ACL2 Support for Floating-point Computations

Matt Kaufmann[0009−0004−5667−4888]⋆ and
J Strother Moore[0000−0002−9628−1702]

University of Texas at Austin, USA (retired)
`{kaufmann,moore}@cs.utexas.edu`

Abstract. Formal methods tools vary widely but generally have logical foundations. For ACL2, a general purpose theorem prover under continuous development since about 1990, having a sound logical foundation is absolutely essential. Many formal tools support execution on concrete data, and ACL2 does so by being compatible with the Common Lisp language: theorems can be proved about Common Lisp functions in the ACL2 subset, and efficient execution is provided by reliance on compiled Common Lisp code. ACL2's arithmetic is based on a straightforward axiomatization of the rationals and Common Lisp provides exact rational arithmetic. But computation based on exact rational arithmetic is relatively slow, so we have recently added support for floating-point operations in ACL2. The challenge is how to do this while preserving the pre-existing axioms for arithmetic and a large regression suite containing verified theorems and other logical tools contributed and used by the entire ACL2 community. We discuss how we have met these challenges, we discuss the limitations of our support for floating-point operations, and we illustrate the resulting system with examples.

Keywords: theorem proving · floating-point · ACL2

1 Reflections on the Career of Cliff Jones by J Moore

In Fall, 2023, when the editors of this book approached me about contributing an original research paper to this Festschrift for Cliff Jones, I initially declined because no such paper was in the works. At the time, Matt Kaufmann and I were busy experimenting with ways to support some limited floating-point arithmetic within ACL2. In fact, we had developed a plausible design and a mock-up of an ACL2 session demonstrating our intended functionality. Convinced it was worth a more serious effort, we embarked on changing ACL2 in accordance with our design and developing several applications that would exploit the new features. But we were far from convinced that we would release this experimental version of ACL2, much less were we inclined to write a paper about it. Reasons for all this *sturm und drang* will be made clear in the main body of this paper.

Meanwhile, I had agreed to write a brief testimonial to Cliff for inclusion in the Festschrift. But as the deadline approached Matt and I began to have more

⋆ corresponding author

confidence that the new version of ACL2 was worthwhile. In fact, we released the new system and asked the editors if we could contribute this article about the work. They very kindly agreed.

But I feel I still owe them, and Cliff, that testimonial. I have known *of* Cliff far longer than I have known him. In fact, I cannot remember when we first met. I got my PhD in Edinburgh in 1973, but Cliff was not in nearby Newcastle then. He was working at IBM in Vienna and made his first marks in formal methods before he got his PhD at Oxford in 1981. I do not think our paths crossed when I was a student. We have seen a lot more of each other since 1999, when my wife and I started spending long summer months in Edinburgh and Cliff and his wife moved to Newcastle. Like Cliff, I spent much of my time working in industry, first at Xerox PARC and SRI and then at Computational Logic, Inc. But regardless of where we were working, we have spent our careers in formal methods. For example, in the 1970s, when Bob Boyer and I were formalizing SRI's Hierarchical Development Methodology, Cliff was working at IBM on the Vienna Definition Language (and later the Vienna Development Method) — two frameworks with the same objective: structured system development facilitating proof.

Often, and quite predictably, we were investigating closely related phenomena — language semantics, industrial practice, inductive invariants, rely-guarantee, specification languages, the role of AI in theorem proving, proofs as certificates that code is correct versus proofs as explanations of why code is correct, abstraction, concurrency, interference, real-time control, compiler correctness, system-level verification, etc.

With such similar beginnings and similar high-level goals it is no wonder our careers have run down parallel tracks. Indeed, the wonder is that they have been literally parallel: they have never crossed! Cliff and I meet whenever we are in the same place and enjoy conversation and exchange of ideas. But we have never collaborated.

We each come at the problems with our own perspective and goals. I build theorem provers intended to be useful to industry, whereas I think of Cliff as more interested in how to build provably correct systems. He explores higher-level ideas for how to specify and verify software and, compared to me, appears agnostic as to one's choice of formal mathematical logic. (Since I build theorem provers, the choice of the logic is binding on me; once chosen, my problem is whether and how we can get the machine to reason effectively in it.) Cliff is bound only by the general principles of sound mathematical reasoning, and a passion to make the work relevant to industrial practice.

Cliff has had a tremendous impact on the field by encouraging others to think about how to think about software. He has graduated around 25 PhD students and a dozen or so Masters students. He was the founding Editor-in-Chief of *Formal Aspects of Computing* in 1989. That journal rather boldly focuses on "the junction of theory and practice." I say "boldly" because that junction can be a hard territory to occupy. I have met many members of both tribes and often found each looked down upon the other. Of course, the best members of either tribe deeply respect and leverage the skills of the other tribe. Cliff is one of those. Stepping into that territory was an act of bravery that indicates his true priorities.

2 Introduction and Background: Part I

This paper is about the extension of the ACL2 system to take advantage of Common Lisp floating-point computations. ACL2 has always used Common Lisp to do computations, even during proofs, as discussed in Section 4. For ACL2 to use floating-point computations, the key idea is to view a floating-point number as its corresponding rational number, with axiomatic support that provides logical justification for floating-point computations. Before we explain this very high-level summary, we provide relevant background.

ACL2 [6] is a formal verification system for (an extension of an applicative subset of) Common Lisp [12]. We axiomatized that language as a first order logic. It is used not only as a specification/modeling language but as the language in which conjectures are stated, and it is also the implementation language of the theorem prover. "ACL2" in fact stands for *A* *C*omputational *L*ogic for *A*pplicative *C*ommon *L*isp. It has been under continuous development by the authors of this paper since 1989 (with early contributions by Bob Boyer). In the early 1990s ACL2 saw its first industrial use [3], and by the mid-2010s, Centaur Technology used it in nightly regression tests to verify modifications to previously verified modules in x86 designs [4]. In fact, by the mid-2010s, ACL2 was used by various companies, including Centaur, IBM, AMD, and Oracle, to verify hardware designs implementing floating-point operations. Basically, implementing floating point requires iterative (recursive) manipulation of bit vectors and Lisp is an excellent language for describing such processes. It is also an excellent language in which to express the operational semantics of hardware design languages and is so used. Proving interesting properties of such designs requires induction, which is among the strengths of the ACL2 theorem prover.

