(Preprint) AAS 13-518

PROGRAMMING PERSPECTIVE ON TIME SCALES

Andrew Main*

Present software and wetware suffer from a failure to grasp the philosophical and practical implications of the existence and availability of multiple time scales. Rethinking from a thoroughly modern point of view, this paper outlines the kind of API by which software of the future can better handle present and future horological needs, addressing both specialized and mundane requirements. Concerning the future of UTC, the new analysis challenges the notion that there is a meaningful decision to be made on whether to abolish leap seconds. Some practical issues in programming around leap seconds are discussed. Some technical requirements are identified, and some ideas that have been mistaken for requirements are unmasked.

PARADIGM SHIFT REQUIRED

Existing time-handling facilities in computer programming extensively rely on an unstated assumption that *all time scales are essentially equivalent*. This assumption is false, and leads to a range of problems. More or less faulty implications of the assumption include:

- Conversions between time scales are semantically insignificant.
- The best way to avoid time scale confusion is to reduce all time data as soon as possible onto a single canonical time scale.
- A clock that is not synchronized to canonical absolute time is necessarily an inaccurate clock, and generates inaccurate timestamps.
- In the usual case where the canonical time scale is UTC, a system with an atomic clock but unaware of the present leap second schedule cannot offer correct clock readings.
- Conversions between timezones, being insignificant, are equally permissible for times in the future and in the past.
- It is vitally important that civil time everywhere have the same basis.
- Sidereal time, Martian time, and the like are inherently difficult to process.
- Time coordinate corrections for relativistic motion require entirely custom software.
- The essential issue for a new time-using application is how it should cope with the awkward aspects of the canonical time scale.

*Scientist at Large, The Perl Foundation, 340 S Lemon Ave #6055, Walnut, California 91789–2706, USA

Where applications run into problems stemming from this assumption, the tendency is to live with the problems. Where that strategy is untenable, the dominant behavior is to model the situation as a single exception to the equivalence paradigm. Quite a few 'single' exceptions have accreted over time. For example, in many systems time scales are interconvertible except in the immediate vicinity of a leap second; also, the usual treatment of timezones is that, while civil time is different in different jurisdictions, it can only differ by offsets of whole minutes.

While this paper principally considers the effects of the faulty paradigm on APIs, the paradigm is also manifest in human thought patterns. The two realms are deeply intertwined. Poor understanding of a problem space begets muddled API design, which begets confusion among API users. The discussion of time scale concepts in this paper is for the most part relevant to both realms.

Having identified a dominant faulty assumption in programming is not to say that programmers are actually unaware that the assumption is false. Indeed, some of the issues mentioned above arise where there is an explicit inequivalence of time scales. The nature of a paradigm is such that its logical consequences are habitually relied upon without formal consideration (or even conscious awareness) of their basis. The use of a paradigm in an application that conflicts with it thus sets up logical contradictions, manifesting as doublethink in humans and as nonsensical behavior in computers.

The present faulty paradigm can be readily understood as merely a non-specialist view of time. The intellectual history of horology offers a slightly more enlightening view: it is only relatively recently that a more sophisticated model of time has become necessary even for specialists. Specifically, Newcomb's model of planetary motion was the first work to reveal the imperfection of Earth rotation as a timepiece.[1–3] The present pain in computer timekeeping arises from the increasing practical consequences of this still-esoteric knowledge.

To truly correct the present problems, the underlying faulty paradigm must be replaced. The straightforward replacement paradigm is that *there are many essentially distinct time scales*. Immediate consequences of this are:

- Conversions between time scales are in general semantically significant.
- To avoid losing information, point-in-time values must generally remain unconverted and carry an explicit indicator of the time scale from which they originate.
- A clock that is not synchronized to any preferred time scale defines its own time scale. The relationship between the clock's time scale and a preferred time scale may be established, either contemporaneously or retrospectively, allowing clock readings to be usefully converted.
- A system with an atomic clock but unaware of the present leap second schedule cannot offer clock readings in UTC directly, but can offer correct clock readings that can later be converted to UTC.
- Timezones are effectively time scales, and conversions between them are semantically significant. Timezone conversions based on historical data are qualitatively different from conversions based on future projections.
- There's no requirement for civil time scales to share any more behavior than do time scales in general.

- Sidereal time, Martian time, and the like are just more time scales.
- Corrections for relativistic motion are just another flavor of conversion between time scales.
- The essential issue for a new time-using application is which time scale it should use.

The remaining sections of this paper will apply the new paradigm, in progressively more practical ways, usually without any further comment.

API SKETCH

This section will develop the core of a time-handling API based on a modern understanding of time scales. The relevant aspects of the API are not specific to any particular programming language, and could (and should) be implemented in any high-level language. The API will be illustrated here in Scheme, which imposes a minimum of distractions.[4] The Scheme language is augmented with the GOOPS object system, which is based on the metaobject protocol (MOP) designed for the Common Lisp Object System (CLOS).[5–8] GOOPS is further augmented with some code shown in the appendix.

While this section is mainly concerned with the API itself, particularly interesting implementation issues will also be discussed as they arise.

Time Values

Because time scales are not equivalent, a point-in-time value must encapsulate not just a numerical time label but also the time scale governing the interpretation of the coordinate. Because different time scales label points in time in different ways, it is also necessary to avoid hard-coding a particular type of time label in the system. In code, the structure for time values is:

```
(define-class <time-scale> ())
(define-class <time-label> ())
(define-class <time-value> ()
  (scale #:init-keyword #:scale
         #:getter time-value-scale)
  (label #:init-keyword #:label
         #:getter time-value-label))
(define (make-time-value s l)
  (make <time-value> #:scale s #:label l))
```

(make-time-value *scale label*) → *time*
(time-value-scale *time*) → *scale*
(time-value-label *time*) → *label*

The definition of a class <time-scale> presumes that time scales will be reified as language-visible objects. For the purposes of the <time-value> structure it is not essential to reify them; they could be represented by name, for example, rather than objects. However, full reification is generally a good idea, and the next subsection will show a specific good reason for reification.

In this structure, a point-in-time value might represent 2002–03–13 19:57:08.852 TT(TAI), and this would be distinct from the point-in-time values representing 2002–03–13 19:57:08.852 TT and

2002–03–13 19:57:08.852 TCG. Where two time scales are completely semantically equivalent, for some purposes one could blur the distinction between them; for example, 2002–03–13 19:57:08.852 TT(TAI) represents exactly the same thing as 2002–03–13 19:56:36.668 TAI. However, in these cases it remains useful to be able to extract both of the numerically-different labels, each with its own time scale reference, and the simplest way to do this is to just keep them as distinct time values.

