Light combinators for finite fields arithmetic

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

Abstract

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

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

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

1. Introduction

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

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

The arithmetic over binary extension of finite fields has many important applications in the domains of theory of codes and in cryptography. Finite fields' arithmetic operations include: addition, subtraction, multiplication, squaring, square root, multiplicative inverse, division and exponentiation.

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

Declarative programming, by its nature, does not permit a tight control on complexity parameters. The scenario has changed in the last twenty years with the introduction of type systems which implicitly guarantee time complexity bounds on the programs they give a type to. This means that they force restrictions on programming schemes which hardly permit to specify an algorithm in a natural way, even if it belongs to the right complexity class. Therefore a certain number of new type systems have been introduced in the last a few years with the declared objective to capture a broader class of polynomial algorithms with respect to the one which was shown to be in the previous systems, [1, 2, 3, 4, 5, 6].

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

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

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

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

INPUT: $a \in \mathbb{F}_{2^m}$, $a \neq 0$.

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

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

Figure 1: Binay-Field inversion as in **Algorithm 2.2** at page 1048 in [9].

relax to have non-linear variables. This programming pattern leads to our soundness result (of arithmetic operations on finite fields with respect to light type systems) which is in Section 4.5. We show (in addition to the other arithmetical operations) that a typeable multiplicative inversion for binary finite fields exists in TFA.

In Section 3, we illustrate the library of λ-terms which TFA gives a type to and which helps to implement arithmetic in binary fields. Figure 3 shows the structure of the functional layers that compose the library.

The lowest layer contains basic definitions introduced in Section 2. The layer *core library* contains all the combinators on basic types. We put particular care while using common functional-programming patterns; we reuse them, whenever possible, while defining other combinators in the library. Moreover, the functionality they provide will hopefully be applicable as programming constructs in future extensions of the library.

Finally, in the *binary-field arithmetic* layer we group all the combinators related to operations over binary polynomials, like addition, multiplication, modular reduction and multiplicative inversion.

In future work, we plan to extend the library by implementing other layers, such as arithmetic of elliptic curves or other cryptographic primitives, on top of the binary-field arithmetic layer.

In the following sections we present type and behavior of combinators specified in the library, while their full definitions as λ-terms are in Appendix A. The choice of giving everything in the plain functional programming language has the advantage that any interpreter for *plain* λ-calculus can be used to evaluate the behavior of the implementation. Moreover, since different interpreters do not evaluate terms in the same way, we plan, in the future, to compare the performance achieved with specific interpreters, starting from the simpler ones, like LCI, to the more sophisticated, like PELCR [10].

We have manually checked that all terms have types in DLAL. Some type inference can be found in [7, 11]. Our Appendix B gives a couple of useful examples too.

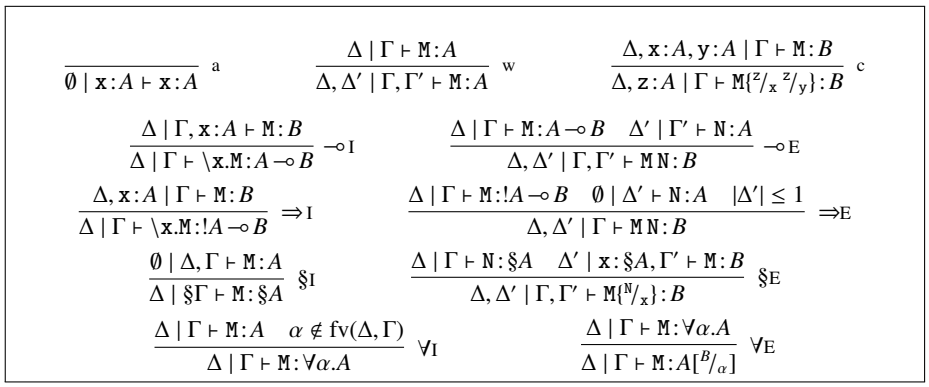


Figure 2: Type assignment system TFA

2. Typeable Functional Assembly

We call Typeable Functional Assembly (TFA) the deductive system in Figure 2. Its rules come from Dual Light Affine Logic (DLAL) [7]. "Assembly" as part of the name comes from our programming experience inside TFA. When programming inside TFA the goal is twofold. Writing the correct λ-term and 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 $\backslash\mathtt{x}.\mathtt{M}$ in place of $(\backslash\mathtt{x}.\mathtt{M})$ 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:

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

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 $\{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n\}$.

In fact, we shall assign *types*, not mere *formulas*, to λ-terms. Introducing the notion of types requires some preliminary definitions.

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

$$\mathbb{B}_n \equiv \forall\alpha.\mathbb{B}_n[\alpha] \text{ with } \mathbb{B}_n[\alpha] \equiv \overbrace{\alpha \multimap \cdots \multimap \alpha}^{n+1} \multimap \alpha \ .$$

Setting $n = 2$ we get "lifted" booleans $\mathbb{B}_2$ with canonical representatives:

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

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

Tuples. They are functions that store a predetermined number of λ-terms:

$$(A_1 \otimes \ldots \otimes A_n) \equiv \forall\alpha.(A_1 \otimes \ldots \otimes A_n)[\alpha] \multimap \alpha \text{ with } (A_1 \otimes \ldots \otimes A_n)[\alpha] \equiv A_1 \multimap \cdots \multimap A_n \multimap \alpha$$

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

$$\mathtt{M} ::= \ldots \mid <\mathtt{M}, \ldots, \mathtt{M}> \mid (\backslash<\mathcal{V}, \ldots, \mathcal{V}>.\mathtt{M}) \ .$$

Then, we extend β-reduction with:

$$(\backslash<\mathtt{x}_1, \ldots, \mathtt{x}_n>.\mathtt{M}) <\mathtt{N}_1, \ldots, \mathtt{N}_n> \rightarrow \mathtt{M}\{{}^{\mathtt{N}_1}/_{\mathtt{N}_1}, \ldots, {}^{\mathtt{N}_n}/_{\mathtt{x}_n}\} \ .$$

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

$$\frac{\Delta_1 \mid \Gamma_1 \vdash \mathtt{M}_1 : A_1 \quad \ldots \quad \Delta_n \mid \Gamma_n \vdash \mathtt{M}_n : A_n}{\Delta_1, \ldots, \Delta_n \mid \Gamma_1, \ldots, \Gamma_n \vdash <\mathtt{M}_1, \ldots, \mathtt{M}_n> : (A_1 \otimes \ldots \otimes A_n)} \otimes\mathrm{I}$$

$$\frac{\Delta \mid \Gamma, \mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n \vdash \mathtt{M} : B}{\Delta \mid \Gamma \vdash \backslash<\mathtt{x}_1, \ldots, \mathtt{x}_n>.\mathtt{M} : (A_1 \otimes \ldots \otimes A_n) \multimap B} \multimap\mathrm{I}_{\otimes}$$

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

$$<\mathtt{M}_1, \ldots, \mathtt{M}_n> \text{stands for } \backslash\mathtt{x.x}\ \mathtt{M}_1 \ldots\ \mathtt{M}_n$$
$$\backslash<\mathtt{x}_1, \ldots, \mathtt{x}_n>.\mathtt{M} \text{ stands for } \backslash\mathtt{p.p}\,(\backslash\mathtt{x}_1 \ldots (\backslash\mathtt{x}_n.\mathtt{M})) \ .$$

Sequences of booleans, or simply Sequences. We denote them by $\mathbb{S}$ and recursively define as:

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

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

2.1. The set $\mathcal{T}$ of types

The set $\mathcal{T}$ of *types* is the quotient $\mathcal{F}/\approx$. We mean that if $\mathtt{M}$ has type $\mathbb{S}$, then we can equivalently use any of the unfolded forms of $\mathbb{S}$ as type of $\mathtt{M}$. The canonical values of type $\mathbb{S}$ are:

$$\begin{aligned} [\varepsilon] &\equiv \backslash\mathtt{tc.t}\ \bot : \mathbb{S} \\ [\mathtt{b}_{n-1} \dots \mathtt{b}_0] &\equiv \backslash\mathtt{tc.c}\ \mathtt{<b}_{n-1}, [\mathtt{b}_{n-2} \dots \mathtt{b}_0]\mathtt{>} : \mathbb{S} \ . \end{aligned} \tag{4}$$

In accordance with (3), the Sequence $[\mathtt{b}_{n-1} \dots \mathtt{b}_0]$ 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 $[\mathtt{b}_{n-2} \dots \mathtt{b}_0]$. The recursive definition of $\mathbb{S}$ should be evidently crucial.

