UTC Time, Formally Verified

Ana de Almeida Borges
University of Barcelona
Barcelona, Spain
Formal Vindications S.L.
Barcelona, Spain
ana.agvb@gmail.com

Mireia González Bedmar
University of Barcelona
Barcelona, Spain
Formal Vindications S.L.
Barcelona, Spain
mireia.gbedmar@formalv.com

Juan Conejero Rodríguez
University of Barcelona
Barcelona, Spain
Formal Vindications S.L.
Barcelona, Spain
jjcoro@proton.me

Eduardo Hermo Reyes
University of Barcelona
Barcelona, Spain
Formal Vindications S.L.
Barcelona, Spain
ehermo.reyes@formalv.com

Joaquim Casals Buñuel
University of Barcelona
Barcelona, Spain
Formal Vindications S.L.
Barcelona, Spain
jcasalsb@formalv.com

Joost J. Joosten
University of Barcelona
Barcelona, Spain
jjoosten@ub.edu

Abstract

FV Time is a small-scale verification project developed in the Coq proof assistant using the Mathematical Components libraries. It is a library for managing conversions between time formats (UTC and timestamps), as well as commonly used functions for time arithmetic. As a library for time conversions, its novelty is the implementation of leap seconds, which are part of the UTC standard but usually not implemented in existing libraries. Since the verified functions of FV Time are reasonably simple yet non-trivial, it nicely illustrates our methodology for verifying software with Coq.

In this paper we present a description of the project, emphasizing the main problems faced while developing the library, as well as some general-purpose solutions that were produced as by-products and may be used in other verification projects. These include a refinement package between proof-oriented MathComp numbers and computation-oriented primitive numbers from the Coq standard library, as well as a set of tactics to automatically prove certain decidable statements over finite ranges through brute-force computation.

CCS Concepts: • **Applied computing** → Law; • **Software and its engineering** → **Formal software verification**; *Software libraries and repositories.*

CPP '24, January 15–16, 2024, London, UK

ACM ISBN 979-8-4007-0488-8/24/01
https://doi.org/10.1145/3636501.3636958

Keywords: Coq, MathComp, formal verification, automation, time, UTC

ACM Reference Format:
Ana de Almeida Borges, Mireia González Bedmar, Juan Conejero Rodríguez, Eduardo Hermo Reyes, Joaquim Casals Buñuel, and Joost J. Joosten. 2024. UTC Time, Formally Verified. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '24), January 15–16, 2024, London, UK*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3636501.3636958

1 Introduction

Coordinated Universal Time (UTC) [42] is the current world standard for keeping time. Although it uses atomic time, it is designed to stay close to solar time, and as such it includes leap seconds. The number of seconds in a minute can be either 59 (if there is a negative leap second), 60 (the regular case), or 61 (if there is a positive leap second). The need for a new leap second is somewhat unpredictable, so the International Earth Rotation and Reference Systems Service announces whether there will be one about six months in advance. The convention is to have at most two leap seconds per year, as the final second of the last day of a month, preferably June or December. As of 2023, there have been 27 positive leap seconds and no negative ones [41], although a recent resolution [18] aims to eliminate future leap seconds, prompted by the various issues and inconveniences they lead to [48].

The vast majority of software uses Unix time [5, 53], which is an implementation of UTC without leap seconds [40]. This is fine for many use cases, and understandable given the unpredictability of UTC for future moments. However, it conflicts with legal regulations that explicitly require UTC. It may seem like 27 seconds are not enough to meaningfully change anything, but in fact even 27 seconds can make a difference in real world legal applications, as well as in critical systems. In particular, in the context of software that interprets and evaluates the log information for driving time

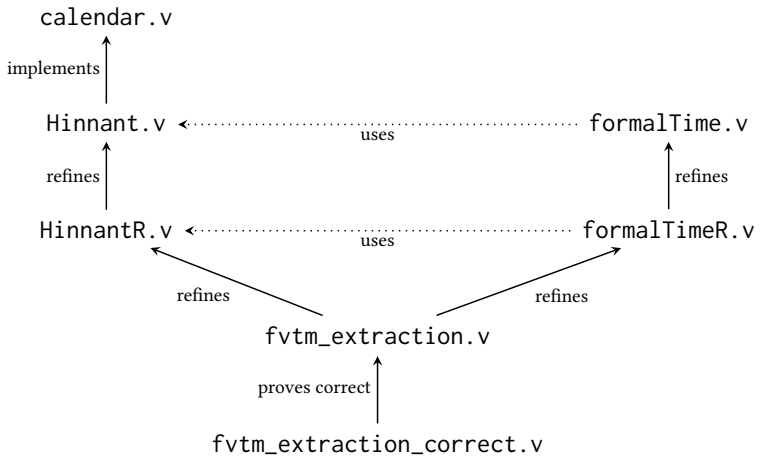

Figure 1. The file structure of FV Time. There is an extra file, `doe_of_yoeK.v`, that only contains auxiliary proofs and definitions and doesn't appear in the diagram.

in the road transport sector according to Regulation (EU) 2016/799 [28], the algorithm that translates from second-resolution data to minute-resolution data as required by the Regulation can give opposite results depending on whether UTC or Unix is used – meaning that there exists a possible data file that gets interpreted as 100% of driving time in UTC and as 0% of driving time in Unix [3].

It was in this context that FV Time [2] was developed. It is a by-product of the collaboration between the University of Barcelona and Formal Vindications S.L., whose main goal is the development of large-scale formally verified software with applications in several critical sectors. FV Time is a small-scale verification project developed in Coq and relying in the MathComp library. It includes conversions between time, represented as a 7-tuple of year, month, day, hour, minute, second, and proof of existence in the chosen paradigm (to avoid ill-formed tuples such as the ones including February 30), and timestamps, represented as the number of seconds since a chosen epoch (for example, year 0, or year 1970, which is the Unix epoch). We also define dates and datestamps, which are the respective concepts without information on the hour, minute, and second. Functions for time and duration arithmetic are provided as well. Leap seconds are tracked via a modifiable parameter, which can be empty for Unix time or updated as appropriate to keep pace with UTC.

This paper reports on the development of FV Time from high-level specifications to executable code integrated with other software. It can serve as a roadmap for other similar verification projects. Section 2 describes the main functions that were verified and our methodology to structure the project, with an emphasis on the issues and solutions that can be of general interest.

We also describe two general purpose Coq libraries: FV Prim63 to MathComp (Section 3) and FV Check Range (Section 4). These libraries were developed as aids to FV Time, but they would be useful in other contexts as well. FV Prim63 to MathComp is a collection of results linking Coq primitive integers to MathComp natural numbers and integers, which are needed when refining primitive integers to those MathComp types. FV Check Range is a small set of automation tactics that solve (provable) goals of the form "for every primitive integer x in the range $a \leq x < b$, we have $f(x)$", where f is a boolean function and a and b are fixed primitive integers. Although these kinds of goals can sometimes be automatically solved using preexisting tactics, our approach simply tests every value of x between a and b, which works regardless of the boolean function f and is quite fast.