The simplest kind of time label is a linear numerical count, such as the count of days described as a Modified Julian Date (MJD). MJDs can be trivially wrapped for use as time labels; they are used by time scales UT1, TT, et al. Generally, time labels should be ordered, which in this case corresponds to the usual ordering of the MJD numbers themselves. In code:

```
(define-class <mjd-label> (<time-label>)
  (mjd #:init-keyword #:mjd #:getter time-label-mjd))
(define-method (<= (a <mjd-label>) (b <mjd-label>))
  (<= (time-label-mjd a) (time-label-mjd b)))
```

(This code presumes that other comparison operators are synthesized from the generic <=. See the appendix.)

The purpose of all this wrapping of a simple number becomes apparent when it becomes necessary to extend this label schema. Points on the UTC time scale (with leap seconds) are for some purposes described in terms of MJD, and are frequently compared to points on pure MJD time scales, for example in expressions such as "TAI − UTC". But because UTC can have more than 86400 seconds per day, whereas UTC's equivalent MJD values are always based on the nominal 86400-second day, it can't just use MJD values. For example, 2012–06–30 23:59:60.864 UTC and 2012–07–01 00:00:00.864 UTC both have MJD 56109.00001.

A neat formalism for UTC's extended day is found by splitting MJD into its integer and fractional components. The integer part is known as the Modified Julian Day Number (MJDN), and this paper will call the fractional part the Modified Julian Day Fraction (MJDF).* The MJDN corresponds to date, and the MJDF to time of day; MJD = MJDN + MJDF, and when it is derived from an MJD there is a constraint that $0 \leqslant \text{MJDF} < 1$.

In UTC the MJDF can be allowed to reach and exceed 1: the MJDN identifies the UTC day, and the MJDF is based on how many seconds have passed since midnight, which can exceed 86400. For example, 2012–06–30 23:59:60.864 UTC has MJDN = 56108 and MJDF = 1.00001, while 2012–07–01 00:00:00.864 UTC has MJDN = 56109 and MJDF = 0.00001. UTC time labels are correctly ordered, both with respect to each other and with respect to plain MJDs, by comparing MJDN first and then MJDF. Arithmetic operations on UTC time labels will nevertheless normally use the MJD. In code:

```
(define-class <mjdish-label> (<time-label>))
(insert-superclass! <mjd-label> <mjdish-label>)
(define-method (time-label-mjdn (lab <mjd-label>))
  (floor (time-label-mjd lab)))
(define-method (time-label-mjdf (lab <mjd-label>))
  (- (time-label-mjd lab) (time-label-mjdn lab)))
```

*The term "MJDF" is a neologism previously used by the author, along with its generalization to other JD-like day counts, in the documentation for a related Perl module ⟨https://metacpan.org/module/Date::JD/⟩.

```
(define-class <mjdnf-label> (<mjdish-label>)
  (mjdn #:init-keyword #:mjdn #:getter time-label-mjdn)
  (mjdf #:init-keyword #:mjdf #:getter time-label-mjdf))
(define-method (time-label-mjd (lab <mjdnf-label>))
  (+ (time-label-mjdn lab) (time-label-mjdf lab)))
(define-method (<= (a <mjdish-label>) (b <mjdish-label>))
  (or (< (time-label-mjdn a) (time-label-mjdn b))
      (and (= (time-label-mjdn a) (time-label-mjdn b))
           (<= (time-label-mjdf a)
               (time-label-mjdf b)))))
```

(See the appendix for the definition of `insert-superclass!`.)

The MJD system is far from unique. The same sequence of days is also numerically labeled using other day counts such as the Truncated Julian Date (TJD). Some other day count systems, principally the Julian Date (JD), follow an obsolete astronomical convention by dividing days at noon instead of midnight. Other day counts have a slightly different class of usage: in the context of Earth rotation time, MJD implicitly refers to the days of UT, but a Chronological Julian Date (CJD) refers to local (or timezone-defined) days that may be offset from UT. All of these classes of Terran day count have been occasionally reinvented. With day count values wrapped up as objects, computers can translate them implicitly, ensuring that they don't get confused and avoiding the need for any global choice of canonical day count system.

There are also day counts that resemble MJD but can't be trivially converted because they refer to a different sequence of days.* For example, the Earth Rotation Angle (ERA) counts stellar days, i.e., rotations of Earth relative to the celestial sphere rather than relative to the Sun.† More exotically, but also of current practical concern, the Mars Sol Date (MSD) counts the Martian solar days of Airy Mean Time (AMT).[10] Code and API concepts should be shared between these day counts, but the API should also provide some type incompatibility to prevent accidental confusion between incompatible day counts.

Other kinds of time label exist, but will not be examined in detail here. The treatment above of the awkward UTC case serves to illustrate how code can cope.

Conversion Operation Dispatch

With the understanding that time scales are not equivalent, the most important operation supported by the API is the conversion of point-in-time values between time scales. The required features of this function will be built up in stages. At its most basic, the plenipotent time scale conversion function is used thus:

`(convert-time` *src-time tgt-scale*`)` $\rightarrow$ *tgt-time*

*Some time scales not linked to Earth rotation, such as TT and TCB, would appear to call for such a separate day count, or even better a count of the SI seconds that are their true base unit, but nevertheless notionally use MJD et al. This illogical situation arises from considerations of historical continuity. Most venerably, Ephemeris Time (ET) was a reinterpretation of a time coordinate that Newcomb had originally intended to be UT.

†The present formal definition of UT1 in terms of ERA, like its predecessors, ostensibly divorces UT1 from the Sun.[9] This paper treats the UT1 $\leftrightarrow$ ERA relationship as if it were actually defined in terms of the Earth-Sun orbit, with the present 'definition' being merely a model that provides a *realization* of UT1. Despite the form of words used by the IAU, the history of redefining UT1 seems more consistent with the model/realization view.

The *src-time* and *tgt-time* values are point-in-time values, each incorporating some choice of time scale. The *tgt-scale* value is such a choice of time scale, taken in isolation. The output, *tgt-time*, represents the same time as *src-time*, but on the time scale *tgt-scale*. In general, *tgt-time* is not semantically equivalent to *src-time*. Where correctness dictates, therefore, *tgt-time* will be *obviously* different from *src-time*. (The importance of this will become apparent in later subsections.) Some conversions have no correct answer: for example, converting a pre-1955 UT1 time value into TAI is impossible because by definition TAI doesn't exist for such times. In such cases the conversion operation must signal an error.

The process that is required to convert between time scales depends a great deal on the particular time scales involved. For example, UTC $\leftrightarrow$ TAI is an exact conversion requiring consulting a leap second table; TAI $\leftrightarrow$ UT1 requires interpolation between measurements of Earth orientation parameters and is subject to measurement error; UT1 $\leftrightarrow$ UT2 is an exact conversion using a fixed formula. Furthermore, given the existing diversity of time scales, a truly general time-handling library should be able to process time scales that are not explicitly built into its implementation, a fortiori novel time scales that didn't exist when the implementation was written. So it is useful for existing time scales, and essential for extensibility, that the implementation be able to invoke modular conversion code. This requires at least a modicum of object orientation in the API: time scales per se must be reified as language-visible objects.