By convention, in every Sequence $[\mathtt{b}_{n-1} \dots \mathtt{b}_0]$, 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*.

2.2. Summing up

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

3. Basic Definitions, Types and the Core Library

From [13], we recall the meaning and the type of the λ-terms that forms the *two* lowermost layers in Figure 3. We also recall their definition in Appendix A.

Cryptographic primitives*: elliptic curves cryptography, linear feedback shift register cryptography, . . .*
Binary-field arithmetic: addition, (modular reduction), square, multiplication, inversion.
Core library: operations on bits (xor, and), operations on sequences (head-tail splitting), operations on words (reverse, drop, conversion to sequence, projections); meta-combinators: fold, map, mapthread, map with state, head-tail scheme.
Basic definitions and types: booleans, tuples, numerals, words, sequences, basic type management and duplication.

Figure 3: Library for binary-field arithmetic

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

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

where $\S[\mathtt{M}] \equiv \backslash \mathtt{x}.\mathtt{M}\,\mathtt{x}$ is the *paragraph lift of* $\mathtt{M}$. As obvious generalization, 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:

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

They iterate the first argument on the second one.

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:

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

that generalize the iterative structures of Church numerals.

Church words. A Church word is a list $\{\mathtt{b}_{n-1}\dots\mathtt{b}_0\}$ whose elements $\mathtt{b}_i$s are booleans, i.e. of type $\mathbb{L}_2 \equiv \mathbb{L}(\mathbb{B}_2)$. By convention. in every Church word $\{\mathtt{b}_{n-1}\dots\mathtt{b}_0\}$, or simply *word*, the *least significant bit* (lsb) is $\mathtt{b}_0$, while the *most significant bit* (msb) is $\mathtt{b}_{n-1}$. The same convention holds for every Sequence $[\mathtt{b}_{n-1}\dots\mathtt{b}_0]$.

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:

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

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:

$$\mathtt{tCast}^{\mathtt{m}} <\mathtt{b}_0, \mathtt{b}_1> \to^* <\mathtt{b}_0, \mathtt{b}_1>.$$

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

$$\mathtt{wSuc}\ \mathtt{b}\ \{\mathtt{b}_{n-1}\dots\mathtt{b}_0\} \to^* \{\mathtt{b}\ \mathtt{b}_{n-1}\dots\mathtt{b}_0\}.$$

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:

$$\mathtt{wCast}^{\mathtt{m}}\ \{\mathtt{b}_{n-1}\dots\mathtt{b}_0\} \to^* \{\mathtt{b}_{n-1}\dots\mathtt{b}_0\}.$$

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

$$\mathtt{w}\nabla_{\mathtt{t}}^{\mathtt{m}}\ \{\mathtt{b}_{n-1}\dots\mathtt{b}_0\} \to^* <\overbrace{\{\mathtt{b}_{n-1}\dots\mathtt{b}_0\},\dots,\{\mathtt{b}_{n-1}\dots\mathtt{b}_0\}}^{\mathtt{t}}>.$$

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:

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

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:

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

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

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

$$\mathtt{sSpl}\,[\mathtt{b}_{n-1} \ldots \mathtt{b}_0] \to^* <\mathtt{b}_{n-1}, [\mathtt{b}_{n-2} \ldots \mathtt{b}_0]>.$$

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

$$\mathtt{wRev}\,\{\mathtt{b}_{n-1} \ldots \mathtt{b}_0\} \to^* \{\mathtt{b}_0 \ldots \mathtt{b}_{n-1}\}.$$

The combinator $\mathtt{wDrop}\bot : \mathbb{L}_2 \multimap \mathbb{L}_2$. It *drops* all the (initial) occurrences[5] of $\bot$ in a word:

$$\mathtt{wDrop}\bot\,\{\bot \ldots \bot\,\mathtt{b}_{n-1} \ldots \mathtt{b}_0\} \to^* \{\mathtt{b}_{n-1} \ldots \mathtt{b}_0\}.$$

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

$$\mathtt{w2s}\,\{\mathtt{b}_{n-1} \ldots \mathtt{b}_0\} \to^* [\mathtt{b}_{n-1} \ldots \mathtt{b}_0].$$

Its type inference is in Appendix B.

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

$$\mathtt{wProj}_1\,\{<\mathtt{a}_{n-1}, \mathtt{b}_{n-1}> \ldots <\mathtt{a}_0, \mathtt{b}_0>\} \to^* \{\mathtt{a}_{n-1} \ldots \mathtt{a}_0\}.$$

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

3.2.1. Meta-combinators

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

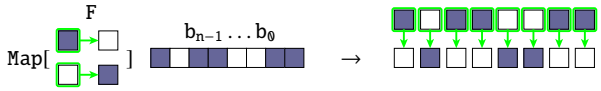
Then, we introduce a new meta-combinator that supplies the main programming pattern to implement BEA as a λ-term of TFA.

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.

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

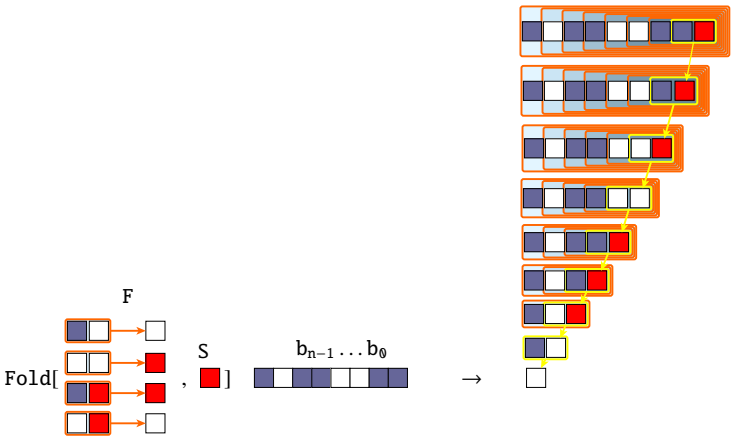
The meta-combinator `Map[·]`. Let $\mathtt{F} : A \multimap B$ be a closed term. Then, $\mathtt{Map[F]} : \mathbb{L}(A) \multimap \mathbb{L}(B)$ applies F to every element of the list that `Map[F]` takes as argument, and yields the final list, assuming $\mathtt{F}\,\mathtt{b_i} \to^* \mathtt{b'_i}$,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

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



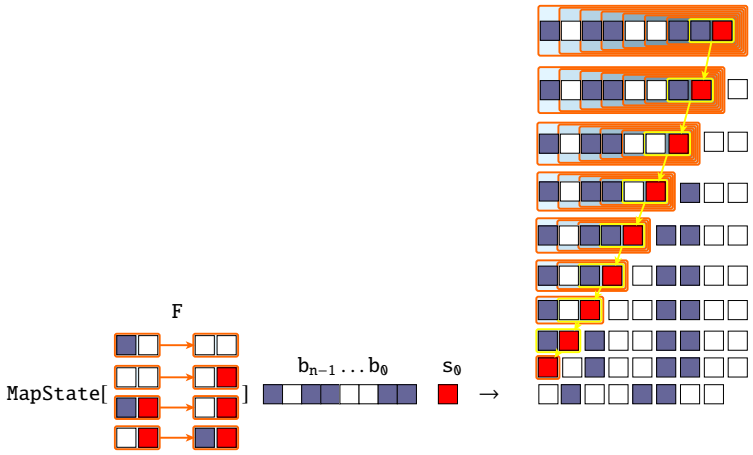
The meta-combinator `Fold[·,·]`. Let $\mathtt{F} : A \multimap B \multimap B$ and $\mathtt{S} : B$ be closed terms. Let also $\mathtt{Cast}^0 : B \multimap \S B$. Then, $\mathtt{Fold[F,S]} : \mathbb{L}(A) \multimap \S B$, starting from the initial value S, iterates F over the input list and builds up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

$$\mathtt{Fold[F,S]}\ \{\mathtt{b}_{n-1} \ldots \mathtt{b}_0\} \to^* \mathtt{b'}\ .$$