We briefly outline our method of obtaining clean extracted code in Section 5. The resulting OCaml code was bundled with a command-line interface called FVTM (Section 6), which can be used by other applications. This makes our time translations available outside the relatively small world of Coq and OCaml.

Finally, Section 7 gives an overview of the existing related work, while Section 8 lists the contributions and conclusions of this project.

2 FV Time

2.1 File Structure

The main goal of FV Time is to provide verified functions translating between UTC times (with leap seconds) and timestamps. We describe the file structure of the library in Figure 1.

As we see throughout this section, the `calendar.v` file describes the main datatypes, such as what it means to be

a UTC time.[1] It also specifies the expected behavior of the translating functions in an intuitive way. We provide a second file, `Hinnant.v`, with alternative implementations of these translating functions. The implementations of the datestamp algorithm and its inverse are inspired by the ones described by Howard Hinnant [39], hence the name of the file.

FV Time also provides functions to perform basic arithmetic on UTC times, such as adding a certain number of hours to a given time, in `formalTime.v`.

Since these algorithms are meant to be extracted from Coq to OCaml for efficient execution, we provide a type refinement for each in the `HinnantR.v` and `formalTimeR.v` files. In other words, there are two versions of each algorithm: one based on proof-friendly datatypes, and one based on extraction and computation-friendly ones. These two versions are proven equivalent under some assumptions.

Finally, the two extraction files `fvtm_extraction.v` and `fvtm_extraction_correct.v` follow the extraction method explained in Section 5.

2.2 Main Data Types

The central data type in FV Time is a representation of moments in time in UTC,[2] which we call `time`. Under the hood it is simply a 6-tuple of natural numbers representing a given year, month, day, hour, minute, and second, together with a proof that the tuple in question forms an existing time. What counts as an existing time depends on the parametrized list of leap seconds.

For convenience and modularity's sake, we define three other relevant types: `date`, `rawDate`, and `rawTime`. A `date` is the part of the `time` with only the year, month, and day, together with a proof that it exists in UTC. The `raw` types are simply the tuples without the proofs. Thus, January 32nd 2000 could be represented as a `rawDate` but not a `date`.

We encode the list of leap seconds as a parameter that can be updated each time a new leap second is announced. The list is actually a list of pairs, where each pair has a date (indicating that a leap second occurs on that date) and a boolean value (where `false` means that it's a positive leap second and `true` that it's a negative one). Since we treat the list as a parameter with unknown contents, it can be instantiated in any way as long as it satisfies the hypotheses we use throughout the theorems: the list must be sorted with respect to the strict order of dates (in particular it doesn't include repeated dates), and all the dates in it must be valid.

[1]Throughout the library and paper, "time" refers to our specification of a UTC point expressed as a date-time, i.e., a 7-tuple of year, month, day, hour, minute, second, and proof of validity. We currently take the second to be the smallest unit, although a finer-grained resolution could be implemented as well. This is described in more detail in Section 2.2.

[2]We further assume that the time occurs before the end of year 9999, for reasons explained in Section 2.3.1.

In particular, FV Time can be used to compute Unix time conversions by using an empty list of leap seconds.

The `raw` types are used in the implementations of every function that operates on dates, such as `datestamp`. It is then possible to compute the datestamp of January 32nd 2000, but we do not wish to prove any facts about the datestamps of such ill-formed dates. For that reason, we use the valid (non-`raw`) versions in the specifications and theorem statements. There is then a disconnect between the specification and the implementation, since they refer to different types. This is easily solved using coercions, i.e., automatically inserted translations between one type and another.

We use a number of coercions in our development, mostly between types and their subtypes, as described in Figure 2. We have a very small type hierarchy. Formalizations of, say, mathematical algebra or large libraries such as MathComp include rich hierarchies [54], and there are existing tools to implement and maintain such large hierarchies such as Hierarchy Builder [17].

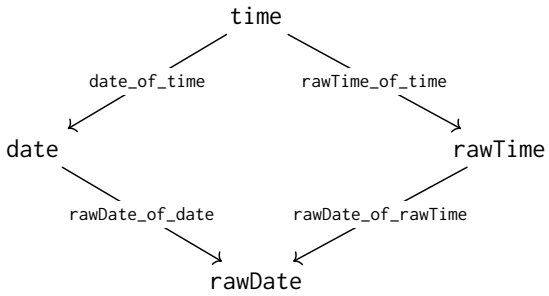

Figure 2. A representation of the four main data types in FV Time and the coercions between them. Each arrow from `X` to `Y` represents the coercion `Y_of_X`.

Still, even with a small hierarchy we do run into some issues. For example, looking at Figure 2, we can see that there are two possible paths from a `time` to a `rawDate`. It happens that these paths are definitionally equivalent, and so in some contexts it is irrelevant which one is chosen. However, sometimes the information that the `date` part of the `time` is valid is crucial, and so the path that goes through the `date` must be picked over the other one. In particular, the following unification problem sometimes arises:

$$\texttt{rawDate\_of\_date}\ ?d = \texttt{rawDate\_of\_rawTime}\ (\texttt{rawTime\_of\_time}\ t) \tag{1}$$

In words, given a `time` t, a `date` $?d$ must be found such that its `rawDate` corresponds to the `rawDate` of the `rawTime` of the `time`. The diamond represented in Figure 2 trivially commutes, so we define the canonical coercion `date_of_time` as the path to solve (1) with $?d := \texttt{date\_of\_time}\ t$.

2.3 Main Functions

The backbone of FV Time is the translations between times and timestamps, which themselves depend on translations between dates and datestamps. In this section we focus on the specification and implementation of these four functions, as well as the proofs that they coincide on well-formed inputs. These terms are listed in Tables 1 and 2.

The relevant files are `calendar.v` and `Hinnant.v`. The former includes the basic definitions of dates and times, as well as all the specifications. The latter includes the efficient implementations and the correctness proofs of the main functions. Note that the specification and implementation of a given function have the same name, so we use the name of the file to clarify which we mean at any given time. Similarly, some lemma names coincide and are thus clarified as well.

2.3.1 Specifications. The specifications of the main functions were primarily chosen to be intuitive. Thus, the `calendar.datestamp` of a `date` d is the size of the set of dates strictly smaller than d, and similarly for `calendar.timestamp`, where the order relations on dates and times are defined as expected. These definitions use the notion of cardinality of a finite set, which is defined in MathComp's `fintype` library [49]. In order to benefit from this library's theory, our types for valid dates and times needed bounds so that they could be declared as a `finType`. While a minimum was already imposed by our definition using natural numbers (a year before 0 cannot be expressed), we arbitrarily set the maximum date as December 31st, 9999. Different end years could be substituted, although if they were large enough there might be problems with overflow during the refinement process (see Section 2.5).