Because the conversion operation depends equally on the source and target time scales, conversion implementors will find it more appropriate to think in terms of a more symmetric interface. The user-oriented function must be defined in terms of the implementors' function:

```
(time-converter src-scale tgt-scale) → func

(define-generic time-converter)
(define (convert-time src-time tgt-scale)
  (make-time-value tgt-scale
    ((time-converter (time-value-scale src-time)
                     tgt-scale)
     (time-value-label src-time))))
```

In the form shown here, the actual conversion code (which is returned by `time-converter`) deals with bare time labels, and so must understand exactly what time scales it is converting between. Conversion code that handles multiple time scale pairs, of which an example will be seen below, must wrap up knowledge of the exact time scales being used in the *func* object that is returned by `time-converter`. If the conversion is implemented in a sufficiently high-level language, this will likely be achieved in the form of a closure. The `time-converter` protocol could instead use the combined scale/label point-in-time objects, requiring less closure-like activity, but this would give more opportunities for code to violate the protocol, and so would only be desirable in an environment where closures are disproportionately expensive.

The generic function `time-converter` employs *multiple dispatch*: the code ("method") that is actually invoked by a call depends on the types of both of the parameters. Many OO languages do not provide this as a native facility, instead being limited to dispatching based on the type of only a single parameter. (Single-dispatch polymorphism is also more commonly expressed as a feature of the dispatching parameter (the "invocant") than as a regular function call.) The lack of

native multiple dispatch is not a fatal barrier to implementing the present API in such languages: multiple dispatch can be implemented non-natively.[11–13] Broadly speaking, the difficulty of doing so is inversely proportional to the amount of metaobject protocol that the language natively supplies.

It is worth examining the extent to which multiple dispatch is required. In many non-time conversion situations, all conversions can pivot around a single canonical format, thus breaking the general double-dispatch conversion case into two single-dispatch legs. That is not possible in the general case for time scale conversions. For example, consider the UT1 ↔ UT2 conversion, which is exactly defined for any time for which UT1 is defined. If forced to go via an atomic time scale such as TAI, the conversion would be incorrectly rendered imprecise for any future time (for which the requisite EOP measurements don't yet exist). Conversely, UTC ↔ TAI would be incorrectly rendered imprecise if forced to go via an Earth rotation time scale. There is no pivot time scale that satisfies both cases, so single dispatch does not suffice.

In many situations, time scale conversions can be meaningfully chained. For example, UTC ↔ UT2 is logically composed of three conversions: UTC ↔ TAI (leap second schedule), TAI ↔ UT1 (EOP measurements or projections*), and UT1 ↔ UT2 (fixed formula). This kind of chaining occurs especially where one time scale is canonically defined in terms of another, as is the case here with both UTC (defined in terms of TAI) and UT2 (defined in terms of UT1). This structure should be exploited in implementing the multiple dispatch of the `convert-time` operation. Simplistically, methods for such derived time scales can be implemented akin to

```
(define-method (time-converter (src (singleton UT2)) tgt)
  (let ((convert-UT1-to-tgt (time-converter UT1 tgt)))
    (lambda (src-time)
      (convert-UT1-to-tgt
        (convert-UT2-to-UT1 src-time)))))
```

(The `singleton` function provides a method specializer identifying a single object. See the appendix for an implementation.)

Because it would get tedious to individually define each such method required for each derived time scale, it would be preferable to build some logic into the generic function that can search for suitably-defined conversions that can be automatically chained. This would involve to some extent implementing the multiple dispatch for this function non-natively even if there is a native facility: in MOP terms it amounts to a non-standard method selection and method combination mechanism.

An obvious question here is whether *all* time scale conversions can be reduced to a uniquely-appropriate chain of primitively-defined conversions. (For the UTC ↔ UT2 case, the UTC ↔ TAI ↔ UT1 ↔ UT2 chain is indeed uniquely appropriate.) This amounts to the question of whether the network of time scales, linked by appropriate primitively-defined conversions, forms an acyclic graph. If this were the case, then the double-dispatch problem would reduce to identifying the unique chain between the two time scales, which could be achieved by relatively simple means. The answer is that in general time scales are not linked by a unique chain, but the situations where they are not are relatively esoteric. So the techniques appropriate to an acyclic graph can form a large part of the conversion dispatch mechanism.

*The comparison between UT1 and atomic time is usually expressed using UTC on the atomic side, but TAI is more logical, and for projections beyond the leap second scheduling horizon it is more practical too. IERS Bulletin A projects 'UT1 − UTC' a year ahead, using a pseudo-UTC that it explicitly defines in terms of TAI.

Cycles in time scale conversions naturally arise with locally-realized reference time scales, such as the raw laboratory time scales that contribute to the computation of TAI. These can in principle be meaningfully compared, as peers, in any combination, and where links have similar quality the most appropriate way to convert between any two that are directly compared is via that comparison link. In practice, recent issues of Circular T show no cycles in the comparison graph behind TAI (although there used to be some).[14] A more fruitful source of cycles is the Network Time Protocol (NTP), which encourages redundant links.[15*] The issue of how best to convert time scales in the presence of such redundant comparisons is beyond the scope of this sketch.

Subjectivity

The API shown so far suffices for situations where the correspondence between two time scales is objectively well defined. In relativistic situations, however, the correspondence depends on a choice of spatial location, or, more strictly, a choice of world line (trajectory through spacetime).

Considering first the theoretically cleanest case, conceptually a time scale exhaustively partitions (foliates) spacetime into a contiguous series of Cauchy surfaces (roughly, unbounded spacelike hypersurfaces), and a coordinate on that time scale identifies a particular Cauchy surface from the series. Each Cauchy surface constitutes the set of events (points in spacetime) that the time scale reckons to be simultaneous. (In the Minkowski spacetime of special relativity, applying Einsteinian simultaneity, the Cauchy surfaces are hyperplanes.) A world line is an inextensible timelike curve in spacetime. The intersection of a Cauchy surface with a world line is a single event. (The definition of a Cauchy surface requires that there be exactly one such intersection.) The transformation from one time scale to another time scale, from the point of view of some subject, can be defined as taking the event where the Cauchy surface defined by the original time value intersects the subject's world line, and then finding the Cauchy surface defined by the target time scale that includes that event.

A degenerate case occurs where the two time scales foliate spacetime into identical sets of simultaneity hypersurfaces; i.e., where the two time scales always agree on simultaneity. For example, TCG, TT, and TAI are all explicitly based on equivalent time axes. In this case the choice of world line makes no difference to the result of the transformation. This is the case that was described as "objectively well defined" at the beginning of this subsection. Observe that it is *not* necessary for the time scales to agree on the temporal distance (duration of the period) between non-simultaneous events (i.e., on the rate at which time passes). Nor is it even necessary for them to agree on the ratios between durations (i.e., they don't have to be merely scaled relative to each other).