Today, for example, Intel runs ACL2 regression tests nightly, not just on floating-point implementations but also on other microprocessor components.

ACL2's success in industry can perhaps be attributed to three things: a longstanding focus on making it useful to industry; the fact that as an efficiently executable programming language formal models can be used both as testable prototypes and build-to specifications subject to formal analysis; and the very talented, passionate, and collaborative ACL2 community. The ACL2 regression suite [1] has thousands of files containing formalized definitions, verified theorems, language models, and verified tools and extensions contributed by the users and used by all. The system and its documentation are available without cost in source code form under a generous 3-clause BSD style license [7]. The documentation is provided in several formats but perhaps the most convenient for most users is `html` format [11]. The documentation written especially for the work described here may be found at [8].

The axioms of ACL2 specify five Common Lisp data types: strings, characters, symbols (with packages), ordered pairs (i.e., lists and trees), and "ACL2 numbers" which include the rationals (which include the integers) and the complex rationals (complex numbers in which both the real and imaginary parts are rational). For simplicity in this paper we are going to ignore the complex rationals and just pretend that, until the work described here, ACL2 arithmetic was limited to the rationals.

ACL2 also includes an axiomatization of the ordinals up to ε_0, a conservative definitional principle for the introduction of recursive functions and an induction principle, both of which rely on the well-foundedness of the ordinals. There are various additional logical features (encapsulation, allowing limited scoping of some names and the conservative introduction of constrained functions, functional instantiation, and limited "second-order" features similar to Lisp's `apply`) and additional programming features (e.g., multiple values, property lists, arrays, single-threaded objects which can be destructively modified, hash tables, and a flexible iteration primitive modeled on Lisp's `loop` statement).

3 Why Are We Interested in Floating Point?

While we can prove facts about implementations of floating-point operations, ACL2 could not directly execute floating-point operations. Of course, it could run the designs it had verified. Indeed, in the above-mentioned floating-point verification projects it is standard operating procedure to run the ACL2 models of the floating-point hardware on millions, sometimes hundreds of millions, of floating-point test vectors to test the model before trying to verify it. But it is terribly inefficient to emulate hardware in software to compute the sum of two floating-point numbers on a machine that has verified floating-point arithmetic hardware. Why not just use the hardware?

Of course, to do this in the spirit of ACL2 we would ideally have floating-point numbers and floating-point operations fully formalized in the logic. We will discuss in Section 5 why that is too ambitious — or at least more ambitious than our current project. Our goal is to allow the ACL2 user to take advantage of floating-point hardware when evaluating ground (variable-free) expressions, even during proofs, without necessarily being able to prove very much about the results — and *without invalidating the thousands of books and tools built and verified in the ACL2 regression suite.*

But why might we want to do floating-point computations? Every number represented as a floating-point object is a rational number. And ACL2 and Common Lisp support exact rational arithmetic. So why not just use rationals? The answer is that even that is too slow. For example, we coded a relatively straightforward (sequential) Gaussian elimination solver for linear equations in Common Lisp. The implementation was optimized for sparse matrices and could use either exact rational arithmetic or double precision floating-point arithmetic. When solving for $\mathbf{x}$ in $\mathbf{Ax} = \mathbf{b}$ for a particular 1036×1036 matrix $\mathbf{A}$ and vector $\mathbf{b}$ of length 1036, it took exact rational arithmetic 158 seconds to compute the exact answer. It took double precision floating-point arithmetic only 0.0036 seconds. The exact rational answer was a vector that took 10 megabytes to print out: an array of 1036 rationals, each of the form p/q for integers p and q in lowest terms, and each of p and q had, on average, 4,891 digits. But the answer was exact. Given that $\mathbf{x}$, the matrix-vector product $\mathbf{Ax}$ was equal to the given $\mathbf{b}$. The double precision answer was much more compact: 1036 double precision floating-point numbers averaging about 20 digits each. The double precision $\mathbf{x}$ was not exact. But the matrix-vector product $\mathbf{Ax}$ was close to $\mathbf{b}$: the Euclidean distance between $\mathbf{Ax}$ and $\mathbf{b}$ was 6.3176103868853046E-18. Perhaps more interesting is the Euclidean distance between the exact solution and the double precision approximation: 1.4438701402655734E-17. In fact, no corresponding components differ by more than 3.0384309630009724E-18. So floating point computations can be very attractive, which is why they are often used in scientific computing when speed is desired and inexactness can be tolerated.

An example of such an application is discussed in “VWSIM: A Circuit Simulator,” by Warren A. Hunt, Jr., Vivek Ramanathan, and J Strother Moore [5]. In fact, the work we did for that project was a major inspiration for the current work. To quote from the introduction of that paper

> We have defined the VWSIM circuit simulator with simulation models for resistors, capacitors, inductors, transmission lines, mutual inductance, Josephson Junctions (JJs), and VWSIM includes voltage, current, and phase sources. VWSIM can simulate an entire circuit model either in the voltage or phase domain.

Voltages and phases are given by floating-point numbers produced by solving a linear equation $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ and $\mathbf{b}$ are derived from the SPICE-compatible circuit description and assumed input values. The solver, which as noted above is a straightforward sequential Gaussian elimination solver for VWSIM, is actually written in Lisp, not ACL2, because at the time of the work (2022) ACL2 did not support floating point. But today, because of the work reported here, it would be possible to verify the termination and type correctness of the solver.

