Genetic Algorithms in Coq: Generalization and Formalization of the Crossover Operator

FELICIDAD AGUADO, JOSÉ LUIS DONCEL, JOSÉ MARÍA MOLINELLI
GILBERTO PÉREZ, CONCEPCIÓN VIDAL
Department of Computer Science
University of A Coruña, Spain

In this article we present the implementation and formal verification, using the **Coq** system [FHB+98], of a generalized version of the crossover operator applied to genetic algorithms (GA) [Hol92]. The first part of this work defines the multiple crossover $\otimes_\ell(p, q)$ of two lists p, q in any finite number of points. This definition generalizes the one given in [UE99] for a maximum number of six points. In the second part, we show that this crossover operator does not depend on the order of the list of points. Then, a more efficient definition of crossover $\oslash_\ell(p, q)$ is provided in the third part. Finally, we formally establish the exact relation between these two definitions, using the notion of differences list: $\otimes_\ell(p, q) = \oslash_{(dif\ \ell)}(p, q)$.

1. INTRODUCTION

Genetic algorithm theory and theorem proving are two different domains of theoretical computer science. Each has its own scientific community and until now little interaction seemed to exist between the two.

The so-called evolutive computation is inspired by natural evolution. Among the evolutive techniques we find genetic algorithms (GA). Basically, they have been used as optimization techniques to find the best solution for a specific problem with respect to a particular criterion given by a fitness function f. This function evaluates each solution and determines how good it is with respect to that criterion. The main idea is that individuals with higher values of f have a higher probability of survival in successive generations.

Genetic algorithm theory was originally developed by a group of researchers from the University of Michigan led by John Holland ([Hol92]). Although there are different types of GAs, they all share the following three processes: selection, reproduction and evaluation. The algorithm repeats these processes cyclically until a stop condition is reached.

The elements of the first population are selected randomly. Usually each element is represented by a string $p = p_1 \dots p_n$ called a chromosome, where each p_i makes reference to a particular characteristic of the individual and belongs to a finite set or alphabet D. The most typical is the binary encoding. In this case, the alphabet D has only two elements, denoted by 0 and 1.

The selection process can be performed in several ways but it is always inspired in the idea that better adapted individuals have more chances of being selected for

This research is partially supported by XUNTA DE GALICIA, research project REGACA, 2006/38.

reproduction. The more common mechanisms used in reproduction are crossover and mutation. In the *single-point crossover*, a position i (with $1 \leq i \leq n$) is randomly selected and then two parents

$$\begin{aligned} p &= p_1 \dots p_n \\ q &= q_1 \dots q_n, \end{aligned}$$

are replaced by their offspring:

$$\begin{aligned} p' &= p_1 \dots p_i q_{i+1} \dots q_n \\ q' &= q_1 \dots q_i p_{i+1} \dots p_n. \end{aligned}$$

Similar to what happens in nature, some random changes may occur in the genetic content of individuals during reproduction; this is called mutation. Binary encoding consists in eventually replacing some value 0 by 1 and vice versa. Although this mutation mechanism should not occur frequently, it contributes to the diversification of the initial population by introducing different or novel characteristics.

To evaluate the individuals of a population, a fitness function is used. Presumably, the population will increase its average fitness over time, in such a way that, by iterating this process, suitable solutions will be found to the problem.

The single-point crossover was generalized by DeJong ([Jon75]) who talked about multipoint crossover when more than one point was involved in the cutting. For example, if we want to cross two parents p and q as above using the points $\{i, j, k\}$, the new offspring will be the following:

$$\begin{aligned} p' &= p_1 \dots p_i q_{i+1} \dots q_j p_{j+1} \dots p_k q_{k+1} \dots q_n \\ q' &= q_1 \dots q_i p_{i+1} \dots p_j q_{j+1} \dots q_k p_{k+1} \dots p_n \end{aligned}$$

There is no consensus in the GA community about the convenience of choosing single-point or multipoint crossovers. Empirical results ([Jon75, ECS89]) do not give conclusive results. Nevertheless, it is accepted that two-point crossover is better than single-point.

For a formalization of this multipoint crossover operator, we use the **Coq** Proof Assistant. This is an implementation of the Calculus of Inductive Constructions (CCI), an intuitionistic higher-order logic with dependent types and inductive types as primitive objects [PM93, CH88]. Coq is both a programming language and a proof assistant. On the one hand, it allows specifications of programs, and on the other, permits formalization and verification of mathematical proofs. The user can introduce definitions and construct proofs in a natural deductive style, which are mechanically checked/tested/analyzed by the system. Proofs in the **Coq** system are constructive, unlike in other systems such as Mizar, HOL or PVS.