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

$$\mathtt{MapState[F]}\ \{\mathtt{b}_{n-1} \ldots \mathtt{b}_0\}\,\mathtt{s}_0 \to^* \{\mathtt{b'}_{n-1} \ldots \mathtt{b'}_0\}\ .$$

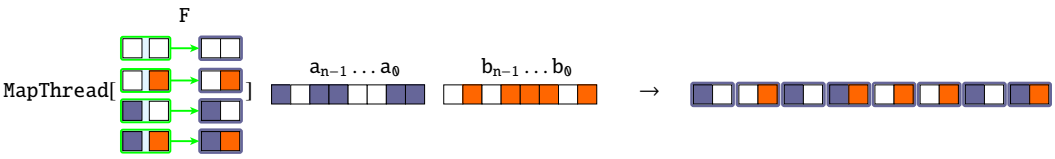


The meta-combinator `MapThread[·]`. Let $\mathtt{F} : \mathbb{B}_2 \multimap \mathbb{B}_2 \multimap A$ be a closed term. Then, $\mathtt{MapThread[F]} : \mathbb{L}_2 \multimap \mathbb{L}_2 \multimap \mathbb{L}(A)$ applies F to the elements of the input list. Specifically, if $\mathtt{F}\ \mathtt{a_i}\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

$$\mathtt{MapThread[F]}\ \{\mathtt{a}_{n-1} \ldots \mathtt{a}_0\}\ \{\mathtt{b}_{n-1} \ldots \mathtt{b}_0\} \to^* \{\mathtt{c}_{n-1} \ldots \mathtt{c}_0\} \quad .$$

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

$$\mathtt{MapThread[\backslash ab.<a,b>]}\ \{\mathtt{a}_{n-1} \ldots \mathtt{a}_0\}\ \{\mathtt{b}_{n-1} \ldots \mathtt{b}_0\} \to^* \{<\mathtt{a}_{n-1}, \mathtt{b}_{n-1}> \ldots <\mathtt{a}_0, \mathtt{b}_0>\} \quad .$$



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

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

where:

- `L` stands for "last (step)". It denotes a closed λ-term with type $X \multimap \alpha$.
- `B[f,e,ft,et,t]` stands for "body (of the step function)". It denotes a closed λ-term with the following two features. It must have type X and the variables `f`, `e`, `ft`, `et` and `t` must be *sub-terms* of `B` that must occur linearly in it.

- `wHTStep[B]` is the step function that must have type $(A \multimap \alpha \multimap \alpha) \multimap A \multimap X \multimap X$.
- `wHTBase` is the base function that must have type X.

So, `wHeadTail[L,B]`: $\mathbb{L}(A) \multimap \mathbb{L}(A)$ and, for example, it can develop the following computation on a list `\g.\y.g b (g a y)`:

$$\begin{aligned}
&\texttt{wHeadTail[L,B]}\ (\texttt{\textbackslash g y.g b (g a y)}) \\
&\to^* \texttt{\textbackslash f x.L (wHTStep[B] f b (wHTStep[B] f a (wHTBase x)))} \qquad (6)\\
&\to^* \texttt{\textbackslash f x.L ((\textbackslash<ft, et, t>.B[f, b, ft, et, t]) B[f, a, \textbackslash e l.l, DummyElement, x])}
\end{aligned}$$

It is an iteration of `wHTStep[B]` from `wHTBase` on the input. If `DummyElement` is different from any possible element of the list, the rightmost occurrence of `B` in (6) knows that the iteration is at its step zero and it can operate on `a` as consequence of this fact. In general, `B` can identify a sequence of iteration steps of predetermined length, say n. Then, `B` can operate on the first n elements of the list in a specific way. The distinguishing invariant of 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 `DummyElement` 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:

$$\begin{aligned}
&\texttt{L} \equiv \texttt{\textbackslash<_, _, l>.l} \qquad (7)\\
&\texttt{B} \equiv \texttt{\textbackslash f e.\textbackslash<ft, et, t>.<f, e, ft et t>}
\end{aligned}$$

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`.

We shall see that BEA*, viewed as a term of* TFA*, relies on some variants of* `wHeadTail[L,S]`.

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 3; 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 [15, 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 $\beta \in \overline{\mathbb{F}}_2$ 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.

In what follows, we denote by $\boldsymbol{n}$ the Church numeral $\overline{n}$, representing the integer $n = \deg p(X)$, and, by $\boldsymbol{p}$, the Church word $\boldsymbol{p} \equiv \left\{ \mathtt{p_n} \ldots \mathtt{p_0} \underbrace{\bot \ldots \bot}_{n-1} \right\}$, where $\mathtt{p_i}$ are the boolean terms associated to the corresponding coefficient p_i of the polynomial $p(X) = \sum p_i X^i$. Note that $\boldsymbol{p}$ has length $2n$. The $\bot$ in the least significative part are included for technical reasons, to simplify the discussion later.

4.1. Addition

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

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

4.2. Modular Reduction

Reduction modulo $p(X)$ is a fundamental building block to keep the size of the operands constrained. We implemented a naïf left-to-right method, assuming that: (1) both $p(X)$ and $n = \deg p(X)$ are fixed (thus given as parameters); (2) the length of the input is $2n$, i.e., we need exactly n repetitions of a basic iteration.
The combinator $\mathtt{wMod}[\boldsymbol{n}, \boldsymbol{p}] : \mathbb{L}_2 \multimap \S\mathbb{F}_{2^n}$ is:

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

where:

$$\begin{aligned}
\mathtt{wModEnd} &\equiv \mathtt{\backslash l.wDrop\bot\,(wRev\,(wProj_1\,l))} \\
\mathtt{wModFun} &\equiv \mathtt{\backslash <e, s>.(\backslash <d, p>.((\backslash <s_0, s_1>.s_0\ S0is1\ S0is0\ S0isB\ d\ p\ s_1)\,s))\,e} \\
\mathtt{S0is1} &\equiv \mathtt{\backslash d\,p\,s.(\backslash <p', p''>.<<Xor\ d\ p', s>, <1, p''>>)\,(b\nabla_2\,p)} \\
\mathtt{S0is0} &\equiv \mathtt{\backslash d\,p\,s.<<d, s>, <0, p>>} \\
\mathtt{S0isB} &\equiv \mathtt{\backslash d\,p\,s.<<\bot, s>, <d, p>>} \\
\mathtt{wModBase[}\boldsymbol{p}\mathtt{]} &\equiv \mathtt{\backslash d.MapThread[\backslash ab.<a, b>]\,(wRev\,d)\,(wRev}\,\boldsymbol{p}\mathtt{)} \ .
\end{aligned}$$

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

$$\mathtt{wModFun} \ \mathtt{<<}\underbrace{\mathtt{d_i, p_i}}_{\text{elem. e}}\mathtt{>,}\ \underbrace{\mathtt{<s_0, p_{i+1}>>}}_{\text{status s}} \ \to^* \ \mathtt{<<}\underbrace{\mathtt{d_i{}', p_{i+1}}}_{\text{e'}}\mathtt{>,}\ \underbrace{\mathtt{<s_0{}', p_i>>}}_{\text{s'}} \ ,$$

where $\mathtt{s_0}$ keeps the m.s.b. of $\{\ldots \mathtt{d_i} \ldots\}$ and it is used to decide whether to reduce or not at this iteration. Thus, $\mathtt{d_i{}'} = \mathtt{d_i} + \mathtt{p_i}$ if $\mathtt{s_0} = \mathtt{1}$; $\mathtt{d_i{}'} = \mathtt{d_i}$ if $\mathtt{s_0} = \mathtt{0}$; and $\mathtt{d_i{}'} = \bot$ when $\mathtt{s_0} = \bot$ (that represents the initial state, when $\mathtt{s_0}$ 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).