That still leaves us with the question: what is the point of having a formal circuit simulator if the voltages and phases being produced by the solver are not exactly accurate? In principle — and we stress we have no immediate intention of doing this — we could prove the solver is correct when running on rationals. Assuming the rest of the simulator could be then proved correct, we would have eliminated a major source of bugs the authors of the above paper have found in other circuit simulators. We could then run the simulator on floating-point inputs and know that the only inaccuracy is due to floating-point operations, not because the authors of the simulator coded the semantics of an `AND` as an `OR` or wrote `X` when they meant `Y`. As for the inexactness, engineers and physicists have long suffered from this problem. For example, it turns out that the inputs to some of these circuits cannot be measured accurately and fast enough to be within 10% of the actual voltage, so it is simply a waste of time to worry about a 10^{-18} error in the voltage of some interior wire. Some models are just ideal descriptions of the design and verification can eliminate the coding bugs and engineering is required to decide what it means. Of course, given sufficient time, we could solve $\mathbf{Ax} = \mathbf{b}$ exactly and see whether that makes a (significant) difference to the predicted behavior of the circuit.

Much more mundane applications in which floating point would be nice and its inexactness could be tolerated is in gathering and analyzing statistics about performance of the theorem prover, perhaps to gather information for weights in machine learning experiments. Instrumenting ACL2 to gather metadata is easiest if the instrumentation itself is in the language ACL2 is written in: ACL2.

Examples like these motivated our work on adding support for floating-point calculations to ACL2. Common Lisp supports both single and double precision floating-point formats, but we decided to support just double precision because it provides so much more accuracy. In Common Lisp, these numbers are called "double-floats" and are a distinct type of object, just as characters, strings, symbols, conses, and rationals are distinct types.

4 Introduction and Background: Part II

To prepare the general reader for our discussion of how we support floating point in ACL2, it is necessary to dive a little deeper into how ACL2 evaluates ground terms like `(+ 2 3)` and what, exactly, is the relationship between ACL2's logic and Common Lisp. And of course we are assuming that Common Lisp implementations actually comply with the Common Lisp standard. The connection to Common Lisp allows ACL2 to serve as a practical programming language. That has always been a design goal for ACL2, which now provides fast execution for floating-point operations.

An important feature for the current context is the notion of the "guard" on a function. Guards allow proof-enforced type-checking, akin to type correctness conditions in PVS [13]. Common Lisp is an untyped language. In principle, any function can be applied to any type of object. But the Common Lisp standard [12] specifies the behavior of primitives only on certain "expected" types or combinations of arguments. ACL2 captures these expectations as guards and guarantees Common Lisp evaluation of an ACL2 ground term is consistent with the ACL2 axioms — provided the guards are all satisfied.

For example, the Common Lisp standard says that `(+ x y)` returns the sum of the rationals `x` and `y`.[1] The ACL2 axioms formalize `+` with the usual axioms about rational arithmetic, e.g., that ACL2's rationals are closed under `+`, that `+` is associative and commutative, that `0` is the identity on numbers, etc. So, `(+ 2 3)` is `5`, `(+ 2 1/2)` is `5/2`, and `(+ 3/17 5/97)` is `376/1649`. But what is `(+ "Monday" 5)`? The standard does not say what `+` returns when given a string as an argument. In fact, the standard says that the implementer of a Common Lisp is free to do anything at all in such a case.

[1] Recall we are ignoring ACL2's handling of complex rationals here. And the standard does not say that `(+ x y)` is the *sum* — in the ordinary mathematical sense of that word — for floating-point numbers because it is not!

But to make the ACL2 axioms as tractable as we can, we add the following axiom, where `rationalp` is the recognizer for rational numbers.

```
(implies (not (rationalp x))
         (equal (+ x y) (+ 0 y)))
```

That is, `+` defaults non-rational arguments to `0`. So according to our axioms `(+ "Monday" 5)` is `5`. As far as we know, there is no Common Lisp implementation that implements this sense of `+`!

But note that the standard said that `+` is sum on *rationals*. That "expectation" is captured in ACL2 by specifying that the guard on `(+ x y)` is `(and (rationalp x) (rationalp y))`. We claim that the ACL2 axioms and Common Lisp evaluation agree *provided every time* `+` *is applied it is applied to rationals.* ACL2 provides *guard verification* for terms and function definitions. Guard verification generates and attempts to prove conjectures that establish that every function call is on arguments satisfying its guards. If a term or function body has been guard verified and the inputs satisfy the guard, then its value under the axioms is the same as its value in Common Lisp.

This means that when a guard-verified ACL2 term needs to be evaluated, and the inputs satisfy the guard, then ACL2 can just ask Common Lisp to evaluate it. But what if the inputs do not satisfy the guard? In that situation, we have to evaluate a version of the term in which all functions are replaced by their logical counterparts. It is easiest to understand by example.

Imagine the ACL2 user submits the following definition.

```
(defun foo (x y)
 (declare (xargs :guard (and (rationalp x) (rationalp y))))
 (+ 3 (+ x y)))
```

(Alternatively, we could have written `(declare (type rational x y))`. But Common Lisp's type specification facilities are not as general as guards.)

This function can be guard verified: if `x` and `y` satisfy the guard that they are both rational, then every `+` in the definition gets a rational input, since the sum of two rationals is a rational.

When `foo` is defined in the ACL2 logic, the ACL2 implementation defines two versions of `foo` in the Common Lisp that is running ACL2. One version of `foo` is exactly as written by the user; we may call this the *Common Lisp counterpart of* `foo`. The second version, the *ACL2 logic version of* `foo`, is (essentially)

```
(defun acl2_*1*_acl2::foo (x y)
 (if (and (rationalp x) (rationalp y))
     (foo x y)
     (acl2_*1*_acl2::+ 3 (acl2_*1*_acl2::+ x y))))
```

Note that the name of this second function is the symbol `foo` in an oddly named symbol package. This package is guaranteed to be different from any package accessible to the ACL2 user. Furthermore, the version of `foo` in that package has a guard of `t` (*true*), so it can be run without error. Importantly, all the primitives of ACL2 have versions defined in that package too. For example,

```
(defun acl2_*1*_acl2::+ (x y)
  (+ (if (rationalp x) x 0)
     (if (rationalp y) y 0)))
```

That version of `+` satisfies the axioms because it defaults its non-rational arguments to `0`.