The `calendar.datestamp` function goes from `date`, the type of valid dates, to `'I_max_datestamp.+1`, the type of natural numbers less than or equal to `max_datestamp`. Hence, on the specification side, the type of the function already ensures that the argument and the result are in the expected range.

To specify the inverse, we start by defining a notion of `next_date`, which goes as expected: add 1 to the day component if doing so results in a valid date; otherwise set the day to 1 and add 1 to the month component if doing so results in a valid date; otherwise set the day to 1, the month to January and add 1 to the year unless the year was 9999 (the maximum year), in which case set the year to 1. Its only particularity is that the successor of the maximum date is the minimum date. This cyclic behavior was chosen as a convenient, simple way to maintain the invariant that the successor of a valid date is always a valid date. We also define `next_time` to be cyclic, and we underline that this definition depends on the parametrized list of leap seconds, as do essentially all the functions related to time.

Given these notions of `next_date` and `next_time`, we define the `calendar.from_datestamp` of a number n as the nth iteration of `next_date` after the minimum date, and similarly for `calendar.from_timestamp`. The spirit of `calendar.from_datestamp` is to describe the counting process one would perform on a real calendar: it is defined as if, given a datestamp n, one counted the days one by one from the epoch up to n, and the last counted day is the result.

2.3.2 Implementations.

`Hinnant.datestamp`. If every month had the same number of days and every year the same number of months, computing the datestamp of a given date would be as straightforward as multiplying each date component by its corresponding number of days and adding everything. Even with different lengths for different months it would not be particularly complicated, but the existence of leap years means that some more care must be taken. In order to have the algorithm be as simple as possible, we internally use shifted years that start on the first day of March and end on the last day of February, as inspired by Hinnant [39]. Thus, the leap day, if it exists, is the last day of the shifted year and doesn't influence the calculation of the datestamp of any day in that year other than itself. Since the rule for leap years in the Gregorian calendar repeats itself over periods of 400 years, we also compute the era (period of 400 years) corresponding to the date.

The only other main part of `Hinnant.datestamp` is calculating how many days there are between the start of a (shifted) year and the first day of each (shifted) month. This can be represented as a table that assigns the appropriate value to each (shifted) month – for March (month 0) it is 0, for April (month 1) it is 31, for May (month 2) it is 61, and so on. However, it turns out that the linear equation $(153 \cdot m + 2)/5$, where m is the ordinal of the (shifted) month, interpolates this table, so we use that instead of storing the table in memory.

`Hinnant.from_datestamp`. As in the previous case, we divide years in 400 year eras and shift everything so that years start in March. With this framing, the algorithm is more obviously the inverse of `Hinnant.datestamp` (a fact that we rely on for the correctness proofs).

Given the number of days since the beginning of the era, finding the year must take into consideration leap years, and finding the month must take into consideration the varying number of days in each month. For the latter we use a linear interpolant of the table matching days in a year to months instead of using the table directly.

`Hinnant.timestamp`. This is a natural extension to time of `Hinnant.datestamp`. In the absence of leap seconds, we could simply add the product of each time component by the amount of seconds in that component (60 seconds per minute, and so on). With leap seconds, we need to additionally calculate the offset generated by them, i.e., the number of extra (positive or negative) seconds that must be added to

Table 1. Names and types of the main functions in FV Time.

	Specification calendar.v	Implementation Hinnant.v
date	datestamp : date → 'I_max_datestamp.+1 from_datestamp : nat → date	datestamp : rawDate → nat from_datestamp : nat → rawDate
time	timestamp : ∀(*ls* : leapSeconds), time *ls* → 'I_(max_timestamp *ls*).+1 from_timestamp : ∀(*ls* : leapSeconds), nat → time *ls*	timestamp : leapSeconds → rawTime → nat from_timestamp : leapSeconds → nat → rawTime

the leap second-less timestamp, and add it accordingly. Such an offset is relatively easy to calculate for a given date d, for we can simply count the number of dates in our list of leap seconds that happened prior to d, and then check whether they were positive or negative to obtain a final offset. This is accomplished by the offset_rd function.

Since the leap second offset can *a priori* be negative (even though there hasn't been a single negative leap second as of 2023), we first calculate the timestamp over the integers and then take its absolute value. This works because even before taking the absolute value we know that the timestamp is positive due to our restrictions on leap seconds: we allow at most one leap second per day (an unimportant restriction, since the international convention allows at most two leap seconds per year). Since there are less days than seconds in any given amount of time, it is not possible to have enough negative leap seconds to obtain a negative timestamp.

Hinnant.from_timestamp. Once we know how to calculate the date corresponding to some datestamp, calculating the time corresponding to some timestamp (i.e., to some number n of seconds since the epoch) is straightforward in the absence of leap seconds. Thus, we first subtract the relevant offset from n and then proceed as if there were no leap seconds.

Obtaining the offsets for this function is slightly more complicated than it was for Hinnant.timestamp, since our list of leap seconds is a list of dates and not of timestamps. Thus, the offset calculator for Hinnant.from_timestamp, called offset_ts, first computes the Hinnant.timestamp of the final second of each date in our list of leap seconds (leap seconds are always the final second of each day by international convention), and then proceeds similarly to the offset computation for Hinnant.timestamp (offset_rd).

2.3.3 Proofs. The specification and implementation of the main functions differ significantly, and so it is hard to directly prove that they match. Instead, we make use of lemmas showing that certain functions are the (left) inverse of others (also known as canceling lemmas) and the following simple result.

Remark 2.1. *Let T and U be types, and $f_1, f_2 : T \to U$ be functions. If there is a function $g : U \to T$ that is both a right inverse of f_1 and a left inverse of f_2, then f_1 and f_2 are extensionally equal.*

We summarize here the correctness proof for timestamp (Theorem 2.2) as an example. The actual Coq statement includes our standard assumptions on the shape of the list of leap seconds (see Section 2.2), omitted here. Note that the theorem statement is about valid times; we make no claim about non-existing times such as any moment during January 32nd. Since timestamp takes a rawTime as an argument, the implicit coercion rawTime_of_time (see Figure 2) is automatically inserted on the left-hand side of the equation. Links to the formalizations of the other proofs can be found in Table 2, which summarizes the results. Note that while the pre-conditions are explicit, the post-conditions are implied by the equality to the specifications.

Theorem 2.2 (timestampE)**.** *For every* time *t:*

Hinnant.timestamp t = calendar.timestamp t.