$$
\begin{aligned}
\texttt{wMultStep} &\equiv \\
&\texttt{\textbackslash s l f x.wBMult[f] (l MSStep[f, wFMult] (MSBase[x] (tCast}^{\texttt{0}}\ \texttt{s})))} \\
\texttt{wBMult[f]} &\equiv \texttt{\textbackslash<w, s>.(\textbackslash<M, m'''>.f <m''', 0> w) s} \\
\texttt{MSStep[f, wFMult]} &\equiv \texttt{\textbackslash e.\textbackslash<w, s>.(\textbackslash<e', s'>.<f e' w, s'>) (wFMult e s)} \\
\texttt{MSBase[x]} &\equiv \texttt{\textbackslash s.<x, s>} \\
\texttt{wFMult} &\equiv \texttt{\textbackslash<m, r>.\textbackslash<M, m'''>.wFMultBody[m, r, M, m''']} \\
\texttt{wFMultBody[m, r, M, m''']} &\equiv \\
&\texttt{(\textbackslash<m', m''>.(\textbackslash<M', M''>.<<m''', bMult[m', M', r]>, <M'', m''>>) (b}\nabla_2\ \texttt{m)) (b}\nabla_2\ \texttt{M)} \\
\texttt{bMult[m', M', r]} &\equiv \texttt{Xor (And m' M') r} \\
\texttt{wMultBase} &\equiv \texttt{\textbackslash m.MapThread[\textbackslash a b.<a, b>] m } \{\varepsilon\}
\end{aligned}
$$

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

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: $\texttt{wSqr} : \mathbb{L}_2 \multimap \mathbb{L}_2$ that performs the bit expansion, and $\texttt{Sqr} : \mathbb{F}_{2^n} \multimap \S\mathbb{F}_{2^n}$ that is the actual square in $\mathbb{F}_{2^n}$. We have:

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

and $\texttt{wSqr} \equiv \texttt{\textbackslash l f x.l wSqrStep[f] x}$, where $\texttt{wSqrStep[f]} \equiv \texttt{\textbackslash e t.f 0 (f e t)}$ 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, $\texttt{wMult} : \mathbb{L}_2 \multimap \mathbb{L}_2 \multimap \S\mathbb{L}_2$ and the field operation $\texttt{Mult} : \mathbb{F}_{2^n} \multimap \mathbb{F}_{2^n} \multimap \S^2\mathbb{F}_{2^n}$, obtained by composing with the modular reduction. We have:

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

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 $\texttt{wMultStep} : \mathbb{B}_2^2 \multimap \mathbb{L}(\mathbb{B}_2^2) \multimap \mathbb{L}(\mathbb{B}_2^2)$ behaves as follows:

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

```
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 F  | FwdM              | fwdm
```

Figure 5: Definition of wInv.

where M is the current bit of the multiplier b, and every $\mathtt{m}_\mathtt{i}$ is a bit of the multiplicand a, and every $\mathtt{r}_\mathtt{i}$ is a bit in the current result. The iteration is enabled by the combinator $\mathtt{wMultBase} : \mathbb{L}_2 \multimap \mathbb{L}(\mathbb{B}_2^2)$, that, on input a, creates $\{<\mathtt{m}_{n-1}, \bot> \ldots <\mathtt{m}_0, \bot>\}$, setting the initial bits of the result to $\bot$. The projection $\mathtt{wProj}_2$ 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 $\mathtt{wFMult} : \mathbb{B}_2^2 \multimap \mathbb{B}_2^2 \multimap (\mathbb{B}_2^2 \otimes \mathbb{B}_2^2)$, that behaves as follows:

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

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

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)))
G2 = \f.\x.f 0 (f 0 (f 0 (f 0 x))) .
```
(10)

`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* the list (11) that `wInv` builds in its first step. We adopt the same name for every list whose tuples have eleven boolean elements with the position meaning that (11) highlights. The `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 is on top; its less significant bit lsb is at the bottom.

The variable `D` which appears in Figure 5 takes the type of a Church numeral and the term which follows `\tw.wRevInit (BkwVst (wRev (FwdVst tw)))` is the step function which is iterated starting from a threaded words built like (11) was. The step function implements steps from 2 through 5 of 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 it takes as input. `BkwdVst` stands for *backward visit*. `wRevInit` reverses the threaded words it gets in input while reinitializing 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 which we call `wFwdVstInput`. Its output is again a threaded words `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 `<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:

"Is `Stop[0]=1`?" (12)

"Does z divide both u and g_1?" (13)

"Does z divide u but not g_1?" (14)

"Neither of the previous questions has positive answer?". (15)

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

Let us assume instead that (13) or (14) hold. Answering the first question requires to verify `U[0]=0` and `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-1`th element of `wFwdVstOutput`. So, `FwdVst` generates the new `i`th element of `wFwdVstOutput` from them which will become the `i-1`th element of `wFwdVstOutput` in the succeeding step:

```
<U[i],FwdV[i-1],g1,FwdG2[i-1],FvdM[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) hold, 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) hold. The elements `FwdV[msb-1]`, `FwdG2[msb-1]` and `FwdM[msb-1]` come from `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, 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 assuming that neither (13) nor (14) hold. This occurs when `U[0]=1`. `FwdVst` must forcefully answer to: "Is u different from 1?". Answering the question requires a complete visit of the threaded words that `FwdVst` takes in input. The visit serves to verify whether some `j>0` exists such that `U[j]=1`. The non existence of `j` implies that `FwdVst` sets `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

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 at least u. In case of positive 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`.

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 `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 `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.

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 0, 0,1,   B,     0,        0,   1,    0,   1>
(f <1,1,Xor 0 0, 0,0,   B,     0,        0,   0,    0,   0>          (20)
(f <0,1,Xor 0 1, 0,1,   B,     0,        0,   1,    0,   1># new lsb
(f <0,B,Xor 1 1, 0,1, . B,     0,        0,   1,    0,   1># orig. lsb
                                                        x))))
```

The threaded words (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># orig. lsb
(f <0,1,Xor 0 1, 0,1,   B,     0,        0,   1,    0,   1># new lsb
(f <1,1,Xor 0 0, 0,0,   B,     0,        0,   0,    0,   0>          (21)
(f <0,0,Xor 0 0, 0,1,   B,     0,        0,   1,    0,   1>
(f <0,1,      0, 0,1,   B,     B,        0,   B,    B,   B># msb
                                                        x))))
```

`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,   1,    0,   1>
(f <1,1,Xor 0 0, 0,0,   B,     0,        0,   0,    0,   0>          (22)
(f <0,0,Xor 0 0, 0,1,   B,     0,        0,   1,    0,   1>
(f <0,1,      0, 0,1,   B,     B,        0,   B,    B,   B> x)))
```

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 0, 0,1,   B,     B,        B,   0,    0,   0>          (23)
(f <1,1,Xor 0 0, 0,0,   B,     B,        B,   0,    0,   0>
(f <0,1,Xor 0 1, 0,1,   B,     B,        B,   0,    0,   0> x)))
```

Let us compare (23) and (20). All the bits of position `U` and `G1` have been shifted while those ones of position `V`, `G2` and `M` have not. Moreover, the bits of position `Stop`, ..., `FwdM` have been reinitialized so that (23) is a consistent input for `FwdVst`. We remark that the whole process of shifting the bits of positions `U` and `G1` requires the concatenation of both `FwdVst` and `BkwdVst` up to some reverse. The first one shifts the bits of position `V`, `G2` and `M` to the left while operates on those of position `U` and `G1`. The latter erases the correct element and fully realizes the shift to the right.

The code of `FwdVst` ***and of*** `BkwdVst`. We recall that `FwdVst` and `BkwdVst` follow the programming pattern of `wHeadTail[L,B]`. The step functions they relies on and their "last step functions" implement branching. Choices of the branching structures depend on the values of the bits that belong to the state or on the values of some bits of `U` or `G1`. We talk of pseudo-code because Figure 7 adds obvious syntactic sugar to the syntax of λ-terms. Let `N` be of type $\mathbb{B}_2$. Then `N M1 M0 MB` is a λ-term which eventually chooses among `M1`, `M0` and `MB`, depending on the normal form `N` evaluates to. The `switch`-structure:

```
switch (N)     {
   case 1: ...
   case 0: ...
   case B: ... }
```
(24)

represent `N M1 M0 MB`. The name of variables in the pseudo-code should recall their meaning. In Figure 7, `stopt` recalls "`Stop` of the tail", i.e. "`Stop` that comes from step `msb-1`". Analogously `rst` is "`RghtShft` that comes from step `msb-1`". Let us focus on the two branches with `stopt=B` and `rst=1` or `rst=0`. They take care of the situations that require the shift to the right of `U` and `G1`. I.e., if we think in general terms, they generate the two elements in (19). If we prefer to think in terms of our example, they generate the two topmost elements in (20). We remark that `LastStepFwdVst` is completely linear. Branching after branching it yields a λ-abstraction that correctly builds required elements that complete the threaded words under construction.