When ACL2 is asked to evaluate a call of `foo`, it makes the corresponding call of `acl2_*1*_acl2::foo`. Thus, for evaluation of `(foo 4 5)`, since `4` and `5` satisfy the guard of `foo`, then Common Lisp simply evaluates `(foo 4 5)` to obtain `12`, which ACL2 reports as its answer, as though it was derived from the axioms. But if the user evaluates `(foo "Monday" 5)`, then when ACL2 asks Common Lisp to evaluate `(acl2_*1*_acl2::foo "Monday" 5)`, the result is `8`, as the axioms specify.[2]

This discussion is critical to what follows because floating-point arithmetic does not follow the laws of rational arithmetic, as we discuss in the next section.

5 Challenges for Supporting Floating-Point Operations in ACL2

Common Lisp has both rational numbers and floating-point numbers, but ACL2 has traditionally not included floating-point numbers. Why not? The fundamental problem is that Common Lisp evaluation with floating-point numbers violates ACL2 semantics. We present two examples of this problem.

Example 1. Consider for example that ACL2 proves `(equal (equal x y) (= x y))`, since in fact `=` is defined logically to be `equal`. That property is violated in Common Lisp, as illustrated by the following log.

```
? (equal 1 1.0)
NIL
? (= 1 1.0)
T
```

The discrepancy is due to Common Lisp semantics [14], where `equal` distinguishes numbers with different types but `=` compares numeric values.

Example 2.

Addition is associative in ACL2, and in fact ACL2 proves `(equal (+ (+ x y) z) (+ x (+ y z)))`. But the following log shows that addition is not associative on Common Lisp double-floats.

[2] Actually, the story is more complicated than this. But this basic idea that every definition in the logic gives rise to two definitions in the underlying Lisp is suggestive of what really happens.

```
? (+ (+ 0.1 0.2) 0.3)
0.6000000000000001
? (+ 0.1 (+ 0.2 0.3))
0.6
```

An obvious approach to integrating double-floats would be to change the axioms of equality and of the arithmetic operators to describe how Common Lisp actually behaves when arguments are of type `double-float`. It would be a monumental task just changing ACL2's source code, where much information about the primitives is built in. But it would have an even more deleterious effect on the regression suite since, for example, it is often not necessary today for the user to restrict to rationals in order to know addition is associative.

6 How ACL2 Supports Floating-Point Operations

We begin this section with an introduction, which presents simple examples that illustrate evaluation using double-floats. That is followed by discussion of three key facets of our approach, which are illustrated with more examples. Then, after summarizing the built-in operators that involve double-floats, we lay out key syntactic restrictions. Next we discuss reasoning about floating-point operations. We conclude this section by discussing limitations.

6.1 Introduction with Examples

Although the logic of ACL2 is generally considered to be untyped, its syntax is restricted so that certain variables may be designated as having type `double-float`: their values are always double-floats during Common Lisp evaluation. The following simple ACL2 definition provides an example: its formal parameters `x`, `y`, and `z` are declared to have type `double-float`, and the addition operation `df+` ("double-float sum") adds the double-float `x` to the double-float sum of `y` and `z`. Note that `df+` is being applied to variables that are declared of type `double-float`, but the outer call of `df+` also has an argument that is itself an application of `df+` and hence designates a double-float value.

```
(defun f1 (x y z)
  (declare (type double-float x y z))
  (df+ x (df+ y z)))
```

Now let us define a function that is to be applied to rational number inputs. We see below that those inputs are converted by `to-df` to double-float numbers, before adding them using `f1` in the given order and in the reverse order, and finally checking for equality of those two sums using the operator `df=`, which compares double-float values.

```
(defun f2 (x y z)
  (declare (type rational x y z))
  (let ((x0 (to-df x))
        (y0 (to-df y))
        (z0 (to-df z)))
    (df= (f1 x0 y0 z0) (f1 z0 y0 x0))))
```

The following log illustrates evaluation of a call of `f2`, after tracing `f1` so that we can monitor the calls of `f1` in Common Lisp. We see that each call of `f1` is made on double-float inputs and returns a double-float. The return value is `NIL` — which represents *false* in Lisp — since the two calls of `f1` returned different values.

```
ACL2 !>(trace$ f1)
 ((F1))
ACL2 !>(f2 1/10 2/10 3/10)
1> (F1 0.1 0.2 0.3)
<1 (F1 0.6)
1> (F1 0.3 0.2 0.1)
<1 (F1 0.6000000000000001)
NIL
```

We see above that reversing the order changes the sum. (In fact double-float addition is commutative but it is not associative.)

6.2 Key Facets of Our Approach

We address the challenges raised in Section 5 with an approach that has the following key facets.

(A) The ACL2 logic simulates floating-point numbers with the rational numbers that they represent. For example, 1.5 is not an ACL2 value, but 3/2 is an ACL2 value that has 1.5 as a floating-point representation.
(B) Certain ACL2 expressions are designated as *df expressions*, or dfs for short. These are the ACL2 expressions that evaluate in Common Lisp to double-floats.
(C) Syntactic restrictions in the form of very limited typed syntax apply to the use of dfs. In particular, each function has an *input signature* and an *output signature* that indicate, respectively, which arguments must be dfs and whether the function's calls are df expressions.[3]

These Key Facets have the following happy consequences.

- The ACL2 logic, supported since 1990, is unchanged: support for floating-point operations only involves adding axioms (including definitions) about new operators such as `df+` and `df=`.
- ACL2 supports Common Lisp computation with double-floats, in spite of their absence from the logic.

