Cleaning up the Tower: Numbers in Scheme

Sebastian Egner
Philips Research Laboratories
sebastian.egner@philips.com

Richard A. Kelsey
Ember Corporation
kelsey@s48.org

Michael Sperber
sperber@deinprogramm.de

Abstract

The R^{5}RS specification of numerical operations leads to unportable and intransparent behavior of programs. Specifically, the notion of "exact/inexact numbers" and the misleading distinction between "real" and "rational" numbers are two primary sources of confusion. Consequently, the way R^{5}RS organizes numbers is significantly less useful than it could be. Based on this diagnosis, we propose to abandon the concept of exact/inexact numbers from Scheme altogether. In this paper, we examine designs in which exact and inexact rounding *operations* are explicitly separated, while there is no distinction between exact and inexact numbers. Through examining alternatives and practical ramifications, we arrive at an alternative proposal for the design of the numerical operations in Scheme.

1 Introduction

The set of numerical operations of a wide-spectrum programming language ideally satisfies the following requirements:

efficiency The programming language's operations are reasonably efficient relative to the capabilities of the underlying machine. In practice, this means that a program can employ fixnum and floating-point arithmetic where reduced precision is acceptable.

accuracy A program computes with numbers without introducing error.

reproducibility The same program, run on different language implementations, will produce the same result.

transparency The programmer can tell when a result is the outcome of inexact operations and thus contains error, or when a computation is reproducible exactly.

In practice, efficiency and accuracy are often in conflict: Accurate computations on non-integral numbers are often (but not always) prohibitively expensive. Fast floating-point arithmetic introduces error. Thus, a realistic programming language must choose a compromise between the two—which introduces the need for transparency. Reproducibility is clearly desirable, but also often in conflict with efficiency—the most efficient method for performing a computation on one machine may be inefficient on another. At least, a programmer should be able to predict whether a program computes a reproducible result. Moreover, as many practical programs as possible should in fact run reproducibly.

Scheme 48 1.1 (default mode)	`7/10`
Petite Chez Scheme 6.0a, Gambit-C 3.0, Scheme 48 1.1 (after `,open floatnums`)	`3152519739159347/4503599627370496`
SCM 5d9	`1`
Chicken 0/1082:	"can not be represented as an exact number"

Table 1. Value of `(inexact->exact 0.7)` in various R^{5}RS Scheme implementations

R^{5}RS [13] provides for *exact* and *inexact* numbers, with the idea that operations on exact numbers are accurate but potentially inefficient and operations on inexact numbers are efficient but introduce error. The intention behind R^{5}RS is to hide the actual machine representation of numbers, and to allow the program (or the programmer) to look at a number object and determine whether it contains error. In theory, this would fulfill a reasonable transparency requirement. In practice, however, the numerical operations in Scheme are anything but transparent.

For a trivial example exhibiting the poor reproducibility of R^{5}RS programs, consider the value of the expression `(inexact->exact 0.7)` in various Scheme systems, all of which are completely R^{5}RS-compliant. Table 1 shows that the results vary wildly—Scheme 48 can even change its behavior at run time. This is only one of a wide variety of problems a programmer faces who tries to predict the outcome of a computation on numbers in a Scheme program. Clearly, R^{5}RS provides for little reproducibility.

What is the cause of these problems? R^{5}RS takes the stance that programs using exact arithmetic are essentially the same as programs using inexact arithmetic. The procedures they use are the same, after all—only the numbers are different. In reality, programs using inexact arithmetic operations are inherently different from programs using only exact operations. By blurring the distinction, R^{5}RS complicates writing many programs dealing with numbers. We know of no design approach which does successfully unite exact and inexact arithmetic into a common set of operations without sacrificing transparency and reproducibility.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission.
Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA. Copyright 2004 Sebastian Egner, Richard A. Kelsey, Michael Sperber.

In this paper, we examine the specific problems with the numerical operations specified in R^{5}RS, and consider alternative designs. Specifically, our contributions are the following:

- We identify the design fallacies concerning the numerical operations specified in R^{5}RS, and their practical consequences.
- We show how to design numerical operations around the idea that *operations* rather than *numbers* are exact or inexact. The design has the following properties:
  - It uses a different numerical tower that is a more appropriate model for realistic program, and which a Scheme system can realistically represent accurately.
  - All standard numerical operations operate exactly on rational numbers of infinite precision.
  - The floating-point operations are separate procedures.
- We examine the choices available in such a design, and discuss their consequences and relative merits. The choices concern the relationship between the rational numbers and the floating-point numbers—whether floating-point numbers count as rationals and vice versa. We also discuss the nature of $\pm\infty$ and "not-a-number," and where they fit in our framework.

  All of the design alternatives we examine allow compromises between efficiency and accuracy similar to what R^{5}RS currently provides, as well as improved transparency and reproducibility: Any program that does not contain calls to floating-point operations always computes exactly and reproducibly, independent of the Scheme implementation it runs on. Rounding conversions from rational to floating-point numbers only occur at clearly identifiable places in a program.
- We identify some weaknesses in the R^{5}RS set of numerical operations as they pertain to the new design, and describe possible approaches to addressing them. This includes the definitions of `quotient`, `remainder`, and `modulo`, the definitions of the rounding operations, and dealing with external representations.

We do not address all the issues concerning numbers that a future revision of the Scheme standard should address. Specifically, we do not discuss the relative merits of tying the Scheme standard to a specific floating-point representation. We do not touch the issue of offering abstractions for explicitly controlling their propagation, as well as the rounding mode of floating-point operations: This is addressed in detail elsewhere, for example in the recent work on floating-point arithmetic in Java [4]. Also, we omit complex numbers and other advanced number representations (such as algebraic numbers, interval arithmetic, cyclotomic fields etc.) from the discussion; they are largely orthogonal to the subject of this paper.

Overview We identify the main problems with the R^{5}RS approach in Section 2. In Section 3, we present a new model for exact rational arithmetic and a set of typical machine representations for it. Section 4 describes how to add inexact arithmetic to the model, along with the design issues arising from this. Section 5 explores design alternatives within our model. In Section 6, details a possible set of exact numerical operations. Some implementation issues are discussed in Section 7. Finally, Section 8 lists some related work, and Section 9 concludes.

2 Problems with the R^{5}RS approach

R^{5}RS specifies that the objects representing numbers in a Scheme system must be organized according to a subtype hierarchy called the *numerical tower*:

$$\text{number} \supseteq \text{complex} \supseteq \text{real} \supseteq \text{rational} \supseteq \text{integer}$$

Section 6.2.3 of R^{5}RS requires implementations to provide "a coherent subset consistent with both the purposes of the implementation and the spirit of the Scheme language." Moreover, implementations may provide "only a limited range of numbers of any type, [...]". As a minimal requirement, exact integers must be present "throughout the range of numbers that may be used for indexes of lists, vectors, and strings [...]".

In addition, Section 6.2.2 specifies that the representation of a number is flagged either exact or inexact and that operations should propagate inexactness as a contagious property. Hence, numbers in R^{5}RS are not just organized according to the numerical tower, but also according to exactness. The exact/inexact distinction is claimed to be "orthogonal to the dimension of type."

The rest of this section enumerates some of the most significant problems with the R^{5}RS specification.

2.1 Not enough numbers

Numbers are in short supply in R^{5}RS. As quoted above, the only numbers a Scheme system must support are indices for arrays, lists and strings. A Scheme system that supports only integers $\{0,\ldots,255\}$ can be perfectly conformant.

Of course, the use of limited-precision fixnum arithmetic can improve performance. However, we conjecture that the cost of allowing the standard arithmetic operations to only support limited precision—the loss of algebraic laws, transparency and reproducibility—is greater than the benefit.

2.2 Unspecified precision of inexact numbers

R^{5}RS puts no constraints on the range or precision of inexact numbers. In particular, the description of `exact->inexact` and `inexact->exact` says

> These procedures implement the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range.

More tellingly, it also says

> If an exact[inexact] argument has no reasonably close inexact[exact] equivalent, then a violation of an implementation restriction may be reported.

The "may" implies that implementations are free to use an arbitrarily inaccurate equivalent. Moreover, the meaning of "reasonably close" is similarly underspecified: Table 1 shows that a call to `inexact->exact` that works fine on one implementation may actually signal an error on another, even if the argument is the same.

2.3 No exact fixnum-only arithmetic

R^{5}RS specifies in Section 6.2.2:

> If two implementations produce exact results for a computation that did not involve inexact intermediate results, the two ultimate results will be mathematically equivalent.

