Can a light typing discipline be compatible with efficient implementation of finite field inversion ?

Daniele Canavese[1], Emanuele Cesena[2], Rachid Ouchary[1], Marco Pedicini[3], and Luca Roversi[4]

[1] Politecnico di Torino, Dip. di Automatica e Informatica, Torino, Italy
daniele.canavese@polito.it, rachid.ouchary@polito.it
[2] Theneeds Inc., San Francisco, CA
ec@theneeds.com
[3] Università degli Studi Roma Tre, Dipartimento di Matematica e Fisica, Roma, Italy
pedicini@mat.uniroma3.it
[4] Università degli Studi di Torino, Dip. di Informatica, Torino, Italy
roversi@di.unito.it

Abstract. We show that an algorithm implementing the Binary-Field Arithmetic operation of *multiplicative inversion* exists as a purely functional term which is typeable in *Dual Light Affine Logic* (DLAL). As a consequense, the set Λ^{DLAL} of functional terms typeable in DLAL is large enough to program the whole set of arithmetic operations. Second, and most important, we show that Λ^{DLAL} can be seen a domain specific language that forces the programmer to think about algorithms under a non standard mental pattern which may results in more essential descriptions of known algorithms which, also, may be more efficient.

1 Introduction

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

1. efficient arithmetics 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 fields has many important applications in the domains of theory of codes and in cryptography. Finite fields' arithmetic operations include: addition, subtraction, multiplication, squaring, square root, multiplicative inverse, division and exponentiation.

Declarative programming, by its nature, does not permit a tight control on complexity parameters, however the scenario is changed in the last twenty years with the introduction of type systems with implicit complexity bounds where implicit means that the inclusion in the complexity class is certified by the type system: if the program is typeable then the type guarantees on execution time. The point in so far considered type systems is that the restrictions on programming schemes hardly permit to specify any algorithm 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.

Our pragmatic workplan is to make fully operational a declarative framework where feasible arithmetics can be programmed in a certified environment ensuring that complexity is polynomial, for this reason we introduced in [1] a variant of the system DLAL [2], that we called TFA (typeable functional assembly) having in mind what kind of programming patterns should be used in arithmetics. In fact, we would like to have an even improved control on these systems in order to certify polynomial computations up to a certain exponent, maybe as a development on the quantitative approach introduced in [3].

We build on our previous paper where we introduced basic materials in order to make arithmetics in 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 their typeability in the formal system: and this is an obstacle in the use of such systems. On the other hand it is maybe because the unusual programming pattern that all these difficulties arise. It is because in these systems iteration is forced to add a marker in the type which blocks nesting of iterations. On the road we discovered and put in practice several patterns derived by classical Map or MapThread terms, and the result contained in this paper completes the work by proving that also the term for inversion in finite fields is in TFA.

The efficient implementation of inversion we started from is known as BEA (binary euclidean algorithm) in the realization given by Fong in [4], see Figure 1. It declines perfectly by the imperative programming toolbox: direct assignments of variables in memory, control flow in the form of a double nested iteration: a goto-statement which creates a loop and an internal while statement. It is obvious that in a declarative programming language there is no goto-statement and even while-loops are to be realised by structural iterations on some data type (typically Church integers). This was a first step in order to write the algorithm in a declarative style, the second one was to simulate those operations with 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: in fact, the point is to think like if terms would be linear terms (any variable is used once), and then very carefully relax to have non-linear variables. This is our main result: in Section 3 we

```
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.
```

Fig. 1: Binay-Field inversion as in **Algorithm 2.2** at page 1048 in [4].

show a typeable multiplicative inversion in TFA. Then in Section 4 we go back to imperative: we give a new implementation of inversion which we called DCEA, the imperative version of the typeable term which implements inversion in TFA. DCEA is the imperative version of the control flow we had to realise in order to comply with typing in TFA. Note that in principle we could perform less efficiently because of the operation rearrangements. In fact, our experimental data show that even with our potentially longer loops we do not loose too much. On the other hand, and this is the main reason to compare the two algorithms in the imperative setting: the regular loops should be easier to be optimised by compiler and easier to be executed by speculative prediction of conditional branches. In Section 5, we have measures of the behaviour of our algorithm DCEA, where it is clear that the new version do not sacrifice efficency for polynomial certification, since it follows very closely timings of BEA. Moreover, by taking timings on different architectures we obtained compatible results in all the cases and we outperformed execution time of BEA while using a machine with UltraSPARC architecture.

2 Typeable Functional Assembly

This work keeps developing the project started in [1] which introduces Typeable Functional Assembly (TFA). TFA is the type assignment for λ-terms in Figure 2. TFA is DLAL [2] whose set of formulas is quotiented by a specific recursive equation we shall recall from [1] in a few. Here we reintroduce the strictly necessary notions about TFA.

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

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

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

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

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

Fig. 2: Type assignment system TFA

Every judgment $\Delta \mid \Gamma \vdash \mathtt{M}:A$ comes with two different kinds of contexts. The formula A is assigned as type to the λ-term $\mathtt{M}$ with hypothesis from the *polynomial context Δ* and the *linear context Γ*.