Proof. By Remark 2.1 it suffices to find a suitable function bridging the implementation and specification of timestamp. We used calendar.from_timestamp and the canceling lemmas calendar.timestampK and cal_from_timestampK (see Figure 3 for a schematic representation of their statements). □

Given the above strategy, the main challenge becomes proving the canceling lemmas. There are three relevant lemmas for time, depicted as the arrows in Figure 3, and three analogous ones for dates, used as stepping stones for the time ones and not shown here. We briefly comment on each of these three results.

On the specification side, lemma calendar.timestampK states that calendar.from_timestamp is a left inverse of calendar.timestamp. Since this is a statement about two specifications, designed to behave nicely with respect to proofs, there were no great difficulties in proving it.

The bridge between the specification and the implementation is provided by cal_from_timestampK, which states that calendar.from_timestamp is a right inverse of Hinnant.timestamp. Fortunately, calendar.from_timestamp is very simple (just iterating next_time), and so this proof follows without too much difficulty using basic arithmetical facts.

Table 2. Theorems stating that the implementations of the main functions meet the specifications.

	Correctness Hinnant.v

date	datestampE : ∀(*d* : date),
	  Hinnant.datestamp *d* = calendar.datestamp *d*
	from_datestampE : ∀(*n* : nat), *n* ≤ max_datestamp →
	  Hinnant.from_datestamp *n* = calendar.from_datestamp *n*

time	timestampE : ∀(*ls* : leapSeconds)(*t* : time *ls*),
	  sorted Order.lt (unzip1 *ls*) →
	  all valid_date (unzip1 *ls*) →
	  Hinnant.timestamp *ls t* = @calendar.timestamp *ls t*
	from_timestampE : ∀(*ls* : leapSeconds)(*n* : nat),
	  sorted Order.lt (unzip1 *ls*) →
	  all valid_date (unzip1 *ls*) →
	  n ≤ max_timestamp *ls* →
	  Hinnant.from_timestamp *ls n* = calendar.from_timestamp *ls n*

Finally, on the implementation side, the auxiliary lemma Hinnant.timestampK (needed for the proof of Hinnant.from_timestampE, which is the correctness theorem for Hinnant.from_timestamp) states that the function Hinnant.from_timestamp is a left inverse of Hinnant.timestamp. Its proof is the most intricate of the three if done with pen and paper, due to the ubiquitous presence of Euclidean division, which doesn't have an inverse. However, once we developed the automation tool FV Check Range (see Section 4), the proof was notably eased.

2.4 Time Arithmetic

When adding and subtracting durations to a given time, the irregular periods that the Gregorian calendar and UTC define must be taken into account. For systems that work in Unix, the issue arises with months and years, because they don't have a constant duration. What some systems do is define arithmetical operations on months and years that don't respect basic arithmetical properties [53]. For example, it's common to define the notion of adding 1 month as adding 1 to the month component of the time. However, the result of this operation is not always valid. Thus, adding 1 month in this sense to 2009-01-31 14:00:00 yields 2009-02-31 14:00:00, which is not a valid time because February doesn't have 31 days. The adopted solution is to correct the wrong component by going back to the previous valid one, so the result would be 2009-02-28 14:00:00. Similarly, it is possible to add any number to the month component, carrying to the year if necessary, and then correct the wrong component. For example, adding 24 months to 2008-02-29 15:00:00 gives 2010-02-28 15:00:00. This operation and the analogously defined subtraction are not mutual inverses, since subtracting 1 month from 2009-02-28 14:00:00 would lead to 2009-01-28 14:00:00, which is some days apart from the original 2009-01-31 14:00:00. These are used for practical purposes such as accounting, monthly interest calculation, or utility bills. However, for general computations on durations this behavior may be undesired.

Our library uses UTC, which means that this problem affects all the components except seconds. Not all minutes have the same duration, nor all hours, nor all days. Our solution is to implement two different types of operations in time arithmetic. The first approach, leading to the so-called shift functions, follows the above logic. The second one is a definition of a standard for durations called formal time, and operations called add_formal that manipulate fixed amounts of seconds and thus do not suffer from the above issue. Since both options are available, users are free to choose the best one for them.

2.4.1 Shift Functions.

The shift functions are defined according to the logic described above. Thus, the shift function shifts a component of the time, carrying to the left if necessary, and then if the result is invalid it performs corrections on the wrong component(s) to return a certain close valid time. The precise specification can be found in the lemma shift_utc_yearsP for years, and similarly for the other date-time components.

2.4.2 Formal Time Arithmetic Functions.

In order to have time arithmetic with the usual arithmetical properties, we have defined formal time, which establishes standard durations for every component. A formal second is an atomic second, a formal minute is 60 formal seconds, and so on.

We have chosen to define a formal month as 30 formal days. Therefore, add_formal_month adds a constant number of seconds ($30 \cdot 24 \cdot 60 \cdot 60$ seconds) to the input. In the example above, adding a formal month to 2009-01-31 14:00:00 yields

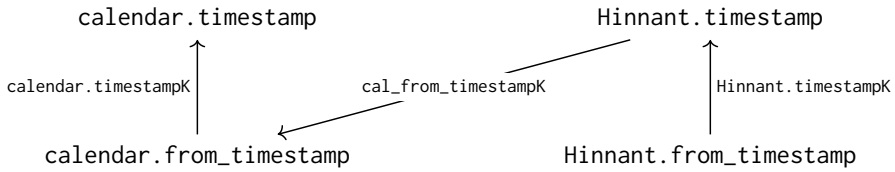

Figure 3. The implementation and specification of the main time functions together with the canceling lemmas used to prove their correctness. Each arrow from f to g represents the proof that f is a left inverse of g.

2009-03-02 14:00:00. The result is always valid by construction, except when it goes beyond the minimum or maximum dates. Subtraction works similarly.

2.5 Type Refinements

The default representation of the natural numbers in Coq (and MathComp) is the unary one, which uses roughly a symbol per unit. It is extremely useful for proving properties about the natural numbers, as one can reason by structural induction: prove first that a property holds for zero, and then that if it holds for n it also holds for its successor. However, it is a very inefficient representation in terms of space. For example, explicitly representing numbers larger than 5000 in Coq makes the code almost unusable. There are ways around this, namely to encode the natural numbers in binary, which is the standard in computer science. The downside is that proofs using natural numbers become more complicated.

The usual solution for this problem is to use what is known as a type refinement. This technique includes an intermediate step where algorithms are defined on top of non-efficient data types and then refined (i.e., redefined) on top of efficient ones. A proof can then be provided to ensure that the refinement kept the relevant properties of the algorithm. This approach has been known and used for quite some time [9].

Our particular approach is the following. For a hypothetical function f with input a natural number and output a natural number, we have a specification using the unary `nat`. Then we write a first implementation, which would be an efficient algorithm for f except it also uses `nat`. We prove a theorem stating that this first implementation behaves as the specification says. The previous steps are described in Section 2.3.