For the purposes of the API, this can all be modeled merely by adding another parameter to the conversion function:

`(convert-time` *src-time tgt-scale subject*`)` → *tgt-time*

The *subject* parameter is an object reifying an observer from whose perspective the conversion is performed. To be used in this way, it must supply the required information about the subject's world line. The manner in which it does this is beyond the scope of this sketch. Different systems

*NTP's clock steering graph is directed, and the protocol quickly breaks any cycles that arise. Even when considering all the graph edges that exist over a period of time, asymmetric configuration means that the graph is largely acyclic. The measurements made are nevertheless applicable in either direction, so for conversion purposes the links are undirected and there is a great deal of cyclicity.

will tackle the issue in different ways, and generic functions offer great flexibility in doing so. User-defined time scale conversion code can accept entirely user-defined world line objects, making it easy to experiment with structures for this parameter. Competing world line mechanisms can even be retrospectively reconciled. Where the subject is a tangible entity such as a celestial body, spacecraft, or astronaut, it is likely to already be reified as a language-visible object for other purposes, and in such cases the existing object can take on the additional role of supplying world line data, rather than using a separate object.

To use the *subject* parameter correctly in common cases doesn't require knowing any detail. There are a small number of world lines that are used for standard purposes, and each of those can be referenced as a named object without needing to delve into the object's internal structure. The Terran geocenter and the Solar System barycenter are of particular importance, and are well defined by the GCRS and BCRS.[16]

The degenerate case where source and target time scales are equivalent for simultaneity will be very common in practice. A great many programs will operate entirely on a single time axis. Such a restriction on the scope of time scale conversions could be usefully enforced, by the use of a world line object that declines to supply any specific information. This would be the most appropriate default choice of *subject* parameter: a program that outgrows this default will naturally call attention to the need to consider subjectivity, by signaling errors.

So far this subsection has assumed some geometric good behavior on the part of spacetime, to make it possible to construct hypersurfaces with the requisite relationship to world lines. Non-trivial topology or extreme curvature could break this assumption. Black holes pose some difficulty for foliation into Cauchy surfaces. More prosaically, world lines and time scales that are of practical interest have limited temporal ranges. One can also consider unphysical world lines and simultaneity hypersurfaces, involving discontinuities or non-standard causal structure, which can result in non-unique conversions no matter how well-behaved the physical system.* So a time scale conversion may have no solutions, or (in particularly exotic situations) multiple solutions.

A lack of solutions is no problem for the API: as mentioned in the previous subsection, the conversion operation must signal an error. Ambiguity is trickier. Multiple solutions could all be returned by redefining the interface to return *a set of* time values, but a finite set system is insufficient. For example, if the simultaneity hypersurfaces of the source time scale are the past light cones of some world line (which makes for a useful, albeit non-Einsteinian, time scale), and the subject is a photon, then a single source time value can correspond to a large range of target time values. It's also somewhat unsatisfactory, and a likely source of bugs, to require such structurally-distinct behavior for this vanishingly exotic situation.† Until there is some real experience with ambiguous conversions, the most practical approach is to signal an error when there are multiple solutions.

Quantitative Inexactness

The time scale conversion process as it has been analyzed so far takes an exact point-in-time input and produces an exact point-in-time output. (The world line input discussed in the preceding subsection has also been treated as exact.) Although there's been mention of some conversions being impossible, and some being subjective, the input and output so far appear to be otherwise

*Any coordinate time scale such as TCB is unphysical with respect to the interior of a black hole, where the hypersurface corresponding to a constant time coordinate is locally timelike.

†This is comparable to the issue mentioned later in this paper regarding the present rarity of leap seconds.

semantically equivalent. This isn't what the new paradigm calls for, and actually produces ridiculous results. For example, consider three NTP nodes that are all mutually compared over links of similar quality. None of the links yields measurements that are perfectly equivalent to chaining the other two (and this is part of the point of having all three links), so converting from one node's time scale back to itself via the loop of three links yields a different result from the starting point. This is nonsensical when applied to values that are semantically exact.

The answer, of course, is that these conversions actually involve some uncertainty, arising from the underlying measurements. For correctness, this uncertainty needs to be represented in the conversion process. The output will not represent an exact point-in-time value, but a range (interval) of such values (on a single time scale), or some other form of inexact value. (This is where it becomes important that the output is permitted to be obviously different from the input: an exact input produces inexact output.) Standard techniques exist for interval arithmetic.[17] In the general case, not only are conversions uncertain, but initial data carry their own uncertainty; so the input to a conversion may also be a range or other form of inexact value.

With this change, the loop of three NTP links is well behaved. Starting with an exact point-in-time value and converting around the loop of three links yields an inexact point-in-time value that is *consistent with* the starting value although not identical to it. Obviously, such a looped conversion is perverse and undesirable. To avoid the loss of precision in the use of redundant conversions, in general point-in-time values should be left in their original form where possible. Conversions should take place as late as possible, when it is known on exactly which time scale points must ultimately be considered.

Because the inexactness occurs in the (usually numerical) label part of a time value, not in the choice of time scale, the interval structure (identifying lower and upper bounds on a value) should be contained in the label slot of the `<time-value>` class. This implies that `time-value-label` can return such an inexact value. It also fits properly into the dispatch mechanism that unwraps time labels. However, the interval structure should *not* be nested inside a non-trivial label structure such as `<mjdnf-label>`. That class defines its own ordering, and information would be lost if an attempt were made to move the inexactness down to its constituent parts. Fortunately a generic implementation of interval arithmetic readily applies to anything that supplies an ordering; very little of it is specific to ordinary numbers.

An inexactness mechanism can also be usefully applied to world line data supplied by the *subject* parameter described in the preceding subsection. (If it is properly integrated into the programming language, inexactness can be applied fairly easily to most uses of numeric values.) This allows, for example, correctly describing a human as having dwelt exclusively within 20 km of the Terran geoid, without having to research the subject's travels in detail. (This particular fuzzy world line should, like the geocenter, be available as a standard named object. This is vital to forestall the falsely-precise use of the geocenter as a proxy for non-astronautical human world lines.)

Qualitative Uncertainty

In addition to measurement error, there is another way that point-in-time values can display uncertainty: the use of sources of doubtful veracity. This is usually not an issue in scientific computation, but is important when dealing with timezones. Generally, the span of a timezone can be divided into three periods: distant past (before about 1980), recent past (since about 1980), and the future. The timezone's behavior in the recent past is usually well attested, with firsthand re-

ports from concerned parties and often good documentation. The distant past is more problematic, with variable documentation quality and no possibility of making fresh observations. Sources may conflict, and information can be entirely missing. The future is a different case again: the truth is not yet determined, and one can only guess at what the behavior will be, usually by supposing that recent behavior patterns will continue indefinitely. It's a near-certainty that this best guess is false in some respect, but it's correct (and, where false, approximately correct) over sufficiently much future territory to be useful.