In TFA we define every single ground data type.

Let $\mathcal{G}$ be a countable set of *variables*. We range over $\mathcal{G}$ by *lowercase Greek letters*. Any *type A* belongs to the quotient $\mathcal{F}/\approx$ of the following language $\mathcal{F}$ *of formulas*:

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

We define the quotient on $\mathcal{F}$ when introducing the *Sequences of booleans* here below. *Uppercase Latin letters A, B, C, D* will range over $\mathcal{F}/\approx$. *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. 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:

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

with $\mathcal{V}$ the set of variables we range over by *any lowercase Latin Teletype letter*. *Uppercase Teletype Latin letters* $\mathtt{M}, \mathtt{N}, \mathtt{P}, \mathtt{Q}, \mathtt{R}$ will range over Λ. We shall write $\backslash\mathtt{x}.\mathtt{M}$ in place of $(\backslash\mathtt{x}.\mathtt{M})$ in absence of ambiguity. Application $((\mathtt{M}_1\mathtt{M}_2)...\mathtt{M}_n)$ is left associative. We shall tend to shorten it as $\mathtt{M}_1\mathtt{M}_2...\mathtt{M}_n$. The set of free variables of $\mathtt{M}$ is $\mathrm{fv}(\mathtt{M})$. The standard β-reduction $\rightarrow^*$ on λ-terms is the reflexive, transitive, and contextual closure of:

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

Both polynomial and linear contexts are maps $\{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n\}$ from the domain of variables $\mathcal{V}$, to the co-domain of formulas $\mathcal{F}$. The difference between the two kinds of context is that variables in the polynomial context may occur an arbitrary number of times in the *subject* $\mathtt{M}$ of $\Delta \mid \Gamma \vdash \mathtt{M}:A$. Every variable in the linear context must occur at most once in $\mathtt{M}$. Every pair $\mathtt{x}:A$ of any kind of context is a *type assignment for a variable*. The notation $\S\Gamma$ is a shorthand for $\{\mathtt{x}_1 : \S A_1, \ldots, \mathtt{x}_n : \S A_n\}$, if Γ is $\{\mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n\}$.

Finite types are functions that project one argument out of the many they have:

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

Finite types with $n = 2$ are *lifted booleans* we denote by $\mathbb{B}_2$. Their canonical representatives are:

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

We need $\bot$ in order to simplify the definition of the functions we want to program.

The type *Tuples* is:

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

If we extend Definition (1) as follows:

$$\mathtt{M} ::= \ldots \mid \langle \mathtt{M}, \ldots, \mathtt{M} \rangle \mid \backslash\langle \mathcal{V}, \ldots, \mathcal{V} \rangle.\mathtt{M}\ .$$

and we extend β-reduction:

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

we can type the new terms with the following rules derivable in TFA:

$$\frac{\Delta_1 \mid \Gamma_1 \vdash \mathtt{M}_1 : A_1 \quad \ldots \quad \Delta_n \mid \Gamma_n \vdash \mathtt{M}_n : A_n}{\Delta_1, \ldots, \Delta_n \mid \Gamma_1, \ldots, \Gamma_n \vdash \langle \mathtt{M}_1, \ldots, \mathtt{M}_n\rangle : (A_1 \otimes \ldots \otimes A_n)} \otimes\mathrm{I} \qquad \frac{\Delta \mid \Gamma, \mathtt{x}_1 : A_1, \ldots, \mathtt{x}_n : A_n \vdash \mathtt{M} : B}{\Delta \mid \Gamma \vdash \backslash\langle \mathtt{x}_1, \ldots, \mathtt{x}_n\rangle.\mathtt{M} : (A_1 \otimes \ldots \otimes A_n) \multimap B} \multimap\mathrm{I}_{\otimes}$$

The *Sequences of booleans*, or simply Sequences, is the following recursive type:

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

The recursive definition of $\approx$ in (3) determines the equivalence relation $\mathcal{F}/\approx$ among formulas of $\mathcal{F}$. As we said we take $\mathcal{F}$ up to such a relation. I.e., if M has type $\mathbb{S}$, then we can equivalently use any of the "unfolded forms" of $\mathbb{S}$ as type of M. The canonical values of type $\mathbb{S}$ are:

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

In accordance with (3), the Sequence $[\mathtt{b}_{n-1} \ldots \mathtt{b}_0]$ that occurs in (4) is a function that takes two constructors as inputs and yields a Sequence. Only the second constructor is used in (4) to build a Sequence out of a pair whose first element is $\mathtt{b}_{n-1}$, and whose second element is — recursively! — another Sequence $[\mathtt{b}_{n-2} \ldots \mathtt{b}_0]$. It is well known that adding recursive equations among the formulas of DLAL is harmless as far as Polynomial time soundness is concerned [5, 2].