This makes it hard for a Scheme system to support only limited-precision integers, as it requires the system to mark results of overflows as inexact—which in turn usually means some loss of efficiency (if boxing is involved in the construction of inexact numbers or a conversion to a different representation) or additional loss of precision (if an additional bit in the fixed-size representation denotes inexactness).[1]

2.4 Numerical promotion loses information

The idea of a "numerical tower" suggests that the more general number types contain the more specific ones. In particular, there is usually a mechanism converting number types automatically when required for an operation, a process often called "numerical promotion", [15, Section 6.5.2.2, §§6, 7], [9, Section 5.6]. In Scheme, automatic conversion can occur both along the integer–number axis and along the exact–inexact axis. Also, information may get lost during any conversion of numbers, even *up* the tower. In MzScheme, for example, the following occurs:

```
(+ (expt 10 309) 0.0) ⟹ +inf.0
```

even though `(expt 10 309)` has an exact representation as an integer.

There are several reasons for loss of information. Bignums can be arbitrarily large, whereas fixed-precision floating-point formats are limited in range. Exact rationals can have arbitrary precision, whereas binary floating-point formats, even if they have arbitrary precision, can only represent binary fractions where the denominator is a power of two.

In effect, the intuition of the numerical tower as a chain of subsets is invalid for all actual Scheme systems having a floating-point representation or imposing limits on the number types.

2.5 Lack of "inexact control-flow"

R^{5}RS defines in Section 6.2.2 which numbers are exact:

> A number is exact if it was written as an exact constant or was derived from exact numbers using only exact operations.

Unfortunately, the following function is perfectly capable of returning an exact result given inexact input:

```
(lambda (x y) (if (< x y) -1 1))
```

Clearly, the reason is that the comparison < returns an exact result, either `#t` or `#f`. This is especially pernicious given that comparisons are one place where the inaccuracies of floating-point numbers may really hurt. In effect, the accuracy of the function's value is no greater than the accuracy of the input—but in Scheme's type system the result is treated as entirely exact.

[1] This seems to be why Bigloo 2.5c, for instance, has `(expt 2 50)` ⇒ `0` but `(exact? (expt 2 50))` ⇒ `#t` (in violation of the standard). Chicken 0/1082, which also does not support bignums, has `(expt 2 50)` ⇒ `1125899906842624.` and `(exact? (expt 2 50))` ⇒ `#f`.

To ensure that an exact result is not dependent on inexact operations the programmer either has to do a careful analysis of the program (in which case any run-time checking is irrelevant) or use exact comparison operations like the following:

```
(define (exact< x y)
  (if (and (exact? x) (exact? y))
      (< x y)
      (error ...)))
```

2.6 Exactness of numerical literals

Section 6.2.4 of R^{5}RS states: "If the written representation of a number has no exactness prefix, the constant may be either inexact or exact. It is inexact if it contains a decimal point, an exponent, or a "`#`" character in the place of a digit, otherwise it is exact."

The consequence is often that the global behavior of a program is governed by the presence or absence of a single decimal point: A program can become intolerably inaccurate through the presence of a decimal point, or intolerably slow through the omission of one. The example described in Figure 1 illustrates why this may be unfortunate.

2.7 Meaning of standard procedures

Some of the standard procedures defined in R^{5}RS only make sense for certain types of numbers, e.g., `gcd` for exact integers or `log` for inexact real or complex numbers.

This is a temptation for implementations to fill in the gap and define things like `(gcd 2.0 6.0)` in the "obvious" way, violating the intended meaning of standard procedures. In the following example (again run under MzScheme), the "greatest common divisor" might be greater than expected:

```
(gcd (expt 2 40) (expt 3 40))  ⟹ 1
(gcd (expt 2 40) (expt 3 40.)) ⟹ 2048.0
```

2.8 Exchanging numbers between Scheme systems

There is no guarantee that two R^{5}RS-compliant Scheme systems can successfully exchange numerical data via the written representations provided by the standard procedures. For exact integers, there seems to be no problem—provided the receiving system covers the range required by the sender. The notation `1/2` already poses a problem because rationals are not mandatory. The specification of `number->string` and `string->number` laudably caters to read/write invariance, but does so only for numbers written and read by the same Scheme system.

3 Exact arithmetic

The analysis in the previous section suggests that the numerical tower of R^{5}RS is not a good model for numerical computations in a computer program—at least not for all of them. Moreover, attributing exactness to numbers in the way R^{5}RS leads to inconsistencies.

In this section and the following two, we examine the consequences of splitting *operations* along the exact/inexact axis instead of the *numbers*. The exact arithmetic operations satisfy strong algebraic properties such as associativity, commutativity, distributivity, total

A typical example of surprises with mixed exact/inexact computation appeared in Sperber's Introductory Computing class. Students had to write a procedure for visualizing the Mandelbrot set. The task boils down to iterating the function $z \mapsto z^2 - c$ for different complex parameters c. For visualization, the procedure `draw-mandelbrot` enumerates points in a rectangle defined by upper left corner, width and height.

Many students observed that their program seemed to "hang" for some inputs, but not for others. This occurred when only literals without decimal point were used as operands for `draw-mandelbrot`—in which case the program computes the iteration using exact fractions. As the iteration progresses, the internal representation of the fraction gets very large very quickly.

Putting a decimal point into one of the numerical literals or placing an `exact->inexact` at almost any point in the program would fix things; there is no recognizably "right" place for it. Students find especially confusing that the seemingly "simpler"—integral—numblers cause problems, while the "more complicated" floating-point numbers do not.

The example illustrates the limited predictability of Scheme programs mixing exact and inexact numbers.

Figure 1. A real-world example

ordering etc. Initially, we consider the exact world only. We show how to add inexact operations later.

We take the following abstract numerical tower as the basis for our numerical operations:

$$\mathbb{Q} \supseteq \mathbb{Q}_{10} \supseteq \mathbb{Q}_2 \supseteq \mathbb{Z} \supseteq \mathbb{Z}_{\geq 0} \supseteq \mathbb{Z}_{>0}.$$

In this chain $\mathbb{Q}$ denotes the rational numbers, $\mathbb{Q}_b$ denotes the b-ary fractions, i.e. the set of rational numbers with denominator a power of b (binary fractions for $b = 2$ and decimal fractions for $b = 10$.)[2] $\mathbb{Z}$ denotes the integers, $\mathbb{Z}_{\geq 0}$ denotes the non-negative integers, and $\mathbb{Z}_{>0}$ denotes the positive integers.

While this view of the rational numbers may appear arbitrary or theoretical at first glance, it identifies and names the kinds of numbers that computer programs typically distinguish. In particular, positive and non-negative integers are so frequent in any sort of program that we propose to name them in the core language itself.

To relate the tower elements to machine representations, we use the following terminology, borrowed from R^{5}RS: *Fixnums* are the fixed-width machine representation for integers—denoted by fixnum. *Bignums* are the arbitrary-width exact representations for arbitrary integers, named bignum. *Flonums* are the fixed-precision floating-point machine representation for rational numbers, named flonum. Finally, *fractions* are the tuple representations for rational numbers, using bignum numerator and denominator, called fraction.

The relationships between the tower elements and the machine representations are as follows:

1. The fixnum representation implements a subset of $\mathbb{Z}$.
2. The bignum representation implements $\mathbb{Z}$, only limited by available memory.
3. The flonum representation implements a subset of $\mathbb{Q}_2$, possibly augmented by special objects like -0, $\pm\infty$ and NaN, which are not elements of $\mathbb{Q}$.
4. Human-readable representations are typically decimal fractions—elements of $\mathbb{Q}_{10}$—at least conceptually.
5. The fraction representation implements $\mathbb{Q}$, only limited by available memory.

We propose that the default operations on rational numbers, that means the standard procedures +, -, *, /, <=, etc., are all exact: Conceptually, they accept rational arguments and return rational results. Of course, implementations may take advantage of more efficient machine representations (employing fixnums and flonums) if possible, but conversion may only take place if no loss of information occurs in the process. Thus, the particular machine representation of a number is purely an efficiency issue.

In practice, this means the following:

- Each number object represents a unique, precisely defined rational number. Rational numbers have conceptually infinite precision.
- Different machine representations of the same rational number may coexist, but they are all equivalent. (Processing time may differ, of course.)
- Rational numbers are treated exactly the same way as R^{5}RS currently treats exact integers and rationals.
- The exact operations satisfy *all* algebraic properties (associativity, commutativity, distributivity, total ordering, etc.) of their mathematical counterparts.

4 Adding inexact arithmetic

Exact operations alone, even combined with explicit rounding, are not sufficiently efficient for many numerical computations. Therefore, the language should provide access to the underlying floating-point hardware, if available, through a default set of inexact operations. For example, `float+` would accept two flonums and return a flonum. By nature, `float+` sacrifices algebraic properties to gain efficient execution. However, by distinguishing exact and inexact operations explicitly, the actual arithmetic used becomes a property of the program, rather than a property of the numbers it processes. (Note that the arithmetic model remains a dynamic property in any language with exact and inexact numbers, even if operations are required to accept only all exact or all inexact argument values.)

