Formalizing Date Arithmetic and Statically Detecting Ambiguities for the Law

Raphaël Monat[1⋆], Aymeric Fromherz[2⋆], and Denis Merigoux[2]

[1] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[2] Inria Paris, France
{raphael.monat,aymeric.fromherz,denis.merigoux}@inria.fr

Abstract. Legal expert systems routinely rely on date computations to determine the eligibility of a citizen to social benefits or whether an application has been filed on time. Unfortunately, date arithmetic exhibits many corner cases, which are handled differently from one library to the other, making faithfully transcribing the law into code error-prone, and possibly leading to heavy financial and legal consequences for users.
In this work, we aim to provide a solid foundation for date arithmetic working on days, months and years. We first present a novel, formal semantics for date computations, and formally establish several semantic properties through a mechanization in the F⋆ proof assistant. Building upon this semantics, we then propose a static analysis by abstract interpretation to automatically detect ambiguities in date computations. We finally integrate our approach in the Catala language, a recent domain-specific language for formalizing computational law, and use it to analyze the Catala implementation of the French housing benefits, leading to the discovery of several date-related ambiguities.

Keywords: Verification, Semantics, Abstract Interpretation

1 Introduction

From filesystems to web servers, time representations are pervasive in modern computer systems. While several libraries and standards were proposed throughout the years, current well-established approaches such as Unix time [53] used in the standard C library or Windows' FILETIME [36] represent dates and time as a number of seconds or nanoseconds that have elapsed since an arbitrary date.

This approach is sufficient for many usecases, in particular when dates are only used for logging purposes, or for determining the chronology of two events. However, it does not permit more complex arithmetic, for instance the addition of months or years, that span a variable number of days. For these usecases, mainstream programming languages offer different libraries that adopt different conventions. For example, Python's `datetime` module [46] forbids the addition of months, while Java's `java.time` library [43] silently rounds invalid dates onto the largest pre-existing date, hiding ambiguous computations from programmers.

⋆ Equal contribution

Given the variety of libraries and behaviors across languages, programming with date arithmetic is thus highly error-prone, and developers' assumptions about how dates behave might vary from project to project. When developing systems whose correctness is critical and that heavily depend on date computations, such as expert legal systems that rule our social and financial lives, this issue becomes highly concerning. As an example, consider the following excerpt from Section 121 of the US Internal Revenue Code [25], which defines the "Exclusion of gain from sale of principal residence".

> In the case of a sale or exchange of property by an unmarried individual whose spouse is deceased on the date of such sale, paragraph (1) shall be applied by substituting "$500,000" for "$250,000" if such sale occurs not later than 2 years after the date of death of such spouse and the requirements of paragraph (2)(A) were met immediately before such date of death.

This paragraph differentiates between two cases, depending on whether a sale occurred *not later than 2 years* after a given date. While applying this paragraph is straightforward in most real-world cases, corner cases raise interesting questions. In particular, when considering leap years, what should be the result of adding two years to February 29th? When manually computing taxes, lawyers would be able to detect the ambiguity, and to reach a decision based on legal precedents. If handled automatically by a computer however, the computation may be done incorrectly; computing February 29 2004 + 2 years in Java using `java.time` would return February 28 2006, while performing the same computation using the `date` utility from Coreutils returns March 1 2006.

Similar computations are pervasive in expert legal systems; the corresponding regulations rely on them to determine whether a citizen is eligible to social benefits or a resident for tax purposes. Errors in such systems can have dramatic consequences; case in point, the incorrect implementation of Louvois, the former French military payroll system, led to several families either receiving over-payments that they had to reimburse years later, or incomplete paychecks totaling a few cents [42]. For such critical software, it is therefore paramount to provide clear semantics for date computations to avoid mistakes based on erroneous assumptions about a library's behavior. Additionally, such a semantics can form the basis for further analyses, paving the way for the automated detection of date-related ambiguities as part of the development process.

Unfortunately, while elegant in theory, a universal semantics for dates and date arithmetic would not be usable in practice; when possible ambiguities are identified in law texts, legislators oftentimes extend or modify the law itself to avoid them. For instance, article 641 of the French civil procedure code [30] specifies that, when adding a positive duration to a date to compute a deadline, the rounding, if needed, should go down. Such articles often have narrow application scopes; similar articles in other branches of the law might either leave rounding unspecified, or adopt a different convention. In the US, date computations when filing motions are heavily specified, however the complexity and amount of corner cases led to no less than 27 subsequent notes and amendments to provide clarifications [14]. Other regulations instead attempt to escape ambiguities due

to month or year additions by reducing such computations to a nonambiguous number of days. Such regulations heavily vary depending on the country and the branch of law considered: acts from the Council of European Communities consider that a month should be treated as 30 days [15], while the Indian Supreme Court took the opposite approach, enacting that the duration of a month for customs purposes is variable [4]. To enable their adoption in a variety of contexts, date libraries therefore require their semantics to be configurable by developers.

The lowest granularity of date arithmetic we focus on is the day level. Our literature review and communications with lawyers in different countries have indeed shown that this kind of date arithmetic is sufficient for the kind of tax and social benefits computations that are the core application target of Catala.

In this paper, we aim to provide a sound foundation for critical software relying on date computations, through the following contributions:

Formally Capturing Date Computations. We first present a formal semantics of date computations (Sec. 2). Our formalization relies on a base semantics, which is universal and does not specify a rounding mode but instead provides facilities to round on-demand. We leverage these facilities to derive a rounding-specific semantics for different rounding policies. We mechanize this semantics in the F* proof assistant, and prove several theorems establishing necessary conditions for, e.g., the monotonicity or associativity of computations (Sec. 3). As part of this mechanization, we also identify seemingly intuitive properties that do not hold in practice, and exhibit counter-examples.

Automatically Detecting Date Ambiguities. Building on the semantics, we define a notion of *rounding-insensitivity*, which captures that the result of evaluating a program's expression does not depend on the chosen rounding policy (Sec. 4). Aiming to automatically identify possibly harmful ambiguities, we then propose a new static analysis based on abstract interpretation [16] targeting this 2-safety hyperproperty. We implement our analysis in the Mopsa static analyzer [28, 29]. We show that with relational numerical abstract domains, our analysis enables precise reasoning. In addition, our implementation provides actionable counter-example hints which will help users understand why a given expression is rounding-sensitive.

Contribution to Date Arithmetic Libraries. To enable the adoption of this work in existing projects, we implement an OCaml library abiding by our formal semantics, which exposes common rounding modes, as well as an option to abort when ambiguous computations are detected. Our library is standalone and open-source, and easily integrable in OCaml developments. We also survey the behavior of mainstream date arithmetic libraries (Sec. 6), and provide litmus tests that can be used to easily understand how a library behaves with respect to date rounding.

Case Study: Integration in the Catala Language. To demonstrate the applicability of our approach in real-world programs, we replace previous han-

$$
\begin{array}{lll}
\text{date unit} & \delta & ::= y \mid m \mid d \\
\text{rounding mode} & r & ::= \uparrow \mid \downarrow \mid \bot \\
\text{values} & v & ::= (y, m, d) \mid \bot \\
\text{expressions} & e & ::= v \mid e \ +_\delta \ n \mid \mathrm{rnd}_r \ e \\
\text{period} & p & ::= (n_d, n_m, n_y)
\end{array}
$$

Fig. 1. Date expressions

dling of dates in the Catala language [34], a recent domain-specific language for formalizing computational law, by our library. We also extend the Mopsa [28, 29] static analyzer to support a subset of the Catala language, enabling us to analyze Catala programs for rounding-insensitivity. We evaluate our approach against an existing Catala implementation of the French housing benefits, and automatically identify several date-related ambiguities in the Catala model. This work is in the process of being upstreamed in the Catala compiler.

2 Formalizing Date Arithmetic

We start this section by presenting a base semantics for date computations, which does not explicitly specify a rounding policy to handle ambiguous dates. Dates expressions are presented in Fig. 1. Dates values are represented in the year-month-day format of the standard Gregorian calendar, where each component will be represented as an integer. We also include a $\bot$ element, which represents an error case. Date expressions consist of either date values, or of the application of one of the date operators. Date expressions also contain variables, however their treatment is straightforward and orthogonal to this work; we omit them as well as their associated environment in our presentation. Operators are of two kinds: the addition $+_\delta$ of n years, months, or days, where n is an integer, and the rounding rnd_r of a date. Our semantics supports three types of rounding: $\mathrm{rnd}_\uparrow$ rounds up the current date to the nearest valid date; $\mathrm{rnd}_\downarrow$ rounds down, and $\mathrm{rnd}_\bot$ raises an error if the current date is invalid. A period is a triple of relative integers, respectively representing the numbers of days, months and years.

We now define a formal semantics for evaluating expressions. We start by describing the semantics of date addition, presented in Fig. 2. To match standard date formats, we start counting at 1 for valid days and months; to simplify the presentation, we will often represent months using their name instead of their number (e.g., `Jan` instead of 1). Our semantics is designed to preserve the following invariant: assuming the date on the left is initially valid, any non-ambiguous computation will return a valid date. When the computation is ambiguous, the resulting date is between the largest smaller and the smallest larger valid date.