2.1 Basic types and (meta-)combinators

Church numerals. Their type is $\mathbb{U} \equiv \forall\alpha.\mathbb{U}[\alpha]$ where $\mathbb{U}[\alpha] \equiv !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ with canonical representatives:

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

Lists. Their type is $\mathbb{L}(A) \equiv \forall\alpha.\mathbb{L}(A)[\alpha]$ where $\mathbb{L}(A)[\alpha] \equiv !(A \multimap \alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ with canonical representatives:

$$\begin{aligned} \{\varepsilon\} &\equiv \backslash \mathtt{f}.\backslash \mathtt{x}.\mathtt{x} : \mathbb{L}(A) \\ \{\mathtt{M}_{n-1} \ldots \mathtt{M}_0\} &\equiv \backslash \mathtt{f}.\backslash \mathtt{x}.\mathtt{f}\ \mathtt{M}_{n-1}(\ldots\ (\mathtt{f}\ \mathtt{M}_0\ \mathtt{x})\ldots) : \mathbb{L}(A) \text{ with } n \text{ occurrences of } \mathtt{f} \end{aligned}$$

Church words $\{\mathtt{b}_{n-1} \ldots \mathtt{b}_0\}$, with all $\mathtt{b}_i$s booleans, are the typical instance of lists with type $\mathbb{L}_2 \equiv \mathbb{L}(\mathbb{B}_2)$ we need. In every Church word $\{\mathtt{b}_{n-1} \ldots \mathtt{b}_0\}$, or simply *word*, the *least significant bit* (l.s.b.) is $\mathtt{b}_0$, while the *most significant bit* (m.s.b.) is $\mathtt{b}_{n-1}$. The same convention holds for every Sequence $[\mathtt{b}_{n-1} \ldots \mathtt{b}_0]$.

The combinator Xor. It ha type $\mathbb{B}_2 \multimap \mathbb{B}_2 \multimap \mathbb{B}_2$ and extends the *exclusive or* on lifted booleans:

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

Whenever one argument is $\bot$ then it gives back the other argument. This is an application oriented choice [1].

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

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

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. If $(\mathtt{F})\,\langle \mathtt{b}_i, \mathtt{s}_i\rangle \rightarrow^* \langle \mathtt{b}'_i, \mathtt{s}_{i+1}\rangle$, for every $0 \leq i \leq n-1$:

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

Meta-combinator MapThread[$\cdot$]. 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. If $((\mathtt{F})\,\mathtt{a}_i)\,\mathtt{b}_i \to^* \mathtt{c}_i$, for every $0 \le i \le n-1$:

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

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

$$\mathtt{MapThread}[\backslash\mathtt{a}.\backslash\mathtt{b}.\langle\mathtt{a},\mathtt{b}\rangle]\ \{\mathtt{a}_{n-1} \dots \mathtt{a}_0\}\ \{\mathtt{b}_{n-1} \dots \mathtt{b}_0\} \to^* \{\langle\mathtt{a}_{n-1},\mathtt{b}_{n-1}\rangle \dots \langle\mathtt{a}_0,\mathtt{b}_0\rangle\}\ .$$

3 Multiplicative inversion as a term of TFA

Preliminaries. We introduce `wHeadTail[L,S]` with parameters L and S. `wHeadTail[L,S]` embodies the core mechanism of predecessor for Church numerals [6, 5] inside typing systems like TFA.

```
wHeadTail[L,S] = \w.\f.\x.L (w (wHTStep f) (wHTBase x))
wHTStep        = \f.\e.\<ft,et,t>.(S f e ft et t)
wHTBase        = \x.<\e.\l.l, DummyElement, x>
```
(5)

Both L and S must be closed λ-terms that play the role of parameters. S stands for "step" and L for "last". For any A and α, let $X \equiv (A \multimap \alpha \multimap \alpha \otimes A \otimes \alpha)$. For any B, if L: $X \multimap B$ and S: $(A \multimap \alpha \multimap \alpha) \multimap A \multimap \alpha \multimap X$, then `wHeadTail`: $\mathbb{L}(A) \multimap \mathbb{L}(B)$. For example, `wHeadTail` can develop the following computation on a list `\f.\x.f b (f a x)`:

```
wHeadTail (\f.\x.f b (f a x))
--> \f.\x.L ((wHTStep f) b ((wHTStep f) a (wHTBase x)))
--> \f.\x.L ((\<ft,et,t>.S f b ft et t) (S f a (\e.\l.l) DummyElement x))
```
(6)

It is an iteration of `wHTStep` from `wHTBase` on the input. If `DummyElement` is different from any possible element of the list, the rightmost occurrence of S in (6) knows that the iteration is at its step zero and it can operate on a as consequence of this fact. In general, S can identify a sequence of iteration steps of predetermined length, say `l`. Then, S can operate on the first `l` elements of the list in a specific way. The distinguishing invariant of the computation pattern that `wHeadTail` develops is that S can have simultaneous stepwise access to two consecutive elements in the list. For example, S 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.