[3] This is not the first use of typed syntax in ACL2. See [2].

We turn now to discuss Key Facets (A), (B), and (C).

According to (A), ACL2 logically treats double-floats as the rational numbers that they represent. ACL2 provides a convenient way to read and write rationals using Common Lisp floating-point notation, by using the prefix "`#d`". In the following example, the input at the prompt is read as 7/4, which is the rational value represented by the double-float, 1.75.

```
ACL2 !>#d1.75
7/4
```

Of course, floating-point numbers are represented using binary digits. So for example, since 3/10 cannot be thus represented, then `0.3` is read by Common Lisp as a number that is only approximately 3/10. The corresponding rational number is shown by the transcript below.

```
ACL2 !>#d0.3
5404319552844595/18014398509481984
```

The `#d` notation is used for printing output when the input form represents a computation that would produce a double-float result in Common Lisp, as suggested by the following example.[4]

```
ACL2 !>(df+ #d1.5 #d0.25)
#d1.75
```

The result is the ACL2 value 7/4. But it is printed with `#d` notation because in Common Lisp, the `df+` operation returns a double-float. This leads us to a discussion of (B).

ACL2 identifies an expression as a df if its Common Lisp evaluation produces a double-float. Thus, any call of `df+` is a df expression. When a df expression is evaluated in the top-level loop, the result is printed using `#d` notation, as illustrated above.

The next examples illustrate df expressions and their evaluation. In the definition of `g1` below, the inputs `x` and `y` occur as dfs because of their `double-float` type declarations. Since `g1` returns a value computed by `df+`, then any call of `g1` is a df.

```
(defun g1 (x y)
  (declare (type double-float x y))
  (df+ x y))
```

We next define a wrapper `g2` for `g1`. This definition uses calls of `to-df`, which are df expressions, to convert rational inputs to df inputs as required by `g1`.

[4] The astute reader may observe that the inputs are the rationals 3/2 and 1/4, yet previous discussion may have suggested that the arguments to `df+` should be df expressions. In fact `df+` is a macro and rational arguments are converted to dfs, as discussed below in Subsection 6.3.

```
(defun g2 (r s)
  (declare (type rational r s))
  (g1 (to-df r) (to-df s)))
```

We have already observed that calls of `g1` are df expressions; thus, calls of `g2` are also dfs.

Let us use these examples to illustrate (C). We have seen that calls of `df+`, `to-df`, `g1`, and `g2` are df expressions; also, some must have df inputs. By using the symbol `:df` to indicate a df input or output and the symbol `*` to indicate an ordinary (not df) input or output, we can write signatures in the form "*input-signature* `=>` *output-signature*", as follows.

```
(df+ :df :df) => :df
(to-df *)     => :df
(g1 :df :df)  => :df
(g2 * *)      => :df
```

These signatures represent the restrictions from (C) regarding which inputs and calls must be dfs. Thus, by (B), they also indicate which input and output values computed by Common Lisp should be double-floats. Let us illustrate these points with the following log, which shows tracing of `g1` and `g2` before evaluating a call of `g2`. We can see in passing from "`2>`" to "`3>`" that `g2` applies `to-df` to convert its rational inputs to corresponding double-floats.

```
ACL2 !>(trace$ g1 g2)
 ((G1) (G2))
ACL2 !>(g2 3/2 1/4)
1> (ACL2_*1*_ACL2::G2 3/2 1/4)
  2> (G2 3/2 1/4)
    3> (G1 1.5 0.25)
    <3 (G1 1.75)
  <2 (G2 1.75)
<1 (ACL2_*1*_ACL2::G2 7/4)
#d1.75
```

Recall that the oddly-named function `acl2_*1*_acl2::g2` is the ACL2 logic version of `g2`.[5] It takes and returns ACL2 objects — in particular, it returns a rational, not a double-float. But the Common Lisp counterpart for `g2` calls the Common Lisp counterpart of `g1`, and these return the double-float 1.75 before `acl2_*1*_acl2::g2` coerces that double-float to a rational number. So the final return value is 7/4, but it is printed as `#d1.75` because the input expression, `(g2 3/2 1/4)`, is a df, since the output signature of `g2` is `:df`. This is consistent with (B), that is, that dfs evaluate in Common Lisp to double-floats; see "`<2 (G2 1.75)`" in the log above.

[5] In ACL2 and Common Lisp, symbol names are case insensitive by default.

6.3 Built-in Operators

Next we lay out the built-in operators that involve double-floats. Technically these are mostly macros. For example, the expression `(df+ x y)` abbreviates the function call `(binary-df+ x y)`. These macros are actually a bit fancier than that might suggest: for example, `(df+ 3 z)` abbreviates `(binary-df+ (to-df 3) z)`, where in Common Lisp, `binary-df+` is an inlined function that is defined to add double-floats efficiently.

Here is the definition of `binary-df+`. An occurrence of the expression `(the double-float EXPR)` tells the Common Lisp compiler that `EXPR` evaluates to a `double-float` value, and it requires ACL2 to prove that this is indeed the case as part of guard verification.

```
(defun binary-df+ (x y)
  (declare (type double-float x y))
  (the double-float
       (+ (the double-float x) (the double-float y))))
```

The operator `to-df` is special: it is a function in ACL2 but in Common Lisp it is a macro, hence there is no runtime cost to evaluating the generated form `(to-df 3)` above. Further such implementation details are beyond the scope of this chapter.

Basic arithmetic operators on dfs include `df+` for addition, already discussed several times above, as well as operators `df*`, `df-`, and `df/` respectively for multiplication, subtraction (also unary negation), and division (also reciprocal). The operator `df-abs` takes the absolute value. IEEE Standard 754 [16] requires that elementary floating-point operations return the mathematically correct result *rounded (as specified by a rounding mode) to the nearest representable floating-point number.* ACL2 is only built on Common Lisps that explicitly include the `:IEEE-FLOATING-POINT` feature, which signifies that the Common Lisp implementors believe they have adhered to the standard.