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

Decision networks analogous to the one in Figure 8 exist for all the components of `wInv`. For example, Figure 9, 10 , 11 and 12 summarize the essentials of the decision network that the step function `SFwdVst` (see Appendix C) of `FwdVst` implements. Again we have to trace how the names of variables in the flow-chart link to the names of variables of the pseudo-code correspond. If we assume we are at step `i`, then `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]`.

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 $\mathtt{F} \equiv \backslash \mathtt{a}_1 \ \ldots \mathtt{a}_{11}.\langle \mathtt{a}_1, \ldots, \mathtt{a}_{11} \rangle : \mathbb{B}_2^{11}$. Figure 13 lists the types of the main components of `wInv`. We remark that `FwdVst`, `BkwdVst`, `LastStepFwdVst` and `wRevInit` map a threaded words to another threaded words. So their composition can be used, as we do, as a step function in a iteration.

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

Both `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 generalizations 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 8. The main point to assure we can give a type to `SFwdVst`, `BFwdVst`, ...

```
LastStepFwdVst =
\f.
\<ft,et,t>. # Element from step i-1.
(\<ut,vt,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.
      (f <0,vt,0,g2t,mt,B,B,1,B,B,B >
         (ft <ut,fwdvt,g1t,fwdg2t,fwdmt,B,snt,1,B,B,B> t ))
     case 0: # of rst. U[0]=0 and G1[0]=1. We are shifting and we
             # have to add a new msb to the threaded words.
      \f.\ft.\ut.\vt.\g1t.\g2t.\mt.\snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
      (f <0,vt,0,g2t,mt,B,B,0,B,B,B >
         (ft <ut,fwdvt,g1t,fwdg2t,fwdmt,B,snt,0,B,B,B> t))
     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`.

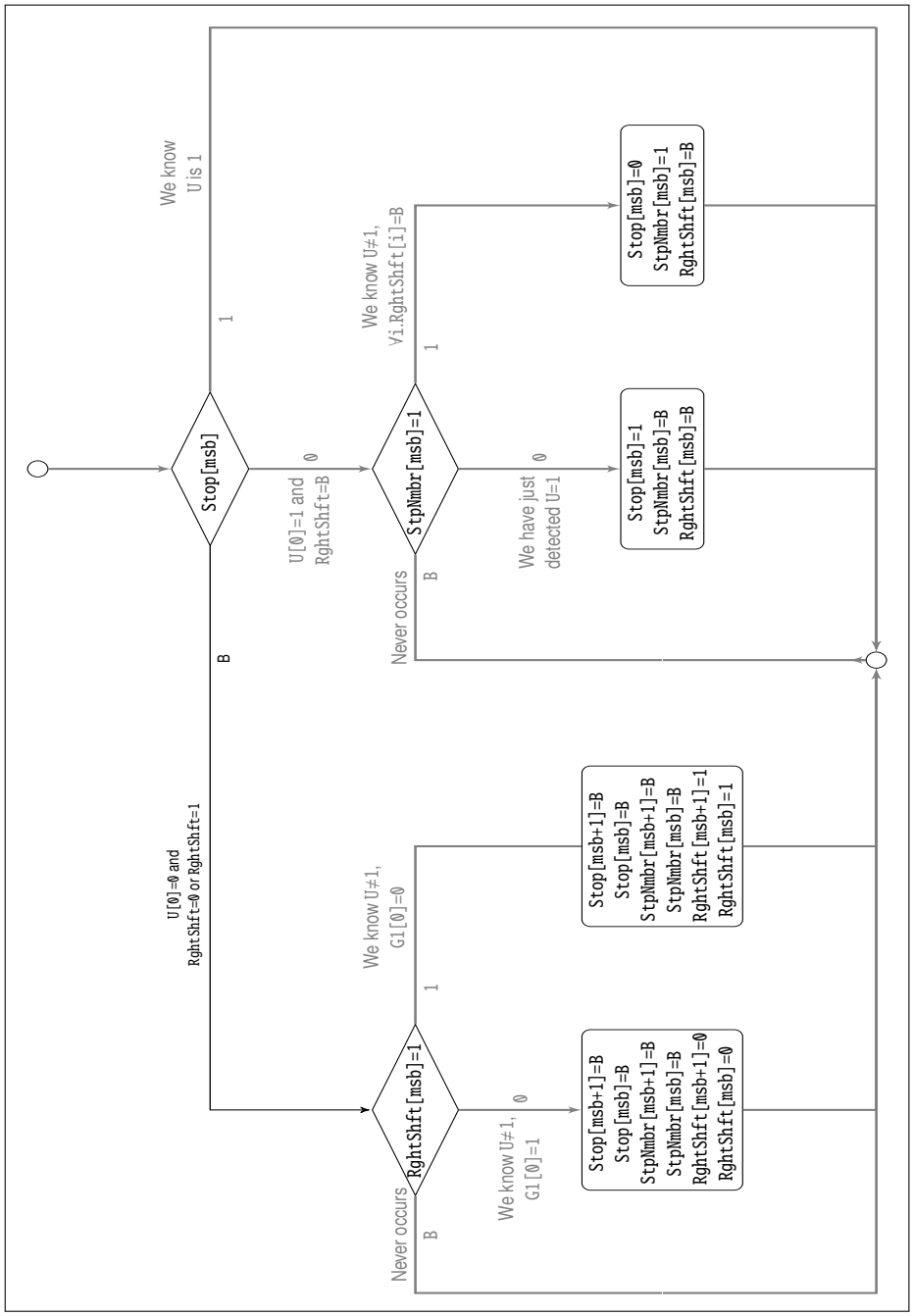


Figure 8: Flow-chart of the decision network that `LastStepFwdVst` implements.

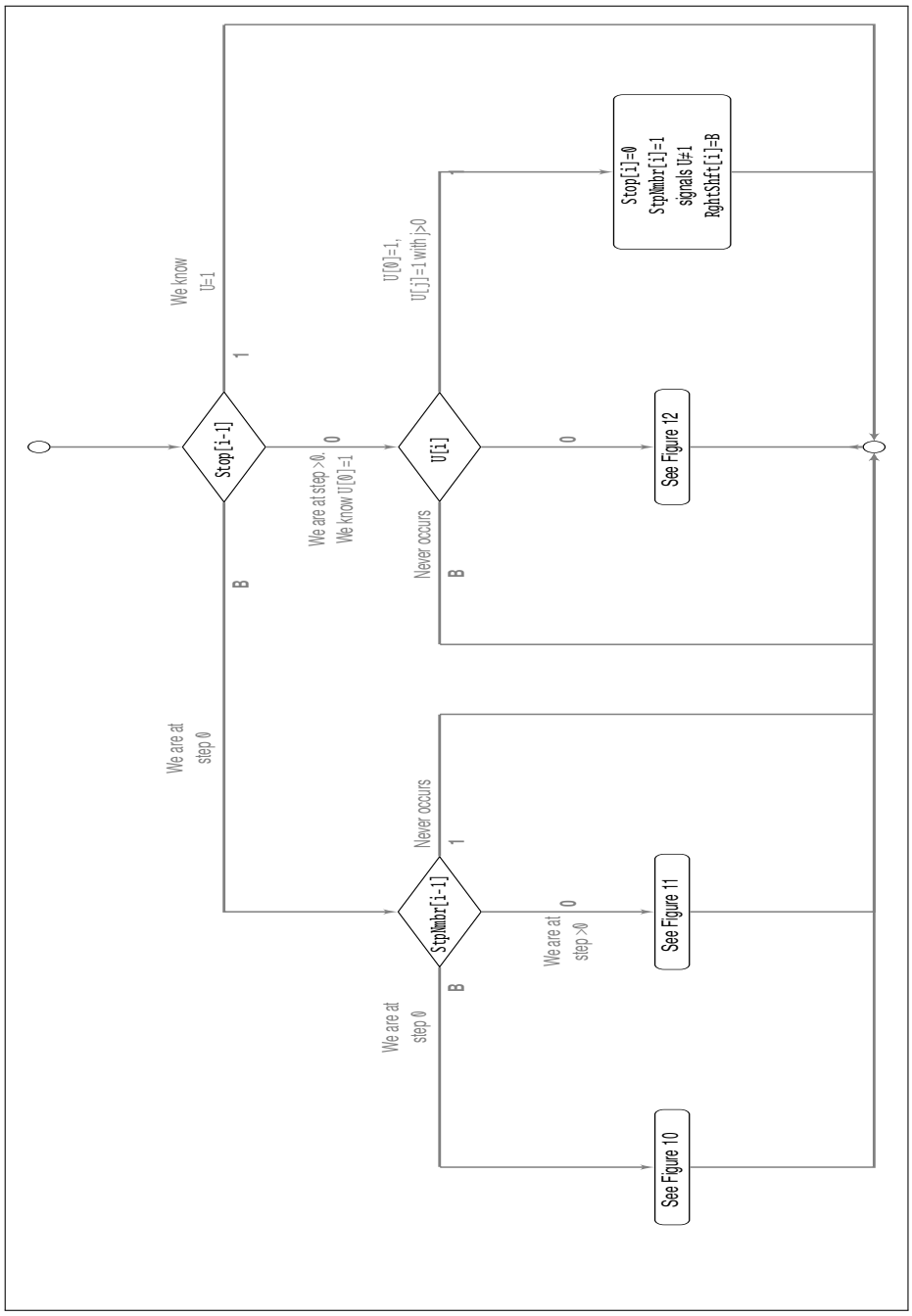


Figure 9: Flow-chart of the decision network that the step function `SFwdVst` of `FwdVst`.

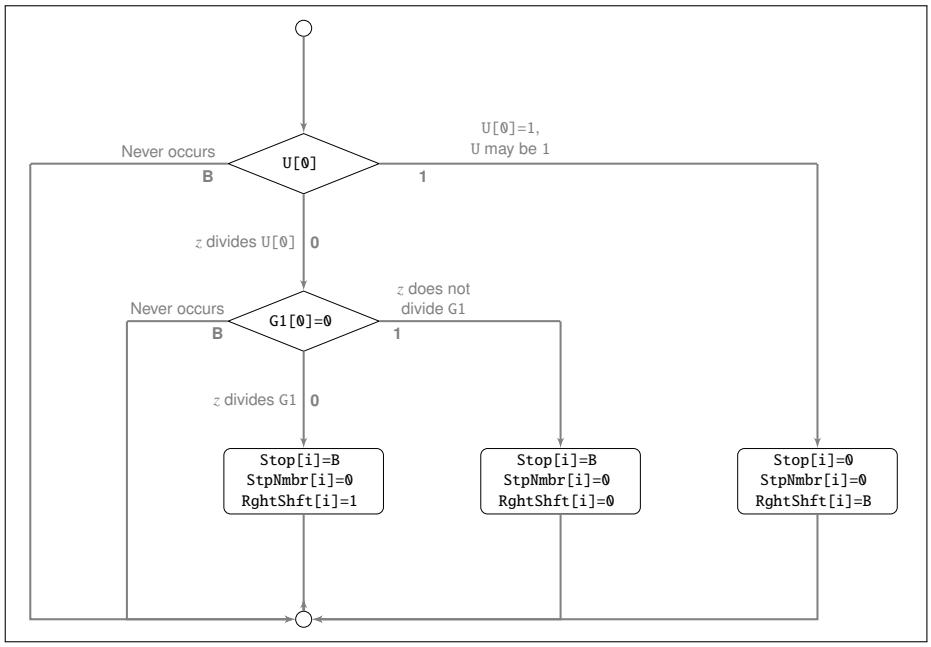


Figure 10: First component of the decision network that the step function `SFwdVst` of `FwdVst` implements.

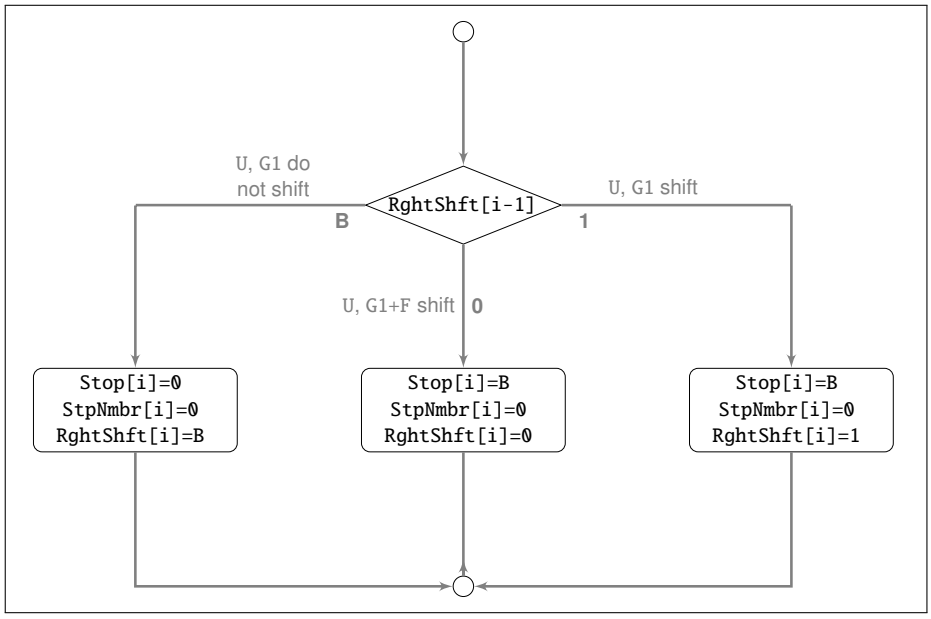


Figure 11: Second component of the decision network that the step function `SFwdVst` of `FwdVst` implements.

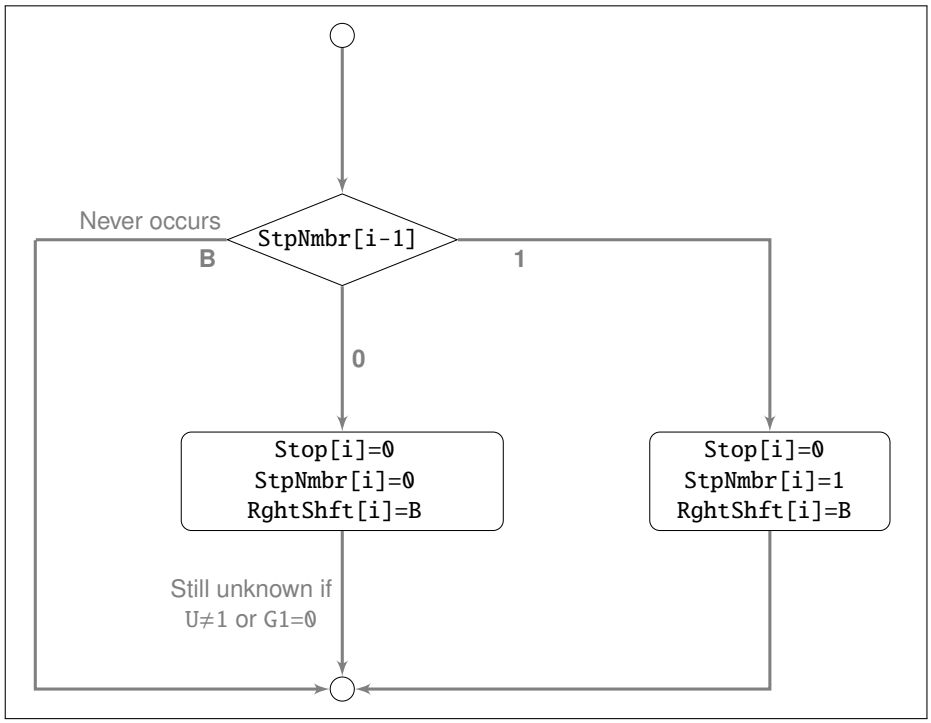


Figure 12: Third component of the decision network that the step function `SFwdVst` of `FwdVst` implements.

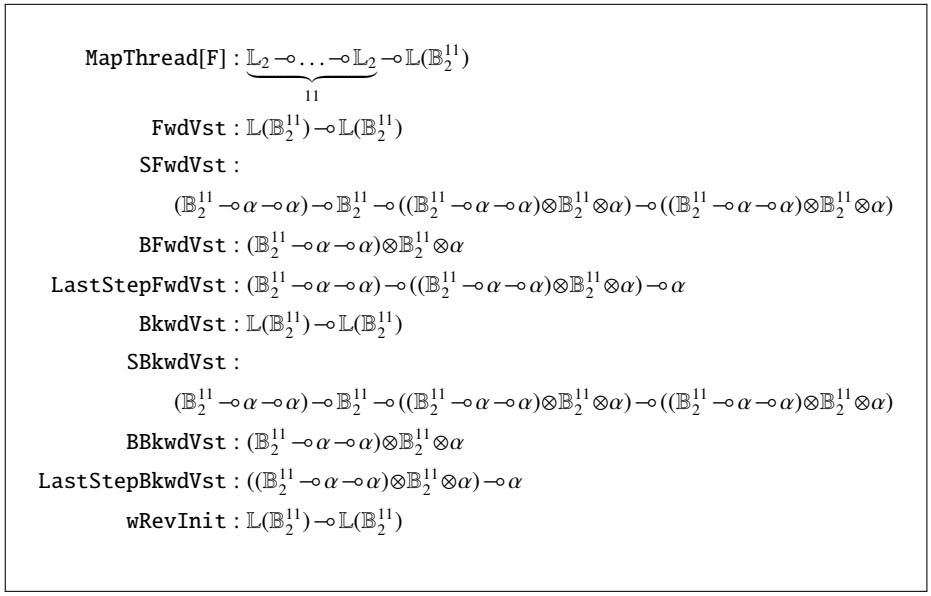


Figure 13: The types of the main sub-terms of `wInv`.

is to organize them so that every possible choice results in a closed term. This maintains as much linear as we can the whole term, so letting it iterable and simply composable.

5. Conclusions and future work

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

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

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

On the other side we missed the development of a complete realistic applicative example, such as elliptic curves cryptography. In the same line, the implementation of symmetric-key cryptographic algorithms (block/stream ciphers, hash functions, ...) looks attractive, thanks to the higher-order bitwise operations at the core of the library.

Next, we shall investigate a compilation process targeting parallelization, which, in general follows from functional programming thanks to the reduced data dependency it embodies. This goal should be feasible because the lambda terms we write to implement finite fields arithmetic exploit programming patterns that can be assimilated to the MapReduce paradigm [16].

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

References

[1] P. Baillot, M. Gaboardi, V. Mogbil, A polytime functional language from Light Linear Logic, in: A. D. Gordon (Ed.), ESOP, Vol. 6012 of Lecture Notes in Computer Science, Springer, 2010, pp. 104–124.

[2] P. Baillot, V. Mogbil, Soft lambda-calculus: A language for polynomial time computation, in: I. Walukiewicz (Ed.), FoSSaCS, Vol. 2987 of Lecture Notes in Computer Science, Springer, 2004, pp. 27–41.

[3] M. Hofmann, Linear types and non-size-increasing polynomial time computation, Information and Computation 183 (1) (2003) 57–85.

[4] M. J. Burrell, R. Cockett, B. F. Redmond, POLA: a language for PTIME programming, in: Tenth International Workshop on Logic and Computational Complexity, Los Angeles, USA, 2009.
URL http://projects.wizardlike.ca/attachments/2/LCC09_19.pdf

[5] U. Dal Lago, M. Hofmann, Bounded linear logic, revisited, Log. Methods Comput. Sci. (Special issue: Selected papers of the conference "Typed Lambda Calculi and Applications 2009") (2010) 4:7, 31.

[6] L. Roversi, Light affine logic as a programming language: a first contribution, Internat. J. Found. Comput. Sci. 11 (1) (2000) 113–152, advances in computing science—Asian'98 (Manila).
URL http://dx.doi.org/10.1142/S0129054100000077

[7] 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