Then we provide a refinement. We define a second implementation of exactly the same algorithm, but this time using the unsigned primitive integers[3] `Uint63.int` and their operations, and taking the possibility of overflow into account.[4] Now we can prove another lemma stating that, whenever the input is small enough, the outputs of both versions coincide. It then follows that the latter implementation meets the specification for such inputs. Analogously, we refine the unary type `ssrint.int` into `Sint63.int`.

The choice of `Uint63.int` and `Sint63.int` was based both on their efficiency and their good properties with respect to extraction, as described in Section 5.

Even if at first sight the refinement phase may appear much easier than the previous one, in truth the difficulty in our case was comparable and arguably higher, for two different reasons. The first one is that the process of refining functions requires having a good set of rewriting lemmas between the operations of the source and target types, which in our case didn't exist. Hence, this project led to the development of such a set of lemmas, called FV Prim63 to MathComp (see Section 3), which is now available for any future projects. The other reason is the nature of the task itself: ensuring the equivalence between the original and the refined versions requires ensuring that all of the intermediate steps will not overflow, or otherwise finding the appropriate bounds for which they don't. This was in itself a very extensive and quite tedious task, eased by our automation tool FV Check Range (see Section 4).

3 FV Prim63 to MathComp

FV Prim63 to MathComp provides locked and unlocked conversions between the proof-oriented libraries `ssrnat` and `ssrint`, and between the computation-oriented libraries `Uint63` and `Sint63`. It also provides an extensive set of lemmas for rewriting between their respective arithmetical operations, with bounds on the numbers as side conditions when needed.

This tool is independent of FV Time, and can be installed and used from any development that decides to take the same refinement path. As explained in Section 2.5, our motivation was to link an abstract specification with efficient code for extraction.

A locked version of a function f is a provably equal but not convertible version of f. In Coq, a locked version of f is achieved by hiding the body of f behind an opaque dummy constant (see the documentation of `locked` at [20]). This prevents the simplification mechanism commonly triggered during proof development from computing the actual value

[3] Primitive integers, or machine integers, are the integers directly supported by the processor and used physically in memory, with binary representation. Programming languages usually provide a type representing primitive integers.

[4] It is necessary to consider overflow because primitive integers are defined cyclically, so that 2^{63} is the same as 0, and so on.

of the function when applied to a specific argument. The need for locked conversions in our setting comes from certain use cases where large constants are needed on the specification side, and as such are expected to be represented as a `nat`. However, we have seen that expressing large numbers with `nat` takes unfeasible amounts of memory. Our solution is to represent such large constants as primitive integers and rely on a coercion `nat_of_uint`, i.e., on an automatically inserted translation from primitive unsigned integers to the unary representation of natural numbers. Crucially, this coercion needs to be locked to prevent simplification from computing the unary representation of the number.

4 FV Check Range

We developed a set of tactics to automatically solve provable decidable goals with up to three free primitive integer variables bounded by a specific primitive integer range. Given a provable goal with base statement of type `bool` (and thus decidable), and given at most three primitive integer variables and the bounds on which to check them, the tactics identify the desired boolean statement, generate a list with all the primitive integers in the relevant range and use `vm_compute` [35] to confirm that the boolean statement indeed holds for every number in range.

These tactics work rather fast, checking ranges with sizes on the order of 10^5 in hundredths of seconds and on the order of 10^7 in two or three seconds (showing an expected linear progression) in one of our machines.

We used these tactics at several points during the development of FV Time and describe here only a particular example:

$$\forall (0 \leq x < 146097)\ x \geq \frac{h(x)}{365} \cdot 365 + \frac{h(x)}{365 \cdot 4} - \frac{h(x)}{365 \cdot 100} \quad (2)$$

where:

$$h(x) = x - \frac{x}{1460} + \frac{x}{36524} - \frac{x}{146096}.$$

Note that these are natural numbers, and so the division operation is Euclidean division, meaning that, for example, it is not always the case that $\frac{x}{y} \cdot y = x$.

As expected, we found that using our automation was significantly easier and faster than translating pen and paper proofs when proving (2). In particular, the proof inspired by a pen and paper strategy had 400 lines in Coq and took some minutes to compile, while the proof using FV Check Range is a one-liner and takes hundredths of seconds.

There are many other automation tactics available in the Coq ecosystem, some of which can be used to solve goals similar to (2) some of the time. See Section 7 for a discussion.

5 Extraction

The concept of extraction is simple: (automatically) translate statements written in Coq to statements written in some other, faster, language [45–47]. The extraction algorithms are not themselves fully formalized yet (although this formalization is work in progress in the MetaCoq project [56]), and so it is possible that errors in the extraction process lead to unexpected discrepancies between the original and the extracted code. Below we describe a method for extracting Coq programs to OCaml so that the resulting OCaml code is clean, reasonably short, and readable.

5.1 Clean Extraction

The main idea is to only extract Coq code that already looks as close as possible to OCaml code, so that the extraction plugin has almost nothing to do. The work of translating the arbitrarily complex original Coq code to Coq code representable in OCaml then falls to the programmer instead of the plugin. This extra work has two advantages: first, one obtains control over the extracted OCaml code, and second, one can still reason about the Coq code that originated it. This means that the distance between the verified Coq code and the unverified OCaml code is much smaller than if one simply relied on the extraction plugin without any pre-processing.

When rewriting the original Coq code to make it representable in OCaml, a common issue is translating those partial functions that were defined using dependent types. Consider, as an example of a partial function, Euclidean division on the natural numbers, `div`. Division is mathematically undefined when the divisor is 0. There are three main ways of implementing such partial functions in Coq, illustrated here with the type signature of division.

1. Forbid the input:

   $$\texttt{div}_{gt0} : \texttt{nat} \to \texttt{nat\_greater\_than\_0} \to \texttt{nat}.$$

2. Output an error:

   $$\texttt{div}_{err} : \texttt{nat} \to \texttt{nat} \to \texttt{nat\_or\_error}.$$

3. Output a default value:

   $$\texttt{div}_{dflt} : \texttt{nat} \to \texttt{nat} \to \texttt{nat}.$$

Forbidding the input can be done by taking advantage of dependent types, which are not representable in OCaml. Our proposed solution is to rewrite any code that uses dependent types without them, dealing with partiality in some of the other two ways. Outputting an error can be done with an option type, which allows to keep error handling inside Coq, useful if our extracted code is going to work as an external library for other projects. On the other hand, outputting a default value has the advantage of yielding a type signature as simple as possible, which keeps compositionality with other functions. Theorems can then take into account the necessary hypotheses on the input. We also extracted functions in this flavor and used this strategy in the implementation side of our development, as seen in Section 2.3.

This transformation contributes to avoiding the presence of `Obj.magic`,[5] and moreover gives us control about what to do in cases where an undesired input is given, which can happen when executing the extracted code, because dependent types are not expressible in OCaml.