So when converting point-in-time values between timezones, the result can be exact in a numerical sense, but qualitatively uncertain. With respect to past timezone behavior, the comments in the Olson timezone database provide a remarkable compendium of uncertainties.* While this information is not presently represented in generated timezone data files, nor in the formal part of the source, it could quite easily be added there. Maintaining formal quality tags would be only a small additional effort, given that current (and good) practice is to discuss the data quality in the comments. A perusal of the Olson database also suggests a simple scale of certainty levels: **5** certain, well attested; **4** nearly certain, subject to a little doubt; **3** probably correct, sources dubious or conflicting; **2** likely specific value; **1** guesswork, not specifically attested; **0** no idea at all.

This kind of uncertainty can be represented in the API in a fairly simple way. Space constraints prevent showing all the code here. The core of it is a simple structure of value and certainty level. A value standing alone implicitly has the highest certainty level. Generic functions for arithmetic and related operations acquire methods that operate on the uncertainty structures: they produce an uncertain result, with the value resulting from applying the underlying operation to the uncertain values, and the minimum certainty level of all the operands. The "no idea at all" certainty level is a special case, because it precludes having any meaningful value: total uncertainty is represented by a distinguished object that does not carry any value, and any arithmetic operation on it returns the same total-uncertainty object. A few special cases must be coded: for example, a multiplication involving a zero yields a zero with the same certainty level as the input zero, no matter how uncertain may be the other multiplicand.

As with the interval structure used to represent quantitative inexactness, the structures for qualitative uncertainty should be contained in the label slot of the `<time-value>` class. Where both forms of uncertainty are present, the quantitative inexactness is logically nested within the qualitative uncertainty.

The data underlying a time scale conversion logically carry the same kind of uncertainty metadata, so a conversion may produce a point-in-time value of lower certainty level than the input point-in-time. The conversion process doesn't have to literally use the generic qualitative-uncertainty mechanism for its data; it suffices that the conversion overall has some conception of its quality.

Aside from human-generated data such as timezone offsets, these qualitative certainty levels could be applied to different kinds of estimate of quantitatively-uncertain parameters. For example, while UT1 − TAI values for the past can be interpolated between actual measurements, UT1 − TAI values for the future are at best extrapolated using models of Earth rotation. Interpolated values between well-recorded measurements, when correctly labeled for quantitative uncertainty, qualify for qualitative certainty level 5. Extrapolations of future values qualify for lower certainty levels, depending on the models used. By this means, the single time conversion API can encompass the needs of very different conversion scenarios. Traditionally, an API that provides high-quality data

*⟨`http://www.iana.org/time-zones`⟩.

from interpolating measurements would have to signal an error when asked about an area lacking measurements suitable for interpolation; a caller wanting the lower-quality extrapolation would have to use a different API that explicitly provides lower-quality results.

The qualitative uncertainty model also supplies a natural way to implement the default *subject* parameter. When introduced above, it was described as "declin[ing] to supply any specific information". The most trivial way to decline to supply information is to signal an error when asked for it. A more sophisticated way would be to supply the appropriate information structure, filled in with certainty-level-0 objects (indicating total uncertainty) in place of actual numeric data. This makes the no-specific-subject value less of a special case. This has real implications for the time scale conversion code. In the error-signaling implementation, the conversion code must specially detect when the choice of world line doesn't matter, and avoid asking for world line data in that case. Using the uncertainty mechanism, the conversion code can freely retrieve world line data that it will ultimately either discard or multiply by zero. (Without further sophistication it's still not totally free in its mode of computation: if a world line parameter would be canceled out by a more complex operation, such as subtracting it from itself, the straightforward uncertainty mechanism won't detect that it can produce an exact result.)

Multiple Knowledge Sources

In several cases touched upon so far, the same conversion could produce different results in different circumstances. Fundamentally, any parameter that must be measured by experiment is subject to better measurement or estimate at a later date, so using the best knowledge available at different dates will yield different results. Often, a parameter is impossible to measure at all before some date: for example, the value of UT1 − TAI for some date in the future cannot be measured, but can only be estimated by extrapolation. In the extreme case, information can be totally lacking before some date. For example, whether UTC (assuming that it continues with leap seconds) will observe a leap second in December 2023 is a total unknown at present, ten years in advance of the event, but will be known with certainty by the time December 2023 arrives.

APIs so far mostly ignore this sort of issue, and just produce different results under different circumstances. This impedes formal analysis, caching, parallelization, and all manner of program manipulation that depends on programs being well behaved. This impediment can be managed by making the knowledge base an explicit parameter to the conversion operation:

`(convert-time` *src-time tgt-scale subject knowledge*`)` → *tgt-time*

The *knowledge* parameter encapsulates such things as a set of EOP measurements that may be consulted and a version of the timezone database. If this is performed fully, the conversion function is now a pure function of its parameters.

Most programs don't need sophisticated control over the knowledge base applied to their computations. It will usually suffice to acquire a single knowledge base for an entire program run. This is readily represented as a non-pure function:

`(current-knowledge)` → *knowledge*

The use of the best current knowledge can be further simplified by making it implicit in a `convert-time` call that doesn't include an explicit knowledge base parameter. Where such defaulting is used, it is important to distinguish between using the best knowledge available at the time

of each operation and using a single knowledge base for all operations in one program run (or all within some other scope). The former means that new knowledge is implicitly used as it becomes available, whereas the latter means that operations always produce consistent results. It is not possible to get both of these benefits from a single default behavior. A useful compromise is to allow the explicit setting (especially with dynamic scope) of the knowledge base that is to be used by default.

Because of the multiplicity of time scale conversion mechanisms that may be invoked, and the diversity of underlying data sets that they require, a general-use knowledge base object can't know in advance what knowledge it encapsulates. It must rather behave as an extensible cache for whatever data a conversion function wishes to retrieve. Some more specialized situations call for knowledge objects that do contain data a priori; for example, an embedded system (such as on a spacecraft) that has no on-demand network access, or a debugging operation that requires reproducing a particular conversion operation repeatedly.

Computation with Time

Most interesting computations on time values amount to simple arithmetic on the time labels applied by some time scale. The subtlety comes in selecting the correct time scale in which to perform the arithmetic. For example, advancing a UTC time value by ten seconds, an operation that is affected by the location of leap seconds, is best done in TAI:

```
(define (advance-by-seconds augend-value addend-secs)
  (convert-time
    (make-time-value TAI
      (make <mjd-label> #:mjd
        (+ (time-label-mjd
             (time-value-label
               (convert-time augend-value TAI)))
           (/ addend-seconds 86400))))
    (time-value-scale augend-value)))
```

No explicit mention of leap seconds is required here, and in fact in the form shown the function is not specific to UTC. If there is uncertainty about the scheduling of leap seconds, it will be incorporated into the result automatically by the `convert-time` operations and, if necessary, propagated through the arithmetic operations.

