Proof Pearl: Constructive Extraction of Cycle Finding Algorithms

Dominique Larchey-Wendling

Proof Pearl: Constructive Extraction of Cycle Finding Algorithms

Dominique Larchey-Wendling

Université de Lorraine, CNRS, LORIA, Nancy, France
dominique.larchey-wendling@loria.fr

Abstract. We present a short implementation of the well-known Tortoise and Hare cycle finding algorithm in the constructive setting of Coq. This algorithm is interesting from a constructive perspective because it is both very simple and potentially non-terminating (depending on the input). To overcome potential non-termination, we encode the given termination argument (there exists a cycle) into a bar inductive predicate that we use as termination certificate. From this development, we extract the standard OCaml implementation of this algorithm. We generalize the method to the full Floyd's algorithm that computes the entry point and the period of the cycle in the iterated sequence, and to the more efficient Brent's algorithm for computing the period only, again with accurate extractions of their respective standard OCaml implementations.

Keywords: Cycle finding · Bar inductive predicates · Partial algorithms in Coq · Correctness by extraction

1 Introduction

The Tortoise and the Hare (T&H for short) in particular and cycle detection [1] in general are standard algorithms that will very likely cross the path of any would-be computer scientist. They aim at detecting cycles in *deterministic* sequences of values, i.e. when the next value depends only on the current value. They have many applications, from pseudorandom number strength measurement, integer factorization through Pollard's rho algorithm [18] or more generally cryptography, etc, even celestial mechanics. But our interest with those algorithms lies more in the framework in which we want to implement and certify them:

- first we want to prove the partial correction of cycle detection algorithms without assuming their termination. Hence, we do not restrict our study to finite domains. In the finitary case indeed, the *pigeon hole principle* ensures that there is always a cycle to detect and termination can be certified by cardinality considerations [9];
- inductive/constructive type theories constitute challenging contexts for these algorithms because they are inherently partial. The reason for that is the undecidability of the existence of a cycle in an arbitrary given sequence. Hence, we need to work with partial recursive functions;

Work partially supported by the TICAMORE project (ANR grant 16-CE91-0002).

– Coq could wrongfully be considered being limited to total functions. To work around this, Hilbert's ϵ-operator (a non-constructive form of the axiom of choice) is sometimes postulated as a convenient way to deal with partial functions [7]. Why not, HOL is based on it? We argue that we can stay fully constructive and we will show that axiom-free Coq can work with such partial recursive functions provided they are precisely specified.

T&H was attributed to Robert W. Floyd by Donald E. Knuth [15] but may in fact be a folk theorem (see [2], footnote 8 on page 21). The idea is to launch from a starting point x_0 both a slow tortoise (the tortoise steps once at each iteration) and a quick hare (the hare steps twice at each iteration). Then the hare will recapture the tortoise if and only if there is a cycle in the sequence from x_0. We refer to [1] for further visual explanations about the origin and intuition behind this very well-known algorithm. The T&H algorithm computes only a meeting point for the two fabulous animals. We will call *Floyd's cycle finding* the algorithm that builds on this technique to compute the entry point and period of the cycle. We also consider *Brent's period only finding algorithm* [6] that proceeds with a slow hare and a so-called "teleporting tortoise."

Because these algorithms do not always terminate, defining the corresponding fully specified Coq fixpoints might be considered challenging. The folklore *fuel* trick could be used to simulate general recursion in Coq. The idea is to use a term of the type $X \to (\texttt{fuel} : \texttt{nat}) \to \texttt{option}\ Y$ to represent a partial recursive function $f : X \rightharpoonup Y$. The `fuel` argument ensures termination, as e.g. a bound on the number of recursive sub-calls. This fuel trick has several problems: computing a big-enough `fuel` value from the input $x : X$ might be as complicated as showing the termination of $f(x)$ itself; but also, the `fuel` argument is *informative* and is thus preserved by extraction, both as a parasitic argument and as a companion program for computing the input value of `fuel` from the value of x; and finally, the output type is now $\texttt{option}\ Y$ instead of Y so an extra `match` construct is necessary. We will show how to replace the informative `fuel` argument with a *non-informative* bar inductive predicate to ensure termination. As such, it is erased by extraction, getting us rid of parasitic arguments. In particular, we obtain an OCaml extraction of T&H certified by less than 80 lines of Coq code.

2 Formalization of the problem

Given a set X and a function $f : X \to X$, we define the n-th iterate $f^n : X \to X$ of f by induction on n: $f^0 = x \mapsto x$ and $f^{n+1} = f \circ f^n$. In Coq, this definition corresponds to the code of the iterator (with a convenient compact f^n notation):

```
Fixpoint iter {X : Type} (f : X → X) n x : X :=
  match n with O ↦ x | S n ↦ f (f^n x) end
where " f^n " := (iter f n).
```

We get the identity $f^{a+b}(x) = f^a\,(f^b(x))$ by induction on a. Given a starting point $x_0 \in X$, we consider the infinite sequence $x_0, f(x_0), f^2(x_0), \ldots, f^n(x_0), \ldots$

of iterates of f on x_0, i.e. the map $n \mapsto f^n(x_0)$. From a classical logic perspective, two mutually exclusive alternatives are possible:

A1 the sequence $n \mapsto f^n(x_0)$ is injective, i.e. $f^i(x_0) \neq f^j(x_0)$ holds unless $i = j$. In this case, there is no cycle in the iterated sequence from x_0;

A2 there exist $i \neq j$ such that $f^i(x_0) = f^j(x_0)$ and in this case, there is a cycle in the iterated sequence from x_0.

It is however not possible to computationally distinguish those two cases: no cycle finding algorithm can be both correct and always terminating. To show this undecidability result, one can reduce the Halting problem to the cycle detection problem (see file cycle_undec.v).

The T&H algorithm terminates exactly when Alternative A2 above holds and loops forever when Alternative A1 holds. There are many equivalent characterizations of the existence of a cycle, which we call the *cyclicity property*.

Proposition 1 (Cyclicity). *For any set X, any function $f : X \to X$ and any $x_0 \in X$, the four following conditions are equivalent:*

1. *there exist $i, j \in \mathbb{N}$ such that $i \neq j$ and $f^i(x_0) = f^j(x_0)$;*
2. *there exist $\lambda, \mu \in \mathbb{N}$ such that $0 < \mu$ and $f^\lambda(x_0) = f^{\lambda+\mu}(x_0)$;*
3. *there exist $\lambda, \mu \in \mathbb{N}$ s.t. $0 < \mu$ and for any $i, j \in \mathbb{N}$, $f^{i+\lambda}(x_0) = f^{i+\lambda+j\mu}(x_0)$;*
4. *there exists $\tau \in \mathbb{N}$ such that $0 < \tau$ and $f^\tau(x_0) = f^{2\tau}(x_0)$.*