By distinguishing exact and inexact operations explictly, we give up a potential source of code reuse: Even if an algorithm works for both exact and inexact operations alike, our proposal requires two different programs—one calling the exact operations, one calling the rounding operations. We are proposing to pay this price because sensible algebraic and numerical algorithms seem to be distinct most of the time.

Of course, practical implementations of the inexact operations will use a limited-precision floating-point representation for numbers. This raises the question of how these representations relate to the other representations for rational numbers. Do the floating-point representations form a subset of the rational representations? What about $\pm\infty$, NaN, and distinct -0? The issue of the special floating-point objects is central to this issue. We discuss $\pm\infty$ and NaN separately from distinct -0:

[2] What we denote as $\mathbb{Q}_b$ is not identical to the algebraic concept of "field of p-adic numbers."

4.1 The case against rational $\pm\infty$ and NaN

The special objects $+\infty$ and $-\infty$ are used in the floating-point world as a mechanism to carry on with a computation in the presence of overflow. They are usually the results of $\mathit{positive}/\mathit{tiny} = +\infty$ and $\mathit{positive}/(-\mathit{tiny}) = -\infty$, which can happen without the programmer being aware of it.

In the exact world, however, the only way of obtaining infinity is a division by zero. The question is whether the system should then signal an error, or return a special object representing infinity. An argument in favor of $\pm\infty$ is that they provide neutral elements for the minimum and maximum, i.e., `(min)` $\Rightarrow +\infty$, `(max)` $\Rightarrow -\infty$.

Nevertheless, an exact division by zero is virtually always a symptom of a genuine programming error or of illegal input data, and the introduction of infinity will only mask this error.

NaN ("not a number") is the strongest form of delaying an error message. NaN is a special object indicating that the result of an arithmetic operation is undefined; one way it could emerge is $(+\infty)+(-\infty) = \text{NaN}$. The advantage of returning NaN instead of raising an error is that the computation still continues, postponing the interpretation of the results to a more convenient point in the program. In this way, NaN is quite useful in numerical computations.

The problem with NaN is that the program control structure will mostly not recognize the NaN case explicitly. Assume we define comparisons with NaN always to result in `#f`, as IEEE 754 does, then

```
(do ((x NaN (+ x 1))) ((> x 10)))
```

will hang but

```
(do ((x NaN (+ x 1))) ((not (<= x 10))))
```

will stop, which is counter-intuitive and may be surprising.

While $\pm\infty$ and NaN are quite useful for inexact computations, there is a high price to pay when they are carried over into the exact world: The rational numbers must be extended by the special objects, and the usual algebraic laws will not hold for the extension anymore. Moreover, the special objects obscure exact programs by masking mistakes.

4.2 The case against rational -0

The purpose of distinguishing a "positive zero" $(+0)$ and a "negative zero" (-0) in a floating-point format is to retain the sign of numbers in the presence of underflow, e.g., $-0 = \mathit{positive}/(-\mathit{huge})$. Since comparisons must allow for tolerances, there is no real harm done identifying $+0$ (positive) with *the* zero, which is neither positive nor negative. The use of signed zeros simplifies dealing with branch cuts [11] and generally helps obtaining meaningful numerical output.

In the exact world, on the other hand, there is no underflow—only memory overflow. Even worse, adding one (or even two) signed "zeros" to the rational numbers completely destroys the rich, clean and simple algebraic structure which the rational numbers do posses. We briefly detail this mathematical fact.

The set $\mathbb{Q}$ of rational numbers equipped with the addition operation $+$ form an abelian group. This means the following:

(C) For all $x, y \in \mathbb{Q} : x+y = y+x$.

(A) For all $x, y, z \in \mathbb{Q} : (x+y)+z = x+(y+z)$.

(Z) There is $Z \in \mathbb{Q}$ such that for all $x \in \mathbb{Q} : Z+x = x$.

(I) For all $x \in \mathbb{Q}$ there is a $y \in \mathbb{Q} : x+y = Z$.

Now take elements $Z, Z' \in \mathbb{Q}$ such that $Z'+x = x$ for all $x \in \mathbb{Q}$ and also $Z+x = x$ for all $x \in \mathbb{Q}$. Then $Z = Z'+Z = Z+Z' = Z'$, where the second equation holds by (C). Consequently, there is *only one* element $Z \in \mathbb{Q}$ having property (Z). Therefore, this element receives the special name 0 (read "zero"). Now if we augment the set $\mathbb{Q}$ into $\mathbb{Q}'$ by forcibly adding another algebraic zero as in $\mathbb{Q}' = \mathbb{Q} \cup \{?\}$ where $?+x = x$ for all $x \in \mathbb{Q}'$ and $? \notin \mathbb{Q}$, then either property (C), or property (Z), or both get lost. This implies that property (I) at least suffers, because the uniqueness of y (which is in fact $-x$) gets lost. This carries on like wildfire, usually destroying nearly *all* algebraic properties at the same time; associativity may survive.

More generally, four different alternatives for dealing with '-0' in the exact world can be identified:

(a) Augment the rational numbers by one (or two) objects behaving like "a zero." Algorithmically, this means that all exact operations must dispatch on these special objects and define some action.

(b) Identify both floating-point values $+0$ and -0 with the rational number 0. In other words, exact operations treat ± 0 and 0 identically.

(c) Represent the floating-point value -0 by some negative rational number, say $-Z$. Conceptually, exact operations first replace -0 by the rational number $-Z$ and then do their work.

(d) It is an error to apply an exact operation to -0.

As explained above, the semantic cost of adding one or more "zeros" is quite high. This is a strong argument against alternative (a). In the other extreme, alternative (d) breaks the symmetry between positive and negative numbers. The problem with alternative (c) is to find a sensible definition of the rational equivalent of -0 (read "negative underflow.") A first approach might be: "-0 behaves like the smallest negative rational larger than any representable float." Unfortunately, there is no such rational number: Let $-f$ denote the largest representable negative float. Then $-f+1/n$, $n \in \{1, 2, 3, \ldots\}$, are not representable and increasingly close to $-f$. So there must be a gap between $-f$ and whichever rational number $-Z$ is choosen as the rational interpretation of '-0'—*unless* the definition reads: "Any $-Z$ for $-f < -Z < 0$ may be chosen as the rational interpretation of '-0';" an approach we do not pursue.

Whatever the choice, a negative number equivalent $-Z$ of -0 will behave surprisingly different from the float -0. For example, repeatedly squaring $-Z$ will soon exhaust memory and printing the square of $-Z$ will print unrecognizably, unless one is willing to sacrifice Scheme's facility to print rationals without loss of information.

Since alternatives (a), (c) and (d) are unattractive, alternative (b) appears to us as the least disadvantagous; there simply seems to be no place for $-0 \neq 0$ in the exact world of rational numbers.

5 Relating exact and inexact arithmetic

As the previous discussion has shown, the special floating-point values -0, $\pm\infty$, and NaN have no place in the exact world—they are not rational numbers. Hence, in the following, we assume that it is an error to apply an exact operation such as + to $\pm\infty$ or NaN, whereas ± 0 are both treated as 0 by the exact operations.

At this point, it is natural to ask whether the inexact numerical operations such as `float+`, `float-` etc. should accept all rational numbers, or only those represented as flonum. If the inexact operations only accept flonum arguments, a Scheme system must provide at least a conversion operation `rational->float`. Similarly, should the exact operations accept flonums (unless they are special values)? In other words, should the domains for exact and inexact operations be completely disjoint, with explicit conversion at all times? Three basic alternative kinds of "type permeability" seem to exist in this spectrum:

#1 The flonum representation is just another partial machine representation for rational numbers (plus special values), and all numerical operations, exact or inexact, accept all rational numbers as arguments. It is, however, an error to apply exact operations to $\pm\infty$ and NaN.

#2 As in #1, flonum is just another partial representation of rational numbers (plus special values), but inexact operations are *only* defined on flonum. Programs make use of the (rounding) operation `rational->float` to convert explicitly.

#3 The flonum representation is completely distinct from implementation of rationals. In other words, the exact operations are not defined on flonum and the inexact operations are undefined for the non-flonum rational numbers. Programs use of `float->rational` and `rational->float` to convert explicitly.

All three alternatives could support a `float?` predicate that answers `#t` for all flonum arguments—including $\pm\infty$ and NaN. A `rational?` predicate would probably behave differently in the different alternatives: Whereas it would answer `#t` to all numbers except for $\pm\infty$ and NaN in #1 and #2, it would naturally be a converse of `float?` in #3. Probably, a `float-not-rational?` predicate that identifies $\pm\infty$ and NaN would also be useful.