There is therefore little value in the API providing specific functions for high-level operations such as this. Many current time APIs have a profusion of such special-purpose functions, which somewhat obscures the key concepts while inevitably not actually handling all required operations. (This is particularly annoying in APIs that impede access to the underlying linear parameters that are needed to implement one's own time arithmetic.) APIs should focus on providing full access to the logical structure of time values, and *enabling* high-level time arithmetic rather than directly implementing it.

WHETHER TO USE LEAP SECONDS

Using Leap Seconds Generally

Should the use of UTC-with-leap-seconds continue? This question is actually meaningless, because UTC doesn't have a single purpose ("the use") about which one could make such a decision.

The meaningful replacement for this question is an array of questions of the form "should UTC-with-leap-seconds be used for application X?".

Sophisticated time-scale users will, inevitably, make their own choice of time scale for each application. The combination of features that UTC offers—almost-constant frequency, contemporaneous availability to microsecond precision, each complete second ascribed to a particular day, days tracking Earth rotation—is good for some applications. So some users will, inevitably, desire a time scale that coordinates TAI and UT1 using leap seconds.

If leap seconds were not scheduled by IERS or other public authority, this wouldn't be a fatal impediment to applications desiring leap seconds. IERS would still publish its predictions of Earth orientation parameters, which is the difficult part of the scheduling job. Anyone can use those predictions to make a reasonable leap second schedule for their private leap-second-based time scale. Furthermore, the network effect, whereby applications with similar needs benefit from using the same time scale as each other, suggests that users of leap-second-based time scales would seek to agree on a common leap second schedule. Consensus would inevitably arise on some single authority or public algorithm to schedule leap seconds for those wanting the facility. In short, if UTC didn't exist it would be necessary to invent it.

A corollary is that if UTC-with-leap-seconds is formally discontinued then it will inevitably be reinvented. Undoubtedly, the consensus replacement would maintain continuity with the historical UTC-with-leap-seconds. (Provided that the name "UTC" is not fatally sullied by being applied to a new time scale that doesn't track UT, the replacement leap-second-based time scale might as well be *called* "UTC" too.) UTC-with-leap-seconds thus *cannot* be effectively abolished.

Using Leap Seconds in Civil Time

Should UTC-with-leap-seconds be used as the basis of civil time? Although this question is more meaningful than the one at the head of the previous subsection, it is so broad as to be largely moot. Asking the question in this form presupposes that there is some international authority that can make this decision for everyone. That's not how civil time works.

The International Meridian Conference of 1884, although it defined a "universal day" (and thus the time scale that is now called "Universal Time"), it stopped short of calling for UT, or the previously-proposed system of timezones consisting of integral hour offsets from UT, to be used for civil time.[18,19] Timezones would be much simpler today if some such proposal had prevailed.

Civil time is actually determined by the effects of thousands of political decisions made by regional authorities. Any similarity of civil time in different regions is an emergent property of the local decisions, not a fundamental constraint. This is illuminatingly recognized by the contributors to the Olson timezone database.* The database names each timezone after a single location within the zone, rather than after the zone's entire geographical extent, because zones frequently split, as regions that formerly had identical civil time go separate ways. Fundamentally, what the Olson database identifies as a timezone is not a distinct real-world entity, with its own ontological inertia, but exists only in the database's model of reality.

Just as civil timezone offsets are determined by fickle politicians, so too is the base civil time scale determined regionally, by legislation and by habit. Others have surveyed the variety of legislative positions.[20] Presently there is relative uniformity in that all countries use round offsets from UT,

*⟨http://www.iana.org/time-zones⟩.

but there's no systematic enforcement of such uniformity. It took nearly ninety years from the International Meridian Conference to reach the present state, the last country (Liberia) abandoning local mean time in 1972. If some TAI + *offset* were to be promoted as a replacement for UTC-with-leap-seconds, there would at best be a similarly piecemeal transition, likely also spanning decades.

Of course, civil authorities are already free to base a country's official civil time on TAI + *offset* rather than on UT. That this has not yet occurred suggests that the acceptance of TAI + *offset* is not presently where GMT was in 1884, but more like where GMT was in 1840. A transition to universal use of TAI + *offset* for civil time may be correspondingly even more protracted. In brief: GMT was adopted in practice on the British railway system starting in 1840, and became dominant on the railways in 1847. It was dominant in British civil practice by 1855, and was adopted legally in 1880. All of that process had already occurred by the time of the International Meridian Conference in 1884. The present public visibility of TAI + *offset* (without being picky about the particular offset) is that it's used internally in GPS, but in that usage it's mostly hidden from users. The present position of TAI + *offset* is thus roughly analogous to GMT's position at the end of 1840, being used by just one railway. There is a marked difference, however, in that whereas GMT's use expanded rapidly over the following few years, TAI + *offset* has been in its present position for around 25 years with no noticeable advance.

In summary, with global civil time not being a single entity, a decision on whether civil time should have leap seconds cannot be directly implemented. Proponents of TAI + *offset* can at best seek to persuade the many relevant authorities.

In the programming world, the decision on what civil time should ideally be is entirely irrelevant. Programs must deal with civil time, whatever that happens to be. The past forty years have been unusually straightforward in this regard, with it being viable to model civil time as always being a round offset from UT (if one ignores the sub-second difference between flavors of UT). The future is likely to require a more rigorous treatment of civil time, just as pre-1972 civil time already does.

PRACTICAL ISSUES AROUND LEAP SECONDS

This section will look at some practical problems around computer processing of leap seconds, which are capable of relatively simple amelioration.

Disseminating Leap Schedule to Machines

Most trivially, there should be a canonical machine-readable statement of the leap second schedule. Although this seems an obvious issue, some subtleties appear to have escaped notice. There are already two machine-readable formats describing the leap second schedule, but neither is entirely satisfactory.

The older format for the schedule, a file named "`tai-utc.dat`", contains an explicit definition of each segment of UTC in terms of TAI.* (It covers the "rubber seconds" era of UTC as well as the present leap seconds.) It is human-readable, and also, by virtue of having a fixed layout, machine-readable. Its main fault is that it does not fully indicate the extent of its applicability. Each segment definition is marked with the date on which it begins to apply, and implicitly applies up to the start of the following segment. The last listed segment, therefore, has no indicated end date. Restating in terms of leap seconds: the file lists the dates of scheduled leap seconds, and so rules out the

*⟨`ftp://maia.usno.navy.mil/ser7/tai-utc.dat`⟩.

possibility of leap seconds occurring between the listed dates, but does not rule out leap seconds occurring on any date following the last scheduled leap.