[8] U. Dal Lago, Context semantics, linear logic, and computational complexity, ACM Transactions on Computational Logic 10 (4) (2009) Art. 25, 32.
URL http://dx.doi.org/10.1109/LICS.2006.21

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

[10] 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

[11] 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).

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

[13] 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.

[14] 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.

[15] 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.

[16] 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`

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

[18] P. Baillot, M. Pedicini, Elementary complexity and Geometry of Interaction, Fundamenta Informaticae 45 (1-2) (2001) 1–31, Typed Lambda Calculi and Applications (L'Aquila, 1999).

[19] M. Pedicini, Remarks on elementary Linear Logic (preliminary report), in: Linear logic 96 (Tokyo), Vol. 3 of Electronic Notes Theoretical Computer Science, Elsevier, Amsterdam, 1996, p. 12 pp. (electronic).

Appendix A. Definition of Basic Combinators

We recall the following definitions from [13].

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

$\mathtt{b}\overline{\nabla}_t$ is $\backslash$b.b $<\overbrace{1\ldots1}^{t}><\overbrace{0\ldots0}^{t}\succ\overbrace{\bot\ldots\bot}^{t}>$, for every $t \geq 2$.

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

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

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

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

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

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

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

`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 ⊥) ⊥ c`.

`sSpl` is `\s.s (\t.<⊥, [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` : $\mathbb{B}_2 \multimap \alpha \multimap \alpha$.

$\mathtt{wDrop}_\bot$ 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]`.

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

$\mathtt{wProj}_2$ 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:

$$\mathtt{MSStep[F, f]} \equiv \mathtt{\backslash e.\backslash <w, s>.(\backslash <e', s'>.<f\ e'\ w, s'>)\ (F\ <e, s>)}$$
$$\mathtt{MSBase[x]} \equiv \mathtt{\backslash s.<x, s>}\ .$$

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[x] (w2s (wRev m))))` with `F` : $\mathbb{B}_2 \multimap \mathbb{B}_2 \multimap A$ closed, `w2s (wRev m)` : $\S\mathbb{S}$ whenever $m : \mathbb{L}_2$ and:

$$\mathtt{MTStep[F, f]} \equiv \mathtt{\backslash a.\backslash <w, s>.(\backslash <b, s'>.<f\ (F\ a\ b)\ w, s'>)\ (sSpl\ s)}$$
$$\mathtt{MTBase[x]} \equiv \mathtt{\backslash x.<x, m>}\ .$$

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

Appendix B. Some examples of type inference

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

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

of Church numerals. We have $\texttt{uSuc} : \mathbb{U} \multimap \mathbb{U}$, in accordance with the type inference:

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

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.

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 $\texttt{tCast}^m$, wSuc, $\texttt{wCast}^m$, $\texttt{w}\nabla_t^m$, 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 $\texttt{L} : X \multimap \alpha$ and $\texttt{B} : X$. The type assignment of wHeadTail[L, B] follows:

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

where Π_1 is:

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

and Π_2 is:

$$\dfrac{\dfrac{}{\emptyset \mid \emptyset \vdash \texttt{wHTStep[B]}:(A\multimap\alpha\multimap\alpha)\multimap(A\multimap X\multimap X)} \quad \dfrac{}{\emptyset \mid \texttt{f}:A\multimap\alpha\multimap\alpha \vdash \texttt{f}:A\multimap\alpha\multimap\alpha}\,\text{a}}{\emptyset \mid \texttt{f}:A\multimap\alpha\multimap\alpha \vdash \texttt{wHTStep[B] f}:A\multimap X\multimap X}\,\multimap\!\text{E}$$

and Π_3 is:

$$\dfrac{\dfrac{\dfrac{}{\emptyset \mid \emptyset \vdash \texttt{L}:X\multimap\alpha} \quad \dfrac{\dfrac{}{\emptyset \mid \texttt{y}:X\multimap X \vdash \texttt{y}:X\multimap X}\,\text{a} \quad \dfrac{\dfrac{}{\emptyset \mid \emptyset \vdash \texttt{wHTBase}:\alpha\multimap X} \quad \dfrac{}{\emptyset \mid \texttt{x}:\alpha \vdash \texttt{x}:\alpha}\,\text{a}}{\emptyset \mid \texttt{x}:\alpha \vdash \texttt{wHTBase x}:X}\,\multimap\!\text{E}}{\emptyset \mid \texttt{y}:X\multimap X, \texttt{x}:\alpha \vdash \texttt{y (wHTBase x)}:X}\,\multimap\!\text{E}}{\emptyset \mid \texttt{y}:X\multimap X, \texttt{x}:\alpha \vdash \texttt{L (y (wHTBase x))}:\alpha\multimap\alpha}\,\multimap\!\text{I}}{\emptyset \mid \texttt{y}:X\multimap X \vdash \texttt{\textbackslash x.L (y (wHTBase x))}:\alpha\multimap\alpha}\,\multimap\!\text{I}}{\emptyset \mid \texttt{y}:\S(X\multimap X) \vdash \texttt{\textbackslash x.L (y (wHTBase x))}:\S(\alpha\multimap\alpha)}\,\S\text{I}$$

.

Appendix C. Pseudocode of the main components of wInv

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