Proof. For $1 \Rightarrow 2$, if $i < j$ then choose $\lambda = i$ and $\mu = j - i$ (exchange i and j if otherwise $j < i$). For $2 \Rightarrow 3$, first show $f^\lambda(x_0) = f^{\lambda+j\mu}(x_0)$ by induction on j. Then $f^{i+\lambda}(x_0) = f^i(f^\lambda(x_0)) = f^i(f^{\lambda+j\mu}(x_0)) = f^{i+\lambda+j\mu}(x_0)$. For $3 \Rightarrow 4$, choose $\tau = (1 + \lambda)\mu$ and derive $f^\tau(x_0) = f^{2\tau}(x_0)$ using $i = (1 + \lambda)\mu - \lambda$ and $j = 1 + \lambda$. For $4 \Rightarrow 1$, choose $i = \tau$ and $j = 2\tau$. That proof is mechanized as Proposition `cyclicity_prop` in file utils.v. □

The functional specification of T&H is to compute a meeting index $\tau \in \mathbb{N}$ such that $0 < \tau$ and $f^\tau(x_0) = f^{2\tau}(x_0)$ (corresponding to Item 4 of Proposition 1), provided such a value exists. Operationally, the algorithm consists in enumerating the sequence of pairs $(f(x_0), f^2(x_0)), (f^2(x_0), f^4(x_0)), \ldots, (f^n(x_0), f^{2n}(x_0)), \ldots$ in an efficient way until the two values $f^n(x_0)$ and $f^{2n}(x_0)$ are equal.

2.1 An OCaml account of the Tortoise and the Hare

The T&H algorithm can be expressed in OCaml as the following two functions:

```
tort_hare_rec : (f : α → α) → (x : α) → (y : α) → int
tortoise_hare : (f : α → α) → (x0 : α) → int
let rec tort_hare_rec f x y =
  if x = y then 0 else 1 + tort_hare_rec f (f x) (f (f y))
let tortoise_hare f x0 = 1 + tort_hare_rec f (f x0) (f (f x0))
```

In general, the tail-recursive version is preferred because tail-recursive functions can be compiled into loops without the help of a stack. The code of the function

`tortoise_hare_tail` contains a sub-function `loop` where the first argument f of `tortoise_hare_tail` is fixed and second argument x_0 is unused.

$$\begin{array}{l}\texttt{tortoise\_hare\_tail} : (f : \alpha \to \alpha) \to (x_0 : \alpha) \to \texttt{int} \\ \quad \texttt{loop} : (n : \texttt{int}) \to (x : \alpha) \to (y : \alpha) \to \texttt{int} \\ \texttt{let tortoise\_hare\_tail } f\ x_0 = \\ \quad \texttt{let rec loop } n\ x\ y = \texttt{if } x = y \texttt{ then } n \texttt{ else loop } (1+n)\ (f\,x)\ (f\,(f\,y)) \\ \quad \texttt{in loop } 1\ (f\,x_0)\ (f\,(f\,x_0))\end{array}$$

Notice that the pre-condition of cyclicity (any item of Proposition 1) is necessary otherwise the above OCaml code does not terminate and is thus incorrect. Any correctness proof must include that cyclicity pre-condition, or a stronger one.

2.2 Goals and contributions

The goal of this work is double:

Goal 1: functional correctness. Using purely constructive means, build fully specified Coq terms that compute a meeting point for the tortoise and the hare, with the sole pre-condition of cyclicity. Reiterate this for Floyd's and Brent's cycle finding algorithms;

Goal 2: operational correctness. Ensure that the extraction of the previous Coq terms give the corresponding standard OCaml implementations. In particular, derive the above implementations of `tortoise_hare` and `tortoise_hare_tail` by extraction.

From these two goals, trusting Coq extraction mechanism, we get the functional correctness of the standard OCaml implementations for free.

For T&H, solving Goal 1 can be viewed as constructing a term of type

$$\texttt{th\_coq} : (\exists \tau,\, 0 < \tau \wedge f^\tau\, x_0 = f^{2\tau}\, x_0) \to \{\tau \mid 0 < \tau \wedge f^\tau\, x_0 = f^{2\tau}\, x_0\}$$

from the assumptions of a type X : `Type`, a procedure $=^?_X$ for deciding equality over X, and a sequence given by $f : X \to X$ and $x_0 : X$. Notice the assumption $=^?_X : \forall x\, y : X, \{x = y\} + \{x \neq y\}$ of an *equality decider* for X that is necessary in Coq. Indeed, unlike OCaml which has a built-in polymorphic equality decider,[1] Coq does (and can) not have equality deciders for every possible type.

Solving Goal 2 means that after extraction of OCaml code from `th_coq`, we get the same function code as `tortoise_hare` (resp. `tortoise_hare_tail`).

This paper is a companion for Coq implementations of cycle finding algorithms. The corresponding source code can be found at

`https://github.com/DmxLarchey/The-Tortoise-and-the-Hare`

The implementation involves around 3000 lines of Coq code but this does not reflect the compactness of our implementation of T&H. Indeed, it contains Floyd's

[1] OCaml equality decider is partially correct, e.g. it throws exceptions on functions.

and Brent's algorithms as well, and there are several accompanying files illustrating certified recursion through bar inductive predicates. To witness the conciseness of our approach, we give a standalone tail-recursive implementation of T&H of less than 80 lines, not counting comments (see th_alone.v). This project compiles under Coq 8.6 and is available under a Free Software license.

The designs of the cycle finding algorithms that we propose are all based on bar inductive predicates used as termination certificates for Coq fixpoint recursion. In Section 3, we give a brief introduction to these predicates from a programmer's point of view and show why they are suited for solving termination problems. The corresponding Coq source code can be found in file bar.v.

In Section 4, we present two fully specified implementations of T&H, one non-tail recursive and one tail-recursive. We give a detailed account of the algorithmic part of the implementation that we isolate from *logical obligations*. We explain how bar inductive predicates are used to separate/postpone termination proofs from algorithmic considerations. The corresponding file is tortoise_hare.v.

In Section 5, we give an overview of the implementation of the full Floyd cycle finding algorithm that computes the characteristic index and period of an iterated sequence. The corresponding Coq file is floyd.v. In Section 6, we give a brief account of our implementation of Brent's period finding algorithm, in fact two implementations: one suited for binary numbers and one suited for unary numbers. The corresponding source code files are brent_bin.v and brent_una.v.

The T&H has already been the subject of implementations in Coq [9,11], but under different requirements. In Section 7, we compare our development with those alternative approaches. From a constructive point of view, we analyse the pre-conditions under which correctness is established in each case.

3 Termination using Bar Inductive Predicates

In this section, we explain how to use *bar inductive predicates* [12] — a constructive and axiom-free form of bar induction[2] — as termination certificates.

As explained in Section 3.2, in the context we use them (decidable terminated cases), these predicates have the same expressive power as the *accessibility predicates* used for well-founded recursion in Coq in the modules `Wf` and `Wellfounded` from the standard library (see also left part of Fig. 1). But we think that bar inductive predicates have several advantages over accessibility predicates:

- compared to the general accessibility predicates of [4], they do not need the simultaneous induction/recursion schemes of Dybjer [10] (not integrated in Coq so far) in case of nested/mutual recursion [17];
- unlike standard accessibility predicates (module `Wf`) which involve thinking about termination before implementing the algorithm, or *inductively defined domain predicates* (see [3] pp 427–432) which involve thinking about termination together with the algorithm, bar inductive predicates focuses on

[2] Conventional bar induction often requires *Brouwer's thesis* which precisely postulates that bar predicates are inductive.

$$\frac{\forall y, R\ y\ x \to \texttt{Acc}\ y}{\texttt{Acc}\ x} \qquad \left|\ \right. \qquad \frac{T\ x}{\texttt{bar}\ x} \qquad \frac{\forall y, R\ x\ y \to \texttt{bar}\ y}{\texttt{bar}\ x}$$

Fig. 1. Inductive rules for `Acc` and `bar` termination certificates.

terminated cases and *recursive sub-calls* so termination proofs can be separated from the algorithm.