The absence of a leap second is as important as its presence, in using UTC. When a Bulletin C is issued that states that there will be no leap second at the next opportunity, this is significant information that must be disseminated, but `tai-utc.dat` *does not change* to reflect this knowledge. The best, not entirely satisfactory, way to handle this omission is to assume that the instance of `tai-utc.dat` hosted by USNO would be updated at least N days in advance of any leap, so that a fresh copy taken from USNO can be taken to imply that there will be no leaps after the last listed until at least today $+ N$. This requires frequent referral directly to USNO, with caches confounding the process rather than being helpful. Furthermore, although there is a formal guarantee of advance notice of a leap second, this guarantee applies to Bulletin C rather than to `tai-utc.dat`, and there is no equivalent guarantee from USNO that would provide a canonical value for N.

The other machine-readable format for the leap second schedule, "`leap-seconds.list`", improves markedly on `tai-utc.dat` by including an explicit indicator of the extent of its applicability.*† It also includes an in-band checksum, in an attempt to catch file corruption, but the checksum mechanism is faulty. The file includes extensive human-readable commentary describing the format of the machine-readable parts. It excludes the commentary from the checksum computation, but is overzealous in doing so, with the result that it also excludes the semantically-significant delimiters and flags between the data values. Omission or transposition of some characters can therefore result in a file that represents a different leap schedule but still passes the checksum test. (Of course, this flawed checksum will still catch many cases of corruption. `tai-utc.dat` lacks any in-band checksum.)

Both leap schedule file formats are designed only to disseminate the complete leap schedule so far. (That is, the entire schedule from 1961 or 1972 up to the most recent scheduling decision.) It's foreseeable that the increasing length of the schedule will discourage the use of downloading the complete schedule as a means of updating systems that have most of the schedule already. It would therefore be nice to have a schedule file format that is capable of representing an excerpt of the schedule in isolation. A file extending from two years ago up to the latest scheduling decision would have bounded size, and would satisfy the bulk of updating requirements. (There would of course still have to be a complete schedule available.)

Neither leap schedule file has entirely clear status. Although the decisions they reflect emanate from IERS, the files are supplied by USNO and NIST respectively. Neither comes with any statement guaranteeing future maintenance. Users of the files are forced to guess and hope regarding how to use the files. This can be rectified merely by documentation.

In summary, considering the above together with more general received wisdom regarding file format design, it is recommended that a future leap schedule file:

- should state both ends of the range of days that it covers;
- should state on which days within its range and in which directions leaps in UTC occur;
- should not include human-readable comments;

*⟨`ftp://utcnist2.colorado.edu/pub/leap-seconds.list`⟩.

†D. L. Mills, *The NTP Timescale and Leap Seconds*, ⟨`http://www.eecis.udel.edu/ mills/leap.html`⟩.

- should have a minimum of syntactic options, ideally none at all;
- should begin with a distinctive sequence ("magic number") for in-band identification of the file format;
- if it includes an in-band checksum, should apply the checksum to all semantically-significant data;
- should avoid including unnecessary metadata;
- should have its format fully and formally described by an accompanying document;
- should be disseminated both in as-complete-as-possible form and in an updating-only form that omits data preceding two years ago;
- should visibly emanate from IERS;
- and should be maintained by a publicly-described procedure.

Disseminating Leap Schedule to Humans

Some of the difficulty with `tai-utc.dat` that is noted above also applies to Bulletin C. Specifically, it doesn't fully explicate which leap opportunities it rules out. The Bulletin's statement that "no positive leap second will be introduced at ..." would suffice if there were such a statement for every leap opportunity, but there is not. Bulletin C only speaks about the leap opportunities in June and December, but the defining standard is clear that there is a leap opportunity every month.[21] (June and December are merely preferred.) Similarly, the statement about "no positive leap second" ignores the option of a negative leap second.

While the wording of the Bulletin doesn't directly affect the operation of UTC-using programs, it does influence programmers' understanding of leap second scheduling, and so indirectly affects the logic that is built into UTC-using programs. It is particularly relevant to logic that interprets `tai-utc.dat`, due to the need for inference to cover the gap in that file's content.

These problems could be resolved by making a small number of changes to the wording of the Bulletin. Firstly, "no positive leap second" should be "no leap second" (thus excluding negative leaps as well as positive). Secondly, after the statement that there will or will not be a leap second at the end of June/December, there should be an additional statement covering the period until the next leap opportunity that IERS will consider. This would be something like "Subsequently there will be no leap seconds in UTC until at least the end of December 2019. The next Bulletin C will indicate whether there will be a leap second at the end of December 2019.". Finally, when listing UTC − TAI values, "until further notice" would be better as something like "until at least 2020 January 1 0h UTC".

Better Leap Scheduling

A major impediment to the correct implementation of leap seconds is their present rarity. Occurring every couple of years, there are few opportunities to test code against the genuine article. Code thus contains lurking bugs, that bite when a leap second eventually occurs.* Even when code

*B. Gondwana, "A story of leaping seconds", 3 July 2012, ⟨`http://blog.fastmail.fm/2012/07/03/a-story-of-leaping-seconds/`⟩.

is correct, there is a lack of confidence about it.[22] A system under active development may change a great deal between leap seconds, bringing much uncertainty about leap handling. In addition to the rarity providing little opportunity to develop institutional knowledge about leap second processing, the large gaps between leaps mean that much of what knowledge is developed may be lost through staff turnover before the next leap.

All of these issues, making each leap second disproportionately expensive to handle, have been proposed as arguments for abandoning leap seconds. With the understanding that leap seconds can't just disappear, a better reaction to these issues is to ameliorate them by making leap seconds reasonably frequent.

The present standard permits leap seconds to occur at the end of any month.[21] There would be plenty of practice with leap seconds if they occurred in most months. Such a scheme has been proposed previously.* Briefly, IERS could schedule a leap whenever doing so is compatible with satisfying the $|\mathrm{UT1}-\mathrm{UTC}|$ bound. For the most part there would be alternating positive and negative leaps.

The present $|\mathrm{UT1}-\mathrm{UTC}|$ bound of 0.9 s doesn't allow for there to be a leap *every* month, even given perfect prediction of UT1 − TAI. Compared to the present strategy of leaping as little as possible, leaping as much as possible consumes an additional 0.5 s of $|\mathrm{UT1}-\mathrm{UTC}|$ bound. There is a tradeoff to be made between using $|\mathrm{UT1}-\mathrm{UTC}|$ leeway to make leaps more frequent and using it to give greater notice of leaps.

Current practice is for leap second decisions to be announced approximately five months in advance of each June/December leap opportunity. As has been noted elsewhere, a longer notice period would ease update requirements, in particular by removing the need for some systems to update in the field at all. There is no particular threshold at which the notice period becomes sufficiently long to reap this benefit. Every marginal increase of notice period can be expected to make the difference for some group of applications. The standard permits greater notice, restricted only by the need to satisfy the bound on $|\mathrm{UT1}-\mathrm{UTC}|$.[21]

The standard's preference for leaping in June and December (and secondary preference for March and September) serves no technical imperative. It seems to be intended for administrative convenience. As noted in the previous subsection, it has led to some confusion about how leap scheduling operates. It also impairs the ability to closely track UT1, by effectively prohibiting the leap opportunities that would minimize the mean value of $|\mathrm{UT1}-\mathrm{UTC}|$. There has been a proposal to eliminate this counterproductive preference, replacing it with a strictly monthly scheduling cadence.† From a software point of view, there's no reason not to do this.