Alternatives #2 and #3 both also require distinct *external* representations for flonum and non-flonum rationals. If an external representation denotes a flonum, it may also be desirable to require representation information to accurately determine the meaning of the literal. (More on the issue of external representation in Section 6.6.) Alternatively, all numerical literals denote rational numbers, and the program must convert them to flonum representation explicitly via `rational->float`.

Alternatives #1 and #2 can both be implemented as conservative extensions of R^{5}RS by the following measures:

- Support integers and rationals of arbitrary precision.
- Have all R^{5}RS numerical operations convert flonum arguments to fraction before proceeding. (Or assert correctness by other means.)
- Interpret "inexact" as "float." Specifically, take `inexact?` to mean `float?` and `exact?` as $\neg$`inexact?`. Define `exact->inexact` and `inexact->exact` as follows:

```
(define (exact->inexact n)
  (if (float? n)
      n
      (rational->float n)))

(define (inexact->exact n)
  (cond
   ((float? n) (float->rational n))
   ((number? n) n)
   (else
     (error ...))))
```

  Note that `(number? NaN)` $\Rightarrow$ `#f` and `(number? ±∞)` $\Rightarrow$ `#f`, while `(float? NaN)` $\Rightarrow$ `#t` and `(float? ±∞)` $\Rightarrow$ `#t`.
- Finally, add operations on flonum with a `float` prefix.

(Of course, `inexact?`, `exact?`, `exact->inexact`, and `inexact->exact` serve no purpose in this new organization of numbers and should disappear eventually.)

The only problem is that of literals: Alternative #1 would work most intuitively if unannotated numerical literals would always represent their rational counterparts exactly. Unfortunately, R^{5}RS requires that the presence of a decimal point or an exponent forces a literal to denote an inexact, and, thus, a floating-point number. Therefore, a true conservative extension still requires that "exact" numerical literals carry a `#e` prefix.

In any case, all alternatives feature full reproducibility for exact computations, and much-improved transparency because the program source code clearly shows when floating-point arithmetic happens. (As for the example in Figure 1: In our design, the program would *always* compute slowly. However, the program now behaves in a much more consistent and less confusing manner, and the cause for the problems is much easier to explain than with R^{5}RS, as is the remedy.)

6 Useful numerical operations

In this section, we disucss alternatives to R^{5}RS's default set of numerical representations. Any such design necessarily represents a subjective choice, however. It should be rich enough to be convenient (e.g. having both < and >) but leave less frequently used operations (like `gcd` and `lcm`) to specialized libraries.[3] Here is possible list of exact operations to be present in the core language of a Scheme system:

`rational?` `decimal-fraction?` `binary-fraction?`
`integer?` `non-negative-integer?` `positive-integer?`
(Section 6.1)
`negative?` `zero?` `non-negative?` `positive?`
`compare` [$= \operatorname{sign}(x-y)$] (Section 6.2)
`<` `<=` `=` `>=` `>` `min` `max` `sign` `abs`
(`if-sign` *x negative zero positive*) (Section 6.2)
`+` `-` `*` `/` `^` [alias `expt`]
`floor` `ceiling` `truncate` `extend` `round` (Section 6.3)
`round-fraction` `floor-log-abs` (Section 6.4)
`div` `mod` (Section 6.5)
`numerator` `denominator`
`string->rational` `rational->string` (Section 6.6)

[3] Of course, fractional arithmetics requires a gcd operation internally—but including rarely used operations in the default set carries a conceptual cost.

We discuss the major deviations from R^{5}RS.

6.1 Numbers

The type predicates `rational?` `decimal-fraction?` `binary-fraction?` `integer?` `non-negative-integer?` `positive-integer?` reflect the abstract chain of numbers as introduced in Section 3.

As mentioned already in Section 3, non-negative and positive integers are exposed because of their ubiquitous nature. Concerning decimal and binary fractions, refer to Section 6.4.

6.2 Comparisons

The additional comparison operations increase programming convenience. With respect to R^{5}RS, there are two major additions: The `compare` procedure and the `if-sign` special form dispatching on the sign of a rational number.

`Compare` has been included for efficiency. All other comparisons can be expressed in terms of a single call to `compare`, which can be implemented without allocating any intermediate objects at all.

`If-sign` has been included because a frequent task in programming is distinguishing between the three possible results of a comparison.

6.3 Rounding rationals to integers

For rounding rationals into integers, the procedures `floor`, `ceiling`, `truncate`, `extend` and `round` provide the rounding modes towards $-\infty$, $+\infty$, 0, $\pm\infty$ and towards the nearest integer. The precise mathematical definitions of these functions are the obvious ones, with the exception of breaking ties in `round`, which breaks ties towards even, just like R^{5}RS and IEEE 754.

All of these operations are useful and common in numerical programs: Breaking ties towards even and towards zero is symmetric in the sense that $\rho(-x) = -\rho(x)$ for all x, where ρ denotes the rounding function. Breaking ties towards $-\infty$ appears naturally in `div` and `mod` as defined in Section 6.5. Finally observe that rounding with breaking ties towards $\pm\infty$ is naturally related to floor and ceiling by $\lceil x-1/2\rceil$ and $\lfloor x+1/2\rfloor$.

6.4 Binary and decimal fractions

By providing a convenient function for rounding rationals into binary and decimal fractions, programs can easily implement floating-point operations of arbitrary precision in the absence of, or in addition to, proper floats. Among others, this provides a natural way of defining external representations for binary and decimal fractions accurately and portably. (A proposal is in Section 6.6.) We propose that

`(round-fraction` *base mantissa round x*`)`

maps the rational x into a number that has *mantissa* significant digits in its *base*-ary expansion and where rounding has been performed by applying the procedure *round* mapping rationals into integers. Figure 4 shows a possible implementation in R^{5}RS, assuming the presence of (bignum) rational arithmetics.

More explicitly, `(round-fraction` $b\ m\ \rho\ x$`)` should result either in 0, or in a number of the form

$$\hat{x} = \mathrm{sign}(x) \cdot (\hat{x}_0.\hat{x}_1 \cdots \hat{x}_{m-1})_b \cdot b^{\hat{e}}, \tag{1}$$

for b-ary digits $\hat{x}_0, \ldots, \hat{x}_{m-1} \in \{0, \ldots, b-1\}$, $\hat{x}_0 \neq 0$, and integer $\hat{e}$. Clearly, this only makes sense for integer b and m where $b \geq 2$ and $m \geq 1$.

Now consider the case $\hat{x} \neq 0$. Then

$$1 = (1.0\cdots)_b \leq (\hat{x}_0.\hat{x}_1 \cdots \hat{x}_{m-1})_b < b = (10.0\cdots)_b.$$

This implies $0 \leq \log_b(\hat{x}_0.\hat{x}_1 \cdots \hat{x}_{m-1})_b < 1$, from which follows

$$\lfloor \log_b |\hat{x}| \rfloor = \hat{e}.$$

This is the primary reason for proposing that

`(floor-log-abs` $b\ x$`)`

computes the largest integer e such that $b^e \leq |x|$ for integer b, $b \geq 2$, and non-zero rational x. Note that e is negative if and only if $|x| < 1$.

Coming back to `round-fraction`, define for $x \neq 0$

$$\hat{x} = \rho\left(x b^{m-e-1}\right) \cdot b^{-(m-e-1)}, \quad e = \lfloor \log_b |x| \rfloor. \tag{2}$$

Clearly, this definition can only result in the form (1) if the rounding function $\rho(-)$ is well-behaved. For this reason, we require that $\rho(u)$ is integer and $|\rho(u) - u| < 1$ for all rational u. This is the case for `round`, `floor`, `ceiling`, `truncate`, and `extend`. (In the case of `round`, even the tighter bound $|\rho(u) - u| \leq 1/2$ holds.) For $x = 0$ define $\hat{x} = 0$.

Under these conditions, the following error bound holds:

$$|\hat{x} - x| < b^{-m+1}|x|.$$

Proof:

$$\begin{aligned} |\hat{x} - x| &= |\rho\left(x b^{m-e-1}\right) b^{-(m-e-1)} - x| \\ &= b^{-(m-e-1)} |\rho\left(x b^{m-e-1}\right) - x b^{m-e-1}| \\ &< b^{-(m-e-1)} \\ &\leq b^{-m+1}|x|. \end{aligned}$$

It remains to be shown that the conditions on $\rho(-)$ imply the form (1). Observe that a positive u is never rounded into a negative $\rho(u)$, and vice versa. This means that we only need to consider $x > 0$. In this case, $b^e \leq x < b^{e+1}$ by definition of e, which implies $b^{m-1} \leq x b^{m-e-1} < b^m$. Applying ρ, we obtain

$$b^{m-1} \leq \rho(x b^{m-e-1}) \leq b^m,$$

because $\lfloor u \rfloor \leq \rho(u) \leq \lceil u \rceil$. Hence, we have shown that $\hat{x}$ has at most m non-zero digits in its b-ary expansion.