We argue that separating/postponing the proof of termination makes the use of bar inductive predicates more versatile. At least, we hope that we illustrate our case here. Of course, a comprehensive comparison with [5] would be necessary to complete our case. The reader could be interested in recent developments that show that the method of bar inductive predicates scales well to more complicated nested/mutual recursive schemes [17].

We do not really introduce new concepts in the section. But we want to stress the links between the notion of *cover-induction* [8] and the notion of *bar inductive predicate* (e.g. inductive bars [12]). We insist on these notions because we will specialize the following generic implementation to get an "extraction friendly" Coq definition of cycle finding algorithms.

3.1 Dependently typed recursion for bar inductive predicates

Let us consider a type X, a unary relation $T : X \to \texttt{Prop}$ and a binary relation $R : X \to X \to \texttt{Prop}$. Here are some possible intuitive interpretations of T and R:

***T x*:** the computation at point x is terminated (no recursive sub-call);
***R x y*:** a call at point x may trigger a recursive sub-call at point y.

We define the inductive predicate $\texttt{bar} : X \to \texttt{Prop}$ which covers points where computation is warrantied to terminate, by the two rules on the right of Fig. 1:

$\texttt{Variables}\ (X : \texttt{Type})\ (T : X \to \texttt{Prop})\ (R : X \to X \to \texttt{Prop}).$
$\texttt{Inductive bar}\ (x : X) : \texttt{Prop} :=$
$\quad \mid \texttt{in\_bar\_0} :\ T\ x \to \texttt{bar}\ x$
$\quad \mid \texttt{in\_bar\_1} :\ (\forall y, R\ x\ y \to \texttt{bar}\ y) \to \texttt{bar}\ x.$

The first rule `in_bar_0` states that a terminated computation terminates. The second rule `in_bar_1` states that if every recursive sub-call y of x terminates then so is the call at x. Notice that the predicate $\texttt{bar}\ x : \texttt{Prop}$ carries no computational content and thus cannot be used to perform computational choices. Termination is only warrantied by the $\texttt{bar}\ x$ predicate, it is not performed by it.

Hence we assume a decider term $T_{\texttt{dec}} : \forall x, \{T\ x\} + \{\neg T\ x\}$ for terminated points. We then define `bar_rect`, a dependently typed recursion principle for $\texttt{bar}\ x$. For this, we need the following inductions hypotheses:

$\texttt{Hypothesis}\quad (T_{\texttt{dec}} : \forall x, \{T\ x\} + \{\neg T\ x\}).$
$\texttt{Variable}\quad\ (P : X \to \texttt{Type}).$
$\texttt{Hypothesis}\quad (H_T : \forall x, T\ x \to P\ x)\ (H_{\texttt{bar}} : \forall x, (\forall y, R\ x\ y \to P\ y) \to P\ x).$

where H_T gives the value for terminated points and $H_{\texttt{bar}}$ combines the values of the recursive sub-calls into a value for the call itself. With these assumptions, we get the following dependently typed induction principle:

$$\begin{array}{l}\texttt{Fixpoint bar\_rect } x\ (H : \texttt{bar } x)\ \{\texttt{struct } H\} : P\ x := \\ \quad \texttt{match } T_{\texttt{dec}}\ x \texttt{ with} \\ \quad\quad \mid \texttt{left } H_x \ \mapsto H_T\ \_\ H_x \\ \quad\quad \mid \texttt{right } H_x \mapsto H_{\texttt{bar}}\ \_\ (\texttt{fun } y\ H_y \mapsto \texttt{bar\_rect } y\ \mathbb{G}_1^?) \\ \quad \texttt{end}.\end{array}$$

where $\mathbb{G}_1^?$ is a proof term for a *logical obligation*:

$$\mathbb{G}_1^? \ \ // \ \ \ldots, x : X, H : \texttt{bar } x, H_x : \neg T\,x, y : X, H_y : R\ x\ y \vdash \texttt{bar } y$$

Notice that for Coq to accept such a `Fixpoint` definition as well-typed, one must ensure that the given proof of goal $\mathbb{G}_1^?$ is a sub-term of the term $H : \texttt{bar } x$ because H is declared as the *structurally decreasing argument* of this fixpoint. Hence, the first step in the proof of $\mathbb{G}_1^?$ is `destruct` H.

The above implementation of `bar_rect` expects the proof term of $\mathbb{G}_1^?$ to be given *before* the actual `Fixpoint` definition of `bar_rect`. This can be mitigated with the use of the very handy `refine` tactic that can delay the proof obligations after the *incomplete proof term* is given (see `bar.v` for details). As a final remark concerning the term `bar_rect`, there are two ways of stopping a chain of recursive sub-calls: the first is obviously to reach a terminated point (i.e. $T\,x$) but the chain can also stop when there is zero recursive sub-calls (i.e. $R\ x\ y$ holds for no y). While the first condition of terminated points is decidable, the second condition of the nonexistence of recursive sub-calls is usually not decidable. Hence when using the accessibility predicate $\texttt{Acc } (\texttt{fun } u\ v \mapsto R\ v\ u \wedge \neg T\,v)\ x$ which mixes both T and R (see Theorem `bar_Acc_eq_dec` below), detecting the first termination condition is less natural.

3.2 Accessibility vs. bar inductive predicates

We show that bar inductive predicates generalize accessibility predicates defined in the Coq standard library module `Wf`,

$$\begin{array}{l}\texttt{Theorem bar\_empty\_Acc\_eq } (X : \texttt{Type})\ (R : X \to X \to \texttt{Prop})\ (x : X) : \\ \qquad \texttt{bar } (\texttt{fun } \_ \mapsto \texttt{False})\ R\ x \iff \texttt{Acc } R^{-1}\ x\end{array}$$

which is obvious from the rules of Fig. 1 because when $T = \texttt{fun } \_ \mapsto \texttt{False}$ is empty, one cannot use rule `in_bar_0`. Then, we show that when $T : X \to \texttt{Prop}$ is (logically) decidable, then `bar` $T\ R$ can be encoded as an accessibility predicate:

$$\begin{array}{l}\texttt{Theorem bar\_Acc\_eq\_dec } X\ (T : X \to \texttt{Prop})\ (R : X \to X \to \texttt{Prop}) : \\ \quad (\forall x, T\,x \vee \neg T\,x) \to \forall x, \texttt{bar } T\ R\ x \iff \texttt{Acc } (\texttt{fun } u\ v \mapsto R\ v\ u \wedge \neg T\,v)\ x\end{array}$$

From our point of view, the advantage of `bar` over `Acc` is that they keep the two forms of termination separate (x is terminated by T vs. x generates no recursive sub-call), making them easier to reason or compute with. Moreover, using

`Acc` incites at using only well-founded relations (or even decreasing measures) whereas `bar` focuses on terminated points/recursive calls and thus can be used more freely as exemplified in Sections 4 and 5.