The function `df-round` is intended to represent rounding as discussed in the IEEE and Common Lisp specs, without tying down the rounding mode. It is constrained to return a rational, and it is used in defining the arithmetic operators. For example, `(binary-df+ x y)` is defined *logically* to be `(df-round (+ x y))`. (The definition of `binary-df+` displayed above is its Common Lisp definition.) Notice that this definition implies that `df+` is commutative, since `+` is commutative.

Conversion functions include `to-df`, discussed above, and `from-df`, which is logically the identity but maps a df to an equal rational, and has the following signature.

```
(from-df :df) => *
```

There is also a function, `dfp`, that recognizes which rationals are representable as double-floats. That function is used mainly in proofs, and `(dfp x)` is defined to be `(and (rationalp x) (= (to-df x) x))`.

Other operations include the square root and common transcendental functions, listed alphabetically below. Their names are derived from corresponding Common Lisp functions by adding the prefix, "`df-`".

```
df-acos df-acosh df-asin df-asinh df-atan df-atanh
df-cos  df-cosh  df-exp  df-expt  df-log  df-pi
df-sin  df-sinh  df-sqrt df-tan   df-tanh
```

We close this discussion by saying a bit more about the function `to-df`. This function is intended to represent the coercion of a rational number to a nearby double-float value, where the rational input is numerically equal to the output exactly when the rational input is representable by a double-float. But while the output of `to-df` is a double-float in Common Lisp, the output is logically a rational number (recall Key Facet (A)) in the ACL2 logic, as the following log illustrates.

```
ACL2 !>(to-df 1/3)
#d0.3333333333333333
ACL2 !>#d0.3333333333333333
6004799503160661/18014398509481984
```

Rationals like that are unpleasant to read, but ACL2 provides (courtesy of Common Lisp) a function, `df-rationalize`, that returns a more pleasant but less exact rational.

```
ACL2 !>(df-rationalize (to-df #d0.3333333333333333))
1/3
```

6.4 More on Syntactic Restrictions

As a useful programming language ACL2 includes many constructs not discussed here but requiring careful consideration when adding support for double-floats. One such construct is multiple-value return. Another is single-threaded objects, or *stobjs* [2], together with several documented related features [11]: abstract stobjs, nested stobjs, global stobjs, and local stobjs. What is more, ACL2 has two forms of syntax, user-facing (untranslated) and internal (translated). [10].

Instead of delving further into such details, we here elaborate a little more on the syntactic restrictions mentioned above in Key Facet (C).

The syntactic restrictions apply to definitions but not to theorems. A definition specifies what we call a set of df *variables*: as we have seen above, the form `(declare (type double-float` $v_1 \ldots v_n$`))` specifies that v_1 through v_n are df variables. For a given set V of df variables (sometimes implicit), we have the following basic rules for determining which expressions are dfs and which syntactic restrictions must hold.

- A variable is a df if and only if it is in V.
- A constant is not a df.

- A function call $(f\, t_1 \ldots t_n)$ respects the signature of f as follows.
  - For each $i < n$, t_i is a df if and only if the ith member of the input signature of f is `:df`.
  - $(f\, t_1 \ldots t_n)$ is a df if and only if the output signature of f is `:df`.

But ACL2 also supports local variable bindings. Recall an earlier definition.

```
(defun f2 (x y z)
  (declare (type rational x y z))
  (let ((x0 (to-df x))
        (y0 (to-df y))
        (z0 (to-df z)))
    (df= (f1 x0 y0 z0) (f1 z0 y0 x0))))
```

There are no top-level df variables here; V is empty. However, variables `x0`, `x1`, and `x2` are each locally bound to df expressions, so they are considered to be df variables in the body of the `let` expression, which is the `df=` call. This example is covered by the following rule.

- Consider a term `(let ((`x_1 e_1`) ...(`x_k e_k`))` dcl_1 `...`dcl_k *body*`)`. Let V_1 be the result of removing all x_i from the set V (of df variables) except for those x_i that some dcl_j declares to be of type `double-float`. Then for each x_i in V_1, e_i must be a df with respect to V. Let V_2 be the set of variables obtained from V_1 as follows: for all x_i not already in V_1, if e_i is a df then put x_i into V_2. Then *body* must satisfy the syntactic restrictions with respect to V_2.

6.5 Reasoning

As noted earlier, the motivation for supporting floating-point computations with ACL2 was to support faster computation. That could also be accomplished by programming in Common Lisp, but ACL2 has the advantage of supporting proofs of properties of its functions. Among the most basic properties it proves are that functions terminate and are applied only to suitable arguments (see the discussion of guard verification in Section 4). Those proofs are typically well supported by ACL2, even when df expressions are involved.

ACL2 can also prove facts about specific evaluations. For example, ACL2 can prove the following formalization of $sin(\pi/2) = 1$.

```
(equal (df-sin (df/ (df-pi) 2)) 1)
```

That proof is performed using evaluation, where the ACL2 function `df-sin` invokes the Common Lisp function `sin` on the result of dividing the Common Lisp approximation to π by 2. The axiomatic foundation for such evaluation includes the implicit table of all Common Lisp computations applying `sin` to a double-float. We omit details other than to point the interested reader to the documentation for ACL2's `partial-encapsulate` feature [9] and comments in ACL2 source file `float-a.lisp`.

However, ACL2 has only limited support for reasoning about floating-point operations when variables are present. For example, ACL2 fails to prove the following.

```
(implies (and (dfp x) (<= 0 x) (<= x 1))
         (<= (df* 2 x) 3))
```

Additional support for floating-point reasoning may be addressed in the future if there is user demand. For example, we may ultimately formalize `df-round` as rounding to nearest even, perhaps following Russinoff's ACL2 formalization of that operation [15].