Our semantics is defined recursively. Consider for instance the addition of a number of days n. If n is small enough to remain in the same month and year, we are in the terminal case and the rule Add-Days applies. The first premise of the rule ensures that the date is initially valid. It relies on an auxiliary function nb_days, omitted for brevity, which computes the number of days for a month in a given year (e.g., 31 for January, and 28 or 29 for February depending

$$\textsc{Add-Year}\ \frac{}{(y,m,d) +_y n \to (y+n,m,d)} \qquad \textsc{Add-Month-Under}\ \frac{m+n<1}{(y,m,d) +_m n \to (y-1,m,d) +_m (n+12)}$$

$$\textsc{Add-Month}\ \frac{1 \leq m+n \leq 12}{(y,m,d) +_m n \to (y,m+n,d)} \qquad \textsc{Add-Month-Over}\ \frac{m+n>12}{(y,m,d) +_m n \to (y+1,m,d) +_m (n-12)}$$

$$\textsc{Add-Days-Over}\ \frac{1 \leq d \leq \text{nb\_days}(y,m) \qquad d+n > \text{nb\_days}(y,m)}{(y,m,d) +_d n \to ((y,m,1) +_m 1) +_d (n - (\text{nb\_days}(y,m) - d) - 1)}$$

$$\textsc{Add-Comp}\ \frac{e \to e'}{e +_\delta n \to e' +_\delta n} \qquad \textsc{Add-Days-Under1}\ \frac{1 < d \leq \text{nb\_days}(y,m) \qquad d+n \leq 0}{(y,m,d) +_d n \to (y,m,1) +_d (d-1+n)}$$

$$\textsc{Add-Days-Err1}\ \frac{d<1}{(y,m,d) +_d n \to \bot} \qquad \textsc{Add-Days-Under2}\ \frac{n+1 \leq 0 \qquad (y,m,1) +_m (-1) \to (y',m',d')}{(y,m,1) +_d n \to (y',m',1) +_d (n + \text{nb\_days}(y',m'))}$$

$$\textsc{Add-Days-Err2}\ \frac{d > \text{nb\_days}(y,m)}{(y,m,d) +_d n \to \bot} \qquad \textsc{Add-Days}\ \frac{1 \leq d \leq \text{nb\_days}(y,m) \qquad 1 \leq d+n \leq \text{nb\_days}(y,m)}{(y,m,d) +_d n \to (y,m,d+n)}$$

Fig. 2. Semantics for date addition

on the year). Otherwise, we either add a month (rule Add-Days-Over) or remove a month (rule Add-Days-Under2) and perform a new addition with an updated number of days. When the initial date is invalid, we return $\bot$ to avoid propagating large errors and maintain important properties about date semantics that we prove in Sec. 3. When composing additions, it might therefore be necessary to apply rounding operators presented later in this section to avoid $\bot$. One last point of interest in these semantics is the dissymmetry between the Add-Days-Over and Add-Days-Under-* rules. Since adding a number of days is never ambiguous, we wish to ensure that, assuming the initial date is valid, we never apply the Add-Days-Err1 or Add-Days-Err2 rules. To do so, when updating the month or year during day addition, we always go through an intermediate state corresponding to the first day of the month, which is always a valid day independently of the month and year. For brevity, we also omit several redundant error cases, where the current month does not belong to the interval $[1;12]$; these cases return $\bot$. Following standard notations, we will denote the transitive closure of our small-step semantics as $\xrightarrow{*}$.

The last step is now to define semantics for rounding, shown in Fig. 3. Compared to additions, the rounding semantics is simpler: if the date is already valid, any mode of rounding leaves the date unchanged (Round-Noop). Otherwise, rounding down (Round-Down) returns the last day of the current month, rounding up (Round-Up) returns the first day of the next month, while the

$$\textsc{Round-Err1}\ \frac{d < 1}{\mathrm{rnd}_r(y,m,d) \to \bot} \qquad \textsc{Round-Err2}\ \frac{d > \mathrm{nb\_days}(y,m)}{\mathrm{rnd}_\bot(y,m,d) \to \bot} \qquad \textsc{Round-Down}\ \frac{d > \mathrm{nb\_days}(y,m)}{\mathrm{rnd}_\downarrow(y,m,d) \to (y,m,\mathrm{nb\_days}(y,m))}$$

$$\textsc{Round-Noop}\ \frac{1 \leq d \leq \mathrm{nb\_days}(y,m)}{\mathrm{rnd}_r(y,m,d) \to (y,m,d)} \qquad \textsc{Round-Up}\ \frac{d > \mathrm{nb\_days}(y,m) \qquad (y,m,d) +_m 1 \xrightarrow{*} (y',m',d')}{\mathrm{rnd}_\uparrow(y,m,d) \to (y',m',1)}$$

Fig. 3. Semantics for date rounding

strict rounding mode (Round-Err2) raises an error. In all cases, if the day is initially negative, rounding returns $\bot$; we will prove in Sec. 3 that this never happens when starting from a valid date.

Separating additions and rounding has several benefits. Different use cases might require different rounding modes, and different ways of adding days, months, and years. For instance, when adding a period such as 1 year and 10 months, some settings might specify that months should be added first, or that rounding must be performed after adding months, and again after adding years; our formal semantics enables this flexibility.

The last remaining step is to define additions not just for individual days, months, or years, but for composite time periods. Building upon our semantics, we can define this generically for a rounding mode r as follows, and avoid the need for users to manually call rounding operators.

$$e +_r (y,m,d) ::= \mathrm{rnd}_r(((e +_y y) +_m m)) +_d d$$

One point of interest in our derived forms is that we only apply rounding after performing addition of years and months. Indeed, adding a year should be equivalent to adding 12 months. However, if we performed rounding after each operation, adding 1 year and 1 month to February 29 2020 with the rounding-up mode would return April 1, 2021 instead of Mar 29, 2021. We emphasize that, in cases where this behavior would be expected, defining derived forms corresponding to this semantics would be straightforward using our base semantics.

Based on this semantics, we can now formally define the notion of an ambiguous date expression in Definition 1.

Definition 1 (Ambiguous expression). *A date expression e is ambiguous if and only if $rnd_\bot(e) \xrightarrow{*} \bot$.*

Note that this intensional definition of ambiguity is equivalent to stating that the an expression e is ambiguous if and only if rounding e in different modes yields different dates.

While the semantics presented in this section focuses on the core, possibly ambiguous computations, our work also includes other non-ambiguous operators (omitted for brevity), e.g., to retrieve the first or last day of a given month. This

allows to encode a variety of patterns, for instance, the second-to-last day of a month by combining date arithmetic with the "last day of month" operator, or to rely on a preprocessing phase if months must be treated as 30 days [15]. Our semantics supports reasoning on computations mixing rounding modes.

3 Mechanizing Semantics

Building upon the semantics presented in the previous section, we now present several properties of interest related to date computations that we will rely upon when designing a static analysis in Sec. 4. As part of our contributions, we mechanize our semantics, related properties and their proofs inside the F* proof assistant [52].

3.1 Semantic properties

As part of our proof development, we separate semantic properties in two categories: properties established on the base semantics, valid for all derived forms, and properties derived on specific rounding modes. In many cases, proofs on derived forms can be performed efficiently by composing lemmas on base semantics, thus simplifying the proof effort. During development, we also encode our OCaml implementation of date computations and corresponding theorems into qcheck [54], a QuickCheck [13] inspired property-based testing framework for OCaml. We mostly used QuickCheck as a fast sanity check before spending time proving lemmas in F*. In particular, our initial intuition for several of the lemmas and theorems presented was often unreliable, omitting corner cases; we used QuickCheck to gain more confidence in our intuition before moving to F*. This encoding allowed us to automatically find most of the counter-examples presented in Sec. 3.2.

We start by proving that expressions in our semantics always evaluate to a value (possibly $\bot$), i.e., reduction is never stuck and it terminates.

Theorem 1 (Normalization). *For any date d, and any integer n, there exists a value v_δ such that $d +_\delta n \xrightarrow{*} v_\delta$.*

In addition to normalization, a useful property about our semantics is a characterization of valid computations: when using any of the non-abort rounding modes, an addition starting from a valid date will always return a valid date; the definition of validity is straightforward, but omitted for brevity. To prove it, we need the following properties on base semantics, which we prove by induction on the reductions.

Lemma 1 (Well-formedness of day addition). *For any valid date d, any integer n, and any value v, $d +_d n \xrightarrow{*} v \Rightarrow v \neq \bot$.*

Lemma 2 (Well-formedness of year/month addition). *For any valid date d, any integer n, any value v, and $\delta \in \{y, m\}$, we have $d +_\delta n \xrightarrow{*} v \Rightarrow v \neq \bot \wedge \mathrm{day\_of}(v) \geq 1$.*

Lemma 3 (Well-formedness of rounding). *For any date d such that $d \neq \bot$, any value v, and $r \in \{\uparrow, \downarrow\}$, we have $rnd_r\ d \xrightarrow{*} v \Rightarrow valid(v)$.*

We can now state the following theorem on the derived semantics.

Theorem 2 (Well-formedness). *For any valid date d, any period p, any value v, and $r \in \{\downarrow, \uparrow\}$, we have $d +_r p \xrightarrow{*} v \Rightarrow valid(v)$.*

We now present several theorems related to the monotonicity of the addition in our semantics. Date comparison is defined in the standard way, as the lexicographical order on (y, m, d). To simplify the presentation, we lift the comparison operators to operate on date expressions, defined as the comparison on the values obtained by evaluating the expressions.

Theorem 3 (Monotonicity). *For any dates d_1, d_2, for any period p, for $r \in \{\downarrow, \uparrow\}$, if $d_1 < d_2$, then $d_1 +_r p \leq d_2 +_r p$.*

A point of interest in this theorem is the discrepancy between bounds: while the bound in the premise is strict, the bound in the conclusion is loose. Unfortunately, a stronger version with strict bounds on both sides does not hold; for instance, two additions involving rounding down of April 30 and April 31 respectively yield the same result. To prove this theorem, we again need several intermediate lemmas operating on base semantics. First, we establish an equivalence between adding years and adding months. We then state and prove several monotonicity properties on the base semantics. The proof of Theorem 3 follows by direct application of these lemmas.

Lemma 4 (Equivalence of year and month addition). *For all date d, for all integer n, $d +_y n = d +_m (12 * n)$.*

Lemma 5 (Monotonicity of year and month addition). *For all dates d_1, d_2, for any integer n, for $\delta \in \{y, m\}$, $d_1 < d_2 \Rightarrow d_1 +_\delta n < d_2 +_\delta n$.*

Lemma 6 (Monotonicity of day addition). *For all valid dates d_1, d_2, for any integer n, $d_1 < d_2 \Rightarrow d_1 +_d n < d_2 +_d n$.*

Lemma 7 (Monotonicity of rounding). *For all dates d_1, d_2, for $r \in \{\downarrow, \uparrow\}$, $d_1 < d_2 \Rightarrow rnd_r(d_1) \leq rnd_r(d_2)$.*

Finally, we state the following lemma, which guarantees that rounding down will always return a smaller date than rounding up. Additionally, when the addition is not ambiguous, the two rounding modes return the same result.