REFERENCES

[1] S. Newcomb, *The Elements of the Four Inner Planets and the Fundamental Constants of Astronomy*. Washington: US Government Printing Office, 1895.

[2] S. Newcomb, *A compendium of spherical astronomy*, pp. 114–117. Macmillan Company, 1906.

[3] H. Spencer Jones, "The rotation of the earth, and the secular accelerations of the sun, moon and planets," *Monthly Notices of the Royal Astronomical Society*, Vol. 99, May 1939, p. 541.

[4] R. Kelsey, W. Clinger, and J. Rees, "Revised^5 report on the algorithmic language scheme," *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, 1998, pp. 7–105.

*E. Davies, "Let's have more leap seconds", 11 Aug. 2003, ⟨http://www.ucolick.org/~sla/navyls/0197.html⟩.

†R. Seaman, *A Proposal to Upgrade UTC*, 2 Apr. 2003, ⟨http://iraf.noao.edu/~seaman/leap/⟩.

[5] Free Software Foundation, *The GOOPS reference manual*, 2006.

[6] G. Kiczales and J. d. Rivieres, *The art of the metaobject protocol*. Cambridge, MA, USA: MIT Press, 1991.

[7] G. L. Steele, *Common LISP: The Language*. Bedford, MA: Digital Press, second ed., 1990.

[8] American National Standards Institute and Information Technology Industry Council, *American National Standard for Information Technology: programming language — Common LISP*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1996. Approved December 8, 1994.

[9] International Astronomical Union, "B1.8: Definition and use of Celestial and Terrestrial Ephemeris Origins," *Resolutions Adopted the 24th General Assembly*, Aug. 2000.

[10] M. Allison and M. McEwen, "A post-Pathfinder evaluation of aerocentric solar coordinates with improved timing recipes for Mars seasonal/diurnal climate studies," *Planetary and Space Science*, Vol. 48, 2000, pp. 215–235, 10.1016/S0032-0633(99)00092-6.

[11] L. Bettini, S. Capecchi, and B. Venneri, "Double Dispatch in C++," *Software—Practice and Experience*, Vol. 36, No. 6, 2006, pp. 581 – 613.

[12] J. Boyland and G. Castagna, "Parasitic Methods: An Implementation of Multi-Methods for Java," *Proceedings of OOPSLA '97: Object-Oriented Programming Systems, Languages and Applications*, New York, ACM Press, Oct. 1997, pp. 66–76.

[13] B. Foote, R. E. Johnson, and J. Noble, "Efficient Multimethods in a Single Dispatch Language," *Proceedings of the European Conference on Object-Oriented Programming*, 2005.

[14] Bureau International des Poids et Mesures, Sevres, France, *Circular T 304*, 13 May 2013.

[15] D. L. Mills, "Internet time synchronization: the Network Time Protocol," *IEEE Transactions on Communications*, Vol. 39, Oct. 1991, pp. 1482–1493.

[16] International Astronomical Union, "B1.3: Definition of Barycentric Celestial Reference System and Geocentric Celestial Reference System," *Resolutions Adopted the 24th General Assembly*, Aug. 2000.

[17] B. Hayes, "A Lucid Interval," *American Scientist*, Vol. 91, No. 6, 2003, pp. 484–488.

[18] S. Fleming, "Uniform Non-Local Time (Terrestrial Time)," Ottawa, Canada, 1876.

[19] *International Conference Held at Washington for the Purpose of Fixing a Prime Meridian and a Universal Day, October 1884: Protocols of the Proceedings*, US Government, 1884.

[20] J. H. Seago, P. K. Seidelmann, and S. Allen, "Legislative Specifications for Coordinating with Universal Time," *Decoupling Civil Timekeeping from Earth Rotation—A Colloquium Exploring Implications of Redefining UTC*, American Astronautical Society, 2011.

[21] International Telecommunications Union, Radiocommunication Bureau, Geneva, *Recommendation ITU-R 460-6 (2002), Standard-frequency and time-signal emissions (Question ITU-R 102/7)*, 2002.

[22] D. Glicksberg, "Risks of First UTC Leap Second in 7 Years," *Forum on Risks to the Public in Computers and Related Systems*, Vol. 24, 28 Aug. 2005.

APPENDIX: SUPPORTING SCHEME CODE

Generic Comparison

Mathematically, a partial or total ordering is fully defined by a single comparison operator, traditionally the one corresponding to less-than-or-equal. Scheme/GOOPS does not implement the corresponding behavior for its generic comparison functions. It is easily added. Given a method for the <= generic function, the following methods implement all the remaining Scheme comparison operators for a total ordering.

```
(define-method (>= a b) (<= b a))
(define-method (< a b) (and (<= a b) (not (<= b a))))
(define-method (> a b) (and (not (<= a b)) (<= b a)))
(define-method (= a b) (and (<= a b) (<= b a)))
(define-method (min a b) (if (<= a b) a b))
(define-method (max a b) (if (<= a b) b a))
```

The methods for >=, <, >, and = are also correct for any partial ordering, but the `min` and `max` methods are not. A partial ordering that is not total can make use of the predicate methods, but in addition to defining a <= method the implementor must also override the default `min` and `max` methods.

Post-Hoc Superclass Definition

The `insert-superclass!` function adds a newly-defined superclass to the inheritance list of a pre-existing subclass. (Or, equivalently, puts a pre-existing subclass into the group reified by a newly-defined superclass.) Its purpose is to allow method specialization on groups of classes where the group was not envisioned in advance of all the classes being defined.

```
(define (insert-superclass! sub super)
  (class-redefinition sub
    (make-class
       (cons super
             (filter
               (lambda (c)
                 (not
                   (memq c (class-direct-supers super))))
               (class-direct-supers sub)))
       (class-direct-slots sub)
       #:metaclass (ensure-metaclass
                      (map class-of (list sub super)) #f)
       #:name (class-name sub)
       #:environment (class-environment sub))))
```

Singleton Method Specializers

The `singleton` function provides a method specializer applying to a single object, rather than a class. A crude implementation of singleton method specializers, sufficient for the purposes of this paper, is merely

```
(define-class <singleton-class> (<class>))
(define (singleton v)
  (let ((cv (class-of v)))
    (if (is-a? cv <singleton-class>)
      cv
      (let ((sc (make-class (list cv) '()
                  #:metaclass <singleton-class>)))
        (change-class v sc)
        sc))))
```

A proper implementation, avoiding the violence that this does to the objects being specialized upon, must subclass the `<generic>` class to modify method selection. GOOPS does not currently support enough MOP to do this cleanly.