To differentiate computational objects from logical information, two sorts (the type of a type) of Coq are basically distinguished: the sort *Prop* for types that

contain logical terms (the propositions), and the sort Set[1], mainly used to describe data types and program specifications.

One of the most attractive aspects of the Coq system is the program *extraction*. As a consequence of the Curry-Howard isomorphism, a constructive algorithm of an object may be "extracted" from the proof of its existence.

This process removes all logical parts inside terms, preserving only those computational information parts [PM89].

In the genetic algorithms context, the formal verification can be used for certifying the correctness of the implementation of some evolutive algorithms and also for proving properties satisfied by the operators used in genetic programming. Moreover, in a previous work [ADM+07], we have formally implemented two crossover operators specifically designed for problems where the chromosomes are permutations of a finite list of elements (like in the classical TSP).

One of the first attempts at the formalization of the foundations of genetic algorithms using a proof system can be found in [UE99].

Taking this work as an initial point and using the Coq proof assistant, we try to establish the basis for the formalization of the basic operators in GA (crossover and mutation) and, at the same time, investigate the formal verification of the properties satisfied by such operators.

In the first section, and generalizing the crossover definition that appears in [UE99], we implement in Coq a first definition of multiple crossover $\otimes_\ell$, where ℓ is any list of natural numbers.

We prove, in the second section of this work, that for the crossover of two strings p and q in some points of a list ℓ, we can suppose that ℓ is in strictly increasing order, because the crossover determined by two different lists ℓ_1 and ℓ_2 coincides only if such lists are the same except perhaps in the order of their elements. This multiple crossover $\otimes_\ell$ requires a double recursion. That is why we propose in the third section of this work an alternative and more efficient crossover definition $\oslash_\ell(p, q)$. We also prove that, if we define, from an increasing ordered list ℓ, the so-called list of differences, $dif\ \ell$, then the result of both crossover operators coincides; that is: $\oslash_{(dif\ \ell)}(p, q) = \otimes_\ell(p, q)$.

2. THE MULTIPLE CROSSOVER OPERATOR

Let D be a non-empty and non-unitary finite set. Let $p = p_1 \dots p_m$ and $q = q_1 \dots q_r$ be two finite sequences (chromosomes) of elements of D. We will represent them in Coq as lists of elements of D (whose set is denoted by $(list\ D)$)[2]. Uchibori and Endou in [UE99] define the crossover

$$crossover(p, q, n) = p \uparrow n\ +\!+\ q \downarrow n,$$

for any pair of lists p and q and any natural number n. Here $+\!+$ represent the concatenation of lists and $\uparrow$, $\downarrow$ are the operators defined by:

[1]Both sorts $Prop$ and Set being of sort $Type$.
[2]The implementation of polymorphic lists in Coq can be found in [INR].

Definition 2.1. Let $p = (p_1, \ldots, p_n)$ be a finite sequence of elements of D (where n is the length of p) and $m \in \mathbb{N}$. We define:

$$p \uparrow m = \begin{cases} (p_1, \ldots, p_m) & \text{if } m < n, \\ p & \text{if } m \geq n \end{cases}$$

$$p \downarrow m = \begin{cases} (p_{m+1}, \ldots, p_n) & \text{if } m < n, \\ [\,] & \text{if } m \geq n \end{cases}$$

where $[\,]$ denotes the empty list.

Uchibori and Endou also define the crossover in two points, n_1 and n_2, as

$$crossover(p, q, n_1, n_2) = crossover(crossover(p, q, n_1), crossover(q, p, n_1), n_2)$$

and repeat this process until the crossover in six points. This construction is generalized by using the recursive definition of the crossover function presented below (see Definition 2.2).

Because each finite list of natural numbers defines a crossover operator (crossover by this sequence of points), we will call this list a **crossover pattern**.

Definition 2.2. Let ℓ (pattern) be a list of natural numbers, the **crossover function** $\otimes_\ell : (list\ D) \times (list\ D) \longrightarrow (list\ D)$ is defined as:

$$\otimes_\ell\ (p, q) = \begin{cases} p & \text{if } \ell = [\,], \\ (\otimes_t\ (p, q) \uparrow n)\ +\!+\ (\otimes_t\ (q, p) \downarrow n) & \text{if } \ell = n :: t \end{cases}$$

where p and q are elements of type $(list\ D)$ and $n :: t$ is the list with head n and tail t.