Theorem 4 (Rounding).

1. *For all date d, for all period p, $d +_\downarrow p \leq d +_\uparrow p$.*
2. *For all date d, for all period p, $d +_\bot p \neq \bot \Rightarrow d +_\downarrow p = d +_\uparrow p = d +_\bot p$.*

We finally characterize the ambiguity of month addition, a property that we will need to prove the soundness of the static analysis presented in Sec. 4.

Theorem 5 (Characterization of ambiguous month additions). *For all valid date d, for all integer n, for all value v such that $d +_m n \xrightarrow{*} v$, we have $\text{nb\_days}(\text{year\_of}(v), \text{month\_of}(v)) < \text{day\_of}(v) \Leftrightarrow rnd_\bot(v) \xrightarrow{*} \bot$.*

3.2 Non-properties and counter-examples

We now present several seemingly intuitive and ideally useful properties about date semantics that do not hold in practice.

Non-Property 1 (Commutativity of addition) *For all date d, for all periods p_1, p_2, for all $r \in \{\downarrow, \uparrow\}$, we have $(d +_r p_1) +_r p_2 = (d +_r p_2) +_r p_1$*

Consider the case where $d =$ March 31, $p_1 = 1$ day, and $p_2 = 1$ month. When adding p_1 first and rounding down, the addition returns April 30, while the result when adding p_2 first will be May 1. Similar examples exist when rounding up, for instance, by setting $d =$ January 29 2023 , $p_1 = 30$ days, and $p_2 = 1$ month.

Non-Property 2 (Associativity of addition) *For all date d, for all periods p_1, p_2, for $r \in \{\downarrow, \uparrow\}$, we have $(d +_r p_1) +_r p_2 = d +_r (p_1 + p_2)$*

Consider the case where $d =$ March 31, $p_1 = 1$ month, and $p_2 = 1$ month. In all rounding modes, adding p_1 followed by p_2 will require rounding, ultimately yielding May 30 or June 1, while directly adding $p_1 + p_2$ returns May 31.

As the addition being associative and commutative is common among most datatypes, we emphasize that its invalidity for dates can be a source of confusion for programmers; common optimizations or rewritings of date computations in a seemingly equivalent way (e.g., replacing 1 month + 1 month by 2 months) can lead to different outcomes. However, these disparities are exclusively due to occurrences of rounding in computations. We thus aim to help programmers when handling date computations by proposing a static analysis that automatically detects when rounding might impact the evaluation of expressions.

4 A Static Analysis For Rounding-Insensitivity

In this section, we leverage our formal semantics to define a sound static analysis automatically verifying date computations programs. Our goal is to statically detect ambiguous computations, whose result depends on the chosen rounding mode. Indeed, when writing programs whose specification is the law, choosing the rounding mode arbitrarily is not a possibility; this would amount to a legal interpretation that exposes the administration operating the program to be challenged in court if the rounding mode is unfavorable to a user. The cost of bearing the responsibility for making technical regulatory choices for administration personnel has been documented by Torny [55].

A naive approach would be to flag any program which contains an ambiguous addition. However, this solution can be overly restrictive: computations can be ambiguous while having no impact on the final outcome of the program. Consider for example the expression `d + 1 month <= March 15 2023`. If no rounding happens when adding `d` and `1 month`, then the expression is obviously safe. Otherwise, we notice that the rounding may only happen to yield the last day of a month, or the next day of the upcoming month. In both cases, comparing

```
1  date current = random_date();
2  date birthday = random_date();
3  date intermediate = birthday + [2 years, 0 months, 0 days];
4  date limit = first_day_of(intermediate);
5  assert(sync(current < limit));
```

Fig. 4. Example extracted from Catala code modeling the French housing benefits

this result with a date in the middle of a month is thus safe. Instead, we consider a more interesting property called *rounding-insensitivity*, capturing that the evaluation of an expression is the same for both rounding modes.

At a high-level, our analysis works by tracking constraints over the day, month, and year of a date, through the YMD domain (Sec. 4.1). The YMD domain is fully parametric in a numerical abstract domain, and works by translating date constraints into numerical constraints. We discuss the choice of numerical abstract domains in Sec. 4.2, in order to obtain the best precision in the presence of linear constraints and unconstrained dates. We analyze the computations with both rounding modes and compare the result to decide rounding-insensitivity, which is a 2-safety hyperproperty. We explain how we lift the YMD domain to these double computations in Sec. 4.3. We implemented our analysis within the Mopsa static analysis platform [28, 29], described in Sec. 4.4. We have taken special care in ensuring that actionable counter-examples can be generated in Sec. 4.5, paving the way for use by non-experts.

We think that abstract interpretation hits a sweet spot to perform this analysis. Its full automation makes it usable by non-specialists, especially with the provided counter-example hints. It allows to derive tailored approximations thanks to Th. 5. The current definition of date addition is recursive and there are non-linear arithmetic constraints involved, which does not work well with SMT.

We use as a motivating example the program shown in Fig. 4. This program has been extracted from a Catala code snippet used to formalize the French housing benefits [33, Sec. 3.1]. We will provide more details on Catala and the extraction to date programs in Sec. 5. In this program, we pick two arbitrary, unconstrained dates, perform a date-duration addition of two years, and project the resulting date onto the first day of its month. The assertion at line 5 expresses the rounding-insensitivity of the comparison between an arbitrary, unconstrained date and the computed date.[3] The `sync` predicate, formally defined in Sec. 4.3, holds if and only if the evaluation of its expression in both rounding modes yields the same result, meaning that the expression is rounding-insensitive.

The programs we consider in this section are written in a standard, toy imperative language whose grammar is presented in Appendix A.

4.1 The YMD domain combinator

The YMD domain translates constraints on the year, month and day of a date into numerical constraints over three integer variables. These numerical constraints are handled by a numerical abstract domain, described in Definition 2.

[3] Here `sync(current < limit)` could be reduced to `sync(limit)`. However our analysis will not need it, and will be able to provide counter-example hints also targeting the values of `current`, improving readability of the output.

$$\text{dates\_dom} : \begin{cases} (\mathcal{V} \to \mathbb{Z}) \to \mathcal{P}(\mathcal{V}) \\ \quad \rho \quad \mapsto \{v \mid \text{year}(v), \text{month}(v), \text{day}(v) \in \text{dom}(\rho)\} \end{cases}$$

$$\gamma_{\text{YMD}} : \begin{cases} \mathcal{N}^\# \to \mathcal{P}(\mathcal{V} \to \mathcal{D}) \\ n^\# \mapsto \bigcup_{\rho \in \gamma_{\mathcal{N}}(n^\#)} \{e \mid \text{dom}(e) = \text{dates\_dom}(\rho) \wedge \forall v \in \text{dom}(e), e(v) = (y, m, d) \\ \qquad \wedge \text{valid}(y, m, d) \wedge y = \rho(\text{year}(v)) \wedge m = \rho(\text{month}(v)) \wedge d = \rho(\text{day}(v))\} \end{cases}$$

Fig. 5. Concretization of the YMD domain

The YMD domain can be seen as a domain combinator, or a functor relying on a numerical abstract domain – we will discuss the chosen instantiation in Sec. 4.2. This domain works at a fixed rounding mode.

Definition 2 (Numerical abstract domain). *In the following, a numerical abstract domain is a lattice $\mathcal{N}^\#$ on which the following operations are defined:*
- *The assignment,* `assign`*, between a variable and an expression in a given abstract environment yields another abstract environment.*
- *The boolean filtering of a state,* `assume`*, filters an abstract environment to enforce that a boolean expression holds.*

This domain is further defined by a concretization function $\gamma_N : \mathcal{N}^\# \to \mathcal{P}(\mathcal{V} \to \mathbb{Z})$ mapping numerical abstract environments to a set of concrete integer environments it represents. We assume the numerical abstract domain is sound.

Given a date variable v, the YMD domain will create new auxiliary (or ghost) variables $\text{year}(v), \text{month}(v), \text{day}(v)$, which do not exist in the original program but simplify reasoning. This is an approach we borrow from the deductive verification community, and that has been used in static analyses both in the work of Chevalier and Feret [12] as well as in Mopsa.

We provide a formal definition of the concretization, which defines the meaning of the YMD domain, and illustrate it on an example.

Definition 3 (Concretization of the YMD domain). *The concretization of the YMD domain is formally defined in Fig. 5. It explains how an abstract numerical environment $n^\# \in \mathcal{N}^\#$ can be interpreted into a set of date environments $e \in \mathcal{V} \to \mathcal{D}$ mapping variables to dates. To construct these date environments, we first pick an integer environment $\rho \in \mathcal{V} \to \mathbb{Z}$ from the concretization of the numerical abstract domain $\gamma_N(n^\#)$. The date environments will have as domain definition the date domain of function ρ,* $\text{dates\_dom}(\rho)$*, which is the set of variables where auxiliary year, month and day variables are defined in ρ. For each of those variables $v \in \text{dates\_dom}(\rho)$, $e(v)$ corresponds to the date defined by the auxiliary variables in ρ, provided that the date is valid.*

Example 1 (Concretization). Let us assume our numerical domain is a map from variables to intervals, and consists of the following state: $n^\# = \text{day}(d) \in [1, 31] \wedge \text{month}(d) \in [1, 12] \wedge \text{year}(d) = 2023$. In that case, the concretization is the set of date environments e defined on variable d such that $e(d)$ can be any valid

date of 2023. Thus, there is a date environment $e \in \gamma_{\text{YMD}}(n^{\#})$ such that $e(d) = (2023, 1, 31)$. However, there is no date environment such that $e(d) = (2023, 2, 29)$ and $e \in \gamma_{\text{YMD}}(n^{\#})$ because the date is invalid (2023 is not a leap year).