```
SFwdVst =
\f.
\<u,v,g1,g2,p,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 u[i]:
                   # -) one for branching
                   # -) one to be inserted in the list
  \<sntb1,sntb2>.  # copies of sn[i-1] for branching
  (switch (stopt) {
    case 1: # of stopt. We checked U=1. wInv must be
            # the identity
     \f.\u.\v.\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,stop,sn,rs,fwdv,fwdg2,fwdm>
     ,ft <ut,vt,g1t,g2t,mt,0,B,rst,fwdvt,fwdg2t,fwdmt> t>
    case 0: # of stopt. We are at a step>0 and we know
            # U[0]=1. We do not have to shift anything
     switch (uba) {
      case 1: # of uba. U contains at least two occurrences
              # of 1. I.e. U[0]=1, U[j]=1 and j>0
       \f.\u.\v.\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.
         ,0   # Stop keeps recording that U[0]=1
         ,1   # StepNumber keeps recording we are at step>0
              # It also signals U[0]=1, U[j]=1 and j>0,
              # This means the whole U!=1
         ,B   # RightShift keeps recording that
              # neither of U, 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.
   <f,<u   # Value of this step
      ,vt  # Value from step i-1
      ,g1  # Value of this step
      ,g2t # Value from step i-1
      ,pt  # 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,vt,g1t,g2t,mt,B,0,1,fwdvt,fwdg2t,fwdmt> t>
  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.
   (\<me1,me2>. # two copies of m to build elements
    <f,<u   # Value of this step
       ,vt  # Value from step i-1
       ,Xor g1 me1 # Values of this step
       ,g2t # Value from step i-1
       ,mt  # 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,g2t,pt,B,0,0,fwdvt,fwdg2t,fwdmt> t>
   ) (m <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.
   <f,<1,v,g1,g2,p
      ,0 # Stop keeps storing that U[0]=1
      ,0 # StepNumber keeps recording
         # we are at step>0
      ,B # RightShift keeps recording that
         # neither of U, G1 shift
      ,B,B,B > # dummy values
   ,ft <ut,vt,g1t,g2t,mt,B,0,B,fwdvt,fwdg2t,fwdmt> t>
 } # switch rst end.
case B: # of sntb2. We are at step 0
        # We must check the value of U[lsb], G1[lsb]
 switch (ubb) {
  case 1: # of ubb. z does not divide U.
          # I.e. U[0]=1. Moreover, U may be 1.
          # I.e. the only bit equal to 1 is U[0]
   \f.\u.\v.\g1.\g2.\m.
      \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
      \ut.\vt.\g1t.\g2t.\mt.
            \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
   <f,<1,v,g1,g2,m
      ,0 # Stop records that U[0]=1
      ,0 # StepNumber 'increases' by 1
      ,B # RightShift records that neither of U, G1 shift
      ,B,B,B > # dummy values
   ,ft <ut,vt,g1t,g2t,mt,B,B,rst,fwdvt,fwdg2t,fwdmt> t>
  case 0: # of ubb. z divides U i.e. U[0]=0
   switch (gb) {
    case 1: # of gb. z does not divide G1, i.e. G1[0]=1.
     \f.\u.\v.\g1.\g2.\m.
        \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
        \ut.\vt.\g1t.\g2t.\mt.
              \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
     (\<me1,me2>. # two copies of m to build elements
      <f,<0
         ,B  # Dummy value. This is the lsb of V.
             # We shall erase it
         ,Xor 1 me1
         ,B  # Dummy value. This is G2[lsb]
             # We shall erase it
         ,B  # Dummy value. This is the M[lsb].
             # We shall erase it
         ,B  # Forward Stop which records that U[0]=0
         ,0  # Forward StepNumber
         ,0  # Forward RightShift which records
             # that U, G1+F must shift
         ,v  # Forward the three bits that
```