A different problem arises when using Coq libraries that were not purpose-built for extraction, as is the case of MathComp. The pervasive use of canonical structures does not lend itself well to extraction. The mere presence of a boolean equality over an `eqType` leads to over 20 lines of almost vacuous OCaml code where often the boolean equality for our desired type could be defined in a couple of much more easily understood lines that need to be included anyway.

Here our proposed solution is to avoid MathComp and other external libraries as much as possible when paring down the functions meant for extraction. It has been our experience that most of the benefit of using external libraries is in the wealth of results about the defined functions, and not in the functions themselves. These are usually not that numerous or hard to redefine using only simple Coq features.

Given the above observations, our proposed procedure for clean extraction is as follows:

0. Suppose the functions that need to be extracted live in a file `original.v`.
1. In a new file (say, `extraction_file.v`) that does not import nor depend on any other file (save perhaps on simple modules such as `List` from the Coq Standard Library), recursively redefine all the functions that are meant to be extracted, i.e., redefine the functions in `original.v` as well as every function mentioned in `original.v`, whether defined in the current project or provided by MathComp or others. Avoid any Coq features not present in OCaml.
2. Extract the functions in `extraction_file.v`.
3. In another file (say, `extraction_file_correct.v`) that imports `original.v`, `extraction_file.v`, and anything else useful or necessary, show that the extracted functions behave the same as the original ones, possibly under some reasonable assumptions.

A simple example following this method can be found in [33].

5.2 What and How Did We Extract?

The only part of the Coq code that makes sense to extract are the implemented algorithms. We started with the functions defined in `HinnantR.v` and `formalTimeR.v` (Step 0). We then redefined all functions to be extracted together with their dependencies in `fvtm_extraction.v` (Step 1) and extracted them in `extraction_command.v` (Step 2). Note that Coq extraction is recursive, so we only needed to list the functions we wished to add to the OCaml user interface, not every function used to define them. The link between the original functions and the functions to be extracted was provided in `fvtm_extraction_correct.v`.

We used `ExtrOcamlBasic` and `ExtrOCamlInt63` from Coq's Standard Library to help with the extraction. The former is a small collection of well-accepted translations, such as mapping Coq's `bool` type to OCaml's, and other such mappings where the types are basically the same in both languages. The latter maps the Coq definitions of `Uint63` and `Sint63` to the very same OCaml module used to implement primitive integers by the Coq kernel.

Our extracted code can be used as a library, taking into account that functions come in two flavors with respect to the way they deal with partiality: we provide versions named `f_plain` that output default values on ill-formed inputs, corresponding to the f_{dflt} described above, and versions named simply `f` that output a more complex type called `possibly`, which includes information about problematic inputs and corresponds to f_{err}. We decided to have the latter because our purpose when extracting was to use the library from other programming languages, for which we wrote a small command-line interface in OCaml that can be compiled as an executable and invoked from any other program, described in Section 6. It was thus convenient to extract versions that detect errors (where we proved lemmas about what errors are detected, and when), instead of implementing the full exception handling in OCaml, which would be prone to bugs.

For each function `f_plain`, there is a lemma `f_plainR` (where the "R" stands for "refinement") showing that `f_plain` meets its specification (i.e., that it behaves like `calendar.f` on the relevant inputs). Furthermore, there are lemmas proving that the error handling for `f` is correct given certain assumptions on the input.

6 FVTM: a Command-line Interface for FV Time

After extraction, we end up with an OCaml library with all the relevant functions of our development. However, OCaml is not the most popular programming language and communication with other languages is non-trivial. Hence, we wrote a command-line interface in OCaml that allows to compile the library as an executable and invoke it from the terminal or from any other programming language. This command-line version of the library is named FVTM (FV Time Manager) [29]. The main functions of the library, i.e., the conversions between UTC times and timestamps, can be tried online at Formal Vindications S.L.'s webpage.[6]

[5] `Obj.magic` is a low-level OCaml function that allows casting any type to any other type. It is purposefully undocumented because it is not meant for the casual user.

[6] https://formalv.com/TimeManager/FVTimeCalculation

7 Related Work

The field of formal verification has been flourishing during the last few decades, both in the area of mathematics formalization [1, 19, 26, 30–32, 50], and also in the realm of software verification [6, 7, 12–15, 38, 44].

FV Time is not the first library to implement UTC (see for example [23, 34, 43, 51]), but to the best of our knowledge it is the first formally verified one either for Unix time or for UTC.

Our approach using refinements has been extensively developed both in specific proof assistant developments, such as Coq [16, 25, 27] and Isabelle/HOL [37]. Some other software verification projects have used it too [12].

It must be noted that in the literature many efforts have been devoted to free the developer from the burden of manually proving every result in the proof assistant. These efforts have yielded several Coq tactics to automatically perform certain proof tasks, such as `micromega` [10], `ring` [36], `interval` [52], `itauto` [11], `sauto` [24], `firstorder` [22], and `auto` and `eauto` [21]. The tactics `auto`, `eauto` and `sauto` are lemma aggregators, meaning that they produce proofs by combining existing lemmas that can be configured by the user, but this wouldn't solve our intricate arithmetical expressions. The `ring` tactic is also of no use, since our goals included Euclidean division, which doesn't form a ring, field or semi-field on the integers. As for `itauto` and `firstorder`, they are solvers for intuitionistic propositional logic and first order logic respectively, and, although extensible, they can't solve our goals to the best of our attempts.

Notably, `micromega` provides a set of tactics for arithmetic, one of which can deal with some instances of Euclidean division. In fact, a translation of (2) to `ssrint` can be automatically solved when using `mczify` [55], an extension of `micromega` designed to work with MathComp numbers. However, it can't yet be solved in the realm of binary or primitive integers. This is likely not a fundamental but a practical shortcoming that could be bridged with some work. Nonetheless, our tactics solve any kind of decidable goals on primitive integers, not only arithmetical expressions, and thus the scope is significantly different from `micromega`'s.

Lastly, `interval` solves interval arithmetic goals on real numbers. It does not seem like it can solve a translation of our arithmetical expressions in particular. Even if it could, it would introduce a significant amount of overhead and unnecessary axioms due to the detour through the (classical) reals.

8 Contributions and Conclusion

We believe that the development of a formalized time library implementing UTC conversions and operations satisfies an existing need in the panorama of software dealing with time. More importantly, its formal verification led to a number of problems, some of which have yielded general-purpose solutions, while others may be more specific but still inspiring for analogous situations.

The first general-purpose development, FV Prim63 to MathComp, provides a translation between proof-oriented types and operations from MathComp (the `ssrnat` and the `ssrint` libraries), and computation-oriented, extraction-friendly types and operations from the Coq Standard Library (the `Uint63` and `Sint63` libraries). We believe that the world of formal verification of software needs powerful, expressive libraries like MathComp for the specification side, and extraction-friendly libraries for the implementation side [8]. FV Prim63 to MathComp fills the gap between them.