The YMD domain handles the following transfer functions:

- Accessors to the day, month or year number of a date. Given a date encoded as a variable v, these functions return the associated variable $\text{day}(v), \text{month}(v)$, $\text{year}(v)$ respectively.
- Projection of a date on the first day of the month: given a date encoded as a variable v, this function creates a new date having the same auxiliary month and year variables. The day auxiliary variable is set to 1. A similar operator working on the last day of the month can be defined.
- The main part of the YMD domain is the transfer function handling month addition and potential rounding originating from this addition. We define it below, argue it is sound, and illustrate it on an example (in Sec. 4.2). As we have proved in Lemma 4, additions on years and months can be reduced to additions on months. Our current, potentially ambiguous, real-world examples taken from legislative code do not use day addition; as it is never ambiguous, we thus do not currently implement it. Given its similarity to month addition, we do not foresee any technical difficulty doing so.
- The YMD domain also provides a transfer function to compare two dates. It is induced by the lexicographic definition of concrete date comparisons and partitions the results to improve the precision.

Transfer function for month addition. We provide a simplified OCaml implementation for the month addition transfer function in Fig. 6. The transfer function takes as parameter a `date`, represented as a variable; a concrete number of months; an input abstract state; and a rounding mode chosen for date computations. It will return a case disjunction[4] of type `cases`: a list of `case`, each consisting in an expression and an abstract state. We start by defining `day, month, year`, which are expressions representing the day, month and year number of `date` through auxiliary variables. The resulting month and year are computed through non-linear expressions. Similarly to the semantics, we encode months as integers to perform arithmetic operations, and start our numbering at 1 for January. The transfer function performs a case disjunction to detect if date rounding will happen, following the characterization of ambiguous month additions (Th. 5). This case disjunction checks whether the day of the date is compatible with the number of days in the resulting month (and year, as February has one more day during leap years). This disjunction is encoded thanks to the `switch` utility, which takes as input an abstract state and a list of tuple of expressions and continuations. Given a tuple `(cond, k)`, the input abstract state is filtered to satisfy the expression `cond` (by delegation to the numerical

[4] These disjunctions can be seen as a partitioning of the abstract state. In this section we consider everything is partitioned to improve the precision. Our implementation supports limiting the number of partitions.

abstract domain). The resulting abstract state is fed to the continuation, which yields a case. The cases we encounter during the addition are:
Rounding to 29 Feb. of a leap year. If the resulting month is February of a leap year, and the current day number is greater than 29, we will have to perform date rounding. We do so using the auxiliary `round` function. Depending on the rounding mode, it either chooses the provided date, or the first of the month afterwards. This date is then returned in its corresponding abstract state using `mk_date`, whose implementation is not detailed.
Rounding to 28 Feb. of a non-leap year. Similar case omitted for brevity.
Rounding to a 30-day month. If the current day number is 31 but the resulting month has 30 days (i.e, it is either April, June, September or November), we also have to perform a rounding, either to the 30th of the resulting month, or the 1st of the month after.
Other cases. No rounding happens, the day number remains the same.

Note that `add_months`, `round` and `is_leap` define syntactic expressions, which will be delegated through `assign` and `assume` to the numerical abstract domain. The expressions at lines 6, 13, 14, 21–22, 26, 28, 30 are not directly evaluated: they will be interpreted by the `assume` of the numerical abstract domain during the evaluation of the `switch` function. The definition of the transfer function for month addition assumes the number of months to add is known as a concrete integer. This is not restrictive in practice: all programs we extracted from Catala in Sec. 5 only perform date-month addition with a concrete number of months.

The proof of soundness of the abstract month addition, is not formalized in F* and omitted for brevity. However, it is a direct application of the characterization of ambiguous month additions established in Th. 5, and proved formally in F*.

The analysis may refine constraints on a day, month or year auxiliary variable. These constraints could then entail new constraints on other auxiliary variables of the same date to represent only valid dates. This propagation phase is performed by the strengthening operator described below, which is sound as it only removes invalid dates, which are not taken into account by the concretization.

Strengthening operator. The strengthening operator enforces the following:
- If the month is February, the day is less than 30.
- If the month is April, June, September of November, the day is less than 31.
- If the date is February 29, we know the current year is a leap year. We enforce that the year number is divisible by 4, which is a necessary condition.

Comparison transfer function. The transfer function for date comparisons is `dates_lt` in Fig. 6; it encodes a lexicographic comparison.

4.2 Instantiating YMD with a combination of numerical domains

The YMD domain is fully generic in the numerical abstract domain it relies on to translate date constraints into constraints over integers. We describe how we

```
1  type case = expr * state
2  type cases = case list
3
4  let switch abs = List.map (fun (cond : expr, k : state -> case) -> k (assume cond abs))
5
6  let is_leap (y : expr) : expr = (y % 4 = 0 && y % 100 <> 0) || (y % 400 = 0)
7
8  let round (r : rounding) (d m y : expr) (abs : state) : case =
9    match r with
10   | RoundDown ->
11     mk_date d m y abs
12   | RoundUp ->
13     let succ_m = 1 + res_month % 12 in
14     let succ_y = y + res_month / 12 in
15     mk_date 1 succ_m succ_y abs
16
17 let add_months (r : rounding) (date : var) (nb_m : int) (abs : state) : cases =
18   let day = day_of date in
19   let month = month_of date in
20   let year = year_of date in
21   let res_month = 1 + (month - 1 + nb_m) % 12 in
22   let res_year = year + (month - 1 + nb_m) / 12 in
23   switch abs
24   [
25     (* Rounding to 29 Feb. of a leap year *)
26     day > 29 && res_month = Feb && is_leap res_year, round r 29 res_month res_year;
27     (* Rounding to 28 Feb. of a non-leap year *)
28     day > 28 && res_month = Feb && not (is_leap res_year), round r 28 res_month res_year;
29     (* Rounding to a 30-day month *)
30     day > 30 && is_one_of res_month [Apr;Jun;Sep;Nov], round r 30 res_month res_year;
31     (* No rounding *)
32     mk_true, mk_date day res_month res_year
33   ]
34
35 let dates_lt (d1 d2 : var) (abs : state) : cases =
36   switch abs
37   [
38     (year_of d1) < (year_of d2), mk_true;
39     (year_of d1) > (year_of d2), mk_false;
40     (year_of d1) = (year_of d2) && (month_of d1 < month_of d2), mk_true;
41     (year_of d1) = (year_of d2) && (month_of d1 > month_of d2), mk_false;
42     (year_of d1) = (year_of d2) && (month_of d1 = month_of d2)
43       && (day_of d1 < day_of d2), mk_true;
44     (year_of d1) = (year_of d2) && (month_of d1 = month_of d2)
45       && (day_of d1 >= day_of d2), mk_false;
46   ]
```

Fig. 6. Abstract transfer functions for month addition and date comparison

chose a combination of numerical abstract domains to get the best precision possible in the presence of non-linearity and unconstrained dates.

We initially started using intervals and congruences for our first tests. Due to its convexity, the interval domain was unable to precisely represent months where the day number may be rounded to 30 days during the date-month addition (line 30 of Fig. 6). Thus, we added a domain of powerset of integers (of size at most 4) to be precise enough for this usecase. When `month` is not a constant, the congruence domain will be unable to precisely represent the resulting month (line 21 of Fig. 6), and refine the potential values of `month` given constraints on `res_month`. This situation happens often in our evaluation; it is shown in our motivating example. We resolved this precision issue by switching from the congruence domain to the relational, linear congruence domain [5]. We also added the polyhedra domain [17] to keep track of equalities between different day,

month and year variables, which happens during analyses on programs with unconstrained dates, as we will show in the upcoming examples.

Our current numerical abstract domain is a reduced product between grids, polyhedra, intervals, and a bounded powerset of integers. The relational domains rely on the Apron library [27]. The approximation of non-linear computations is performed through linearization techniques [37].

Example 2. Let us consider the program below picking an arbitrary, unconstrained date `d` and then adding one month to `d`. We illustrate the different cases of the transfer function `add_months` in this case, assuming we round down.

```
date d = random_date(); date d2 = d + [0 years, 1 months, 0 days];
```

Rounding to 29 Feb. of a leap year. In the first case of the transfer function, the numerical domain is able to deduce from the expression `day > 29 && res_month = Feb` that the day of `d` is either 30 or 31, and the month is January. In the rounding down mode, `d2` is thus February 29th. The relational domain additionally expresses that $\text{year}(d) = \text{year}(d2)$.
Rounding to 28 Feb. of a non-leap year. Similar case, omitted for brevity.
Rounding to a 30-day month. The numerical abstract domain infers that `d` represents the 31st of March, May, August or October, tracked thanks to the bounded set of integers domain. As we round down, we deduce that the day of `d2` is 30, and $\text{month}(\texttt{d}) \in \{\texttt{Apr}, \texttt{Jun}, \texttt{Sep}, \texttt{Nov}\}$. In that case, the relational domain can also infer that $\text{year}(d) = \text{year}(d2)$, as `m / 12` will always be zero.
Other cases. In the last case, the intervals and powerset domains cannot express interesting constraints on `d` and `d2`. The relational domains are however able to capture key relations:

- The day does not change as there is no rounding: $\text{day}(\texttt{d}) = \text{day}(\texttt{d2})$.
- Thanks to the grids domain [5] we can infer linear relations modulo a constant, and thus that the month of `d2` is the month after `d`, even if the year changes: $\text{month}(\texttt{d2}) \equiv_{12} \text{month}(\texttt{d}) + 1$, where $\equiv_{12}$ denotes congruence modulo 12. Note that since $\text{month}(\texttt{d2})$ is not a constant, the non-relational congruence domain is not sufficient to express this relation.
- The year number may be the same, or the successor provided that the month of `d` is December. We lose a bit of precision, as the last month always creates a year increase in the concrete.
  $12\,\text{year}(\texttt{d}) + \text{month}(\texttt{d}) \leq 12\,\text{year}(\texttt{d2}) + 11 \wedge 12\,\text{year}(\texttt{d2}) \leq 12\,\text{year}(\texttt{d}) + \text{month}(\texttt{d}) + 1$

Example 3 (Addition and strengthening). We use our running example from Fig. 4, and show what the date addition and the strengthening operator yield for dates `birthday` and `intermediate`. In this example, we assume the dates are rounded up. As we add two years to `birthday`, two of the four cases described in the month addition previously presented will not apply; we omit them below.
Rounding to 28 Feb. of a non-leap year. In that case, `birthday` is a Feb. 29th, and `intermediate` rounds up to March 1st. We additionally know that $\text{year}(\texttt{birthday}) + 2 = \text{year}(\texttt{intermediate})$. The strengthening ensures that $\text{year}(\texttt{birthday})$ is divisible by 4.