Note that `round-fraction` ignores several details of actual floating-point formats: The exponent of `round-fraction` is unlimited in magnitude, which means overflow and mantissa denormalization ($\hat{x}_0 = 0$) do not occur. Also underflow, the production of a number of magnitude too small to be represented, is not detected; it is simply rounded to zero.

6.5 `Div` and `mod`

Given an unlimited integer type, it is a trivial matter to derive signed and unsigned integer types of finite range from it by modular reduction. For example, arithmetic using 32-bit signed two's-complement behaves like computing with the residue classes "mod 2^{32}," where the set $\{-2^{31},\ldots,2^{31}-1\}$ represents the residue classes. Likewise, unsigned 32-bit arithmetic also behaves like computing "mod 2^{32}," but using a different set of representatives: $\{0,\ldots,2^{32}-1\}$.

Unfortunately, the R^{5}RS-operations `quotient`, `remainder`, and `modulo` are not ideal for this purpose. In the following example, `remainder` fails to transport the additive group structure of the integers over to the residues modulo 3.

```
(define (r x) (remainder x 3))
(r (+    -2     3 ))  ⟹  1
(r (+ (r -2) (r 3)))  ⟹ -2
```

In fact, `modulo` should have been used, producing residues in $\{0,1,2\}$. For modular reduction with symmetric residues, i.e. in $\{-1,0,1\}$ in the example, it is necessary to define a more complicated reduction altogether.

Therefore we propose operations div and mod (with Scheme counterparts `div` and `mod`), defined on all integers x,y, by the following properties

$$x = (x \operatorname{div} y)\cdot y + (x \bmod y), \tag{3}$$

$$\begin{array}{rcccll} 0 & \leq & (x \bmod y) & < & y & \text{if } y>0, \\ y/2 & \leq & (x \bmod y) & < & -y/2 & \text{if } y<0, \end{array} \tag{4}$$

$$x \operatorname{div} y \text{ is integer, and } x \operatorname{div} 0 = 0. \tag{5}$$

In other words, the sign of the modulus y determines which system of representatives of the residue class ring $\mathbb{Z}/y\mathbb{Z}$ is being chosen, either non-negative ($y>0$), symmetric around zero ($y<0$), or the integers ($y=0$).

The definition above implies

$$x \operatorname{div} y = \begin{cases} \lfloor \frac{x}{y} \rfloor & \text{if} \quad y>0, \\ 0 & \text{if} \quad y=0, \\ \lceil \frac{x}{y} - \frac{1}{2} \rceil & \text{if} \quad y<0. \end{cases}$$

This simplicity is the reason why the definition can be extended literally to define div and mod for all rational x,y. Mathematically, it even makes sense for all real x,y. For example, $(x \bmod 2\pi)$ and $(x \bmod -2\pi)$ both reduces x modulo 2π, and

$$0 \leq (x \bmod 2\pi) < 2\pi \text{ and } -\pi \leq (x \bmod -2\pi) < \pi.$$

Since div and mod offer both conventions which make sense, the R^{5}RS procedures `modulo`, `remainder`, and `quotient` can easily be defined in terms of `div` and `mod`. Of course it is also possible the other way around, albeit with more effort. Figures 2 and 3 show the definitions, respectively.

6.6 External Representations

We discuss some of the issues regarding external representatives arising from our design proposal in this section.

External representations occur in several contexts:

- literals in program source code,
- the output of `write` and the input of `read`,
- numbers printed out for human readers,
- numbers printed for consumption by other (non-Scheme) programs and read from other programs.

```
(define (quotient  n1 n2)
  (* (sign n1) (sign n2) (div (abs n1) (abs n2))))

(define (remainder n1 n2)
  (* (sign n1) (mod (abs n1) (abs n2))))

(define (modulo n1 n2)
  (* (sign n2) (mod (* (sign n2) n1) (abs n2))))
```

Figure 2. Defining `quotient`, `remainder`, `modulo` in terms of `div`, `mod`, `sign`, and `abs`.

```
(define (div x y)
  (cond
   ((positive? y)
    (let ((n (* (numerator x)
                (denominator y)))
          (d (* (denominator x)
                (numerator y))))
      (if (negative? n)
          (- (quotient (- d n 1) d))
          (quotient n d))))
   ((zero? y)
    0)
   ((negative? y)
    (let ((n (* -2
                (numerator x)
                (denominator y)))
          (d (* (denominator x)
                (- (numerator y)))))
      (if (< n d)
          (- (quotient (- d n) (* 2 d)))
          (quotient (+ n d -1) (* 2 d)))))))

(define (mod x y)
  (- x (* (div x y) y)))
```

Figure 3. Defining `mod` and `div` using R^{5}RS, assuming exact rational arithmetics.

Since the number formats used for consumption by humans and non-Scheme programs vary wildly and uncontrollably, they are properly the subject of one or probably several libraries and beyond the scope of this paper. We focus on literal syntax and on the syntax used by `read` and `write`.

In Scheme, to preserve some of the desirable properties of the language, the literal syntax must be compatible with the format used by `read` and `write`.

The simple-minded approach to the external-representation issue is to just have one uniform external representation for all machine number formats—each representation stands for a unique rational number, and converting a number to its representation is an exact operation. However, many floating-point numbers have quite long representations as fractions, making this choice prohibitive in terms of both space (for storage of the representation) and time (for converting back and forth between the numbers and their representation).

```
(define (floor-log-abs base x)
  (define (log b x e b^e offset)
    (let ((b^e+1 (* b^e b)))
      (if (> b^e+1 x)
          (if (= b^e x) e (+ e offset))
          (log b x (+ e 1) b^e+1 offset))))
  (let ((abs-x (abs x)))
    (if (>= abs-x 1)
        (log base abs-x 0 1 0)
        (- (log base (/ 1 abs-x) 0 1 1)))))

(define (round-fraction base mantissa round x)
  (if (zero? x)
      0
      (let ((k (- mantissa
                  (floor-log-abs base x)
                  1)))
        (* (round (* x (expt base k)))
           (expt base (- k))))))
```

Figure 4. `Floor-log-abs` and `round-fraction` as defined in Sections 6.3 and 6.4, implemented in R^{5}RS, assuming rational arithmetics.

Hence, it is desirable to be able to use a shorter, floating-point (in the true sense of "using a point") external representation for numbers, preferably using the familiar decimal-point format. In that case, read/write invariance requires tagging the result explicitly as a floating-point number. Moreover, to better support the exchange of external representations between different Scheme systems, or to support distinguishing between several machine floating-point formats used by a single Scheme system, it is desirable to provide information about the nature of the floating-point format used.

We suggest using a suffix indicating the length of the binary mantissa of the floating-point format. Thus, In our proposal, `0.7` would always denote 7/10 (unless R^{5}RS compatibility is important, see Section 5), whereas the IEEE 754 64-bit float closest to 0.7 would print as `0.7|52`, which is equal to `3152519739159347/4503599627370496`. We call this format the *mantissa-width tagged format*.

From the point of view of communication, the mantissa-width tagged format is not so much an indicator for "floating point" but rather a source coding (compression) method for a frequently used subset of the rational numbers—binary fractions. The mantissa-width tagged format for binary fractions achieves accuracy without loss of performance.

The mantissa-width tagged format can be specified accurately in terms of the procedures `round-fraction` and `round` of Sections 6.4 and 6.3. To be specific, we propose procedures `string->rational` and `rational->string` (serving the function of R^{5}RS's `string->number` and `number->string`) that convert between internal and external representations of rational numbers. Apart from the usual formats (base 2/8/10/16, fractions via /, and decimal scientific "e"-notation), `string->rational` understands the number syntax

scientific`|`*mantissa*

and interprets it as

`(round-fraction 2` *mantissa* `round` *scientific*`)`

`Rational->string` and `string->rational` satisfy the "read/write-invariance" property of R^{5}RS: For each rational number x (in the sense of `rational?`), the following holds:

`(= (string->rational (rational->string` x`))` x`)`

(Note that our `=` is an exact comparison, unlike the `=` of R^{5}RS, which is the reason R^{5}RS formulates this property in terms of `eqv?`.)

To summarize, we suggest the following (partly departing from R^{5}RS):

- Each external number representation without annotation denotes exactly the rational number the "learned in high school interpretation" would assign it. That is, `0.7` $= 7/10$ and `1.3e-2` $= 13/1000$.
- The mantissa-width tagged format specifies a *binary* fraction (like a floating point number) by *decimal* digits: `0.7|5` $= 11/16$ and `0.7|52` $= 3152519739159347 \cdot 2^{-52}$.
- The `#e` and `#i` prefixes go away.