To specify the previous definitions in Coq and to formally prove their properties, first, we should declare in Coq the type of data D. As it is a set, we will associate it with the type *Set* because, from the point of view of program extraction, *Set* is the type of the propositions with constructive content.

```
Variable D: Set.
```

When we define the operators $\uparrow$, $\downarrow$ and $\otimes$ in Coq[3], we call them, respectively, $first$, *cutting*[4] and *crossover* (for each list ℓ, $\otimes_\ell$ is the crossover ℓ).

```
Fixpoint first (m: nat) (p: (list D)){struct p}: (list D) :=
  match m, p with
  |  0, _ => nil
  |  _, nil => nil
  |  S x, h :: p1 => h :: first x p1
  end.

Fixpoint cutting (m: nat) (p: (list D)){struct p}: (list D) :=
  match m, p with
  | 0, x => x
```

[3]All definitions, lemmas and theorems written in a `typewriter font` have been formalized in the Coq system.

[4]Note that the functions $first$ and *cutting* are the functions *take* and *drop* in Haskell.

```
  |  _, nil => nil
  | S x, _ :: p1 => cutting x  p1
  end.

Fixpoint crossover (l: list nat) (p q: (list D)){struct l}: (list D):=
  match l with
  |  nil => p
  | n :: t => (first n (crossover t p q)) ++ (cutting n (crossover t q p))
  end.
```

There are some properties of $\uparrow$ and $\downarrow$ operators in [DN91, Kot93, UE99] which are verified using the MIZAR system. Here we have used Coq to (formally) prove some of these properties and also to obtain some new ones. We begin with technical results, some of them appearing in [DN91]. Their proofs are obtained by simplifying (beta reduction) and using the implementation of $\uparrow$ and $\downarrow$ and the recurrence schemes in Coq. These results will serve to automate later proofs. We include all of them in the following proposition.

Proposition 2.3. If $p, q \in (list\ D)$ and $n, m \in \mathbb{N}$, it holds that:

—$p \uparrow 0 \ = \ [\,]$
—$p \downarrow 0 \ = p$
—$[\,] \uparrow n \ = \ [\,]$
—$[\,] \downarrow n \ = \ [\,]$
—If $p \downarrow 0 \ = q$, then $p = q$
—$length\ p \ \leq \ n \ \Rightarrow \ (p \uparrow n) \ = \ p$
—$length\ p \ \leq \ n \ \Rightarrow \ (p \downarrow n) \ = \ [\,]$
—$(p \ ++ q) \uparrow n = (p \uparrow n) \ ++ (q \uparrow (n - length\ p))$[5]
—$(p \ ++ q) \uparrow (length\ p) \ = \ p$
—$(p \ ++ q) \uparrow ((length\ p) + n) \ = \ p \ ++ (q \uparrow n)$
—$n \ \leq \ length\ p \ \Rightarrow \ (p \ ++ q) \uparrow n \ = \ p \ \uparrow n$
—$length\ p \ = \ length\ q \ \Rightarrow \ length\ (p \uparrow n) \ = \ length\ (q \uparrow n)$
—$length\ (p \uparrow n) \ = \ min\ (n, (length\ p))$
—$((p \uparrow m) \uparrow n) \ = \ (p \uparrow min(n, m)) \ = \ ((p \uparrow n) \uparrow m)$
—$(p \ ++ q) \downarrow n = (p \downarrow n) \ ++ (q \downarrow (n - length\ p))$
—$n \leq length\ p \Rightarrow (p \ ++ q) \downarrow n \ = \ (p \downarrow n) \ ++ q$
—$length\ p \leq n \Rightarrow (p \ ++ q) \downarrow n = (q \downarrow (n - length\ p))$
—$length\ p \ = \ length\ q \ \Rightarrow \ length\ (p \downarrow n) \ = \ length\ (q \downarrow n)$
—$length\ (p \downarrow n) \ = \ (length\ p) - n$
—$(p \downarrow m) \downarrow n \ = p \downarrow (n + m) \ = \ (p \downarrow n) \downarrow m$
—$(p \uparrow n) \ ++ (p \downarrow n) \ = \ p.$
—$(p \downarrow n) \uparrow m \ = (p \uparrow (m + n)) \ \downarrow n$
—$(p \uparrow n) \downarrow m \ = (p \downarrow m) \ \uparrow (n - m)$
—$(p \uparrow n) \downarrow n \ = \ [\,]$

[5]Note that subtraction $n - m$ in Coq is 0 when $n \leq m$.