No rounding. The day and month numbers of `birthday` and `intermediate` are equal. The year condition is similar to the one provided in Ex. 2.

Example 4 (Comparison). Let us continue with our running example, assuming we are focusing on the partition where `intermediate` has been rounded up to March 1st (as shown in Ex. 3). In that case, `limit` is equal to `intermediate`. Assuming the comparison `current < limit` holds, we have three different cases, described by the line number in Fig. 6. Line 38 yields year(`current`) $<$ year(`limit`). Line 40 enforces year(`current`) $=$ year(`limit`), month(`current`) $<$ month(`limit`), so month(`current`) $\in \{$`Jan`, `Feb`$\}$. Line 42 yields that the year and month numbers of `current` and `limit` are the same and day(`current`) $<$ day(`limit`). This last case is impossible given that $1 \leq$ day(`current`) ≤ 31 and day(`limit`) $= 1$.

4.3 Lifting to both rounding modes

The YMD domain operates at a given, fixed rounding mode. In this section, we leverage the YMD domain to perform date computations in both rounding modes and thus prove rounding-insensitivity. This lifting is inspired by Delmas et al. [21], who analyze product programs to prove endianness portability of C programs. Here, we keep the product of programs implicit: only the rounding mode changes between the two executions we will consider.

We start by explaining how the concrete semantics are lifted from a single rounding mode to both. We assume we have a semantics of expressions (respectively statements) $\mathbb{E}_r[\![expr]\!]$ (resp. $\mathbb{S}_r[\![stmt]\!]$) parameterized by a date rounding mode $r \in \{\uparrow, \downarrow\}$. They take as input sets of environments ($\mathcal{E} = \mathcal{V} \to \text{Val}$) mapping variables to values (which are either integers or dates), and produce values (resp. environments).

$$\mathbb{E}_r[\![expr]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\text{Val}) \qquad \mathbb{S}_r[\![stmt]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$$

We define in Fig. 7 the concrete semantics evaluating expressions and statements over both rounding modes, written respectively $\mathbb{E}_\updownarrow[\![expr]\!]$ and $\mathbb{S}_\updownarrow[\![stmt]\!]$. We do not delve into the details of product programs, which are defined in the work of Delmas et al. [21]. In this semantics, the state is now duplicated: $\mathcal{D} = \mathcal{E} \times \mathcal{E}$. We ensure that random operations return the same value in both rounding modes, to avoid spurious desynchronizations. The sync predicate returns true if and only if the expression evaluates to the same values in both rounding modes, capturing the rounding-insensitivity of the contained expression. We use it in the programs we analyze to target the expressions we want to check, as we have already seen in Fig. 4. The evaluation of other expressions is performed pointwise on both rounding modes, and similarly for the assignments.

Definition 4. *An expression e is* rounding-insensitive *in a state d if and only if $\mathbb{E}_\updownarrow[\![sync(e)]\!](\{d\}) = \{(true, true)\}$. This property is encoded in programs by the statement* `assert(sync(e))`.

$$\mathbb{E}_{\updownarrow}[\![expr]\!] : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathrm{Val}^2)$$
$$\mathbb{E}_{\updownarrow}[\![\mathrm{random\_date}()]\!](D) = \{(d,d) \mid d \in \mathbb{Z}^3, valid(d)\}$$
$$\mathbb{E}_{\updownarrow}[\![\mathrm{sync}(e)]\!](D) = \bigcup_{(\rho_\uparrow,\rho_\downarrow)\in\mathcal{D}} \{(b_u == b_d, b_u == b_d) \mid (b_u, b_d) = \mathbb{E}_{\updownarrow}[\![e]\!](\rho_\uparrow,\rho_\downarrow)\}$$
$$\mathbb{E}_{\updownarrow}[\![expr]\!](D) = \bigcup_{(\rho_\uparrow,\rho_\downarrow)\in\mathcal{D}} \{(v_\uparrow, v_\downarrow) \mid v_\uparrow = \mathbb{E}_\uparrow[\![e]\!]\rho_\uparrow, v_\downarrow = \mathbb{E}_\downarrow[\![e]\!]\rho_\downarrow)\}$$
$$\mathbb{S}_{\updownarrow}[\![stmt]\!] : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$$
$$\mathbb{S}_{\updownarrow}[\![v = e]\!](D) = \bigcup_{(\rho_\uparrow,\rho_\downarrow)\in D} \{(\mathbb{S}_\uparrow[\![v = v_\uparrow]\!]\rho_\uparrow, \mathbb{S}_\downarrow[\![v = v_\downarrow]\!]\rho_\downarrow), (v_\uparrow, v_\downarrow) \in \mathbb{E}_{\updownarrow}[\![e]\!]\{\rho_\uparrow,\rho_\downarrow\}\}$$

Fig. 7. Concrete semantics over double evaluation of rounding modes

The abstract semantics mimics the concrete behavior, but works on a single abstract state instead of a set of concrete double states. The double state is represented by duplicating variables according to their rounding mode in the numerical abstract domain. A variable `x` is thus written $\uparrow$`x` (resp. $\downarrow$`x`) to represent the variable when the upper (resp. lower) rounding mode is used. This duplication is performed in a shallow fashion to improve usability: when performing an assignment `x = e`, if `e` evaluates into the same value in both rounding modes, the variable `x` will not be duplicated into the numerical abstract domain.

Example 5 (Rounding-sensitivity of the comparison). Back to our running example, we have shown so far how the YMD domain analyzes the program when rounding up (Ex. 4). Continuing with the same relational abstract domain, we show part of the abstract state in the partition focusing on rounding to Feb. 28 of a non-leap year in Eq. (1). In the rounding mode down, `intermediate` rounds to Feb. 28, and thus `limit` rounds down to Feb. 1st.

$$\begin{aligned}
&\mathrm{day}(\texttt{current}) \in [1,31], \mathrm{month}(\texttt{current}) \in [1,12], \mathrm{year}(\texttt{current}) \in [-\infty,+\infty]\\
&\mathrm{day}(\texttt{birthday}) = 29, \mathrm{month}(\texttt{birthday}) = \texttt{Feb}, \mathrm{year}(\texttt{birthday}) \equiv_4 0\\
&\uparrow\mathrm{day}(\texttt{intermediate}) = 1, \uparrow\mathrm{month}(\texttt{intermediate}) = \texttt{Mar}\\
&\downarrow\mathrm{day}(\texttt{intermediate}) = 28, \downarrow\mathrm{month}(\texttt{intermediate}) = \texttt{Feb}\\
&\downarrow\mathrm{year}(\texttt{intermediate}) = \uparrow\mathrm{year}(\texttt{intermediate}) = \mathrm{year}(\texttt{birthday}) + 2\\
&\uparrow\mathrm{day}(\texttt{limit}) = 1, \uparrow\mathrm{month}(\texttt{limit}) = \texttt{Mar}\ \downarrow\mathrm{day}(\texttt{limit}) = 1, \downarrow\mathrm{month}(\texttt{limit}) = \texttt{Feb}\\
&\downarrow\mathrm{year}(\texttt{limit}) = \uparrow\mathrm{year}(\texttt{limit}) = \mathrm{year}(\texttt{birthday}) + 2
\end{aligned} \tag{1}$$

We exhibit an abstract state where we cannot prove that the expression `current < limit` is rounding-insensitive. The static analysis will consider all cases in the comparison and the evaluation in both rounding modes. For the sake of presentation here, we only highlight one case. The date comparison operator between `current` and the rounded up version of `limit` yields a partition where the years are the same and the month number is less. This partition refines the abstract state above with the following constraints:

$$\mathrm{year}(\texttt{current}) = \uparrow\mathrm{year}(\texttt{limit}) \wedge \uparrow\mathrm{month}(\texttt{limit}) < \mathrm{month}(\texttt{current}) = \texttt{Mar} \tag{2}$$

Let us now consider the case where the comparison with the rounded down version of `limit` does not hold, when the years and months are the same but the days are not. We get the following additional constraints:

$$\begin{aligned}&\text{year}(\texttt{current}) =\downarrow\text{year}(\texttt{limit}) \wedge \text{month}(\texttt{current}) =\downarrow\text{month}(\texttt{limit}) = \texttt{Feb} \wedge \\ &\text{day}(\texttt{current}) \geq\downarrow\text{day}(\texttt{limit}) = 1\end{aligned} \tag{3}$$

Combining the constraints from Eqs. (2) and (3) on the abstract state from Eq. (1) gives the following result on `current`:

$$\text{year}(\texttt{current}) = \text{year}(\texttt{birthday}) + 2 \wedge \text{month}(\texttt{current}) = \texttt{Feb} \tag{4}$$

To summarize, our analysis has been unable to prove the rounding-insensitivity of the expression `current < limit`, in particular in the case of the abstract state presented in Eq. (1), refined with constraints from Eqs. (2) and (3). Thanks to partitioning and relational abstract domains, we know that the proof fails when `birthday` is a Feb. 29th (of a year y which is divisible by 4, a sound but not complete way to express it is leap). In that case, `intermediate` will either be Feb. 28th or March 1st of $y + 2$. This entails that `limit` will either be Feb. 1st or March 1st of $y + 2$. In the cases where `current` is a day of February of $y + 2$ (Eq. (4)), the comparison will effectively be rounding-sensitive.

The original program did not contain any constraints on `birthday` or `current`. Note that if we add in the program that the day of `birthday` is less than 28, our analysis is able to automatically prove the program to be rounding-insensitive.

4.4 Implementation

We implemented our approach in the Mopsa static analysis platform [28, 29]. Mopsa is able to analyze C, Python and multilanguage Python/C programs [40, 41, 44], to prove the absence of runtime errors, and to perform portability analysis of C programs [21]. We modified the front-end of a toy imperative language also available in Mopsa to analyze programs performing date arithmetic. We chose to extend this language for our analysis as we do not require advanced features from C nor Python. Thanks to Mopsa's modular architecture, we have been able to reuse iterators for intraprocedural analysis with little code changes.