L is the last operation we can perform on the final triple that the iteration of `wHeadTail` yields. For example, let:

```
L = \<_,_,l>.l
S = \f.\e.\ft.\e.\t.<f,e,ft et t>
```
(7)

We implement a λ-term that pops the last element out of the input list. Developing from (5) we get:

```
..--> \f.\x.L ((\<ft,et,t>.S f b ft et t) (S f a (\e.\l.l) DummyElement x))
--> \f.\x.L ((\<ft,et,t>.S f b ft et t) <f, a, (\e.\l.l) DummyElement x>)
--> \f.\x.L ((\<ft,et,t>.S f b ft et t) <f, a, x>)
--> \f.\x.L (S f b f a x)
--> \f.\x.L <f, b, f a x>)
--> \f.\x.f a x
```
(8)

***Main details about* `wInv`.** We reformulate BEA in Fig. 1 as a λ-term `wInv` of TFA as in Fig. 3. `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 can 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)))
```
(9)

```
wInv = \U.  # Word in input.
       (wProj # Extract the bits of G1 from the resulting 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 function 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
```

Fig. 3: Definition of `wInv`.

`wInv` builds an initial list by applying `MapThread` to the four words in (9) and to further seven words which play the role of a state of the computation. In our example, the whole initial list is:

```
wInvInput =
\f.\x.#           |------- This is a state ----------|
      #           v                                  v
      #  U V G1 G2 M Stop StpNmb RghtShft FwdV FwdG2 FwdM
      f <0,1, 0, 0,1,   0,     B,       B,   0,    0,   0> # msb
     (f <1,0, 0, 0,0,   0,     B,       B,   0,    0,   0>
     (f <0,1, 0, 0,1,   0,     B,       B,   0,    0,   0>
     (f <0,1, 1, 0,1,   0      B,       B,   0,    0,   0> # lsb
                                                         x)))
```
(10)

We call *threaded words* the initial list that `wInv` builds as its first step which is customary to fix some notation and terminology on. Every *element* of the threaded words is a tuple of booleans. Let `U[i]` be the bit of column `U` in the `i`th element of `wInvInput`. Analogous notation on `V`, `G1`, etc. exists. 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 in `wInvInput`. Analogous notation holds for arbitrary sub-sequences we need to project out of `U`, ..., `FwdM`. The most significant bit msb of `wInvInput` is on top; its less significant bit lsb is at the bottom.

The term `D` is a Church numeral. It iterates `\tw.wRevInit (BkwVst (wRev (FwdVst tw)))`, a step function, starting from `wInvInput`. The step function implements steps 2 through 5 of BEA in Fig. 1. The iteration by `D` implements the outermost loop that 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`.

The implementation of FwdVst follows the pattern of wHeadTail. It can distinguish its step zero, and its last step. Yet, for every 0<i<=msb, FwdVst has access to elements <U,V,..,FwdM>[i] and <U,V,..,FwdM>[i-1].

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?” (11)

“Does z divide both u and g_1?” (12)

“Does z divide u but not g_1?” (13)

“Neither of the previous questions has positive answer?” (14)

If (11) 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 (12) or (13) hold. Answering the first question requires to verify U[0]=0 and G1[0]=0. Answering the second one needs to check both U[0]=0 and G1[0]=1. Under our conditions, just after reading wInvInput, FwdVst generates the following first element of the whole threaded words it has to build:

```
<U[0],B,X,B,B,B,0,Y,FwdV[0],FwdG2[0],FwdM[0]>
```
(15)

If (12) holds, then X is G[0] and Y is 1. If (13) holds, then X is Xor G[0] M[0] and Y is 0. It is worth remarking that we record V[0], G2[0] and M[0] in FwdV[0], FwdG2[0] and FwdM[0]. Finally, we replace them with B.

After the generation of the first element (15), for every 0<i<=msb, the iteration that FwdVst implements proceeds as follows. Two elements constitute the input 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]}
```
(16)

From them, FwdVst generates the following element which we see as being the output at step i:

```
<U[i],FwdV[i-1],X,FwdG2[i-1],FvdM[i-1],B,0,Y,V[i],G2[i],M[i]>
```
(17)

Yet, X and Y depend on u and g_1 being divisible by z. The element (17) becomes the i-1th element in the succeeding step.

Finally, under the above condition that (12) or (13) hold, the last step of FwdVst adds two elements to the threaded words it builds. If l is the length of the threaded words that FwdVst starts from, the two elements are:

```
<U[l],FwdV[l-1],X,FwdG2[l-1],FwdM[l-1],B,0,Y,V[l],G2[l],M[l]>
<0,FwdV[l],0,FwdG2[l],FwdM[l],B,0,Y,B,B,B>
```
(18)

where X and Y keeps depending on which between (12) or (13) hold.

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 two elements of a list to propagate some bits from step i-1 to step i — amounts to shifting the bits in positions V, G2 and M one step to their *left*. Leaving the bits of position U and G1 as they were will result in shifting them one step to their right. We shall clarify how and why when we describe 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] preserving the position of every other element.

The description of how FwdVst works concludes by assuming that neither (12) nor (13) 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 an element to the threaded words such that `<Stop,StpNmb>[msb]=<0,1>`. This records that the result of `FwdVst` must be subject the implementation in TFA of Step 4 and 5 of BEA in Fig.1.

To sum up, one of the goal of `FwdVst` is to let the last element of the threaded words it produces contain `<Stop,StpNmbr,RghtShft>` in one of the three configurations of Fig. 4.

Let `l` be the position of the last element of the result of `FwdVst`.