3.3 Constructive epsilon via bar inductive predicates

As a first illustration of using bar inductive predicates, we show how to implement *Constructive Indefinite Ground Description* defined in the standard library module `ConstructiveEpsilon`.

$$\texttt{Theorem Constructive\_Epsilon } (Q : \texttt{nat} \to \texttt{Prop}) : \\ \big(\forall n, \{Q\,n\} + \{\neg Q\,n\}\big) \to (\exists n, Q\ n) \to \{n : \texttt{nat} \mid Q\ n\}.$$

We instantiate `bar_rect` with $(T := Q)$, $(R\,x\,y := \texttt{S}\,x = y)$ and $(P\ \_ := \{x \mid Q\ x\})$. We only have to transform the termination certificate $\exists n, Q\,n$ into a bar inductive predicate at the purely logical/`Prop` level. For this, we show $(\exists n, Q\,n) \to \texttt{bar}\ Q\ R\ 0$: from $Q\,n$ deduce $\texttt{bar}\ Q\ R\ n$ using `in_bar_0` and then $\texttt{bar}\ Q\ R\ (n-1)$,... down to $\texttt{bar}\ Q\ R\ 0$ by descending induction[3] using `in_bar_1`.

Notice that using the previous development, we can already implement the functional specification of the T&H algorithm:

$$\texttt{th\_min} : (\exists \tau, 0 < \tau \wedge f^\tau\,x_0 = f^{2\tau}\,x_0) \to \{\tau \mid 0 < \tau \wedge f^\tau\,x_0 = f^{2\tau}\,x_0\}$$

by application of `Constructive_Epsilon` with $(Q\,n := 0 < n \wedge f^n\,x_0 = f^{2n}\,x_0)$. Indeed, such $Q : \texttt{nat} \to \texttt{Prop}$ is computationally decidable as both $< : \texttt{nat} \to \texttt{nat} \to \texttt{Prop}$ and $=_X : X \to X \to \texttt{Prop}$ are computationally decidable.[4]

However, this approach will not give us the operational specification of T&H because this implementation of `th_min` using `Constructive_Epsilon` extracts into the inefficient unbounded minimization algorithm on the right-hand side (see also th_min.v). There, $\mu\texttt{min}$ corresponds to unbounded minimization. This `th_min` program recomputes $f^n\,x_0$ and $f^{2n}\,x_0$ for each value of n before a cycle is detected, making it really inefficient.

```
let th_min f x0 =
  let rec μmin n =
    if (n = 0) or (f^n x0 ≠ f^2n x0)
    then μmin (1 + n)
    else n
  in μmin 0
```

4 The Tortoise and the Hare via Bar Inductive Predicates

In this section, we use the methodology of Section 3 (i.e. termination via bar inductive predicates) to design a fully specified implementation of the T&H algorithm that satisfies Goals 1 and 2 of Section 2.2. We could use `bar_rect` to implement this algorithm but we do not use it directly. Indeed, we want to finely control the computational content of our terms so that we can extract the

[3] descending induction is implemented by `nat_rev_ind` in file utils.v.

[4] for $=_X$, this is precisely the assumption of the $=_X^?$ equality decider.

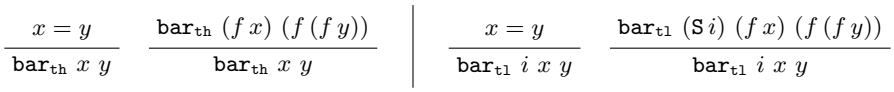

Fig. 2. Inductive rules for $\texttt{bar}_{\texttt{th}} : X \to X \to \texttt{Prop}$ and $\texttt{bar}_{\texttt{tl}} : \texttt{nat} \to X \to X \to \texttt{Prop}$.

expected OCaml code accurately. However, we will mimic the implementation of `bar_rect` several times. The corresponding file for this section is tortoise_hare.v.

The T&H detects potential cycles in the iterated values of an endo-function $f : X \to X$. As explained in Section 2.2, for the remaining of this section, we assume the following pre-conditions for the hare to recapture the tortoise:

$$\begin{array}{ll}\texttt{Variables} & (X : \texttt{Type})\ (=^?_X : \forall x\, y : X, \{x = y\} + \{x \neq y\}) \\ & (f : X \to X)\ (x_0 : X)\ (H_0 : \exists \tau,\, 0 < \tau \wedge f^\tau\, x_0 = f^{2\tau}\, x_0).\end{array}$$

that is a type X with an equality decider $=^?_X$, a sequence f starting at point x_0 satisfying a cyclicity assumption H_0 (see Proposition 1). These pre-conditions are not minimal for establishing the correctness of T&H[5] but we do think they are general enough to accommodate most use cases of T&H.

4.1 A non-tail recursive implementation

Let us start with the non-tail recursive implementation of T&H, as is done in the OCaml code of `tortoise_hare` (see Section 2.1). We define a bar inductive predicate which will be used as termination certificate for the main loop `tort_hare_rec`. Compared to the generic inductive definition of `bar` of Section 3, $\texttt{bar}_{\texttt{th}}$ is a binary (instead of unary) bar predicate specialized with $(T\, x\, y := x = y)$ and $(R\, x\, y\, u\, v := u = f\, x \wedge v = f\, (f\, y))$:

$$\begin{array}{l}\texttt{Inductive}\ \texttt{bar}_{\texttt{th}}\ (x\ y : X) : \texttt{Prop} := \\ \quad \mid \texttt{in\_bar\_th\_0} : \ x = y \qquad\qquad\qquad\quad \to\ \texttt{bar}_{\texttt{th}}\ x\ y \\ \quad \mid \texttt{in\_bar\_th\_1} : \ \texttt{bar}_{\texttt{th}}\ (f\, x)\ (f\, (f\, y)) \ \to\ \texttt{bar}_{\texttt{th}}\ x\ y\end{array}$$

This definition matches the inductive rules of Fig. 2 (left part). We define a fully specified Coq term `tort_hare_rec` mimicking both the OCaml code of Section 2.1 (with the addition of a termination certificate of type $\texttt{bar}_{\texttt{th}}\, x\, y$) and the code of `bar_rect` of Section 3:

$$\begin{array}{l}\texttt{Fixpoint}\ \texttt{tort\_hare\_rec}\ x\ y\ (H : \texttt{bar}_{\texttt{th}}\ x\ y) : \{k \mid f^k\, x = f^{2k}\, y\} := \\ \quad \texttt{match}\ x =^?_X y\ \texttt{with} \\ \qquad \mid \texttt{left}\ E \ \mapsto \texttt{exist}\ \_\ 0\ \mathbb{G}^?_1 \\ \qquad \mid \texttt{right}\ C \mapsto \texttt{match}\ \texttt{tort\_hare\_rec}\ (f\, x)\ (f\, (f\, y))\ \mathbb{G}^?_2\ \texttt{with} \\ \qquad\qquad\qquad \mid \texttt{exist}\ \_\ k\ H_k \mapsto \texttt{exist}\ \_\ (\texttt{S}\ k)\ \mathbb{G}^?_3 \\ \qquad\qquad\quad \texttt{end} \\ \quad \texttt{end}.\end{array}$$