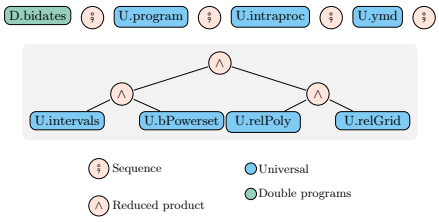

Fig. 8. Date analysis configuration

The configuration used by Mopsa for our analysis is illustrated in Fig. 8. The "D.bidates" domain corresponds to the abstract domain and transfer functions described in Sec. 4.3. The "U.ymd" domain is the YMD domain (Sec. 4.1). The last part enclosed in a gray box corresponds to the numerical abstract domain on top of which the YMD domain was built (Sec. 4.2).

```
5: assert(sync(current < limit));
                  ^^^^^^^^^^^^^^
Desynchronization detected: (current < limit). Hints:
↑month(limit) = 3, ↑day(limit) = 1, ↓month(limit) = 2, ↓day(limit) = 1,
↑month(intermediate) = 3, ↑day(intermediate) = 1,
↓month(intermediate) = 2, ↓day(intermediate) = 28,
month(birthday) = 2, day(birthday) = 29, month(current) = 2, day(current) = [1,29],
year(birthday) =[4] 0, year(current) = ↑year(intermediate) = ↑year(limit)
= ↓year(intermediate) = ↓year(limit) = year(birthday) + 2
```

Fig. 9. Mopsa's output on the running example

4.5 Generating counter-example hints

We have extended our implementation to provide counter-examples hints when a synchronization assertion cannot be proved safe. Given our usecase, it is paramount to provide meaningful feedback to users translating law articles into Catala code so they understand why their date computations might be ambiguous (Sec. 5). These hints are precise constraints on the considered program that may lead an expression to be rounding-sensitive. They are especially helpful to provide more precise date ranges for unconstrained dates that may affect rounding sensitivity. As our approach is incomplete, these hints may be spurious; we however did not encounter this issue in our case study on Catala programs.

This generation of counter-example hints is atypical for static analyses by abstract interpretation. This approach is permitted here by a simplified setting (variables are assigned once, and the abstract state is partitioned to ensure a high precision) and the use of powerful relational abstract domains. In a general setting with multiple variable assignments, joins and widenings, most approaches need to perform backward analyses [1, 38, 49].

This generation of hints works in two steps: it first starts by heuristically selecting the best partition of the abstract state. The YMD domain may partition the abstract state in order to keep the best precision. Our heuristic selects the partition with the highest number of desynchronized variables (meaning there has been significant roundings), and the highest number of auxiliary variables for days and months which are constants. The second step of the hint generation extracts the relevant constraints from the considered abstract state. This extraction starts by collecting all date variables defined in the program. For these variables, we evaluate the auxiliary day, month and year variables into intervals, and keep only intervals providing meaningful information (i.e., intervals strictly included in $[1, 31]$ for day variables, strictly included in $[1, 12]$ for month variables, and bounded intervals for year variables). We then project the relational abstract domain onto the set of auxiliary variables where no meaningful intervals has been extracted to provide linear relations for those. We show in Fig. 9 the exact, unedited output of the hints generated by Mopsa in the case of our running example and highlight their readability. They correspond exactly to the constraints previously described in Ex. 5.

5 Case Study: Application to Catala

This section highlights how the results and methods established in the previous section can be applied in the setting of legal expert systems, and more specifically

within Catala [34], a recent domain-specific programming language designed to be understandable by lawyers and close to the structure of legal texts, with formal semantics that clearly define its behavior to reduce discrepancies between legal texts and their implementation.

We start by describing rulings and implementations of the law where precise and well-defined date arithmetic is paramount to ensure expected results. Then, we describe how Catala's implementation of date rounding has recently evolved: from the issues we noticed in Catala's previous off-the-shelf implementation, to the port to our date calculation library and the introduction of a function-local rounding definition when legal references or interpretations are known, reducing the number of cases where the rounding mode is unspecified. We finish by explaining the latest implemented feature, which allows the Catala compiler to extract date computations and relies on Mopsa to (dis)prove rounding-insensitivity.

5.1 Date arithmetic and the law

Critical software relying on date computations is commonly used by companies or government agencies to automatically enforce legal dispositions, e.g., to check if an application has been filed within the correct time period, to compute age-related conditions, or to aggregate periods between dates and compare the result to a fixed duration for eligibility calculation.

In all these cases, there can be heavy financial and legal consequences when a date computation goes wrong or is subject to diverging interpretation. In the case Bowles v. Russell, 551 U.S. 205 (2007) cited by Bailey [7], the court gave Bowles a 17-days notice to file an appeal but this notice was incorrectly computed from Rule 4(a)(6) and paragraph 2107(c), as it should have been 14 days. When Bowles filed his appeal on the 17^{th} day, the court system dismissed the appeal on the basis that Bowles should have filed on the 14^{th} day and not trust the notice the court gave him earlier. In more mundane cases, an incorrect date computation can deprive someone of their social benefits, or impose a higher late fee than what should be.

These doubts about date computation in software applying the law are all the more concerning that previous research in code open-sourced by French government agencies did not show a great deal of transparency or trustful practices on that particular matter. For instance, the custom programming language M, used by the French tax authority to compute income tax [35], encodes dates as mere floating-point numbers where the date is just a decimal number in the format DDMMYYYY. The French unemployment agency, whose IT system is mostly implemented in Java, uses a custom date library for its computation (`fr.unedic.util.temps.Damj`) but its implementation is omitted from their only open-source release [47].

5.2 Catala's policy about date rounding

Recently, the Catala project [24, 34] has aimed to bring more accountability and transparency to programs computing taxes or social benefits inside government

agencies. The Catala language is specifically designed to allow the easy translation of computational law into code; in particular, it is based on prioritized default logic [10], which enables programmers to closely follow the base case/exception pattern that permeates the law. To increase confidence and explainability in its programs, Catala also comes with a formal semantics which is formalized in the F* proof assistant. These formal semantics mostly focus on Catala's default calculus, the encoding of prioritized default logic as a programming language, and do not specify all Catala expressions, including date computations.

Initially, the semantics of the date computations was defined by the behavior of the `calendar` OCaml library [50] used inside its interpreter. However, this library relies on the POSIX behavior which is not always monotonic and may appear quirky (for instance, it computes `Jan 31st + 1 month` as `March 3rd` for non-leap years) despite its very complete set of features. These unusual behaviors prompted a deeper investigation about the corner cases of date computations and led to the implementation of the library presented in this paper. While now integrated in the Catala interpreter, our library is standalone, and freely available with an open-source license. As the Catala compiler is implemented in OCaml, so is our library[5], currently packaged with opam; however, by relying on our semantics, its implementation is straightforward. We do not foresee any difficulty porting it to other languages, and plan to do so to support more of the Catala backends, including Python and JavaScript.

The default behavior of our date computation library inside the Catala interpreter is to raise a runtime exception whenever a date rounding is needed during a computation. This choice of behavior has been made conservatively because the decision to round up or down date computations in software enforcing legal rules is itself a legal rule that has to be specified, as we described in the introduction of this paper. To avoid runtime exceptions, rounding rules can be specified at the scope level (a precise definition of Catala's scopes is outside the range of this paper, but it can be considered as a sort of function in Catala) and should be justified, for example by a legal reference or interpretation.

We applied this methodology to fix the code of the biggest Catala program so far, which computes the French housing benefits [32]. Articles L822-4, R823-4 of the construction and housing Code, as well as article L512-3 of the social security Code, all feature a comparison of the age of the user to an age constant. However, as the input to the Catala program is not the age of the user but their birth date, we know such a comparison can be ambiguous if the user was born on February 29$^{\text{th}}$ on a leap year and if the current date is March 1$^{\text{st}}$. In those situations, we took the decision to round up the date addition, as shown in Fig. 10, with the `date rounding increasing` mention. We are currently trying to contact the relevant government agencies operating the system for clarifications about how this issue should be handled.

[5] Our F* formalization can be extracted to executable but non-idiomatic OCaml code. In practice, we thus manually reimplement our library in OCaml to use features such as named arguments or exceptions to provide a more idiomatic API.

```
declaration scope CheckingAgeInferiorEqual:
  input birth_date content date
  input current_date content date
  input target_age content duration # always a number of years
  output age_is_inferior_or_equal_target content boolean

scope CheckingAgeInferiorEqual:
  definition age_is_inferior_or_equal_target equals
    birth_date + target_age <= current_date
  date rounding increasing
```

Fig. 10. Catala code for checking the age of the user is lower than a constant

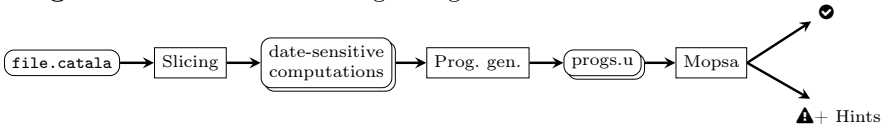

Fig. 11. Catala date ambiguity analysis pipeline

To best benefit the recipient and be in line with the general principle underpinning legal interpretations of social security law in France, a better solution would be to perform the computation twice, by rounding up and down, and select the outcome most favorable to the user in case of disagreement. The flexibility offered by our library allows us to do that, and we intend to explore this avenue in future work. Being able to control precisely where the rounding is done and how is key for developers and maintainers of such programs, as they are responsible for the legal effect of the program itself [22].

5.3 Detecting potentially ambiguous computations

Choosing the rounding mode for each date computation allows us to precisely control the outcome of ambiguous computations. However, given the pervasiveness of such computations in legal texts, it is also extremely tedious, and figuring out the cases where an ambiguous computation could happen is complex. For these reasons, we expect some developers to delay this step and wait for incidents to figure out the policy of the institution operating the program on the matter. But figuring out this policy might itself be tricky because of the automation frontier [33] strictly separating the developers from the decision-makers in charge of legal policy decisions.