Two basic properties that *can* be proved are commutativity of addition and multiplication, as follows.

```
(equal (df+ x y) (df+ y x))
(equal (df* x y) (df* y x))
```

These prove because `(df+ x y)` is defined to be `(df-round (+ x y))` (as noted earlier) and `(df* x y)` is defined to be `(df-round (* x y))`. Notice by the way the use of `equal` instead of `df=`; either is OK, but only `df=` would be allowed in definitions.

Unlike commutativity, associativity cannot be proved for `df+` or `df*`, since it fails.

```
ACL2 !>(df+ #d0.1 (df+ #d0.2 #d0.3))
#d0.6
ACL2 !>(df+ (df+ #d0.1 #d0.2) #d0.3)
#d0.6000000000000001
```

ACL2 can prove the following trivial theorem, as `from-df` is logically the identity function.

```
(equal (from-df x) x)
```

Of course, `to-df` is not the identity function; for example, 1/3 is not a representable rational, so `(to-df 1/3)` cannot equal 1/3. But ACL2 can prove:

```
(dfp (to-df x)).
```

However, ACL2 can prove that `to-df` is idempotent.

```
(equal (to-df (to-df x)) (to-df x))
```

Here are two simple theorems provable by ACL2.

```
(implies (dfp x)
         (equal (df- x x)
                0))
(implies (and (dfp x)
              (not (equal x 0)))
         (equal (df/ x x) 1))
```

Finally, we note that two Common Lisp implementations may prove contradictory theorems, because the IEEE spec does not make requirements on all floating-point operations. The following examples illustrate the fact that indeed, different Lisp implementations can compute slightly different values for trigonometric functions.

```
;;; ACL2 built on LispWorks:
(equal (df-sin (df* 2 *df-pi*)) #d-2.4492127076447545E-16)

;;; ACL2 built on other than LispWorks:
(equal (df-sin (df* 2 *df-pi*)) #d-2.4492935982947064E-16)
```

However, it has long been part of the ACL2 soundness claim that a single proof development should not be performed using more than one Lisp implementation.

6.6 Limitations

Our approach is limited by syntactic restrictions. In particular, one cannot form lists of double-floats using ACL2, since the list-formation function, `cons`, takes non-df inputs. (Arrays of double-floats provide an efficient workaround, and are used in Section 7. These are fields of single-threaded objects (stobjs); further discussion is beyond the scope here.)

But ACL2 also inherits a limitation from Common Lisp: floating-point computations do not always yield mathematically exact results. The following examples reflect attempts to compute $sin(\pi/6) = 1/2$ and $cos(\pi/2) = 0$.

```
ACL2 !>(df-sin (df* (to-df 1/6) (df-pi)))
#d0.49999999999999994
ACL2 !>(df-cos (df* #d0.5 (df-pi)))
#d6.123233995736766E-17
```

There are several reasons why the computed values are not exact. One is that 1/6 is not representable as a double-float, so `(to-df 1/6)` is not numerically equal to 1/6. A second is that π is not a rational number, hence not numerically equal to the value of `(df-pi)`. Third, the multiplications performed by `df*` may involve rounding. And finally, the mathematical *sin* and *cos* functions are transcendental and hence cannot be expected to be perfectly represented by `df-sin` and `df-cos` since they logically return rationals.

IEEE Standard 754 comprehends non-numeric results, including NaNs and infinities, from floating-point computations. These are completely avoided in

ACL2, which causes an error when a df operation would return such a result. Implementation details on how that is accomplished are generally beyond the present scope, but we give a brief outline for those who want to understand more by exploring the ACL2 source code. Four of the host Lisps support causing errors for what would be non-numeric results, and the function `break-on-overflow-and-nan` exploits such support provided by CCL, SBCL, GCL, and CMUCL. For the other two supported host Lisps, LispWorks and Allegro CL, the primitive df operations (such as `binary-df+`) are introduced with macros `defun-df-binary` and `defun-df-unary`, each of which lays down a call of macro `df-signal?` to check the results.

That IEEE standard also discusses a negative zero, typically written as -0.0. But this is not an issue, since ACL2 has only rational numbers, not true floating-point numbers. In particular, the input `#d-0.0` is read in as the rational number 0.

Finally, handling of underflow and overflow reflect Common Lisp. Here are examples.

- An attempt to read `#d1E310` causes an overflow error.
- The input `#d1E-500` is read as the number 0.

7 A Realistic Example

As noted in Section 3, one application of ACL2 is circuit simulation, which in the case of [5] involves solving systems of linear equations $\mathbf{Ax} = \mathbf{b}$ for sparse matrices $\mathbf{A}$. That paper described a solver written in Lisp, not ACL2, because at the time ACL2 did not support floating-point computations. Now that we have double-floats in ACL2 we have coded a new solver in ACL2 and we describe it briefly here. It is important to note that this solver is in no way competitive with the numerous linear algebra packages available today that exploit platform- and OS-specific hardware, memory hierarchies, block memory operations, instruction level parallelism, and other features of superscalar processors, clusters, and supercomputers.

Our solver is just a straightforward, sequential ACL2 implementation of a Gaussian elimination-style solver but the matrix representation is designed for sparse $\mathbf{A}$: abstractly each row is a list of double-float coefficients optionally separated by single elements denoting zero-filled "gaps" of specified lengths. But dfs cannot be stored in conses, so we store the coefficients in a `double-float` array called the "heap" which is a component of a mutable single-threaded object; the index of each coefficient is stored in the row as a natural number. The basic operation of adding the product of a scalar and a row to another row is straightforward. The df arithmetic expression involved is

```
(df+ (df* scalar
          (coefi (the (integer 0 *) (car row1)) heap))
     (coefi (the (integer 0 *) (car row2)) heap)).
```

We organize rows into blocks by the number of leading zeros and implement pivoting so as to use the row with the largest (absolute value) coefficient to cancel the leading coefficient of every other row in the block. As we reduce **A** to triangular form we also maintain a "program" that, when run on any given **b**, will perform the same transformation that would have occurred had we augmented **A** with **b** in an extra column. This allows us reduce **A** once and then quickly solve for multiple **b**s.