Note that we expect the mantissa-width tagged format to occur only rarely in numerical literals—the programmer can simply specify a rational number and rely on the automatic conversion for `float` operations.

The R^{5}RS requirement that `number->string` must use the minimum number of digits for decimal-point external representations must be adjusted for `rational->string`, as there might be several different representations for the same number. For example, `11/32` $=$ `0.34|4` $=$ `0.34375`: Although the mantissa-width tagged format is shorter, the purely decimal format is arguably clearer.

Consequently, we propose to require the minimum number of digits only within one particular number format, but give the implementations the freedom to choose the format. Nevertheless, printing with the absolute minimum of characters is also possible and even computationally inexpensive.

7 Implementation Issues

In this section, we address the most important implementation issues that arise with our proposal:

7.1 Exact operations on flonums

In design alternatives #1 and #2, numbers represented as flonums will be converted into fractions when an exact operation requires it. This might lead to surprises in terms of time and memory consumed, because exact representations can and generally do grow quickly with arithmetic depth. This is the price of exactness.

However, if problems arise from exact operations on flonums, they are easy to detect (slow execution) and have a specific remedy: Replace exact operations by inexact operations and investigate numerical stability. R^{5}RS, on the other hand, makes it much harder to identify and systematically fix this kind of problems because exactness is not a static property of the program. In other words, the programmer must investigate the run-time propagation of inexactness in order to understand the algorithm actually being executed.

7.2 Generic arithmetic

The exact arithmetic operations need to dispatch on the representations of their arguments—a typical implementation will at least use separate representations for fixnums, bignums, and true fractions. This is no different from the situation in R^{5}RS, and a Scheme system can employ the same technique as before to perform the dispatch—for example, via exhaustive case analysis or a suitable exception system.

7.3 Coercion of constants

If number literals containing a decimal point (and without a mantissa-width specification) are interpreted as rationals, and floating-point operations accept rational arguments (as in design alternative #1), the implementation will typically need to convert the rational number to a floating-point representation. This may be a relatively expensive operation, and a straightforward program may perform it often. To reduce the cost, an implementation could memoize the floating-point approximation of a rational number, or perform a static analysis to determine what literals are used exclusively as arguments to floating-point operations. We conjecture that a simple analysis would be quite effective for most realistic programs.

7.4 Fixnum arithmetic

Many Scheme implementations already use fixnum arithmetic to optimize common-case numerical operations. However, implementations might want to offer exclusively fixnum arithmetic to optimize away the generic-arithmetic dispatch and the overflow detection. Doing this in the default set of numerical operations on exact numbers is already in violation of R^{5}RS. (See Section 2.)

Thus, the best way of offering fixnum-only operations would be through a set of separate procedures, analogous to the floating-point operations, with their algebraic meaning defined as calculating "mod $\pm 2^w$", $w \in \{8, 16, 32, 64\}$, as proposed in Section 6.5.

7.5 Floating-point representation

We have said nothing about the particular machine floating-point representation a Scheme system may use or should be required to use by a standard. This is a touchy issue—requiring, say, a particular IEEE 754 representation would lead to completely reproducible computations, but, depending on the hardware a program runs on, results in an unacceptable loss in either accuracy or efficiency [4, 12] and might pose a considerable obstacle for implementations on platforms not supporting this representation natively.

For this reason, we would expect a standard to specify that the floating-point operations use the widest floating-point format the underlying hardware supports efficiently. In practice, this would probably mean IEEE 754 double extended on the Intel x87 or the 68xxx architecture, and IEEE 754 double on, say, the PowerPC, or the Alpha.

Of course, implementations could also offer sets of floating-point operations specific to a specific machine representation or with parameters (e.g. multiprecision.) However, as few programs seem to require this degree of control, it should probably not be included into the core language by default.

7.6 Floating-point storage

The choice of the storage format for large quantities of floating-point numbers is independent of the choice of the format used for computations. Uniform vectors that explicitly specify the floating-point format used, such as those proposed in SRFI 4 [8] are an appropriate mechanism for this.

7.7 Mantissa-width tagged format

Reading the mantissa-width tagged format proposed in Section 6.6 can be done efficiently using Clinger's method [3, 2].

Similarly, printing the mantissa-width tagged format using the minimum number of total digits can be reduced to Burger and Dybvig's efficient method for printing a binary fraction as an approximate decimal fraction [19, 1]. The most important difference is that the mantissa width may vary with the number being printed. In effect, the mantissa-width tagged format can often be shorter, as for example in `1e9|1` $= 2^{30}$. Whether the system should really use the mantissa-width tagged format in this case is a different matter.

8 Related Work

Some Scheme implementations targeted at high performance programs—such as Chez Scheme [6], and Bigloo [17]—offer specialized numerical operations for floating-point numbers. This underlines the need for separating floating-point arithmetic from the usual generic arithmetic for performance reason, but does not really address the concerns raised in this paper: The remaining numerical operations are unaffected in these systems. Gambit-C [7] offers a declaration which locally declares all R^{5}RS numerical operations to perform floating-point arithmetic—again, for performance reasons.

The teaching languages of DrScheme [5] use exact arithmetic by default, to spare beginning students the confusion of programming with mixed exact and inexact floating-point arithmetic.

Objective Caml [14] keeps the domains and types for floating-point numbers completely separate from that of integers: A program cannot use them interchangeably, it must explicit convert. The floating-point operations have names different from the integer operations. (`+.` for floating point addition, etc.) Keeping the floating-point numbers separate from the rest is easier in Objective Caml than it is in Scheme because Caml does not have built-in rational numbers. Hence, there is no choice but the read `0.7` as a float.

Haskell 98 [10] also has a sophisticated type hierarchy for its numerical types, including rational numbers and single- and double-precision floating-point numbers. It keeps the various numerical types separate, but uses its type class mechanism to use a single set of operators for all numerical types and make parts of the numerical domains look like subtype hierarchies. Just like our proposal, Haskell mandates that a literal containing a decimal dots represents its corresponding rational number accurately. Two methods `fromInteger` and `fromRational`, overloaded over their respective result types, negotiate between literals and the contexts that receive them. Ambiguities concerning the numerical types are frequent, which is why the `default` declaration can specify a strategy for resolving them.

Common Lisp [18] does not have inexactness as a property of numbers orthogonal to the representation type. However, numerical operations will always convert rational arguments to float arguments

if any other arguments are floats. Comparisons between floats and rationals always convert the floats to rationals. Unlike Scheme, Common Lisp does at least give a recommendation for the minimum precision offered by the various floating-point operations, which, we conjecture, reduces the variance between different Common Lisp systems considerably. However, the basic arithmetic operations are still overloaded and do not always respect the various algebraic laws.

Mathematica [20] provides an arbitrary-precision floating-point representation and applies a mechanism of decreasing precision during inexact computation. In practice, however, this approach suffers from the same weaknesses as R^{5}RS: When inexact numbers enter the computation, it is usually time to design a new program. Moreover, the automatic decreasing of precision makes it difficult to run entire computations at a higher precision; a stray `1.0` (default precision) instead of a `N[1, 50]` (high precision) propagates its low precision uncontrollably, usually ruining the calculation.

An alternative approach to preserve read/write invariance (and a number of the other issues raised in this paper) would be to *fix* the floating-point representation in the language specification once and for all, as for example has been done in Java [9]. In that case, no tagging is necessary. The controversy around this approach suggests against it [12].

Scheme has long been one of the few languages to specify that a round-trip of conversion of a number to an external representation and back should preserve that number. Hence, it comes as little surprise that the most important publications about efficient and accurate algorithms to achieve this purpose come from the Scheme community [3, 2, 19, 1].

9 Conclusion

In Section 1.1, R^{5}RS says:

> Scheme's model of arithmetic is designed to remain as independent as possible of the particular ways in which numbers are represented within a computer. [...] Thus the distinction between integer and real arithmetic, so important to many programming languages, does not appear in Scheme.

We have shown that the behavior of realistic programs is in fact very much dependent on the particular number representations chosen by an implementation. The distinction between integer and real arithmetic is important to many other languages because it is important to programs. Following this design guideline, R^{5}RS makes is very difficult to write portable programs employing inexact arithmetic: Inexact arithmetic is too underspecified to allow a programmer to predict what a particular program will do running in different Scheme implementations. At the heart of the problem is the notion of inexact numbers itself—a more useful basis for the design of a set of numerical operations is attributing inexactness to the operations rather than the numbers.