To help developers reach out to the legal services of their institution with concrete examples of where things can go wrong before production incidents, we integrated the semantics and abstract domains presented in this paper inside the ongoing initiative to provide a proof platform for Catala programs [20]. By connecting the Catala compiler to the Mopsa static analyzer, we are able to check whether a date computation can be ambiguous in the context of the program, and often exhibit a counter-example if it is the case. We present in Fig. 11 our analysis pipeline. It consists of three main phases: program slicing, verification condition crafting, and analysis – which may generate counter-examples.

First, we scan the Catala program in one of its intermediate representation and look for Catala expressions susceptible of raising a runtime exception be-

cause of an ambiguous date computation. We use classic techniques of program slicing for this step, selecting only the target sub-expression and then adding the definitions of variables used in that sub-expression recursively to extract a small, self-contained program with sufficient information to be analyzed. This will simplify the counter-example hint generation of Mopsa, which outputs constraints on variables rather than subexpressions of a computation.

Second, we augment the sliced program with the assertions and other information about its variables that are declared in the original Catala program to further constrain the search space. So far, our analysis is intraprocedural, but we are planning to implement an inlining pass to make it inter-procedural. We then translate the sliced program to Mopsa's toy language (using the `.u` extension), which can then be fed to the static analyzer.

Finally, we run Mopsa on the generated program. As we have mentioned in Sec. 4.5, Mopsa is able to exhibit potential counter-examples hints. While these hints are approximate due to incompleteness of the analysis, they are often sufficient to yield real, actionable counter-examples on the Catala programs that we analyzed. We extract relevant intervals and linear constraints and display them to the user, in the format illustrated by Fig. 9. While the intervals and constraints presented are descriptive, and sufficient for a programmer to identify concrete counter-examples, they can however be difficult to grasp for non-experts. Formatting these constraints in a more readable format is an interesting question, requiring further interaction with lawyers; we leave it as future work.

The implementation of housing benefits in Catala currently consists of about 20,000 lines (including the text of the law directly specifying it) that were written prior to this work. While automatically analyzing this implementation using our verification pipeline, we found issues in two date computations (one of them being our running example). In both cases, Mopsa was able to provide actionable counter-example hints. Several other computations were age computations, which are now handled by a custom scope with a legally interpreted date rounding mode, as shown in Fig. 10. Finally, remaining computations rely on durations defined outside of the analyzed scope, which requires an inter-scope analysis in Catala, which is being implemented. In the meantime, we performed a manual duration extraction in these cases and detected 16 new unsafe (rounding-sensitive) date comparisons, which are real issues. In all cases, the provided counter-example hints are actionable. In 10 cases, the issues can only happen with a current date before 2023. By constraining the year to be greater or equal to 2023, these 10 cases are proved safe. All date arithmetic programs we have currently extracted or written are small and analyzed within three seconds.

As the number of Catala programs grows, we hope to apply our analyzer at a larger scale, possibly suggesting future avenues for improvement.

6 Related Work

We start by surveying the behavior of mainstream implementations of date arithmetic. We created a suite of litmus tests involving date-duration additions, and

the expected result depending on the rounding mode. We wrote test drivers for each library, running those tests to decide which rounding mode applies.

The `java.time` library [43] provides a `LocalDate` class for dates and a `Period` class to express durations. In our tests, the addition is performed by rounding down. This behavior is explicitly described in the documentation [26]. To the best of our knowledge, there is no option to use another rounding mode, or fail during ambiguous computations. In the Python standard library, the `datetime` module [46] provides a `date` class and `timedelta` to express durations. However these durations cannot be defined in terms of months, but only in terms of days. A third party library called `dateutil` [45] provides a replacement feature, `relativedelta`, able to express durations in months and years. This library seems widely used, as it ranks within the top 20 most downloaded Python packages. On our tests, this library rounds down. This seems to be confirmed by the documentation stating that "adding one month will never cross the month boundary." Similarly to Java, this rounding behavior is not configurable. The `boost` C++ [9] and the `luxon` [31] JavaScript libraries exhibit similar behaviors.

The `coreutils` implementation of date arithmetic follows a different principle, which is not expressible in our semantics. When adding months, this implementation first computes an adjusted date which might not be valid. This adjusted date d_a is then normalized using POSIX's `mktime` function. For example, adding one month to 2023-03-31 yields adjusted date 2023-04-31, which does not exist and is normalized into 2023-05-01. In this case, the behavior is the same as the upper rounding. There are however cases where its behavior differs: adding one month to 2023-01-31 yields adjusted date 2023-02-31, which is normalized into 2023-03-03. This behavior breaks monotonicity of the addition in the date argument (2023-02-01 + 1 MONTH is 2023-03-01). In ambiguous computations, the debug mode of the `date` utility outputs a warning with the following message "when adding relative months/years, it is recommended to specify the 15th of the months" – which is a sufficient condition to avoid any ambiguity. This semantics is also followed by the `calendar` [50] library of OCaml.

We finish this survey with the case of spreadsheet editors (such as Google Sheets), and highlight an inconsistent behavior we have found in them. The `EDATE` function adds a given number of months to a date. In our experiments, this function silently rounds down. As such, adding 18 years (that is, 216 months) to 2004-02-29 yields the date 2022-02-28. These spreadsheets applications also offer the `DATEDIF` function, which can compute the duration in years between two dates. In that case, `DATEDIF`(2004-02-29, 2022-02-28) yields 17 years (18 years are reached when the second date is 2022-03-01). This behavior is inconsistent with `EDATE`. Cheng and Rival [11] focus on performing a type analysis of spreadsheet applications, given that a runtime type casting may silently happen and provide unwanted results (similarly to what JavaScript does). This analysis supports a variety of types, including dates, but as it focuses on type information there is no mention of the value semantics of operations on dates.

The book of Reingold and Dershowitz [48] can be seen as the hacker's delight of calendar computations, with many efficient formulas for day additions, and a

wide range of different calendars being presented. Their work does not mention nor address the issue of month addition, and potential date rounding, which is at the core of our work. Although we have not needed it for now, we could leverage their approach to optimize the recursive computations of our library. Similarly, ISO 8601 defines the representation of dates in the Gregorian calendar, but does not address date-duration additions with years or months.

The Formal Vindications start-up developed a mechanized, formally verified implementation of a time management library [2, 3] in Coq, computing over dates and time, including specific technical points (timezones, leap seconds). Their duration of a month is defined as 30 days. Some recent changes allow to round down dates. A similar effort was developed in Lean 4 by Bailey [6], but this library only supports the addition of days to a date. As a reminder, the Catala project currently targets laws that do not need to go beyond the precision of a day in terms of time management. Formal Vindications developed a formally verified, high-precision tachograph software for enforcing truck drivers scheduling laws [19].

We finish this related work by highlighting similarities between floating-point and date arithmetic. Floating-point arithmetic is more complex and widely used, but both settings have rounding operators with different modes available. This similarity has guided us in our search for properties that hold and counter-examples presented in Sec. 3. The static analysis to prove non-ambiguity of date computations presented in Sec. 4 can be seen as the abstract execution of the computation under both rounding modes, to compare results. To the best of our knowledge, no such static analysis for floating-point programs try to bound the difference in computations between two rounding modes. Tools such as Daisy [8, 18], Fluctuat [23] and FPTaylor [51] usually aim at upper-bounding errors between ideal computations over reals and a machine computation using floating-point.

7 Conclusion and Future Work

Legal expert systems rely on date computations, which are ambiguously defined in some corner cases. There are different ways of solving these ambiguities through different rounding operators, where no operator prevails over the others. We have thus defined semantics for date computations, taking into account these ambiguities to either raise errors, or round the result (either up or down). This semantics has been implemented into a publicly available OCaml library. We have studied this semantics and have formally proved several properties they satisfy, and exhibited counter-examples to usual properties they do not satisfy. We have defined and implemented an analysis that is able to prove an expression to be *rounding-insensitive* in a given program. This analysis relies on partitioning and relational abstract domains to maintain the best possible precision, and can generate understandable counter-examples hints. Both our library and the rounding-sensitivity analysis have been integrated within the Catala language – which focuses on implementing computational laws. Through our analysis, we

found rounding-sensitivity issues in the implementation of the French housing benefits in Catala. We surveyed the behavior of mainstream date arithmetic libraries, and developed litmus tests that can be used to test new libraries.

There are limitations to our static analysis: its soundness has not been proved mechanically, but the proofs simply lift theorems that have been formally verified. The current analyzed language is a core imperative language which was sufficient for our case studies. Having an inter-scope analysis within the Catala to Mopsa translation would improve our precision in the case study. We plan to craft human-readable error messages from Mopsa's output. We believe the relevant constraints are already properly extracted by Mopsa and the rest of the work consists in engineering, in order to inverse the translation from Catala date computations to Mopsa programs.

In spite of these limitations, we believe this paper to be a crucial step into clarifying and improving the robustness of many computer programs implementing "business logic", often overlooked by formal methods. The widespread use of date arithmetic in programs used by companies or government agencies to operate massive financial transfer should have prompted a formal analysis of date rounding a long time ago, but the existing literature only indicates a recent interest from the formal methods community on the matter.

This work was triggered by the problems we found during interdisciplinary investigations about French housing benefits using the Catala programming language. From these investigations surfaced the need for various formal analysis, which we have thus started integrating into the programming language. We hope to further develop the integration of static analysis into the Catala proof platform, thus benefiting both legal and computer science users by including formal methods advances into development processes of Catala programs.

Artifact Availability Statement. All our development is under open-source licenses, public or in the process of being upstreamed into a public development. To foster reproducibility of our results, we provide an artefact [39] containing the formal proofs written in F*, our date calculation library, and our ambiguity detection analysis as well as supporting evidence of our case study.

Acknowledgements. We thank the anonymous reviewers for their constructive feedback and support of our work. We are obliged to Abdelraouf Ouadjaout for making his implementation of partitioning within Mopsa available to us. We are grateful to David Delmas for the discussions around double semantics, Liane Huttner & Sarah Lawsky for the interesting discussions around the properties this work targets, and Louis Gesbert for his technical help around the Catala compiler. We appreciated the many discussions and valuable feedback about this work we got from the whole Catala team.