As direct consequences of this work, it is worth noting that this project catalyzed the addition of signed primitive integers (`Sint63`) to Coq. The unsigned version was already available, but when we needed the signed version as well, one of the authors teamed up with the Coq developers to make it happen [4]. Furthermore, we opened a number of minor issues in the Coq bug tracker and helped fix some of them. The improvements live on in the Coq versions since released.

Another general-purpose by-product has been the development of FV Check Range, which provides a set of tactics to automatically solve decidable statements with up to three variables bounded by a specific primitive integer range. These tactics have reduced our development time noticeably, and we hope they serve the same purpose for other teams. Future work includes extending this tool to work with an arbitrary number of variables.

Regarding extraction, we have presented a methodology to minimize possible bugs (crucial before verified extraction for Coq is completed) and obtain clean and simple extracted code. This methodology could serve as a first model of good-practice for other projects.

Finally, we have developed a non-trivial formalized library that is still simple enough to be an accessible example and road map to other libraries. In particular, we have dealt with a number of typical things such as subtyping, partial functions, algorithm and type refinements, extraction, and interface creation.

Acknowledgments

This project wouldn't have been possible without the ideas and help of several people. We are grateful to Guillermo Errezil for prompting the project and to Jasper Hugunin for the idea behind FV Check Range's implementation. We thank Cyril Cohen for his essential help and advice on using Coq. Yannick Forster shared some advice on improving some parts of this paper, for which we are grateful. Finally, we would like to thank the Coq community at large for all the discussions and support.

All authors have received research support in a consortium between the University of Barcelona and Formal Vindications S.L. under the grant RTC-2017-6740-7 of the Spanish Ministry of Science and Universities.

Author Roles. **Ana de Almeida Borges:** Conceptualization, Methodology, Software, Visualization, Writing - original draft, review & editing; **Mireia González Bedmar:** Conceptualization, Methodology, Software, Visualization, Writing - review & editing; **Juan Conejero Rodríguez:** Conceptualization, Methodology, Software; **Eduardo Hermo Reyes:** Conceptualization, Software, Writing - review & editing; **Joaquim Casals Buñuel:** Software, Visualization; **Joost J. Joosten:** Funding acquisition, Project administration, Writing - review & editing.[7]

References

[1] Ana de Almeida Borges. 2022. Towards a Coq formalization of a quantified modal logic. In *Proceedings of the 4th International Workshop on Automated Reasoning in Quantified Non-Classical Logics (ARQNL 2022)*, Christoph Benzmüller and Jens Otten (Eds.). CEUR, Haifa, Israel, 13–27. https://ceur-ws.org/Vol-3326/ARQNL2022_paper1.pdf

[2] Ana de Almeida Borges, Joaquim Casals Buñuel, Juan Conejero Rodríguez, Mireia González Bedmar, and Eduardo Hermo Reyes. 2023. The FormalV Library. https://gitlab.com/formalv/formalv/-/releases/1.2.0 Version 1.2.0.

[3] Ana de Almeida Borges, Juan Conejero Rodríguez, David Fernández-Duque, Mireia González Bedmar, and Joost J. Joosten. 2021. To drive or not to drive: A logical and computational analysis of European transport regulations. *Information and Computation* 280 (2021), 104636. https://doi.org/10.1016/j.ic.2020.104636

[4] Ana de Almeida Borges, Guillaume Melquiond, and Pierre Roux. 2021. Signed primitive integers. https://github.com/coq/coq/pull/13559

[5] Android. 2022. Date Class. Android Reference Manual. https://developer.android.com/reference/java/util/Date

[6] Andrew W. Appel. 2011. Verified Software Toolchain. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

[7] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2, Article 7 (2015), 31 pages. https://doi.org/10.1145/2701415

[8] Andrew W Appel. 2022. Coq's vibrant ecosystem for verification engineering (invited talk). In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, New York, NY, United States, 2–11. https://doi.org/10.1145/3497775.3503951

[9] R.J.R. Back. 1981. On correct refinement of programs. *J. Comput. System Sci.* 23, 1 (1981), 49–68. https://doi.org/10.1016/0022-0000(81)90005-2

[10] Frédéric Besson. 2007. Fast Reflexive Arithmetic Tactics the Linear Case and Beyond. In *Types for Proofs and Programs*, Thorsten Altenkirch and Conor McBride (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 48–62. https://doi.org/10.1007/978-3-540-74464-1_4

[11] Frédéric Besson. 2021. Itauto: An Extensible Intuitionistic SAT Solver. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:18. https://doi.org/10.4230/LIPIcs.ITP.2021.9

[12] Dominique Cansell, J. Paul Gibson, and Dominique Méry. 2007. Refinement: A Constructive Approach to Formal Software Design for a Secure e-voting Interface. *Electronic Notes in Theoretical Computer Science* 183 (2007), 39–55. https://doi.org/10.1016/j.entcs.2007.01.060 Proceedings of the First International Workshop on Formal Methods for Interactive Systems (FMIS 2006).

[13] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

[14] Adam Chlipala. 2010. A Verified Compiler for an Impure Functional Language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 93–106. https://doi.org/10.1145/1706299.1706312

[15] Adam Chlipala. 2011. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. *SIGPLAN Not.* 46, 6 (jun 2011), 234–245. https://doi.org/10.1145/1993316.1993526

[16] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. 2013. Refinements for Free!. In *Certified Programs and Proofs (Lecture Notes in Computer Science, Vol. 8307)*, Georges Gonthier and Michael Norrish (Eds.). Springer, Cham, 147–162. https://doi.org/10.1007/978-3-319-03545-1_10

[17] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. 2020. Hierarchy Builder: Algebraic hierarchies made easy in Coq with Elpi. In *FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction (5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020), 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl, Paris, France, 34:1–34:21. https://doi.org/10.4230/LIPIcs.FSCD.2020.34

[18] Comité international des poids et mesures. 2022. Resolutions of the General Conference on Weights and Measures (27th meeting). https://www.bipm.org/documents/20126/77765681/Resolutions-2022.pdf/281f3160-fc56-3e63-dbf7-77b76500990f

[19] Ernesto Copello, Nora Szasz, and Álvaro Tasistro. 2018. Machine-checked Proof of the Church-Rosser Theorem for the Lambda Calculus Using the Barendregt Variable Convention in Constructive Type Theory. *Electronic Notes in Theoretical Computer Science* 338 (2018), 79–95. https://doi.org/10.1016/j.entcs.2018.10.006 The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).