We have designed the basis for such a set of numerical operations, and identified design alternatives within its framework. The most important property of our design is that the default numerical operations are always exact. Floating-point arithmetic is relegated to a separate set of operations. Most of the choices available within the design concern the degree of separation between the inexact and exact worlds. However, all of the alternatives we propose have more pleasant properties than what R^{5}RS currently requires—in particular, greater transparency and full reproducibility for exact computations. They also require similar, if not less implementation effort. We have also identified some weaknesses in the set of numerical operations offered by R^{5}RS, and proposed alternatives.

Arguably, the result is still "strange" in that it is unlike basically every other programming language. We conjecture that this difference is good and necessary: In particular, most programming languages do not offer infinite-precision integers and rational numbers at all, which reduces the design space, but comes with its own problems: Limited precision of the various numerical types along with implicit coercion rules often cause programming errors and non-reproducible behavior. Of the languages that do support infinite-precision integers and rationals, only Common Lisp stands out, which takes a less principled but otherwise similar approach to Scheme. We conjecture that programmers experience similar surprises in Common Lisp as in Scheme. However, given Common Lisp's much tighter specification and as much fewer Common Lisp systems exist than Scheme systems, these surprises may not matter as much in practice. All in all, we believe that Scheme is special enough to warrant a special design for its numerical operations.

10 References

[1] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 108–116, Philadelphia, PA, USA, May 1996. ACM Press.

[2] William D. Clinger. How to read floating point numbers accurately. In PLDI 1990 [16], pages 92–101.

[3] William D. Clinger. How to read floating point numbers accurately. In Kathryn S. McKinley, editor, *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection*. ACM, April 2004. SIGPLAN Notices 39(4).

[4] Joseph Darcy. Adding IEEE 754 floating point support to Java. Master's thesis, University of California at Berkeley, 1998. `http://www.cs.berkeley.edu/~darcy/Borneo/spec.html`.

[5] *PLT DrScheme: Programming Environment Manual*, May 2004. Version 207.

[6] R. Kent Dybvig. *Chez Scheme User's Guide*. Cadence Research Systems, 1998. `http://www.scheme.com/csug/index.html`.

[7] Marc Feeley. *Gambit-C, version 3.0, A portable implementation of Scheme*, 3.0 edition, May 1998. `http://www.iro.umontreal.ca/~gambit/doc/gambit-c.html`.

[8] Marc Feeley. SRFI 4: Homogeneous numeric vector datatypes. `http://srfi.schemers.org/srfi-14`, May 1999.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[10] Haskell 98, a non-strict, purely functional language. `http://www.haskell.org/definition`, December 1998.

[11] William Kahan. Branch cuts for complex elementary functions, or much ado about nothing's sign bit. In A. Iserles and M.J.D. Powell, editors, *The State of the Art in Numerical Analysis*. Clarendon Press, 1987.

[12] William W. Kahan and Joseph D. Darcy. How Java's floating-point hurts everyone everywhere. `http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf`, March 1998.

[13] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[14] Xavier Leroy. *The Objective Caml system release 3.08 , Documentation and user's manual*. INRIA, France, July 2004. `http://pauillac.inria.fr/caml`.

[15] International Standards Organization. Programming language — C, 1999. ISO/IEC 9899.

[16] *Proc. Conference on Programming Language Design and Implementation '90*, White Plains, New York, USA, June 1990. ACM.

[17] Manuel Serrano. *Bigloo—A "practical Scheme compiler"—User manual for version 2.6d*, April 2004. `http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html/`.

[18] Guy Steele. *Common LISP: The Language*. Digital Press, Bedford, MA, 2nd edition, 1990.

[19] Guy L. Steele and Jon L. White. How to print floating-point numbers accurately. In PLDI 1990 [16], pages 112–126.

[20] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, 5th edition, 2003.

A Mantissa-Width Tagged Format

In this appendix we show how to print the mantissa-width tagged format efficiently. Given a binary fraction x (i.e. $x = a2^b$, integer a,b), we are to construct a decimal fraction y (i.e., $y = c10^d$, integer c,d) and a positive integer w such that "$y\,|\,w$" reads as x according to the definition of the format given in Section 6.6.

Obviously, it is always possible to find *some* "$y\,|\,w$" for x because binary fractions are also decimal fractions. (Note $a/2^b = (a5^b)/10^b$.) This appendix shows how to obtain the *shortest* such notation for a given x.

Let us begin by giving a concise formal definition of the mantissa-width tagged notation. For non-zero rational y and positive integer w, define "$y|w$" to represent the rational number

$$\sigma_w(y) = \rho\left(y2^{-\lfloor\log_2|y|\rfloor+w-1}\right)\cdot 2^{\lfloor\log_2|y|\rfloor-w+1},$$

where $\rho(-)$ denotes rounding to the nearest integer, with ties broken towards even. In other words, $\rho(z)$ is either $\lfloor z\rfloor$ or $\lceil z\rceil$, depending on which one is closer, and $\rho(n+1/2) = n$ for integer n, if and only if n is even. For $y = 0$ we define $\sigma_w(y) = 0$.

Since $\rho(-)$ is symmetric, i.e., $\rho(-z) = -\rho(z)$, so is $\sigma_w(-)$. For this reason we only consider positive x,y for the remainder of this section.

The following lemma characterizes $\sigma_w(-)$.

Lemma 1 The function $\sigma_w(-)$ is non-decreasing and piecewise constant, with a fixed point in every piece.

More explicitly, for integer k and $w \geq 1$ define

$$y_{w,k} = \left(1 + \frac{k \bmod 2^{w-1}}{2^{w-1}}\right) 2^{k \operatorname{div} 2^{w-1}},$$

and

$$u_{w,k} = \left(1 + \frac{(k \bmod 2^{w-1}) + 1/2}{2^{w-1}}\right) 2^{k \operatorname{div} 2^{w-1}}.$$

These values are ordered linearly as

$$\cdots < y_{w,k-1} < u_{w,k-1} < y_{w,k} < u_{w,k} < \cdots.$$

Then $\sigma_w(y) = y_{w,k}$ for all $y \in (u_{w,k-1}, u_{w,k})$ and

$$\sigma_w(u_{w,k}) = \begin{cases} y_{w,k} & \text{if } k \text{ is even and } w \geq 2, \\ y_{w,k+1} & \text{otherwise.} \end{cases}$$

A.0.0.1 Proof. The function $\sigma_w(-)$ is piecewise constant by construction. Moreover, jumps in $\sigma_w(y)$ can only occur at places where $y2^{w-\lfloor\log_2 y\rfloor-1}$ is of the form $n+1/2$ for some integer n, or if y is a power of two. (The first condition comes from $\rho(-)$, the second from $\lfloor\log_2 -\rfloor$.) This is where we need to check that the function is non-decreasing.

Let us inspect the behavior of $\sigma_w(y)$ at these potential transition points. First consider $y = 2^e$ for integer e and a positive real parameter ε, chosen so small that no other potential transition point lies in the open interval $(y-\varepsilon, y+\varepsilon)$. Such an ε exists because the only accumulation point of the potential transitions is zero, which is excluded. Then, $\lfloor\log_2(y-\varepsilon)\rfloor = e-1$ and $\lfloor\log_2 y\rfloor = \lfloor\log_2(y+\varepsilon)\rfloor = e$. This implies

$$\begin{aligned} \sigma_w(y-\varepsilon) &= \rho\left(2^w - \varepsilon 2^{w-e}\right)2^{-(w-e)} = y, \\ \sigma_w(y) &= \rho\left(2^{w-1}\right)2^{-(w-e-1)} = y, \\ \sigma_w(y+\varepsilon) &= \rho\left(2^{w-1} + \varepsilon 2^{w-e-1}\right)2^{-(w-e-1)} = y. \end{aligned}$$

Hence, $\sigma_w(-)$ is in fact constant across the transitions at powers of two. Moreover, $\sigma_w(2^e) = 2^e$ for all integer e. Note that $y_{w,k} = 2^{k/2^{w-1}}$ if k is divisible by 2^{w-1}.

Now consider the other type of potential transition point, i.e., $y2^{w-\lfloor\log_2 y\rfloor-1} = n+1/2$ for some integer n. Again, let $\varepsilon > 0$ such that there is no other potential transition point in $(y-\varepsilon, y+\varepsilon)$. Then $\lfloor\log_2(y+s\varepsilon)\rfloor = \lfloor\log_2 y\rfloor = e$ for all $s \in \{-1,0,1\}$ and

$$\sigma_w(y+s\varepsilon) = \rho\left(n + \frac{1}{2} + s\varepsilon 2^{w-e-1}\right)2^{-w+e+1}.$$

The term $r = \rho\left(n+\frac{1}{2}+s\varepsilon 2^{w-e-1}\right)$ is either n or $n+1$ depending on s. If $s=-1$ then $r=n$, if $s=1$ then $r=n+1$, and if $s=0$ then it depends on the tie-breaking rule of ρ. The "round to even" results in $r=n$ if and only if n is even. Whatever the tie-breaking rule,

$$\sigma_w(y-\varepsilon) \leq \sigma_w(y) \leq \sigma_w(y+\varepsilon).$$

Hence, $\sigma_w(-)$ is non-decreasing at each transition point, which means that the entire function is non-decreasing.