[5] see th_rel.v where an arbitrary decidable relation $R : X \to X \to \texttt{Prop}$ replaces $=_X$.

where $\mathbb{G}_1^?$, $\mathbb{G}_2^?$ and $\mathbb{G}_3^?$ are three proof terms of the following types:

$$\begin{array}{lll}\mathbb{G}_1^? & // & \ldots, E : x = y \vdash f^0\ x = f^{2.0}\ y \\ \mathbb{G}_2^? & // & \ldots, C : x \neq y, H : \mathtt{bar_{th}}\ x\ y \vdash \mathtt{bar_{th}}\ (f\,x)\ (f\,(f\,y)) \\ \mathbb{G}_3^? & // & \ldots, H_k : f^k\,(f\,x) = f^{2k}\,(f\,(f\,y)) \vdash f^{\mathtt{S}\,k}\ x = f^{2(\mathtt{S}\,k)}\ y\end{array}$$

These can be established before the `Fixpoint` definition of `tort_hare_rec` or else (preferably), using the Coq `refine` tactic, after the statement of the computational part of `tort_hare_rec`, as remaining logical obligations (see tortoise_hare.v for exact Coq code). Recall that the termination certificate H must structurally decrease, i.e. the proof term for $\mathbb{G}_2^?$ must be a sub-term of H.

We can now define `tortoise_hare` by calling `tort_hare_rec` but we need to provide a termination certificate:

```
Definition tortoise_hare : {τ | 0 < τ ∧ f^τ x0 = f^(2τ) x0} :=
  match tort_hare_rec (f x0) (f (f x0)) G1? with
    | exist _ k Hk ↦ exist _ (S k) G2?
  end.
```

There are two remaining logical obligations, $\mathbb{G}_1^?$ being the termination certificate:

$$\begin{array}{lll}\mathbb{G}_1^? & // & \ldots, H_0 : \exists \tau, 0 < \tau \wedge f^\tau\,x_0 = f^{2\tau}\,x_0 \vdash \mathtt{bar\_th}\ (f\,x_0)\ (f\,(f\,x_0)) \\ \mathbb{G}_2^? & // & \ldots, H_k : f^k\,(f\,x_0) = f^{2k}\,(f\,(f\,x_0)) \vdash 0 < \mathtt{S}\,k \wedge f^{\mathtt{S}\,k}\ x_0 = f^{2(\mathtt{S}\,k)}\ x_0\end{array}$$

We prove $\mathbb{G}_1^?$ as follows: from H_0, we (non-computationally) deduce m such that $0 < m$ and $f^m\,x_0 = f^{2m}\,x_0$. Using `in_bar_th_0` we immediately get $\mathtt{bar_{th}}\ (f^m\,x_0)\ (f^{2m}\,x_0)$. Then using `in_bar_th_1` repeatedly from m, $m-1$, ... down to 1 we get $\mathtt{bar_{th}}\ (f^1\,x_0)\ (f^2\,x_0)$. $\mathbb{G}_2^?$ is obtained by trivial computations over `nat` using the `f_equal`/`omega` tactics.

The Coq command `Recursive Extraction tortoise_hare` produces the corresponding OCaml code of Section 2.1 except that the OCaml type `int` is replaced with `nat` and the OCaml built-in equality decider is replaced with (a to be provided implementation of) $=_X^?$.

4.2 A tail-recursive implementation

Now we proceed with the tail-recursive implementation of T&H. We define a ternary bar inductive predicate corresponding to the recursive call of the `loop` in the OCaml code of `tort_hare_tail` in Section 2.1:

```
Inductive bar_tl (i : nat) (x y : X) : Prop :=
  | in_bar_tl_0 : x = y                          → bar_tl i x y
  | in_bar_tl_1 : bar_tl (S i) (f x) (f (f y))   → bar_tl i x y
```

the corresponding inductive rules being described in Fig. 2 (right part). Then we can define the internal `loop` of `tort_hare_tail` by a (local) fixpoint over

the fourth argument of type $\texttt{bar}_{\texttt{tl}}\ i\ x\ y$:

$$\begin{array}{l}\texttt{Fixpoint loop}\ i\ x\ y\ (H : \texttt{bar}_{\texttt{tl}}\ i\ x\ y) : \{k \mid i \leqslant k \wedge f^{k-i}\, x = f^{2(k-i)}\, y\} := \\ \quad \texttt{match}\ x =^{?}_{X} y\ \texttt{with} \\ \quad\quad \mid \texttt{left}\ E \ \mapsto \texttt{exist}\ \_\ i\ \mathbb{G}^{?}_{1} \\ \quad\quad \mid \texttt{right}\ C \mapsto \texttt{match loop}\ (\texttt{S}\, i)\ (f\, x)\ (f\, (f\, y))\ \mathbb{G}^{?}_{2}\ \texttt{with} \\ \quad\quad\quad\quad\quad \mid \texttt{exist}\ \_\ k\ H_k \mapsto \texttt{exist}\ \_\ k\ \mathbb{G}^{?}_{3} \\ \quad\quad\quad\quad \texttt{end} \\ \quad \texttt{end}.\end{array}$$

where $\mathbb{G}^{?}_{1}$, $\mathbb{G}^{?}_{2}$ and $\mathbb{G}^{?}_{3}$ are three proof terms of the following types:

$$\begin{array}{lll}\mathbb{G}^{?}_{1} & // & \ldots, E : x = y \vdash i \leqslant i \wedge f^{i-i}\ x = f^{2(i-i)}\ y \\ \mathbb{G}^{?}_{2} & // & \ldots, C : x \neq y, H : \texttt{bar}_{\texttt{tl}}\ i\ x\ y \vdash \texttt{bar}_{\texttt{tl}}\ (\texttt{S}\, i)\ (f\, x)\ (f\, (f\, y)) \\ \mathbb{G}^{?}_{3} & // & \ldots, H_k : \texttt{S}\, i \leqslant k \wedge f^{(k-\texttt{S}\, i)}\, (f\, x) = f^{2(k-\texttt{S}\, i)}\, (f^2\, y) \vdash i \leqslant k \wedge f^{k-i}\ x = f^{2(k-i)}\ y\end{array}$$

and $\mathbb{G}^{?}_{2}$ is the termination certificate and must be a sub-term of H. Then we proceed with the implementation of `tortoise_hare_tail` which calls `loop`:

$$\begin{array}{l}\texttt{Definition tortoise\_hare\_tail} : \{\tau \mid 0 < \tau \wedge f^{\tau}\, x_0 = f^{2\tau}\, x_0\} := \\ \quad \texttt{match loop}\ 1\ (f\, x_0)\ (f\, (f\, x_0))\ \mathbb{G}^{?}_{1}\ \texttt{with} \\ \quad\quad \mid \texttt{exist}\ \_\ k\ H_k \mapsto \texttt{exist}\ \_\ k\ \mathbb{G}^{?}_{2} \\ \quad \texttt{end}.\end{array}$$

We must provide a termination certificate $\mathbb{G}^{?}_{1}$ and establish the specification $\mathbb{G}^{?}_{2}$:

$$\begin{array}{lll}\mathbb{G}^{?}_{1} & // & \ldots, H_0 : \exists \tau, 0 < \tau \wedge f^{\tau}\, x_0 = f^{2\tau}\, x_0 \vdash \texttt{bar}_{\texttt{tl}}\ 1\ (f\, x_0)\ (f\, (f\, x_0)) \\ \mathbb{G}^{?}_{2} & // & \ldots, H_k : 1 \leqslant k \wedge f^{k-1}\, (f\, x_0) = f^{2(k-1)}\, (f\, (f\, x_0)) \vdash 0 < k \wedge f^{k}\ x_0 = f^{2k}\ x_0\end{array}$$

$\mathbb{G}^{?}_{1}$ is proved by descending induction much like what is done in the non-tail recursive case and $\mathbb{G}^{?}_{2}$ is quite trivial to obtain using the `f_equal`/`omega` tactics.