[20] Coq Development Team. 2023. Coq.ssr.ssreflect. Coq Standard Library Version 8.17.1. https://coq.inria.fr/doc/V8.17.1/stdlib/Coq.ssr.ssreflect.html

[21] Coq Development Team. 2023. Programmable proof search. Coq Reference Manual Version 8.17.1. https://coq.github.io/doc/v8.17/refman/proofs/automatic-tactics/auto.html

[22] Pierre Corbineau. 2004. First-Order Reasoning in the Calculus of Inductive Constructions. In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 162–177. https://doi.org/10.1007/978-3-540-24849-1_11

[23] cppreference.com Team. 2021. `utc_clock`. C++ Reference Manual. https://en.cppreference.com/mwiki/index.php?title=cpp/chrono/utc_clock&oldid=134878

[24] Łukasz Czajka. 2020. Practical Proof Search for Coq by Type Inhabitation. In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 28–57. https://doi.org/10.1007/978-3-030-51054-1_3

[25] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive synthesis of abstract data types in a proof assistant. *ACM SIGPLAN Notices* 50, 1 (2015), 689–700. https://doi.org/10.1145/2775051.2677006

[26] William DeMeo. 2022. The Agda Universal Algebra Library. http://ualib.org/

[27] Maxime Dénès, Anders Mörtberg, and Vincent Siles. 2012. A Refinement-Based Approach to Computational Algebra in Coq. In

[7] CRediT author statement as described in https://casrai.org/credit/.

ITP 2012: Interactive Theorem Proving (Lecture Notes in Computer Science, Vol. 7406), Lennart Beringer and Amy Felty (Eds.). Springer, Berlin, Heidelberg, 83–98. https://doi.org/10.1007/978-3-642-32347-8_7

[28] European Parliament and Council of the European Union. 2016. COMMISSION IMPLEMENTING REGULATION (EU) 2016/799 of 18 March 2016 implementing Regulation (EU) No 165/2014 of the European Parliament and of the Council laying down the requirements for the construction, testing, installation, operation and repair of tachographs and their components. Official Journal of the European Union. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32016R0799&qid=1688983367069

[29] Formal Vindications S.L. 2022. How-To: Use of the FV Time Manager on Windows, Linux and other platforms through its command line interface. https://formalv.gitlab.io/fv-docs/fvtm-tech-spec.pdf

[30] Yannick Forster, Dominik Kirst, and Gert Smolka. 2019. On Synthetic Undecidability in Coq, with an Application to the Entscheidungsproblem. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 38–51. https://doi.org/10.1145/3293880.3294091

[31] Georges Gonthier. 2008. Formal proof – the four-color theorem. *Notices of the American Mathematical Society* 55, 11 (2008), 1382–1393.

[32] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. 2013. A Machine-Checked Proof of the odd order theorem. In *Interactive Theorem Proving*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 163–179. https://doi.org/10.1007/978-3-642-39634-2_14

[33] Mireia González Bedmar. 2022. A method for clean extraction: example. https://gitlab.com/formalv/extraction-model

[34] Google. 2022. Unsmear. https://github.com/google/unsmear

[35] Benjamin Grégoire and Xavier Leroy. 2002. A compiled implementation of strong reduction. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. ACM, New York, NY, United States, 235–246. https://doi.org/10.1145/581478.581501

[36] Benjamin Grégoire and Assia Mahboubi. 2005. Proving Equalities in a Commutative Ring Done Right in Coq. In *Theorem Proving in Higher Order Logics*, Joe Hurd and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 98–113. https://doi.org/10.1007/11541868_7

[37] Florian Haftmann, Alexander Krauss, Ondřej Kunčar, and Tobias Nipkow. 2013. Data Refinement in Isabelle/HOL. In *Interactive Theorem Proving*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 100–115. https://doi.org/10.1007/978-3-642-39634-2_10

[38] John Harrison. 1999. A Machine-Checked Theory of Floating Point Arithmetic. In *Theorem Proving in Higher Order Logics*, Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 113–130. https://doi.org/10.1007/3-540-48256-3_9

[39] Howard Hinnant. 2019. chrono-Compatible Low-Level Date Algorithms. https://howardhinnant.github.io/date_algorithms.html

[40] IEEE and The Open Group. 2018. The Open Group Base Specifications Issue 7. https://pubs.opengroup.org/onlinepubs/9699919799/

[41] IETF. 2016. Leap seconds list. https://www.ietf.org/timezones/data/leap-seconds.list

[42] ITU Radiocommunication Assembly. 2002. Recommendation ITU-R TF.460-6: Standard-frequency and time-signal emissions. *International Telecommunication Union* (2002). https://www.itu.int/dms_pubrec/itu-r/rec/tf/R-REC-TF.460-6-200202-I!!PDF-E.pdf

[43] Daan Leijen. 2016. UTC time calculation. The Koka Programming Language Documentation. https://koka-lang.github.io/koka/doc/std_time_utc.html

[44] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (Jul 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[45] Pierre Letouzey. 2003. A New Extraction for Coq. In *Types for Proofs and Programs*, Herman Geuvers and Freek Wiedijk (Eds.). Springer, Berlin, Heidelberg, 200–219. https://doi.org/10.1007/3-540-39185-1_12

[46] Pierre Letouzey. 2004. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Ph. D. Dissertation. Université Paris Sud-Paris XI.

[47] Pierre Letouzey. 2008. Extraction in Coq: An overview. In *Logic and Theory of Algorithms*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.). Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 359–369. https://doi.org/10.1007/978-3-540-69407-6_39

[48] Judah Levine, Patrizia Tavella, and Martin Milton. 2023. Towards a consensus on a continuous coordinated universal time. *Metrologia* 60, 1 (2023), 014001. https://doi.org/10.1088/1681-7575/ac9da5

[49] Assia Mahboubi. 2013. The rooster and the butterflies. In *International Conference on Intelligent Computer Mathematics*. Springer-Verlag, Berlin, Heidelberg, 1–18. https://doi.org/10.1007/978-3-642-39320-4_1

[50] Mathematical Components Team. 2007. The Mathematical Components library. https://math-comp.github.io/

[51] MathWorks. 2022. leapseconds. MATLAB Reference Manual. https://www.mathworks.com/help/matlab/ref/leapseconds.html

[52] Guillaume Melquiond. 2023. Coq Interval. https://coqinterval.gitlabpages.inria.fr/

[53] Microsoft. 2022. DateTime.Ticks. Microsoft .NET 6 Documentation. https://docs.microsoft.com/en-us/dotnet/api/system.datetime.ticks?view=net-6.0

[54] Kazuhiko Sakaguchi. 2020. Validating Mathematical Structures. In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 138–157. https://doi.org/10.1007/978-3-030-51054-1_8

[55] Kazuhiko Sakaguchi. 2021. Mczify. https://github.com/math-comp/mczify

[56] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2023. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. (2023). https://inria.hal.science/hal-04077552 hal-04077552.

Received 2023-09-19; accepted 2023-11-25