Bibliography

[1] Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: TACAS, Lecture Notes in Computer Science, vol. 7214, pp. 157–172, Springer (2012)

[2] de Almeida Borges, A., Bedmar, M.G., Rodríguez, J.J.C., Reyes, E.H., Buñuel, J.C., Joosten, J.J.: UTC time, formally verified. In: CPP, pp. 2–13, ACM (2024)

[3] de Almeida Borges, A., Buñuel, Q.C., Rodriguez, J.C., Bedmar, M.G., Reyes, E.H.: Formal vindication time library. `https://gitlab.com/formalv/formalv/-/tree/dev/theories/time` (2023)

[4] Arora, C.M.: What is the concept of "month" while computing limitation period under the custom act? `https://itatonline.org/digest/articles/what-is-the-concept-of-month-while-computing-limitation-period-under-the-custom-act/` (2020)

[5] Bagnara, R., Dobson, K.L., Hill, P.M., Mundell, M., Zaffanella, E.: Grids: A domain for analyzing the distribution of numerical values. In: LOPSTR, Lecture Notes in Computer Science, vol. 4407, pp. 219–235, Springer (2006)

[6] Bailey, C.: A date and time library for lean 4, implementing the proleptic gregorian calendar. `https://github.com/ammkrn/timelib` (2023)

[7] Bailey, C.: Research keynote at the Programming Languages and the Law workshop (POPL). `https://www.youtube.com/watch?v=_0ZycH8NB4s` (2023)

[8] Becker, H., Zyuzin, N., Monat, R., Darulova, E., Myreen, M.O., Fox, A.C.J.: A verified certificate checker for finite-precision error bounds in coq and HOL4. In: FMCAD, pp. 1–10, IEEE (2018)

[9] Boost contributors: The Boost c++ libraries. `https://www.boost.org/` (2023)

[10] Brewka, G., Eiter, T.: Prioritizing default logic. In: Intellectics and Computational Logic: Papers in Honor of Wolfgang Bibel, pp. 27–45, Springer (2000)

[11] Cheng, T., Rival, X.: Static analysis of spreadsheet applications for type-unsafe operations detection. In: ESOP, Lecture Notes in Computer Science, vol. 9032, pp. 26–52, Springer (2015)

[12] Chevalier, M., Feret, J.: Sharing ghost variables in a collection of abstract domains. In: VMCAI, Lecture Notes in Computer Science, vol. 11990, pp. 158–179, Springer (2020)

[13] Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the International Conference on Functional Programming (ICFP) (2000)

[14] Cornell Law School: Rule 6. computing and extending time; time for motion papers. `https://www.law.cornell.edu/rules/frcp/rule_6` (2016)

[15] Council of the European Communities: Regulation (eec, euratom) no 1182/71 of the council of 3 june 1971. `https://eur-lex.europa.eu/`

LexUriServ/LexUriServ.do?uri=CELEX%3A31971R1182%3AEN%3AHTML (1971)
[16] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL) (1977)
[17] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–96, ACM Press (1978)
[18] Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - framework for analysis and optimization of numerical programs (tool paper). In: TACAS (1), Lecture Notes in Computer Science, vol. 10805, pp. 270–287, Springer (2018)
[19] de Almeida Borges, A., Conejero Rodríguez, J.J., Fernández-Duque, D., González Bedmar, M., Joosten, J.J.: To drive or not to drive: A logical and computational analysis of european transport regulations. Information and Computation (2021), 26th International Symposium on Temporal Representation and Reasoning
[20] Delaët, A., Merigoux, D., Fromherz, A.: Turning Catala into a Proof Platform for the Law. In: Workshop on Programming Languages and the Law at POPL 2022, Philadelphia, United States (Jan 2022)
[21] Delmas, D., Ouadjaout, A., Miné, A.: Static analysis of endian portability by abstract interpretation. In: SAS, Lecture Notes in Computer Science, vol. 12913, pp. 102–123, Springer (2021)
[22] Diver, L., McBride, P., Medvedeva, M., Banerjee, A., D'hondt, E., Duarte, T., Dushi, D., Gori, G., van den Hoven, E., Meessen, P., Hildebrandt, M.: Typology of legal technologies (2022)
[23] Goubault, E., Putot, S.: Static analysis of finite precision computations. In: VMCAI, Lecture Notes in Computer Science, vol. 6538, pp. 232–247, Springer (2011)
[24] Huttner, L., Merigoux, D.: Catala: Moving Towards the Future of Legal Expert Systems. Artificial Intelligence and Law (Aug 2022)
[25] Internal Revenue Service: Exclusion of gain from sale of principal residence. https://www.law.cornell.edu/uscode/text/26/121 (2017)
[26] Java™ Platform Standard Ed. 8: Class LocalDate. https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusMonths-long- (2023)
[27] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV, Lecture Notes in Computer Science, vol. 5643, pp. 661–667, Springer (2009)
[28] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: VSTTE, Lecture Notes in Computer Science, vol. 12031, pp. 1–18, Springer (2019)
[29] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: The MOPSA static analyzer (2023), URL https://gitlab.com/mopsa/mopsa-analyzer/
[30] Legifrance: Article 641 - code de procédure civile. https://www.legifrance.gouv.fr/codes/article_lc/LEGIARTI000006411002 (1976)

[31] Luxon contributors: Luxon — a library for working with dates and times in javascript. https://github.com/moment/luxon (2023)
[32] Merigoux, D.: Experience report: implementing a real-world, medium-sized program derived from a legislative specification. In: Workshop on Programming Languages and the Law 2023, POPL 2023, Boston (MA), United States (Jan 2023)
[33] Merigoux, D., Alauzen, M., Slimani, L.: Rules, Computation and Politics: Scrutinizing Unnoticed Programming Choices in French Housing Benefits. Journal of Cross-disciplinary Research in Computational Law **1**(3) (2023), (forthcoming)
[34] Merigoux, D., Chataing, N., Protzenko, J.: Catala: a programming language for the law. In: Proceedings of the International Conference on Functional Programming (ICFP) (2021)
[35] Merigoux, D., Monat, R., Protzenko, J.: A modern compiler for the french tax code. p. 71–82, CC 2021, Association for Computing Machinery (2021)
[36] Microsoft: Filetime (minwinbase.h). https://learn.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-filetime (2021)
[37] Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: VMCAI, Lecture Notes in Computer Science, vol. 3855, pp. 348–363, Springer (2006)
[38] Miné, A.: Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. Sci. Comput. Program. **93**, 154–182 (2014)
[39] Monat, R., Fromherz, A., Merigoux, D.: Formalizing Date Arithmetic and Statically Detecting Ambiguities for the Law (Artifact) (Jan 2024), https://doi.org/10.5281/zenodo.10460049
[40] Monat, R., Ouadjaout, A., Miné, A.: Static type analysis by abstract interpretation of python programs. In: ECOOP, LIPIcs, vol. 166, pp. 17:1–17:29, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
[41] Monat, R., Ouadjaout, A., Miné, A.: A multilanguage static analysis of python programs with native C extensions. In: SAS, Lecture Notes in Computer Science, vol. 12913, pp. 323–345, Springer (2021)
[42] Monin, J.: Louvois, le logiciel qui a mis l'armée à terre. https://www.radiofrance.fr/franceinter/podcasts/secrets-d-info/louvois-le-logiciel-qui-a-mis-l-armee-a-terre-8752880 (2018)
[43] Oracle: Package java.time. https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html (2023)
[44] Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: SAS, Lecture Notes in Computer Science, vol. 12389, pp. 223–247, Springer (2020)
[45] Python-dateutil contributors: python-dateutil — useful extensions to the standard python datetime features. https://github.com/dateutil/dateutil (2023)
[46] Python Software Foundation: datetime — basic date and time types. https://docs.python.org/3/library/datetime.html (2023)

[47] Pôle Emploi: Calcul de l'allocation d'aide au retour à l'emploi (are). https://www.pole-emploi.org/opendata/calcul-de-lallocation-daide-au-r.html (2018)
[48] Reingold, E.M., Dershowitz, N.: Calendrical Calculations: The Ultimate Edition. Cambridge University Press (2018)
[49] Rival, X.: Understanding the origin of alarms in astrée. In: SAS, Lecture Notes in Computer Science, vol. 3672, pp. 303–319, Springer (2005)
[50] Signoles, J.: The calendar ocaml library (2011), URL https://github.com/ocaml-community/calendar
[51] Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. ACM Trans. Program. Lang. Syst. **41**(1), 2:1–2:39 (2019)
[52] Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoué, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL) (2016)
[53] The Open Group: time. https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/functions/time.html (2017)
[54] The QCheck contributors: Q-check: quickchecking library for ocaml. https://github.com/c-cube/qcheck (2023)
[55] Torny, D.: L'administration sanitaire entre contraintes techniques et contraintes juridiques. L'exemple des maladies émergentes. Revue générale de droit médical (6) (2005), URL https://shs.hal.science/halshs-01249084

A Definition of the toy imperative language used to analyze date programs

The programs we consider in this section are written in a standard, toy imperative language whose grammar is presented in Appendix A. Fig. 12, consisting of assignments of date expressions, and assertions to express rounding-insensitivity of given expressions. The date expressions can be date literals, variables, fresh arbitrary dates (`random_date`), projections of a date on the first day of its month (`first_day_of`), accessors to the current day (`day_of`), month (`month_of`), or year (`year_of`), and additions between a date and a duration (expressed in years, months and days). The semantics of assignments and assertions is defined as usual, while the semantics of addition has been presented in Sec. 2. The semantics of the `sync` operator used to express rounding insensitivity will be defined in Sec. 4.3.

variables	$v \in \mathcal{V}$	
type	t	$::=$ date $\mid$ int
literal	l	$::= z \in \mathbb{Z} \mid [z_1$ years, z_2 months, z_3 days$]$
expr	e	$::= l \mid v \mid$ random_date() $\mid e_1 + e_2 \mid$ day_of$(e_1) \mid$ month_of$(e_1) \mid$ year_of$(e_1) \mid$ first_day_of(e_1)
boolean expr	b	$::= e_1 < e_2 \mid e_1 == e_2 \mid$ sync(b)
statement	s	$::= t\, v = e \mid$ assert(b)
program	p	$::= () \mid s; p$

Fig. 12. Grammar of analyzed programs

This work is licensed under a Creative Commons “Attribution 4.0 International” license.