1. `<Stop,StpNmbr,RghtShft>[l]=<1,_,_>`. `FwdVst` verifies that *u* is 1. I.e., `U[0]=1` and `U[i]=0` for every `i>0`.
2. `<Stop,StpNmbr,RghtShft>[l]=<0,1,_>`. `FwdVst` verifies that *z* does not divide *u* and that *u* is different from 1. I.e., there exist two distinct index `i` and `j` such that `U[i]=1` and `U[j]=1`.
3. `<Stop,StpNmbr,RghtShft>[l]=<B,_,0>` or `<Stop,StpNmbr,RghtShft>[l]=<B,_,1>`. `FwdVst` verifies that *z* divides at least *u* at step zero, i.e. that `U[0]=0`. Simultaneously, `FwdVst` also checks if *z* divides at least *u*. In case of positive answer `FwdVst` bitwise adds `G1` and `M` in the course of its whole iteration.

Fig. 4: Relevant combinations of `<Stop,StpNmb,RghtShft>` as given by `FwdVst`.

Then, `wRev` reverses the result of `FwdVst` exchanging lsb and msb. We call `wBkwdVstInput` the threaded words 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 Fig. 4, 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 Fig. 4, we know that *z* does not divide *u* and that *u* is different from 1. In this case `BkwdVst` implements Step 3 and 4 of BEA in Fig. 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]`. 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,_,X>` with `X` different from `B`. It means `FwdVst` verified that one between (12) and (13) holds. Then, `BkwdVst` pops the msb of the threaded words it takes as input. It exploits its behavioral pattern *à la* `wHeadTail`. Erasing the msb is equivalent to erase the lsb of the list that `FwdVst` yields as its result. I.e., we realize the one-step shift to the right of at least `U`, while `V`, `G2` and `M` which were shifted one place to the left, survive the erasure.

Running example. Let us focus on (10) which we apply `FwdVst` to. `FwdVst` can check `U[0]=0` and `G1[0]=1` and determines that (13) 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>
       (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> # original lsb.
                                                                   x))))
```
(19)

The threaded words (19) is the input of `wRev` which yields 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> # original 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>
      (f <0,0,Xor 0 0, 0,1,   B,      0,        0,   1,    0,    1>                (20)
      (f <0,1,      0, 0,1,   B,      B,        0,   B,    B,    B> # msb
                                                                  x))))
```

`BkwdVst` applies to (20). 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 (20). 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>
      (f <0,0,Xor 0 0, 0,1,   B,      0,        0,   1,    0,    1>               (21)
      (f <0,1,      0, 0,1,   B,      B,        0,   B,    B,    B> x)))
```

Finally, `wRevInit` reverses (21), 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>
      (f <1,1,Xor 0 0, 0,0,   B,      B,        B,   0,    0,    0>               (22)
      (f <0,1,Xor 0 1, 0,1,   B,      B,        B,   0,    0,    0> x)))
```

Let us compare (22) and (19). 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 (22) 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 drops the correct element and fully realizes the shift to the right we need.

The code of `FwdVst` ***and of*** `BkwdVst`. We recall that `FwdVst` and `BkwdVst` follow the programming pattern of `wHeadTail`. The step functions they relies on and their "last step functions", implement branching. Choices depend on the values of the bits that belong to the state or on the values of bits in position `U` and `G1`.

Because of space constraints we include `LastStepFwdVst` only. It is the last step of `FwdVst`. The aim is to give an idea of the branching structure of the terms that `wInv` contains. We make use of simple syntactic sugar to introduce `LastStepFwdVst`. Let `N` be of type $\mathbb{B}$. 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. For the sake of readability, we represent `N M1 M0 MB` by means of a `switch`-structure:

```
switch (N)     {
   case 1: ...
   case 0: ...                                     (23)
   case B: ... }
```

`LastStepFwdVst` is in Fig 5. The name `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 (18). If we prefer to think in terms of our example, they generate the two topmost elements in (19). 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.

```
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: # 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: # So we have also RghtShft=B and U[0]=1.
      switch (snt) {
        case 1: # 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: # Here we detect that U=1 and we set Step=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: # 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: # We have U[0]=0 and RghtShft=0 or RghtShft=1.
      switch (rst) {
        case 1: # 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: # 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: # 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
```

Fig. 5: Definition of LastStepFwdVst.

4 An imperative version of wInv and its correctness

In Section 3, we reformulate the imperative algorithm BEA in [4] as a functional program which TFA can give a type to. In this section we go in the opposite direction. From `wInv` we synthesize an imperative program DCEA (DLAL Certified Euclidean Algorithm). `wInv` suggests how to reorganize some steps of BEA so that DCEA becomes a slight variation of BEA.