The extracted OCaml code corresponds to the `tortoise_hare_tail` implementation of Section 2.1. The file th_alone.v contains a standalone implementation of `tortoise_hare_tail` in less than 80 lines, not counting comments.

5 Floyd's Cycle Finding Algorithm in Coq

In this section, we give an overview of Floyd's index and period finding algorithm as implemented in the file floyd.v. It has the same pre-conditions as the T&H algorithms of Section 4:

$$\begin{array}{ll}\texttt{Variables} & (X : \texttt{Type})\ (=^{?}_{X} : \forall x\, y : X, \{x = y\} + \{x \neq y\}) \\ & (f : X \to X)\ (x_0 : X)\ (H_0 : \exists \tau,\, 0 < \tau \wedge f^{\tau}\, x_0 = f^{2\tau}\, x_0).\end{array}$$

It does not only finds a meeting point for the tortoise and the hare but computes the characteristic pair of values (λ, μ) of the cycle which satisfy the predicate `cycle_spec` with the following body:

$$\begin{array}{l}\texttt{Definition cycle\_spec}\ (\lambda\ \mu : \texttt{nat}) : \texttt{Prop} := \\ 0 < \mu \wedge f^{\lambda}\, x_0 = f^{\lambda+\mu}\, x_0 \wedge \forall i\, j,\ i < j \to f^{i}\, x_0 = f^{j}\, x_0 \to \lambda \leqslant i \wedge \mu\ \texttt{div}\ (j - i).\end{array}$$

where `div` represents the *divisibility order* over `nat` (i.e. d `div` n means $\exists q, n = qd$). The *index* λ is such that $f^\lambda(x_0)$ is the entry point of the cycle and $\mu > 0$ is the *period* (or length) of the cycle. The third conjunct $(\forall i\, j,\, i < j \to \ldots)$ states that any (non-empty) cycle $i \rightsquigarrow j$ occurs after λ and has a length divisible by μ.

Hence under the above pre-conditions, Floyd's algorithm has the functional specification `floyd_find_cycle` : $\{\lambda : \texttt{nat}\ \&\ \{\mu : \texttt{nat} \mid \texttt{cycle\_spec}\ \lambda\ \mu\}\}$. The operational specification is simply that the Coq term extracts to a standard OCaml implementation derived from [1]. `floyd_find_cycle` is implemented as the combination of three sub-terms, `floyd_meeting_pt` which first computes a meeting point for the tortoise and the hare, then `floyd_index` that computes the index and finally `floyd_period` that computes the period. We describe these three sub-terms in specific sub-sections, each sub-section potentially having its own set of additional pre-conditions, mimicking Coq sectioning mechanism. For each of these terms, we use a tailored `bar` inductive predicate to ensure termination under the corresponding pre-conditions.

5.1 Computing a meeting point

The term `floyd_meeting_pt` needs no further pre-conditions. We use the inductive $\texttt{bar}_{\texttt{th}} : X \to X \to \texttt{Prop}$ of Fig. 2 as termination certificate, the same that we used for the non-tail recursive `tortoise_hare` implementation. However, we do get a tail-recursive term here because we only compute a meeting point, not its index in the sequence as in the $\texttt{bar}_{\texttt{tl}}$/`tortoise_hare_tail` case.

$\texttt{Let bar\_th\_meet} : \forall x\ y,\ \texttt{bar}_{\texttt{th}}\ x\ y \to \{c : X \mid \exists k,\, c = f^k\, x \wedge c = f^{2k}\, y\}.$
$\texttt{Definition floyd\_meeting\_pt} : \{c \mid \exists \tau,\, 0 < \tau \wedge c = f^\tau\, x_0 \wedge c = f^{2\tau}\, x_0\}.$

We define `bar_th_meet` as a local fixpoint using the same technique as in the $\texttt{bar}_{\texttt{th}}$/`tortoise_hare` case. Then, we show $H'_0 : \texttt{bar}_{\texttt{th}}\ (f\, x_0)\ (f\, (f\, x_0))$ as a consequence of H_0 and we derive `floyd_meeting_pt` from the following instance `bar_th_meet` $(f\, x_0)\ (f\, (f\, x_0))\ H'_0$.

5.2 Computing the index

The term `floyd_index` uses the post-condition of `floyd_meeting_pt` as a further pre-condition, i.e. a meeting point c for the tortoise and the hare. We use the predicate $\texttt{bar}_{\texttt{in}} : \texttt{nat} \to X \to X \to \texttt{Prop}$ of Fig. 3 as termination certificate.

$\texttt{Variables}\ (c : X)\ (H_c : \exists \tau,\, 0 < \tau \wedge c = f^\tau\, x_0 \wedge c = f^{2\tau}\, x_0).$
$\texttt{Let bar\_in\_inv}\ (i : \texttt{nat})\ (x\ y : X) :$
$\qquad \texttt{bar}_{\texttt{in}}\ i\ x\ y \to \texttt{least\_le}\ (\texttt{fun}\ n \mapsto i \leqslant n \wedge f^{n-i}\, x = f^{n-i}\, y).$
$\texttt{Definition floyd\_index} : \texttt{least\_le}\,(\texttt{fun}\ l \mapsto \exists k,\, 0 < k \wedge f^l\, x_0 = f^{k+l}\, x_0).$

We define `bar_in_inv` as a local fixpoint where `least_le` P is the least[6] $n : \texttt{nat}$ which satisfies $P\, n$. We show $H'_c : \texttt{bar}_{\texttt{in}}\ 0\ x_0\ c$ as a consequence of H_c and we derive `floyd_index` from the following instance `bar_in_inv` $0\ x_0\ c\ H'_c$.

[6] least for the natural order $\leqslant$ over `nat`.

$$\frac{x = y}{\texttt{bar}_{\texttt{in}}\ i\ x\ y} \qquad \frac{\texttt{bar}_{\texttt{in}}\ (\texttt{S}\, i)\ (f\, x)\ (f\, y)}{\texttt{bar}_{\texttt{in}}\ i\ x\ y} \quad \Bigg| \quad \frac{c = y}{\texttt{bar}_{\texttt{pe}}\ i\ y} \qquad \frac{\texttt{bar}_{\texttt{pe}}\ (\texttt{S}\, i)\ (f\, y)}{\texttt{bar}_{\texttt{pe}}\ i\ y}$$

Fig. 3. Inductive rules for $\texttt{bar}_{\texttt{in}} : \texttt{nat} \to X \to X \to \texttt{Prop}$ and $\texttt{bar}_{\texttt{pe}} : \texttt{nat} \to X \to \texttt{Prop}$.