The classical crossover operator used in genetic algorithms works with strings of the same length. That is why, throughout this paper, we will assume that our lists of elements of D have this property. First, we prove that, when we apply the crossover operator to these lists, it preserves the length of the strings. This result will prove extremely useful in some results afterwards.

THEOREM 2.4. *For any $\ell \in (list\ \mathbb{N})$ and any $p, q \in (list\ D)$ with length p = length q, it holds that*

$$length\ (\otimes_\ell\ (p, q)) = length\ p.$$

```
Lemma length_crossover: forall (l: (list nat)) (p q: (list D)),
   length p = length q -> length (crossover l p q) = length p.
```

The following theorem guarantees that if two lists ℓ_1 and ℓ_2 define the same crossover operator, the result remains true when we add the same list ℓ at the beginning of both lists.

THEOREM 2.5. *Let p, q be elements of $(list\ D)$ with length p = length q and let ℓ, ℓ_1, ℓ_2 be elements of $(list\ \mathbb{N})$. If $\otimes_{\ell_1}\ (p, q)\ =\ \otimes_{\ell_2}\ (p, q)$, then*

$$\otimes_{(\ell +\!+ \ell_1)}\ (p, q)\ =\ \otimes_{(\ell +\!+ \ell_2)}\ (p, q).$$

```
Theorem add_list: forall (l l1 l2: (list nat)), (forall (p q: (list D)),
   length p = length q -> crossover l1 p q = crossover l2 p q) ->
      (forall (p q: (list D)), length p = length q ->
         crossover (l ++ l1) p q = crossover (l ++ l2) p q).
```

We include now two theorems which have turned out to be fundamental in the development of our work. In addition, they generalize all the results concerning the crossover operator appearing in [UE99].

THEOREM 2.6. *Taking $\ell \in (list\ \mathbb{N})$ and also $p, q \in (list\ D)$, it can be proven that:*

—$\otimes_{(0::\ell)}\ (p, q)\ =\ \otimes_\ell\ (q, p)$
—$\otimes_{(0::[\,])}\ (p, q)\ =\ q$
—$\otimes_{(n::\ell)}\ (p, q) = \otimes_\ell\ (p, q)$, *if length p = length $q \leq n$.*

THEOREM 2.7. *Let $\ell \in (list\ \mathbb{N})$, $n, m \in \mathbb{N}$ and $p, q \in (list\ D)$ with length p = length q. It holds that:*

$$\otimes_{(n\ ::\ (m\ ::\ \ell))}\ (p, q)\ =\ \otimes_{(m\ ::\ (n\ ::\ \ell))}\ (p, q)$$

and

$$\otimes_{(n\ ::\ (n\ ::\ \ell))}\ (p, q)\ =\ \otimes_\ell\ (p, q)\ .$$

```
Theorem swap_crossover2: forall (l: (list nat)) (p q: (list D)) (n m: nat),
   length p = length q ->
      crossover (n :: (m ::l)) p q = crossover (m :: (n ::l)) p q.

Theorem elim_rep: forall (l: (list nat)) (p q: (list D)) (n: nat),
   length p = length q -> crossover (n :: (n ::l)) p q = crossover l p q.
```

3. CANONICAL LISTS AND PATTERN CROSSOVER

The multipoint crossover operator used in the GA theory discussed in the Introduction depends on the sites where we are going to cut the strings, but the order of the points is not relevant. Our crossover operator $\otimes_\ell\ (p, q)$ also verifies, as a consequence of 2.5 and 2.7 that, if ℓ_1 and ℓ_2 are two lists with the same elements, but in different order, then

$$\otimes_{\ell_1}(p, q) = \otimes_{\ell_2}(p, q),$$

for any pair of strings p and q of the same length.

Then, we can always suppose that ℓ is a list in strictly increasing order, and if this is not the case, we should reorder its elements. Moreover, 2.7 reveals that the repeated elements in a list can be eliminated without any change in the result of the crossover operator.

So, for any list ℓ, we know that $\otimes_\ell(p, q) = \otimes_{\ell'}(p, q)$, ℓ' being the list obtained with the same elements of ℓ but in strictly increasing order and after deleting the repeated elements two by two. Given any list ℓ of natural numbers, one of the objectives of this section is to formally implement in Coq a function that returns the list ℓ' which now has the required properties and called the *canonical list* of ℓ.

