Science of Computer Programming 111 (2015) 365–394

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Light combinators for finite fields arithmetic

D. Canavese [a], E. Cesena [b], R. Ouchary [a], M. Pedicini [c], L. Roversi [d]

[a] Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy
[b] Theneeds Inc., San Francisco, CA, USA
[c] Università degli Studi Roma Tre, Dipartimento di Matematica e Fisica, Roma, Italy
[d] Università degli Studi di Torino, Dipartimento di Informatica, Torino, Italy

ARTICLE INFO

Article history:
Received 26 August 2013
Received in revised form 26 February 2015
Accepted 5 April 2015
Available online 28 April 2015

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

ABSTRACT

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

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

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

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

1. Introduction

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

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

E-mail 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).

http://dx.doi.org/10.1016/j.scico.2015.04.001
0167-6423/

INPUT: $a \in \mathbb{F}_{2^m}$, $a \neq 0$.
OUTPUT: $a^{-1} \bmod f$.

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

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

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

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

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

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

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

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

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

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

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

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

We told that the encoding of BEA as `wInv` relies on the light predecessor patterns which is specific of type assignments that come from Light Affine Logic. The relevance of a new programming pattern, or abstraction, may not be immediately evident. For example, the `MapReduce` paradigm have been exploited as in [7] far after its introduction which, morally, occurs in [8]. Of course, we are not supporting the idea that light predecessor pattern is, or will be, as relevant as `MapReduce`! However, the work [9], which we see as a natural companion of this one, helps pursuing the idea that something interesting in connection with light predecessor pattern exists. In [9] we show that the design of `wInv` in fact suggests to rewrite BEA in Fig. 1 in a new imperative algorithm DCEA. We do not recall it here. Suffice it to say that DCEA rearranges the statements in BEA. On standard architectures, under the same optimisations, the speed of `C` implementations of BEA and DCEA are comparable with a slight prevalence of BEA. Instead, on `ARM` architectures, under the same optimisations, DCEA can be up to 20% faster than BEA. Fully investigation of why this happens is on-going work.