```
                # must shift to the left
             ,g2
             ,me2 >
         ,ft <ut,vt,g1t,g2t,mt,B,B,rst,fwdvt,fwdg2t,fwdmt> t>
        ) (m <1,1> <0,0> <B,B>)
       case 0: # of gb. z divides G1, i.e. G1[0]=0
        \f.\u.\v.\g1.\g2.\m.
           \stop.\sn.\rs.\fwdv.\fwdg2.\fwdm.
           \ut.\vt.\g1t.\g2t.\mt.
                 \snt.\rst.\fwdvt.\fwdg2t.\fwdmt.\t.
        <f,<0
           ,B  # Dummy value. This is V[lsb].
               # We shall erase it
           ,0
           ,B  # Dummy value. This is G2[lsb].
               # We shall erase it
           ,B  # Dummy value. This is M[lsb].
               # We shall erase it
           ,B  # Forward Stop which records that U[0]=0
           ,0  # Forward StepNumber
           ,1  # Forward RightShift which records
               # that U, G1 must shift.
           ,v  # Forwarding the three bits that
               # must shift to the left
           ,g2
           ,m >
        ,ft <ut,vt,g1t,g2t,mt,B,B,rst,fwdvt,fwdgbt,fwdpt> t>
       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, p, 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[i] may serve for branching.
                                 # The second one serves to
                                 # build a new state.
   (snt   <1,1>   <0,0>   <B,B>) # Both copies of sn[i-1] serve
                                 # for branching.
) et
```

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

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

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

```
SBkwdVst =
\f.
```

```
\<u,v,g1,g2,m,stop,sn,rs,_,_,_>.
\<ft,et,t>.
(\<ut,vt,g1t,g2t,mt,stopt,snt,rst,_,_,_>.
 (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,B,B,B> # Dummy values.
       >
    ,ft <ut,vt,g1t,g2t,mt,1,snt,rst,B,B,B> 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 (rst) {
     case 1: # of rst. 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 (rst) {
  case 1: # of rst. So U[0]=0. Keep propagating
          # RightSihft=1. The last step will compute
          # the predecessor of the input threaded words
          # to implement the shift to the right U and one
          # between G1 or G1+F
   \u.\v.\g1.\g2.\m.\stop.\sn.\rs.
   \ut.\vt.\g1t.\g2t.\mt.\stopt.\snt.\rst.
   <f,<u,v,g1,g2,m,
      ,B  # Propagation of Stop=B.
      ,sn # Dummy value.
      ,1  # Keep propagating RightSihft=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 RightSihft
       ,B,B,B> x
   >
  case 0: # of rst. Never occurs because the base case, i.e.
          # stopt=B and rst=B and Stop=B, sets RightShift=1
          # which the case here above with rst=1 keeps
          # propagating. This is not a mistake because it is
          # important to calculate the predecessor in the
          # course of the very last step
   \u.\v.\g1.\g2.\m.\stop.\sn.\rs.
   \ut.\vt.\g1t.\g2t.\mt.\stopt.\snt.\rst.
   <f,<u,v,g1,g2,m,
      ,B  # Propagation of Stop=B
      ,sn # Dummy values
      ,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 RightSihft=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> # Dummy values.
     ,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 # StepNumper=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
            # RightSihft=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 values.
        ,1 # Propagate RightSihft=1. I.e. we shall calculate
           # the predecessor on the threaded words in input.
           # The predecessor realizes the shift to the right.
           # Propagating 0 in place of 1 would yield the
           # same result
        ,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,u,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
```