When applied to the 1036 $\times$ 1036 matrix **A** mentioned in Section 3, (where 99.75% of the entries are zero), this code reduces the matrix in 0.00137 seconds on a MacBook Pro running ACL2 in SBCL. This is approximately twice as fast as the Lisp (not ACL2) solver reported in [5] (though the speed up is due to the change of matrix representation, not df). By further refining the implementation to use a stobj containing resizable arrays for rows and for blocks we reduced the time for the df solver on this problem to 0.0008 seconds. The answers computed and their accuracy were identical to those described in Section 3.

We have not yet verified termination or the guards of this solver. The reason is that we are still refining it. Because the basic algorithm is just Gaussian elimination, it will not scale to significantly larger matrices. We aim to implement better algorithms going forward, now that we can do double-float calculations in ACL2 with decent efficiency.

The solver, named `df-solver-v9.lisp`, and the 1036 x 1036 matrix example, named `big-a-and-big-b.lsp`, may be found in the ACL2 regression suite [1] under the directory `projects/gaussian-elim-solvers/`.

8 Conclusion

We have added support for doing floating-point calculations within the ACL2 system while preserving the pre-existing axioms for arithmetic and a large regression suite containing verified theorems and other logical tools. We provide only limited support for reasoning about floating-point arithmetic but enough support to enable use of the underlying (and IEEE 754 compliant) floating-point hardware for calculations within ACL2, including in proofs. This is especially meaningful in light of the fact that much of today's floating-point hardware designs have been verified (often by ACL2) as being compliant. All of this is just another small step toward the goal, shared by all of us in the formal methods community and especially by Cliff Jones, of making formal methods a practical and widely used tool in industrial hardware and software development.

Acknowledgments. We thank ForrestHunt, Inc. for support for this work. We especially thank Warren Hunt for encouraging us to pursue support for floating point in ACL2. For a long time we did not see a way to do it without changing the logic (e.g., the axioms for equality and arithmetic). That in turn would have required changes to the regression suite too numerous to mention. But Warren was insistent that we keep trying and we ultimately discovered the method described here. Thank you Warren, not just for the support but for the encouragement.

We also thank the reviewers for excellent feedback, which has resulted in a number of improvements.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. ACL2 User Community: ACL2 community books, https://github.com/acl2/acl2/tree/master/books
2. Boyer, R.S., Moore, J.S.: Single-threaded objects in ACL2. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2257, pp. 9–27. Springer (2002). https://doi.org/10.1007/3-540-45587-6_3, https://doi.org/10.1007/3-540-45587-6_3
3. Brock, B., Kaufmann, M., Moore, J.S.: ACL2 theorems about commercial microprocessors. In: Srivas, M., Camilleri, A. (eds.) Formal Methods in Computer-Aided Design (FMCAD'96). pp. 275–293. Springer-Verlag, LNCS 1166, Heidelberg (November 1996). https://doi.org/10.1007/BFb0031816, http://www.cs.utexas.edu/users/moore/publications/bkm96.ps.Z
4. Hunt, Jr., W., Kaufmann, M., Moore, J.S., Slobodova, A.: Industrial hardware and software verification with ACL2. In: Verified Trustworthy Software Systems. vol. 375. The Royal Society (2017). https://doi.org/10.1098/rsta.2015.0399, (Article Number 20150399)
5. Hunt, Jr., W.A., Ramanathan, V., Moore, J.S.: Vwsim: A circuit simulator. In: Sumners, R., Chau, C. (eds.) Proceedings Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, 26th-27th May 2022. Electronic Proceedings in Theoretical Computer Science, vol. 359, pp. 61–75. Open Publishing Association (2022). https://doi.org/10.4204/EPTCS.359.7
6. Kaufmann, M., Manolios, P., J S. Moore: Computer-Aided Reasoning: An Approach. Kluwer Academic Press, Boston, MA. (2000). https://doi.org/10.1007/978-1-4615-4449-4
7. Kaufmann, M., Moore, J.S.: The ACL2 home page. In: http://www.cs.utexas.edu/users/moore/acl2/. Dept. of Computer Sciences, Univ. of Texas at Austin (2024)
8. Kaufmann, M., J S. Moore: ACL2 Documentation for DF, see http://acl2.org/manual/index.html?topic=ACL2____DF
9. Kaufmann, M., J S. Moore: ACL2 Documentation for Partial Encapsulation, see http://acl2.org/manual/index.html?topic=ACL2____PARTIAL-ENCAPSULATE
10. Kaufmann, M., J S. Moore: ACL2 Documentation for TERM, see http://acl2.org/manual/index.html?topic=ACL2____TERM
11. Kaufmann, M., J S. Moore, The ACL2 Community: The Combined ACL2+Books User's Manual. http://acl2.org/manual/index.html (2021)
12. LispWorks: Common Lisp Documentation, see http://www.lispworks.com/documentation/common-lisp.html
13. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) 11th International Conference on Automated Deduction (CADE). pp. 748–752. Lecture Notes in Artificial Intelligence, Vol 607, Springer-Verlag, Heidelberg (June 1992)
14. Pitman, K.: The Common Lisp HyperSpec, see https://www.lispworks.com/documentation/HyperSpec/Front/

15. Russinoff, D.M.: Formal Verification of Floating-Point Hardware Design - A Mathematical Approach, Second Edition. Springer (2022). https://doi.org/10.1007/978-3-030-87181-9, https://doi.org/10.1007/978-3-030-87181-9
16. Standards Committee of the IEEE Computer Society: IEEE standard for binary floating-point arithmetic. Tech. Rep. IEEE Std. 754-1985, IEEE, 345 East 47th Street, New York, NY 10017 (1985)