First, we implement in Coq a function that, given ℓ a list of natural numbers in strictly increasing order and n a natural number, returns a new list which is also in strictly increasing order. The following function removes n from ℓ when n is an element of ℓ and inserts n in the right position of ℓ in other case.

```
Fixpoint insert_sd (n:nat) (l: (list nat)){struct l}: (list nat):=
  match l with
  | nil => (n :: nil)
  | n1 :: l1 => match lt_eq_lt_dec n n1 with
     |inleft (left _) => n :: l         (* n  < n1 *)
     |inleft (right_) => l1             (* n  = n1 *)
     |inright _ => n1 :: insert_sd n l1 (* n1 < n *)
                end
  end.
```

Now, with this function *insert_sd*, we obtain another function *canon* that returns the canonical list of any list ℓ.

```
Fixpoint canon (l: (list nat)){struct l}: (list nat) :=
  match l with
  | nil => nil
  | n1 :: l1 => insert_sd n1 (canon l1)
  end.
```

To obtain a formal proof of certain properties of the lists in strictly increasing order we need a predicate implementing this concept. The intuitive idea we usually have about this predicate is "a list where any element is strictly less than the next one".

```
Inductive st_crec: (list nat) -> Prop :=
  | st_crec_nil:  st_crec nil
  | st_crec_unit: forall (n: nat), (st_crec (n :: nil))
  | st_crec_cons: forall (n1 n2: nat) (l: (list nat)),
       (n1 < n2) -> st_crec (n2 :: l) -> st_crec (n1 :: n2 :: l).
```

As expected, if ℓ is a canonical list, then ℓ is a strictly increasing ordered list and the function *canon* does not modify the lists with this property.

THEOREM 3.1. *Given any* $\ell \in (list\ \mathbb{N})$ *and* $n, m \in \mathbb{N}$*, it follows that:*

—$(st\_crec\ (n :: \ell)) \wedge (m\ <\ n)\ \Rightarrow\ (st\_crec\ (m :: \ell))$
—$st\_crec\ \ell\ \Rightarrow\ st\_crec\ (insert\_sd\ n\ \ell)$
—*The canonical lists are strictly increasing ordered lists.*
—$st\_crec\ \ell\ \Rightarrow\ (canon\ \ell)\ =\ \ell$

As mentioned previously, the crossover operator defined by ℓ is exactly the same as the one defined by its canonical list.

Now, we have all the ingredients to prove that:

THEOREM 3.2. *For* $\ell \in (list\ \mathbb{N})$*,* $p, q \in (list\ D)$ *of the same length and* $n \in \mathbb{N}$*, it follows that:*

—$\otimes_{(insert\_sd\ n\ \ell)}\ (p, q)\ =\ \otimes_{n\ ::\ \ell}\ (p, q)$.
—$\otimes_{(canon\ \ell)}\ (p, q)\ =\ \otimes_{\ell}\ (p, q)$.

```
Theorem crossover_ins: forall (l: (list nat)) (p q: (list D)) (n: nat),
   length p = length q -> crossover (insert_sd n l) p q = crossover (n :: l) p q.

Theorem crossover_can_eq: forall (l: (list nat)) (p q: (list D)),
   length p = length q -> crossover (canon l) p q = crossover l p q.
```

Immediately, we obtain the corollary:

Corollary 3.3. Take $\ell_1, \ell_2 \in (list\ \mathbb{N})$ such that $(canon\ \ell_1)\ =\ (canon\ \ell_2)$. It can be proven that

$$\otimes_{\ell_1}\ (p, q)\ =\ \otimes_{\ell_2}\ (p, q),$$

for any pair of strings $p, q \in (list\ D)$ of the same length.

```
Theorem crossover_canon: forall (l1 l2: (list nat)),
   canon l1 = canon l2 -> (forall (p q: (list D)),
      length p = length q -> crossover l1 p q = crossover l2 p q).
```

To prove the reciprocate of Corollary 3.3, we need to assume that D is a set with at least two different elements (this always occurs when we work with genetic algorithms) and also we need to compare any pair of elements of D. As Coq works with intuitionistic logic, we must assume the existence of a decidability algorithm for D. This is going to be a parameter of our specification.

```
Variable a b: D.

Axiom two_el: ~ (a = b).

Parameter decD: forall d d': D, {d = d'} + {~ (d = d')}.
```

Making use of the decidability algorithm assumed before, we obtain the proof of the decidability with respect to the equality of the lists whose elements are in D.