5.3 Computing the period

The further pre-condition of `floyd_period` is a point c which belongs to a (non-empty) cycle, a direct consequence of the post-condition of `floyd_meeting_pt`. Termination is certified by the predicate $\texttt{bar}_{\texttt{pe}} : \texttt{nat} \to X \to \texttt{Prop}$ of Fig. 3.

$\texttt{Variables}\ \ (c : X)\ (H_c : \exists k,\, 0 < k \wedge c = f^k\, c).$
$\texttt{Let}\ \ \texttt{bar\_pe\_inv}\ i\ x : \texttt{bar}_{\texttt{pe}}\ i\ x \to \texttt{least\_le}\ (\texttt{fun}\ n \mapsto i \leqslant n \wedge x = f^{n-i}\, y).$
$\texttt{Definition}\ \ \texttt{floyd\_period} : \texttt{least\_div}\ (\texttt{fun}\ n \mapsto 0 < n \wedge c = f^n\, c).$

We define `bar_pe_inv` as a local fixpoint. We prove $H'_c : \texttt{bar}_{\texttt{pe}}\ 1\ (f\, c)$ using H_c and we get `floyd_period` from the following instance $\texttt{bar\_pe\_inv}\ 1\ (f\, c)\ H'_c$. Here, $\texttt{least\_div}\ P$ is the least n s.t. $P\ n$ for the divisibility order `div`.

5.4 Gluing all together

We finish with the term `floyd_find_cycle`:

$\texttt{Definition}\ \ \texttt{floyd\_find\_cycle} : \{\lambda : \texttt{nat}\ \&\ \{\mu : \texttt{nat} \mid \texttt{cycle\_spec}\ \lambda\ \mu\}\}.$

It needs no further pre-conditions than those given at the beginning Section 5. It is composed of the successive applications of `floyd_meeting_pt`, `floyd_index` and `floyd_period` where the post-condition of `floyd_meeting_pt` serves as input for the extra pre-conditions of `floyd_index` and `floyd_period`. We conclude with a short proof that the computed index and period satisfy `cycle_spec`. The extracted OCaml program corresponds to the following code with the same remarks regarding $=^?_X$ and `nat`/`int` as with `tortoise_hare` from Section 4.1.

```
let floyd_meeting_pt f x0 =
  let rec loop x y = if x = y then x else loop (f x) (f^2 y) in loop (f x0) (f^2 x0)
let floyd_index f x0 c =
  let rec loop i x y = if x = y then i else loop (1 + i) (f x) (f y) in loop 0 x0 c
let floyd_period f c =
  let rec loop i y = if c = y then i else loop (1 + i) (f y) in loop 1 (f c)
let floyd_find_cycle f x0 =
  let c = floyd_meeting_pt f x0 in (floyd_index f x0 c, floyd_period f c)
```

6 Brent's Period Finding Algorithm

In the file brent_bin.v, we propose a correctness proof of Brent's algorithm in the same spirit as what was done for Floyd's cycle finding algorithm of Section 5.

Brent's algorithm [6] only computes the period μ of the cycle. The index λ can be computed afterwards by using two tortoises separated by μ steps. Brent's algorithm is more efficient than T&H: it can be proved that a run of Brent's algorithm on (f, x_0) always generates less calls to f than a run of the T&H (or Floyd's cycle finding) algorithm on the same input (see [14], Sect. 7.1.2).

In this section, we just describe the functional specification and the operational specification (i.e. the extracted OCaml code) of Brent's algorithm of which we propose two implementations. The first one in file brent_bin.v is suited for a binary representation of natural numbers, but it is not efficient with unary natural numbers. The other implementation in file brent_una.v is also efficient on unary natural numbers such as those of type `nat`.

Given a type X : `Type`, an equality decider $=^?_X : \forall x\, y : X, \{x = y\} + \{x \neq y\}$, input values $f : X \to X$ and $x_0 : X$, and a cycle existence certificate (see Proposition 1), Brent's algorithm computes a μ satisfying the specification `period_spec` : $\forall \mu : \texttt{nat}, \texttt{Prop}$ with the following body

$$0 < \mu \wedge (\exists \lambda, f^{\lambda}\, x_0 = f^{\lambda+\mu}\, x_0) \wedge \forall i\, j, i < j \to f^i\, x_0 = f^j\, x_0 \to \mu \texttt{ div } (j - i)$$

that is, it computes the period of the cycle. The term `brent_bin` extracts to something close to the following OCaml code:

```
let brent_bin f x0 =
  let rec loop p l x y =
    if      x = y then l
    else if p = l then loop 2p 1 y (f y)
    else               loop p (1 + l) x (f y)
  in loop 1 1 x0 (f x0)
```

where `int` is replaced with `nat`, $(x = y)$ with $(x =^?_X y)$ and $(p = l)$ with (`eq_nat_dec` $x\, y$). However this code is not optimal with a unary representation of numbers such as `nat`: in particular $2p$ and $p = l$ are slow (linear) to compute.

To get a more efficient implementation, one should either use a binary representation of numbers or switch to `brent_una` which has the same specification as the binary version but extracts to the following OCaml code:

```
let brent_una f x0 =
  let rec loop l p m x y =
    if      x = y then l
    else if m = 0 then loop 1 (1 + p) p y (f y)
    else               loop (1 + l) (1 + p) (m - 1) x (f y)
  in loop 1 1 0 x0 (f x0)
```

This code is much better suited for unary numbers. In particular, $m = 0$ and $m - 1$ are computed via pattern-matching on m in constant time.

7 Correctness by Extraction and Related Works

Correctness is a property of programs *with respect to a given specification*. As trivial as this remark may seem, it is important to keep it in mind because the

purpose of extraction is to erase the logical content of Coq programs and to keep only their computational content: *specifications are erased by extraction.* One cannot claim that a program `extract`(t) is correct just because it has been extracted from a Coq term $t : T$. The correctness property is only ensured with respect to the particular specification T that (by the way) had just been erased.

7.1 Correctness of the Tortoise and the Hare

We illustrate this critical aspect of extraction on the non-tail recursive OCaml implementation of T&H. The type of `tortoise_hare` is

$$\texttt{Definition tortoise\_hare } \{X\}\ (=^?_X : \forall x\ y : X, \{x = y\} + \{x \neq y\})\ f\ x_0 :$$
$$(H_0 : \exists \tau, 0 < \tau \wedge f^\tau\, x_0 = f^{2\tau}\, x_0) \to \{\tau : \texttt{nat} \mid 0 < \tau \wedge f^\tau\, x_0 = f^{2\tau}\, x_0\}.$$

as reported from Section 4. The pre-conditions of this specification are all the logical properties of the input parameters, i.e. the fact that $=^?_X$ is an equality decider for X and the cyclicity property H_0. The post-condition is the fact that τ is a meeting index for the fabulous animals. By extraction, the OCaml code `tortoise_hare` of Section 2.1 is correct w.r.t. to this specification. In the extracted program however, the pre-conditions on $=^?_X$ and of cyclicity, and the post-conditions $0 < \tau$ and $f^\tau\, x_0 = f^{2\tau}\, x_0$ have disappeared.