In Table 1, we present DCEA and BEA side by side. The outermost loop exists in DCEA as well. The new variable *side* in DCEA takes `Fwd` and `Bkwd` as its values. When *side* is `Fwd`, DCEA mimics `FwdVst` as defined in `wInv`; it checks if z divides at least u and behaves accordingly. If true, *side* gets set to `Bkwd`. Then, the block B completes the shift to the right of u and g_1 exactly as `BkwdVst` would do in `wInv`. On the contrary, if *side* is `Fwd` and z does not divide u, then *side* eventually becomes `Bkwd`. Then DCEA compares the degrees of u and v exactly like `BkwdVst` in `wInv`. Finally, DCEA checks $u = 1$ just before switching from *side* = `Fwd` to *side* = `Bkwd`. We recall that this is what `FwdVst` does when z does not divide u and the computation is going to be controlled by `BkwdVst`.

```
INPUT: $a \in \mathbb{F}_{2^m}, a \neq 0$
OUTPUT: $a^{-1} \mod f$
 1: $u \leftarrow a, v \leftarrow f, g_1 \leftarrow 1, g_2 \leftarrow 0$, side ← Fwd
 2: if side = Fwd then
 3:     if $z$ divides $u$ and $z$ do not divides $g_1$ then      (A)
 4:         $g_1 \leftarrow g_1 + f$
 5:     end if
 6:     side ← Bkwd
 7: else
 8:     if $z$ divides $u$ then                                   (B)
 9:         $u \leftarrow u/z$ ; $g_1 \leftarrow g_1/z$
10:     else
11:         if deg($u$) < deg($v$) then                           (C)
12:             $u \leftrightarrow v$; $g_1 \leftrightarrow g_2$
13:         end if
14:         $u \leftarrow u + v$; $g_1 \leftarrow g_1 + g_2$
15:     end if
16:     side ← Fwd
17: end if
18: if $u = 1$ then                                               (D)
19:     return $g_1$
20: end if
21: go to 2
```

(a) DCEA.

```
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
 3:     $u \leftarrow u/z$                                        (E)
 4:     if $z$ divides $g_1$ then
 5:         $g_1 \leftarrow g_1/z$
 6:     else
 7:         $g_1 \leftarrow (g_1 + f)/z$
 8:     end if
 9: end while
10: if $u = 1$ then                                               (F)
11:     return $g_1$
12: end if
13: if deg($u$) < deg($v$) then                                   (G)
14:     $u \leftrightarrow v$; $g_1 \leftrightarrow g_2$
15: end if
16: $u \leftarrow u + v$; $g_1 \leftarrow g_1 + g_2$
17: go to 2
```

(b) BEA.

Table 1: Algorithms that compute the multiplicative inversion for a given binary finite field.

Section 5 gives an account of experiments on the performance of DCEA, BEA and OpenSSL. We anticipate that, despite DCEA might look a trivial syntactical modification of BEA, it can behave better than BEA, and outperforms OpenSSL.

We now give evidence about the equivalence of DCEA and BEA. Proving the equivalence of DCEA and BEA amounts to show that they behave the same, starting from identical pre-conditions. Two significant cases exists.

- Let $z \mid u$, i.e. let z divide u. BEA iterates E in the loop that starts at line 2 until $z \nmid u$. Under the same pre-condition DCEA iterates the sequence A, D, B, D until $z \nmid u$, i.e. until z does not divide u. In both cases it should be evident that the post-conditions coincide.
- Let $u \neq 1$ and $z \nmid u$. BEA skips both E and F to execute G. DCEA skips A, D, and B to finally interpret C. The set of instructions in G and C coincide.

As soon as DCEA and BEA check $u = 1$ they are obviously equivalent because neither of the two alter any among u, v, g_1 and g_2.

5 Experimental results

Calculating the multiplicative inverse of a finite field is typically the most time-consuming operation. A high-speed implementation is frequently much more than desirable, since its running-time can have a significant impact on several cryptographic algorithms and protocols. We present the results of a number of tests aiming to compare the execution time of DCEA, BEA and the implementation of the inversion that the OpenSSL library [7] supplies.

We performed our tests on four different hardware architectures whose specifications are given in Table 2. We remark that the "Core2", "Pentium4" and "UltraSPARC" platforms are PC microprocessors while the "ATmega328" is a microcontroller. Note also that for the tests we used an Arduino Duemilanove[5] prototype board which integrates an ATmega328.

CPU	Clock	Word size	Compiler	Optimization flags
Intel Core2 Duo P8600	2.40GHz	64 bit	gcc 4.7.3	-O4
Sun UltraSPARC IIe	500 MHz	64 bit	gcc 4.2.1	-O4
Intel Pentium4	3.00GHz	32 bit	gcc 4.4.5	-O4
ATmega328	16MHz	16 bit	avr-gcc 4.3.2	-Os

Table 2: Specifications of the test platforms.

We compare the performances of the aforementioned algorithms on the following binary fields:

- the field $\mathbb{F}_{2^{61}}[z]/(z^{61} + z^5 + z^2 + z + 1)$;
- the field $\mathbb{F}_{2^{163}}[z]/(z^{163} + z^7 + z^6 + z^3 + 1)$, recommended in the standard FIPS 186-3 [8];
- the field $\mathbb{F}_{2^{233}}[z]/(z^{233} + z^{74} + 1)$, also recommended in the standard FIPS 186-3 [8].

We recall that the field degrees 61, 163 and 233 are the dimension of the field, that is the length n of the threaded words we uniformly manipulate with `wInv` in Section 3.

5.1 Comparing the running-time of DCEA and BEA

In the following paragraphs, we briefly explain our findings regarding the performances of the DCEA with respect to the BEA.