A result required to achieve our goal is:

THEOREM 3.4. *For any non-empty and strictly increasing ordered list* $\ell \in (list\ \mathbb{N})$*, we can find two strings* $p, q \in (list\ D)$ *of the same length such that:*

$$\otimes_\ell\ (p, q)\ \neq\ p$$

```
Lemma canon_crossover_lnil: forall (l: (list nat)),
   st_crec l -> ~(l=nil) -> exists p: (list D), exists q:(list D),
      length p = length q /\ ~(crossover l p q = p).
```

Now, we can prove:

THEOREM 3.5. *For any pair of different lists* $\ell_1, \ell_2 \in (list\ \mathbb{N})$ *in strictly increasing order, it is possible to find* $p, q \in (list\ D)$ *of the same length satisfying:*

$$\otimes_{\ell_1}\ (p, q)\ \neq\ \otimes_{\ell_2}\ (p, q).$$

```
Lemma canon_crossover: forall (l1 l2: (list nat)),
   st_crec l1 -> st_crec l2 -> ~(l1=l2) -> exists p: (list D), exists q:(list D),
      length p = length q /\~(crossover l1 p q = crossover l2 p q).
```

Finally, the reciprocate of 3.3 is:

Corollary 3.6. Let $\ell_1, \ell_2 \in (list\ \mathbb{N})$ be two pairs of lists with *canon* $\ell_1 \neq$ *canon* ℓ_2. We can find two strings $p, q \in (list\ D)$ of the same length verifying:

$$\otimes_{\ell_1}\ (p, q)\ \neq\ \otimes_{\ell_2}\ (p, q).$$

```
Lemma cor_canon_crossover: forall (l1 l2: (list nat)),
   ~(canon l1 = canon l2) -> exists p: (list D), exists q:(list D),
      length p = length q /\ ~(crossover l1 p q = crossover l2 p q).
```

4. CROSSOVER WITH DIFFERENCES

If we analyze, from a computational point of view, the calculation of the crossover operator $\otimes_\ell(p, q)$, for any pair of strings p, q, we conclude that it is not very efficient because, when the list ℓ is of the form $\ell = n :: t$, we need a double recursion in t to compute the crossover $\otimes_\ell(p, q)$ (see Definition 2.2). In the previous section, we have proven that we can always suppose that ℓ is a list in strictly increasing order. Now, we propose an alternative definition of crossover, more efficient than the previous one.

Definition 4.1. Let ℓ be any list of natural numbers. The function *n_crossover* $\oslash_\ell : (list\ D) \times (list\ D) \longrightarrow (list\ D)$ is defined as:

$$\oslash_\ell(p, q) = \begin{cases} p & \text{if } \ell = [\,], \\ (p \uparrow n) + +(\oslash_t\ (q \downarrow n, p \downarrow n) & \text{if } \ell = n :: t \end{cases}$$

```
Fixpoint n_crossover (l:list nat) (p1 p2: (list D)){struct l}: (list D) :=
  match l with
  |  nil => p1
  | n :: t => (first n p1) ++ (n_crossover t (cutting n p2) (cutting n p1))
  end.
```

This new crossover operator also conserves the length of the strings.

THEOREM 4.2. *Let* ℓ *be any list of natural numbers* $\ell \in (list\ \mathbb{N})$ *and* $p, q \in (list\ D)$*. If* $length\ p = length\ q$*, then* $length\ (\oslash_\ell\ (p, q)) = length\ p$*.*

```
Lemma length_n_crossover: forall (l: (list nat)) (p q: (list D)),
   length p = length q -> length (n_crossover l p q) = length p.
```

It is easy to prove that $\otimes_\ell(p,q) = \oslash_\ell(p,q)$, if $\ell = [\,]$ or $\ell = n :: [\,]$ and n is any natural number. This is also true in the general case, that is, we take a list $\ell = \{n_1, \ldots, n_k\}$ in strictly increasing order, and use it to construct another list called *differences list*, denoted by $dif\ \ell = \{n_1, m_2, \ldots, m_k\}$ with $m_i = n_i - n_{i-1}$, for each $2 \leq i \leq k$. We shall prove that the multiple crossover with pattern ℓ is the same as the new one defined with $dif\ \ell$,

$$\otimes_\ell(p,q) = \oslash_{(dif\ \ell)}(p,q).$$

To obtain the differences list of ℓ, we implement a recursive function in Coq:

```
Fixpoint dif_rec (l: (list nat))(n: nat){struct l}: (list nat) :=
  match l with
  | nil => nil
  | n1 :: l1 => (n1 - n) :: (dif_rec l1 n1)
  end.

Definition dif (l: (list nat)) := dif_rec l 0.
```

Note that ℓ must be a strictly increasing ordered list.