Finally we show how to obtain the expressions for $y_{w,k}$ and $u_{w,k}$. Since every positive y can uniquely be decomposed into $y = \hat{y}2^e$ for integer e and $1 \leq \hat{y} < 2$, the transitions are characterized by

$$\lfloor\log_2 y\rfloor = e, \quad \hat{y} = y2^{-\lfloor\log_2 y\rfloor} = \left(n + \frac{1}{2}\right)2^{-w+1}.$$

for integer n. The condition $1 \leq \hat{y} < 2$ restricts the range of n to $\{2^{w-1}, \ldots, 2^w - 1\}$. Therefore, we define the parameter $k = e2^{w-1} + (n - 2^{w-1})$ indexing the transition points $u_{w,k}$ as defined

in the lemma. It is easy to show that $u_{w,k} < u_{w,k+1}$ for all k. In a similar fashion, it can be shown that $y_{w,k}$ is a fixed point of $\sigma_w(-)$ and that $u_{w,k-1} < y_{w,k} < u_{w,k}$ for all k. Finally, a careful analysis of the effect of tie-breaking yields the value $\sigma_w(-)$ at the transition points, which concludes the proof. □

Let $x = a2^b$ be a positive binary fraction with integer a, b and odd a. Then a and b are uniquely defined by x.

According to Lemma 1, there is (y,w) such that $\sigma_w(y) = x$ if and only if $x = y_{w,k}$ for some integer k. Moreover, this k relates to x and w by

$$\begin{aligned} k \operatorname{div} 2^{w-1} &= \lfloor \log_2 x \rfloor = b + \lfloor \log_2 a \rfloor, \\ k \bmod 2^{w-1} &= \left(x 2^{-\lfloor \log_2 x \rfloor} - 1\right) 2^{w-1} \\ &= \left(a 2^{-\lfloor \log_2 a \rfloor} - 1\right) 2^{w-1} \\ &= a 2^{-\lfloor \log_2 a \rfloor + w - 1} - 2^{w-1}. \end{aligned}$$

Since a is odd, the last right-hand side is an integer if and only if $-\lfloor \log_2 a \rfloor + w - 1 \geq 0$. Hence, there is a y such that $\sigma_w(y) = x$ if and only if

$$w \geq w_{\min}(x) = \lfloor \log_2 a \rfloor + 1.$$

The following theorem refines this result by characterizing *all* y for which $\sigma_w(y) = x$.

Lemma 2 Let $x = a2^b > 0$ for integer a, b and a odd, and let w be positive integer. Then the set of all y such that $\sigma_w(y) = x$ is given as

$$\sigma_w^{-1}(x) = \begin{cases} \emptyset & \text{if } w < w_{\min}(x), \\ [u_{w,k-1}, u_{w,k}) & \text{if } w = w_{\min}(x) = 1, \\ [u_{w,k-1}, u_{w,k}] & \text{if } w \geq w_{\min}(x) \geq 2, k \text{ even}, \\ (u_{w,k-1}, u_{w,k}) & \text{if } w \geq w_{\min}(x) \geq 2, k \text{ odd}, \end{cases}$$

where k is defined as

$$k = \left(b + \lfloor \log_2 a \rfloor + a 2^{-\lfloor \log_2 a \rfloor} - 1\right) 2^{w-1}.$$

Moreover,

$$\sigma_{w_{\min}}^{-1}(x) \supseteq \sigma_{w_{\min}+1}^{-1}(x) \supseteq \sigma_{w_{\min}+2}^{-1}(x) \supseteq \cdots.$$

A.0.0.2 Proof. The stated form of $\sigma_w^{-1}(x)$ is a consequence of Lemma 1, with careful analysis of the boundaries.

It remains to be shown that the $\sigma_w^{-1}(x)$ form a tightening chain. By induction over k, it is sufficient to show

$$u_{w,k-1} < u_{w+1,2k-1} < y_{w,k} = y_{w+1,2k} < u_{w+1,2k} < u_{w,k}.$$

Note that $2k$ corresponds to $w+1$. By Lemma 1,

$$\begin{aligned} u_{w,k-1} < y_{w,k} < u_{w,k}, \\ u_{w+1,2k-1} < y_{w+1,2k} < u_{w+1,2k}, \\ x = y_{w,k} = y_{w+1,2k}. \end{aligned}$$

Decompose $k = k_1 2^{w-1} + k_0$ for $0 \leq k_0 < 2^{w-1}$. Then $2k = k_1 2^w + 2k_0$, where $0 \leq 2k_0 < 2^w$. Hence,

$$\begin{aligned} (u_{w,k} - u_{w+1,2k}) \cdot 2^{-k_1} &= \frac{k_0 + 1/2}{2^{w-1}} - \frac{2k_0 + 1/2}{2^w} \\ &= 2^{-w-1} > 0. \end{aligned}$$

The calculation has shown $u_{w+1,2k} < u_{w,k}$. For the remaining inequality decompose $k - 1 = k_3 2^{w-1} + k_2$ for $0 \leq k_2 < 2^{w-1}$. Then $2k - 1 = k_3 2^w + 2k_2 + 1$, where $0 < 2k_2 + 1 < 2^w$. A similar calculation as before shows $u_{w,k-1} < u_{w+1,2k-1}$, and this concludes the proof. □

For reference, we also investigate the specialization of the previous lemma in which $w = w_{\min}(x)$.

Lemma 3 For $x = a2^b > 0$, integer a, b, and a odd,

$$\sigma_{w_{\min}(x)}^{-1}(x) = \begin{cases} [x - 2^{b-2}, x + 2^{b-1}) & \text{if } a = 1, \\ (x - 2^{b-1}, x + 2^{b-1}) & \text{otherwise.} \end{cases}$$

A.0.0.3 Proof. Consider the case $a = 1$. Then $w = 1$, $k = b$, $u_{w,k} = a2^b + 2^{b-1}$, and $u_{w,k-1} = a2^b - 2^{b-2}$ (sic!).

Now consider the case $a > 1$. By definition of $w = w_{\min}(x)$ it is $2^{w-1} \leq a < 2^w$. Since a is odd and $w \geq 2$ it is also $a \neq 2^{w-1}$. Hence, $1 \leq a - 2^{w-1} < 2^w$. This implies that k and $k-1$ decompose modulo 2^{w-1} into

$$\begin{aligned} k \bmod 2^{w-1} &= a - 2^{w-1}, \\ (k-1) \bmod 2^{w-1} &= a - 2^{w-1} - 1, \\ k \operatorname{div} 2^{w-1} = (k-1) \operatorname{div} 2^{w-1} &= b + w - 1. \end{aligned}$$

Using these equations in $u_{w,k-1}$ and $u_{w,k}$ (Lemma 1) and $\sigma_w^{-1}(x)$ (Lemma 2), and observing that k is odd, shows the stated forms of the bounds.□

Finally, we will apply the results to printing x as "$y \,|\, w$".

As Lemma 2 states, $\sigma_w^{-1}(x) = \emptyset$ if $w < w_{\min}(x)$. This means, the smallest w for which we can hope to find a suitable y at all is $w_{\min}(x)$. But Lemma 2 also states an inclusion chain, showing that choosing w *larger* than $w_{\min}(x)$ can only decrease our choice for y. Moreover, choosing w larger will eventually increase the length of the printed representation of w. Hence, the optimal choice is $w = w_{\min}(x)$, as defined before Lemma 2.

Once we have choosen $w = w_{\min}(x)$, Lemma 3 shows the interval in which we need to look for a suitable y. And this is exactly what we need to apply the Burger/Dybvig algorithm [1, Section 2.2] to the printing problem: Their algorithm computes increasingly close decimal approximations of x until one of them is contained in a given interval.

To be concrete, the notation used in the formulation of the Burger/Dybvig algorithm translates to our notation as follows: $v = x$ (normalized), $f = a$, $b = 2$, $e = b$, $p = w$, $B = 10$, $V = y = c10^d$, $(d_1 d_2 \cdots d_n)_{10} = c$, $k - n = d$. Applied to the mantissa-width tagged format, we need to replace the bounds for termination as follows: $low = x - 2^{b-1-[w=1]}$, $high = x + 2^{b-1}$, where $[w = 1] = 1$ if $w = 1$ and 0 otherwise. Finally, we need to modify the termination criterium (1): $> low$ must be replaced by $\geq low$ in case $w = 1$. These modifications also apply to Burger and Dybvig's improved algorithms (without rational arithmetics and with efficient scaling).