In this context, the term "*generic* algorithm" refers to an implementation of a multiplicative inversion which can operate on any field $\mathbb{F}_{2^n}[z]/(m(z))$, with a generic degree n and modulus $m(z)$. On the other hand, the term "*specialized* algorithm" identifies a specific implementation that can only work on a given field $\mathbb{F}_{2^n}[z]/(m(z))$, with a fixed degree n and modulus $m(z)$. Note that fixing n and $m(z)$ *a-priori* usually allows to better optimize the code, so a specialized algorithm is typically faster than its generic counterpart.

For instance, the Listing 1.1 shows the C code used to test the DCEA algorithm over $\mathbb{F}_{2^{61}}$ on the Core2 platform. The snippet does not precisely reflect the pseudo-code listed in Table 1a since several ad-hoc optimizations were applied, such as:

- elimination of the selection on the *side* variable;
- elimination of the selection in the block A of Table 1a via a Shannon expansion (lines 31 – 32);
- the comparison between the polynomial degrees is computed using a simple and efficient integer comparison (line 37).

```c
#define SHIFT(i) \
  tmp1 = u[i] >> 1; \
  tmp2 = (u[i + 1] & 1) << 63; \
  tmp3 = g1[i] >> 1; \
  tmp4 = (g1[i + 1] & 1) << 63; \
  u[i] = tmp1 ^ tmp2; \
  g1[i] = tmp3 ^ tmp4

#define SUM(i) \
  u[i] ^= v[i]; \
  g1[i] ^= g2[i]

#define SWAP_SUM(i) \
  u[i] ^= v[i]; \
  g1[i] ^= g2[i]; \
  v[i] ^= u[i]; \
  g2[i] ^= g1[i]

void dcea61(uint64_t r[], uint64_t a[])
{
  uint64_t u[2] = { a[0] };
  uint64_t v[2] = { (1l << 61) ^ (1 << 5) ^ (1 << 2) ^ (1 << 1) ^ 1 };
  uint64_t g1[2] = { 1 };
  uint64_t g2[2] = { 0 };
  uint64_t x0;
  uint64_t tmp1, tmp2, tmp3, tmp4;

  do
    if ((u[0] & 1) == 0)
    {
      x0 = ~((g1[0] & 1l) - 1l);
      g1[0] ^= ((1l << 61) ^ (1 << 5) ^ (1 << 2) ^ (1 << 1) ^ 1) & x0;
      SHIFT(0);
      u[1] >>= 1;
      g1[1] >>= 1;
    }
    else if (u[1] < v[1] || u[0] < v[0])
    {
      SWAP_SUM(0);
      SWAP_SUM(1);
    }
    else
    {
      SUM(0);
      SUM(1);
    }
  while (u[0] != 1 || u[1] != 0);

  r[0] = g1[0];
}
```

Listing 1.1: The DCEA algorithm C code specialized for the field $\mathbb{F}_{2^{61}}$ and the Core2 architectures.

Algorithm	Core2		Pentium4		UltraSPARC		ATmega328	
	BEA	DCEA	BEA	DCEA	BEA	DCEA	BEA	DCEA
$\mathbb{F}_{2^{61}}$: generic algorithm	2.63	2.64	6.10	6.15	10.52	10.89	7596.13	7685.26
$\mathbb{F}_{2^{61}}$: specialized algorithm	0.70	0.78	1.45	1.47	2.99	2.58	1108.59	1159.87
$\mathbb{F}_{2^{163}}$: generic algorithm	25.14	25.43	41.77	41.48	102.11	107.29	59755.80	60160.20
$\mathbb{F}_{2^{163}}$: specialized algorithm	5.92	5.94	13.77	13.78	38.54	30.01	15250.45	15504.68
$\mathbb{F}_{2^{233}}$: generic algorithm	46.34	46.83	91.22	91.23	199.23	205.62	122471.00	123576.85
$\mathbb{F}_{2^{233}}$: specialized algorithm	10.83	10.88	30.30	30.75	86.34	82.39	36039.93	36695.22

Table 3: Average time in μs that BEA and DCEA take to compute an inversion.

In Table 3, we compare the average running-time in microseconds (μs) that DCEA and BEA need to calculate a single inversion on a given field. If we exclude the UltraSPARC architecture, DCEA can be effectively seen as a variant of BEA. Both programs perform essentially equally with a slight prevalence of BEA. However, on UltraSPARC, the specialized version of DCEA is significantly faster than the BEA. It achieves a 1.28 speedup factor over $\mathbb{F}_{2^{163}}$.

5.2 Comparing the running-time of DCEA and OpenSSL

OpenSSL [7] is one of the most used and widespread cryptographic toolkits for securing a network communication, especially in the open-source world. Table 4 shows the average running-times of OpenSSL 1.0.1e and DCEA, both generic and specialized, to calculate a single inversion on the given field on an Intel Core2 CPU. The time is in microseconds (μs.)

