,1> <0,0> <B,B>)) (g1 <1,1> <0,0> <B,B>)
     case 0: # of rst. deg(U)>deg(V) detected.
      \u.\v.\g1.\g2.\m.\stop.\sn.\rs.
      \ut.\vt.\g1t.\g2t.\mt.\stopt.\snt.\rst.
      ((\<va,vb>.\<g2a,g2b>.
        <f,<Xor u va,vb,Xor g1 g2a,g2a,m
           ,0 # Propagate stop=0.
           ,0 # Propagate deg(U) > deg(V).
           ,rs,B,B,B>
        ,ft <ut,vt,g1t,g2t,mt,0,snt,0,B,B,B> t>
       ) (v <1,1> <0,0> <B,B>)) (g2 <1,1> <0,0> <B,B>)
     case B: # of rst. Relation between deg(U), deg(V)
             # still unknown
      \u.\v.\g1.\g2.\m.\stop.\sn.\rs.
      \ut.\vt.\g1t.\g2t.\mt.\stopt.\snt.\rst.
      ((\<va,vb>.\<g2a,g2b>.
        <f,<Xor u va,vb,Xor g1 g2a,g2b,m
           ,0 # Propagate Stop=0.
           ,B # Set StepNumber=B to propagate that the
              # relation between deg(U) and deg(V)
              # is unknown
           ,rs,B,B,B>
        ,ft <ut,vt,g1t,g2t,mt,0,snt,B,B,B,B> t>
       ) (v <1,1> <0,0> <B,B>)) (g2 <1,1> <0,0> <B,B>)
    } # switch rst
   case B: # of stopt
    switch (rs) {
     case 1: # of rs. So U[0]=0. Keep propagating
             # RightShift=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.\g1t.\g2t.\mt.\stopt.\snt.\rst.
      <f,<u,v,g1,g2,m,
         ,B  # Propagation of Stop=B.
         ,sn #
         ,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,g1t,g2t,mt,B,snt
          ,1    # Propagates the previous value of RightShift
          ,B,B,B> x
      >
     case 0: # of rs. Never occurs because the base case, i.e.
             # stopt=B and rs=B and Stop=B, sets RightShift=1
             # which the case here above with rs=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.\g1t.\g2t.\mt.\stopt.\snt.\rst.
      <f,<u,v,g1,g2,m,
         ,B  # Propagation of Stop=B
         ,sn #
         ,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,g1t,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.\g1t.\g2t.\mt.\stopt.\snt.\rst.
   <f,<u,v,g1,g2,m,
      ,1 # Propagation of Stop=1.
      ,B,B,B,B,B> #
   ,ft <ut,vt,g1t,g2t,mt,stopt,snt,rst,B,B,B> x>
  case 0: # of stop. I.e. U[0]=1, U!=1.
          # 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.\g1t.\g2t.\mt.\stopt.\snt.\rst.
       (\<g2a,g2b>.
         <f,<Xor 1 1,1,Xor g1 g2a,g2a,m
            ,0 # Propagate Stop=0
            ,B # StepNumber=B says we do not know
               # the relation between deg(U), deg(V)
            ,rs,B,B,B>
         ,ft <ut,vt,g1t,g2t,mt,stopt,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.\g1t.\g2t.\mt.\stopt.\snt.\rst.
       (\<g2a,g2b>.
         <f,<Xor 1 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,g1t,g2t,mt,stopt,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.\g1t.\g2t.\mt.\stopt.\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,g1t,g2t,mt,stopt,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.\g1t.\g2t.\mt.\stopt.\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,g1t,g2t,mt,stopt,snt,rst,B,B,B> t>
            ) (g2 <1,1> <0,0> <B,B>)
           case B: # of v. Never occurs.
            SBkwVst45NeverOccurs
          } # switch v
         case B: # of u. Never occurs.
          SBkwVst45NeverOccurs
        } # 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.
      \u.\v.\g1.\g2.\m.\stop.\sn.\rs.
      \ut.\vt.\g1t.\g2t.\mt.\stopt.\snt.\rst.
      <f,<u,v,g1,g2,m,
         ,B # Propagation of Stop=B.
         ,B # Dummy value.
         ,1 # Propagate RightShift=1. I.e. we shall calculate
            # the predecessor on the threaded words in input.
            # The predecessor realises the shift to the right.
            # Propagating 0 in place of 1 would yield the
            # same result
         ,B,B,B> # Dummy values.
      ,ft <ut,vt,g1t,g2t,mt,stopt,snt,rst,B,B,B> x>
     } # switch stop
    } # switch rst
   } # switch stopt
 ) u v g1 g2 m stop sn rs ut vt g1t g2t mt stopt snt rst
) et

where

SBkwVst45NeverOccurs =
\u.\v.\g1.\g2.\m.
\stop.\sn.\rs.\ut.\vt.\g1t.\g2t.\mt.
\stopt.\snt.\rst.
<f,<u,v,g1,g2,m,stop,sn,rs,B,B,B>
```

```
,ft <ut,vt,g1t,g2t,mt,stopt,snt,rst,B,B,B> t>
```

```
LastStepBkwdVst =
\<f,e,t>.
(\<u,v,g1,g2,m,_,_,_,_,_,_>.
 ( switch (stop) {
     case 1: # of stop says that U=1. Do nothing
       \u.\v.\g1.\g2.\m.f <u,v,g1,g2,m,B,B,B,B,B,B> 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
           \u.\v.\g1.\g2.\m.
           ((\<ua,ub>.\<g1a,g1b>.
              f <Xor v ua,ub,Xor g2 g1a,g1b,m,B,B,B,B,B,B> t
            ) (u <1,1> <0,0> <B,B>)) (g1 <1,1> <0,0> <B,B>)
         case 0: # of rs. deg(U)>deg(V) detected.
           \u.\v.\g1.\g2.\m.
           ((\<va,vb>.\<g2a,g2b>.
             f <Xor u va,vb,Xor g1 g2a,g2b,m,B,B,B,B,B,B> t
            ) (v <1,1> <0,0> <B,B>)) (g2 <1,1> <0,0> <B,B>)
         case B: # of rs. We know deg(U)=deg(V)
           \u.\v.\g1.\g2.\m.
           ((\<va,vb>.\<g2a,g2b>.
             f <Xor u va,vb,Xor g1 g2a,g2b,m,B,B,B,B,B,B> t
            ) (v <1,1> <0,0> <B,B>)) (g2 <1,1> <0,0> <B,B>)
       } # switch rs
     case B: # of stop. Conclude an iteration that must
             # implement a shift to the right. Do not insert
             # the last element of the threaded list. I.e.,
             # calculate the predecessor
       \u.\v.\g1.\g2.\m.t
   } # switch stop
  ) u v g1 g2 m
) e
```

```
wRevInit =
\w.\f.w (wRevInitS f) wRevInitB

wRevInitS =
\f. \e. (\<u,v,g1,g2,m,stop,_,_,_,_,_>.
        (\e.\r.\z.r (f <u,v,g1,g2,m,stop,B,B,0,0,0> z)) e

wRevInitB = \x.x
```