The following auxiliary lemmas are necessary to prove the main result of this section:

Lemma 4.3. Let $\ell \in (list\ \mathbb{N})$, $p, q \in (list\ D)$ with the same length and $n \in \mathbb{N}$ such that $(n :: \ell)$ is a strictly increasing ordered list. We can prove that

$$(\otimes_\ell\,(p,q)) \uparrow n \;=\; p \uparrow n.$$

```
Lemma first_elem_crossover: forall (l: (list nat)) (p q: (list D)) (n: nat),
   length p = length q -> st_crec (n :: l) -> first n (crossover l p q) = first n p.
```

Lemma 4.4. Let $\ell \in (list\ \mathbb{N})$, $p, q \in (list\ D)$ with the same length and $n \in \mathbb{N}$ such that $(n :: \ell)$ is a strictly increasing ordered list. It holds that:

$$\oslash_{(dif\_rec\ \ell\ n)}\,(p \downarrow n, q \downarrow n) \;=\; (\oslash_{(dif\ \ell)}(p,q)) \downarrow n.$$

```
Lemma dif_rec_Dif_crossover: forall (l: (list nat)) (p q: (list D))(n: nat),
   length p = length q -> st_crec (n :: l)->
      n_crossover (dif_rec l n) (cutting n p) (cutting n q) =
             cutting n (n_crossover (dif l) p q).
```

Finally, we verify:

THEOREM 4.5. *Let $p, q \in (list\ D)$ with the same length and $\ell \in (list\ \mathbb{N})$ a strictly increasing ordered list. It can be proven that:*

$$\oslash_{(dif\ \ell)}\,(p,q) \;=\; \otimes_\ell(p,q).$$

```
Theorem n_crossover_eq_crossover: forall (l: (list nat)) (p1 p2: (list D)),
   length p1 = length p2 -> (st_crec l) ->
      n_crossover (dif l) p1 p2 = crossover l p1 p2.
```

Proof:

If we proceed by structural induction over the list ℓ, we obtain two cases:

(1) If $\ell = [\,]$, it follows directly that $\oslash_{[\,]}(p,q) = p = \otimes_{[\,]}(p,q)$.
(2) In the case $\ell = a :: \ell_1$, we have:

$$\oslash_{(dif\ (a :: \ell_1))}\ (p,q) = p \uparrow a +\!+ \oslash_{(dif\_rec\ \ell_1\ a)}\ (q \downarrow a, p \downarrow a).$$

As the list $a :: \ell_1$ is in strictly increasing order, by 4.3 we have that:

$$(\otimes_{\ell_1}\ (p,q)) \uparrow a = p \uparrow a.$$

Now, if we apply 4.4 and the recurrence hypothesis, we obtain:

$$\oslash_{(dif\_rec\ \ell_1\ a)}\ (q \downarrow a, p \downarrow a) = (\oslash_{(dif\ \ell_1)}(q,p)) \downarrow a = (\otimes_{\ell}(q,p)) \downarrow a$$

and then, from the above equalities, it can be proven that:
$\oslash_{(dif\ (a :: \ell_1))}\ (p,q) = \otimes_{\ell_1}\ (p,q)) \uparrow a +\!+ (\otimes_{\ell_1}\ (q,p)) \downarrow a = \otimes_{(a :: \ell_1)}(p,q)$.

5. EQUIVALENCE WITH CLASSICAL FORMALIZATION

For genetic algorithms, the classical crossover with one point starts with two strings (parents) and after the reproduction process obtains two strings (children)[6] in the following way:

$$\odot_n\ (p,q) = (p \uparrow n +\!+ q \downarrow n, q \uparrow n +\!+ p \downarrow n).$$

The corresponding implementation in Coq of this operator would be:

```
Definition crossover_s (n: nat) (t: (list D)*(list D)) :=
   (((first n (fst t)) ++ (cutting n (snd t)),
    ((first n (snd t)) ++ (cutting n (fst t))))).
```

In the same way, the multiple crossover with a list of points can be seen as the composition (sequential concatenation) of the corresponding simple crossover operators:

$$\bigcirc_{[n_1;n_2;\ldots;n_k]}\ (p,q) = \odot_{n_1}\ (\odot_{n_2}(\ldots(\odot_{n_k}\ (p,q))))$$

or more formally:

Definition 5.1. If ℓ (pattern) is any list of natural numbers, we define the **crossover function** $\bigcirc_\ell : (list\ D) \times (list\ D) \longrightarrow (list\ D) \times (list\ D)$ as:

$$\bigcirc_\ell(p,q) = \begin{cases} (p,q) & \text{if } \ell = [\,], \\ (\odot_n\ (\bigcirc_t\ (p,q))) & \text{if } \ell = n :: t. \end{cases}$$

We implement this function in Coq in the following way:

```
Fixpoint crossover_m (l: (list nat)) (t: (list D)*(list D)){struct l}:
  (list D)*(list D) :=
```

[6]Usually in the context of GA the size of the population remains fixed after the reproduction process.

```
match l with
|  nil => ((fst t),(snd t))
| n :: l1 => crossover_s n (crossover_m l1 ((fst t),(snd t)))
end.
```

We include now a result which shows that our crossover definition does not distort the "classical" crossover operator used in genetic algorithms:

THEOREM 5.2. *For any* $\ell \in (list\ \mathbb{N})$ *and* $p, q \in (list\ D)$*, we have verified that:*

$$\bigcirc_\ell\ (p,q)\ =\ (\otimes_\ell\ (p,q), \otimes_\ell\ (q,p)).$$

```
Lemma equiv_crossover: forall (l: (list nat)) (p q: (list D)),
   (crossover_m l (p,q)) = ((crossover l p q),(crossover l q p)).
```

6. CONCLUSIONS AND FUTURE WORK

Starting from Uchibori and Endou [UE99], we develop a generalization of multiple crossover definitions (in a GA context) and a formally verified implementation of this operator in a functional programming language. This operator can be directly used in the implementation of any genetic algorithm. The following objectives have been achieved:

—The properties of the paper [UE99] and some new ones referring to the generalized crossover operator have been formally verified.

—It has been proven that multiple crossover is not related to the order of the points where it is applied.

—A more efficient definition of multiple crossover (based on "lists of differences") has been developed and implemented; moreover, its equivalence with the initial definition was formally proven.

—And, finally, we introduce a direct definition of multiple crossover closer to the concept used normally in GA. We demonstrate its equivalence with the definition previously implemented in this paper.

This work uses the Coq proof assistant and its system of code extraction. Some of the final results were suggested by the very development of the proofs of some properties. The required effort for the formalization of the results has allowed to clarify some points of the proofs. Moreover, in some cases it has also led to a more detailed proof of the statements.

The work could be completed in two ways:

—It would be interesting to specify and prove intrinsic properties of the mutation operator in Coq. Another point would be to create a GA library in Coq and to develop general tactics of automatization in Fields and Ring tactics style which would improve the efficiency of the development.

—We are also interested in the formalization of the crossover operator and its properties in the PVS system ([COR+95]), which would allow us to analyze, in a pragmatic way, the similarities and differences existing between both Proof Assistants.

References

[ADM+07] F. Aguado, J. L. Doncel, J. M. Molinelli, G. Pérez, C. Vidal, and A. Vieites. Certified genetic algorithms: Crossover operators for permutations. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, editors, *EUROCAST*, volume 4739 of *Lecture Notes in Computer Science*, pages 282–289. Springer, 2007.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.

[DN91] A. Darmochwal and Y. Nakamura. The topological space E2T. Arcs, line segments and special polygonal arcs. *Journal of Formalized Mathematics*, 3:617–621, 1991.

[ECS89] L. J. Eshelman, R. A. Caruana, and J. D. Schaffer. Biases in the crossover landscape. In *Proceedings of the third international conference on Genetic algorithms*, pages 10–19, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[FHB+98] J. C. Filliâtre, H. Herbelin, B. Barras, S. Boutin, C. Cornes, J. Courant, C. Murthy, C. Parent, Ch. Paulin-Mohring, A. Saïbi, and B. Werner. The coq proof assistant reference manual. Technical report, 1998.

[Hol92] J. H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1992.

[INR] INRIA. The coq standard library. http://coq.inria.fr/library/.

[Jon75] K. A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Ann Arbor, MI, USA, 1975.

[Kot93] J. Kotowicz. Functions and finite sequences of real numbers. *Journal of Formalized Mathematics*, 5:1–4, 1993.

[PM89] Ch. Paulin-Mhoring. *Extraction de programmes dans le calcul des constructions.* Thése de doctorat, Université de Paris VII, January 1989.

[PM93] Ch. Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, London, UK, 1993. Springer-Verlag.

[UE99] A. Uchibori and N. Endou. Basic properties of genetic algorithm. *Journal of Formalized Mathematics*, 8:151–160, 1999.