Now consider this "alternative" (and cheating) implementation of T&H:

```
Variables (X : Type) (=?X : ...) (f : X → X) (x0 : X) (H0 : False).
Fixpoint th_false_rec x y (H : False) : nat :=
  match x =?X y with
    | left E  ↦ 0
    | right C ↦ S (th_false_rec (f x) (f (f y)) match H with end)
  end.
Definition th_false := S (th_false_rec (f x0) (f (f x0)) H0).
```

The pre-conditions for `th_false` are the same as those of `tortoise_hare` except that cyclicity has been replaced with absurdity (H_0 : `False`). The post-condition has been erased. Yet, up to renaming, extraction of OCaml code from `th_false` and from `tortoise_hare` yields the *very same program.* So the OCaml program `tortoise_hare`/`th_false` is correct w.r.t. two very different specifications: one is useful (cyclicity) and one is useless (absurdity).

The file infinite_loop.ml contains `th_false` and gives other illustrations of abuses of extraction in absurd contexts. As a conclusion, before using or running an extracted algorithm, one should first check for assumptions in the specification using the Coq command `Print Assumptions` which lists all the potentially hidden pre-conditions such as axioms or parameters which are usually not displayed in the types of terms in Coq to avoid bloating them.

7.2 Comparison with Related Works

Given the stunning simplicity of T&H, our implementation is hardly the first attempt at certifying this algorithm. As for Coq, may be there are others, but

we are aware of two previous developments. One (unpublished) proof by J.C. Filliâtre [11] and another, more recent, by J.F. Dufourd [9].

Illustrating how Hilbert's ϵ-operator could be used to manage partial functions within Coq was the main goal of his implementation [11] (private communication with J.C. Filliâtre); and the correctness of T&H was not really a goal of that project. The use of `epsilon`/`epsilon_spec` to *axiomatize* Hilbert's ϵ-operator is, from a constructive point of view, our main criticism against that implementation. Hence, while it is true that the term `find_cycle` of fillia_orig.v extracts to some OCaml code very similar to `tortoise_hare_tail` of Section 2.1, the corresponding specification has stronger pre-conditions than our own. Following the observations on correctness of Section 7.1, even if erased by extraction, Hilbert's ϵ-operator is still a pre-condition and a particularly strong form of the axiom of choice. Anything that exists can be reified with this operator. While not necessarily contradictory in itself, that kind of axiom is incompatible with several extensions of Coq [7]. We do not think it can be accepted from a constructive point of view because admitting the ϵ-operator allows to write non-recursive functions in Coq. We suggest the interested reader to consult the file collatz.v to see how the ϵ-operator "solves" the Halting problem or the Collatz problem [16]. It is our claim that although `find_cycle` implements some correctness property of T&H, the pre-conditions under which this correctness is achieved cannot be constructively or computationally satisfied. In the file fillia_modif.v, we propose a modified version of [11] where the ϵ-operator is replaced with `Constructive_Epsilon` from Section 3.3. This requires in-depth changes, in particular, on the induction principle used to ensure termination. We additionally mention a more recent Isabelle/HOL proof of P. Gammie [13] which seems to reuse the technique of J.C. Filliâtre.

The work of J.F. Dufourd [9] is based on different assumptions and the correctness proof of T&H that he obtains derives from a quite large library on functional orbits over finite domains of around 20 000 lines of code. The pre-conditions do not assume cyclicity. Instead there is a stronger assumption of finiteness of the domain, from which cyclicity can be derived using the pigeon hole principle (PHP). Admittedly, this finiteness assumption is not unreasonable, even constructively: most use cases of cycle finding algorithms occur over finite domains. However, it is our understanding that Pollard's rho algorithm [18] is run on a finite domain of *unkown (i.e. non-informative) cardinality*. As far as we can understand J.F. Dufourd's code, his inductive proofs are cardinality based. Hence an informative bound on the cardinal of the domain is likely a pre-condition for correctness. Thus, we think that his correctness proof might not be applicable to Pollard's rho algorithm (non-informative finiteness, see below).

On the other hand, modifying our specification of `tortoise_hare_tail` to replace cyclicity by finiteness involves the PHP (see php.v). In that case, finiteness could be expressed as the predicate $(\exists l : \texttt{list}\, X, \forall x : X, \texttt{In}\ x\ l)$ which postulates the *non-informative existence* of a list covering all the type X. Hence we would obtain a specification compatible with the context of Pollard's rho algorithm. This short development can be found in th_finite.v.

References

1. Cycle detection — Wikipedia, The Free Encyclopedia
2. Aumasson, J.P., Meier, W., Phan, R., Henzen, L.: The Hash Function BLAKE. Springer Publishing Company, Incorporated (2014)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
4. Bove, A., Capretta, V.: Modelling general recursion in type theory. Math. Structures Comput. Sci. **15**(4), 671–708 (2005)
5. Bove, A., Krauss, A., Sozeau, M.: Partiality and Recursion in Interactive Theorem Provers – An Overview. Math. Structures Comput. Sci. **26**(1), 38–88 (2016)
6. Brent, R.P.: An improved Monte Carlo factorization algorithm. BIT Numerical Mathematics **20**(2), 176–184 (Jun 1980)
7. Castéran, P.: Utilisation en Coq de l'opérateur de description (2007), `http://jfla.inria.fr/2007/actes/PDF/03_casteran.pdf`
8. Coen, C.S., Valentini, S.: General Recursion and Formal Topology. In: PAR-10, Partiality and Recursion in Interactive Theorem Provers. EPiC Series in Computing, vol. 5, pp. 72–83. EasyChair (2012)
9. Dufourd, J.F.: Formal study of functional orbits in finite domains. Theoret. Comput. Sci. **564**, 63–88 (2015)
10. Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory. J. Symb. Log. **65**(2), 525–549 (2000)
11. Filliâtre, J.C.: Tortoise and the hare algorithm (2007), `https://github.com/coq-contribs/tortoise-hare-algorithm`
12. Fridlender, D.: An Interpretation of the Fan Theorem in Type Theory. In: TYPES'98 Selected Papers. Lecture Notes in Comput. Sci., vol. 1657, pp. 93–105. Springer (1999)
13. Gammie, P.: The Tortoise and Hare Algorithm (2015), `https://www.isa-afp.org/entries/TortoiseHare.html`
14. Joux, A.: Algorithmic Cryptanalysis. Cryptography and Network Security, Chapman & Hall/CRC (2009)
15. Knuth, D.E.: The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
16. Lagarias, J.: The Ultimate Challenge: The $3x+1$ Problem. American Mathematical Society (2010)
17. Larchey-Wendling, D., Monin, J.F.: Simulating Induction-Recursion for Partial Algorithms. In: TYPES 2018, Braga, Portugal (2018)
18. Pollard, J.M.: A monte carlo method for factorization. BIT Numerical Mathematics **15**(3), 331–334 (Sep 1975)