Field	OpenSSL	DCEA	
		Generic	Specialized
$\mathbb{F}_{2^{61}}$	3.97	2.64	0.78
$\mathbb{F}_{2^{163}}$	13.05	25.43	5.90
$\mathbb{F}_{2^{233}}$	20.58	46.83	10.88

Table 4: Average time in μs that OpenSSL algorithm and DCEA take to compute an inversion.

In Fig. 6, we graphically depicts the data in Table 4 to better emphasize the differences among the performances of OpenSSL 1.0.1e and DCEA.

We remark that OpenSSL implements MAIA (Modified Almost Inverse Algorithm) which is a generic version of BEA. The specialized versions of DCEA always outperform the OpenSSL, especially on the small fields. The generic version of DCEA seems faster only on $\mathbb{F}_{2^{61}}$. In this case it is twice as fast. This is an interesting result since several algebraic structures, such as hyperelliptic curves, can guarantee a high level of security using fields with a low order.

6 Conclusions and future work

We introduce the functional program `wInv`. It has type in TFA and it implements the multiplicative inverse in a given binary field of arbitrary, but fixed, degree. We complete a project started in [1]. The goal of [1] is 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

[5] http://arduino.cc/en/Main/ArduinoBoardDuemilanove

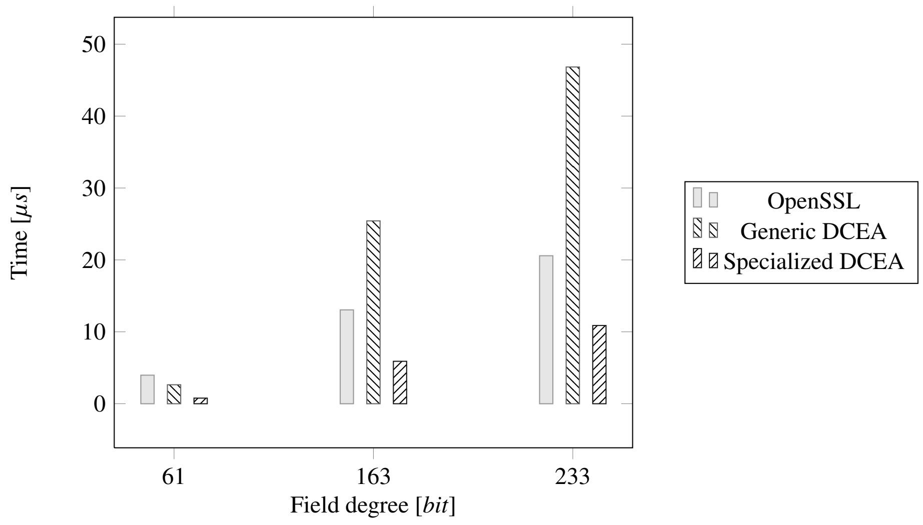

Fig. 6: Comparing the implementation of inversion in OpenSSL and DCEA.

that the expressivity of languages like the one we used is too weak to program anything interesting. By teh way, we remark that the existence of `wInv` gives an alternative proof that inversion has a polynomial cost.

We further strengthen the evidence that programming with a language with its roots inside ICC, like the λ-terms typed by TFA are, can be rewarding. From `wInv` we synthesize the iterative algorithm DCEA which implements the inversion. Tests on typical hardware architectures show that DCEA is essentially as fast as BEA (DCEA is about 1% slower than BEA). Instead, tests on an UltraSPARC architecture show that DCEA performs better than BEA. One of our future goals is to certify that this is not a singularity and to obtain clues at why DCEA seems to perform better than BEA on UltraSPARC architectures.

A second goal is to assess to which extent we can exploit the functional programming patterns we use in the definition of DCEA to design a (specific domain) imperative language. The hope is to design one characterized by two seemingly contrasting features. It should be expressive enough in terms of programs we can write but its constructs would be limited enough to imply the application of good compiling techniques and optimizations able to fully exploit architecture like, for example, UltraSPARC.

References

1. Cesena, E., Pedicini, M., Roversi, L.: Typing a Core Binary-Field Arithmetic in a Light Logic. In Peña, R., van Eekelen, M., Shkaravska, O., eds.: Foundational and Practical Aspects of Resource Analysis (subtitle: 2nd International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA 2011). Volume 7177 of LNCS., Springer (2012) 19 – 35
2. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda calculus. I&C **207**(1) (2009) 41–62
3. Dal Lago, U.: Context semantics, linear logic, and computational complexity. ACM Trans. Comput. Log. **10**(4) (2009) Art. 25, 32
4. Fong, K., Hankerson, D., Lopez, J., Menezes, A.: Field inversion and point halving revisited. IEEE Trans. Comput. **53**(8) (2004) 1047–1059
5. Asperti, A., Roversi, L.: Intuitionistic light affine logic. ACM ToCL **3**(1) (2002) 1–39
6. Roversi, L.: A P-Time Completeness Proof for Light Logics. In: Ninth Annual Conference of the EACSL (CSL'99). Volume 1683 of Lecture Notes in Computer Science., Madrid (Spain), Springer-Verlag (September 1999) 469 – 483
7. The OpenSSL team: OpenSSL: The Open Source toolkit for SSL/TLS (2013)
8. National Institute of Standards and Technology: FIPS PUB 186-3 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS) (June 2009)