Boundary Number Systems -- James

William Bricken

January 2001

James Numbers

James calculus uses three types of containers/boundaries to represent all types of numbers. Several unique numerical concepts arise from this approach. *Generalized cardinality* applies to negative and fractional counts, as well as to integer counts. The *generalized inverse* unifies subtraction, division, roots, and logarithms into a single concept and operation. The *James imaginary*, **J**, removes all inverses by embedding them in an imaginary operation. J can be used for numerical computation as an alternative to using inverse operations.

The non-imaginary part of this presentation closely follows Jeff James' 1993 masters thesis under Dr. William Bricken at the University of Washington (thus the name J).

Boundary Units

Three containers define the types of numerical objects. Configurations of these containers define numerical operations. Similar to Kauffman numbers, rules for James forms apply independently to each space, regardless of nesting. As well, all forms have a direct interpretation in standard notations, even during transformation steps. This makes James numbers easy to understand. However the routes that they take to achieve computation are generally very unusual.

James Form	Interpretation
( )	e^0 = 1
[ ]	ln 0 = negative infinity
< >	negative 0 = 0

Each elementary unit container is empty, forming the ground, or constant, forms. Each elementary container can be interpreted as a ground object, and as the operation of containing nothing. In that sense, the void serves as the fundamental ground of all objects and operations.

The round container, ( ), raises e to the power of its contents. When it is empty, the contents are zero, and the value of the boundary is e^0, which can also be interpreted as the object one.

The square container, [ ], takes the logarithm of its contents, and is the inverse of the round boundary.

The angle container, <>, converts its contents to additive inverse; it multiplies by -1.

Boundary Operators

Each container operates on its contents with the following semantics:

James Form	Interpretation
(A)	e^A
[A]	ln A
<A>	- A (generalized)

The exponent and logarithm transforms can be in an arbitrary base. Let the base be represented by #. Then the following remains true:

( )	#^0 = 1
[ ]	log# 0 = negative infinity
(A)	#^A
[A]	log# A

The base of natural logarithms, e, is most convenient as a specific choice, since many irrationals are defined in terms of e.

Integers

James integers are expressed in stroke notation. There is no provision for a power-oriented notation for integers, however the calculus itself uses power transformations extensively.

0	void
1	( )
2	( )( )
3	( )( )( )
...	

Varieties of numbers occur through configurations of the three containers, with empty containers forming a computational ground. The calculus emphasizes algebraic forms, and is clumsy for arithmetic evaluation.

Since stroke representation is rather clumsy, we will use decimal numbers to abbreviate stroke numbers throughout this section.

Algebraic Operations

Addition is sharing space. All forms inside the same container, that is, all forms sharing a space, are joined by implicit addition. Multiplication and power are specific configurations of ( ) and [ ] containers, both of which keep track of the appropriate exponential or logarithmic space. Multiplication is adding logarithms then converting back the non-logarithmic space. Power is adding the loglog form of the base to the log of the exponent.

Addition	A+B	A B
Multiplication	A*B	([A] [B])
Power	A^B	(([[A]][B]))

The round and square boundaries can be read as exponents and natural logs, providing James forms with a direct interpretation:

Operation	*James Form*	*Interpretation*
A+B	A B	A + B
A*B	([A][B])	e^(ln A + ln B) = e^ln A * e^ln B A * B
A^B	(([[A]][B]))	e^(e^(lnln A + ln B)) = e^(e^lnln A * e^ln B) e^((ln A) * B) e^(ln A^B) A ^ B

It is fair to say that round and square boundaries are simply a convenient way write complex exponents, since they introduce no new transformation rules. Similar to Spencer-Brown numbers, James notation could use a single container by indexing the depth of containments: even is exponent ( ), odd is logarithm [ ]. Similar to Kauffman numbers, a fourth boundary type could be used for a depth-oriented positional notation.

Inverse Operations

Subtraction is sharing a space with an additive inverse form, <B>. Division is sharing deeper space with a multiplicative inverse form, <[B]>. Taking a root is sharing an even deeper space with the multiplicative inverse form.

Subtraction	A–B	A < B >
Division	A/B	([A] <[B]>)
Root	A^(1/B)	(([[A]]<[B]>))

The angle container, <>, serves as the inversion concept for all inverse operations. The operations are distinguished by which forms are contained in angle boundaries, and by the depth of nesting of exp-log transforms.

Reduction Rules (Axiomatic basis)

Computation is achieved through application of three reduction rules:

`([A]) = [(A)] = A`	*Involution*
`(A [B]) (A [C]) = (A [B C])`	*Distribution*
`A <A> = void`	*Inversion*

The distribution rule in standard notation would read:

```
e^(A+ln B) + e^(A+ln C) = e^(A+ln(B+C))
```

Proof:

```
e^(A+ln B) = (e^A)*(e^ln B) = B*(e^A)
e^(A+ln C) = (e^A)*(e^ln C) = C*(e^A)

B*(e^A) + C*(e^A) = (e^A)(B+C)
                  = (e^A)(e^ln(B+C))
                  = e^(A + ln(B+C))
```

Alternatively, we could convert the distributive rule into a multiplicative rather than an additive form:

`( A [B]) ( A [C]) = ( A [B C])`	additive
`([A][B]) ([A][C]) = ([A][B C])`	multiplicative

which reads more conventionally as:

```
(A*B)+(A*C) = A*(B+C)
```

and more unconventionally as exponents and logs:

```
e^(ln A + ln B) + e^(ln A + ln C) = e^(ln A + ln(B+C))
```

Note that the multiplicative representation uses `[A]` rather than `A`. This is not a significant difference, since any form can be bounded by `[ ]` due to involution:

```
A = [(A)]
```

Algebraic Proof

James calculus is an algebraic, equational system. The primary transformations are substitution and replacement of equals for equals. Proof consists of a series of transformations from one form into another.

The standard substitution strategies are all available in the boundary calculus. Given an equation A=?=B, the two forms can be demonstrated to be equal by:

Convert one form into the other form.

Convert both forms into the same third form

Standardize the equation to a void-equivalent and reduce to void.

To standardize to a void-equivalent, we place all terms on one side of the equation, leaving the other side void. Unlike conventional algebra, there is only one operation, Inversion, to move all terms to one side of an equation:

```
A     = B

A <B> = B <B> = void
```

The Form of Numbers

All conventional numbers are represented as nested configurations of containers. These configurations specify both the pattern of a particular type of number, and the sequence of exp-log transformations necessary to compute that number.

Type	*Standard form*	*James form*
zero	0	void
one	1	()
natural	n	()()..n = ([n][()])
negative integer	-n	<()()..n> = <([n][()])>
rational	m/n	([m]<[n]>)
irrational	a^-b	(([[a]]<[b]>))
transcendental	e	(())
	PI	([[<()>]] ([[<()>]] <[2]>))
complex	i	(([[<()>]] <[2]>))
	a + bi	a ([b] ([[<()>]]<[2]>))
infinity	inf	<[]>

The Form of Numerical Computation

In the container representation, the relationships between numerical operations become overt. Essentially, any operation is applying the pair (...[...]...) to a particular part of the existing form.

Addition begins with no boundaries. Like stroke arithmetic, addition (and its inverse subtraction) is putting forms in the same space. Any space can be considered to be contained by a ([...]) pair.

Multiplication (and its inverse division) involves converting to natural logs with [...] and then back to powers of e with (...).

Power (and its inverse root) is another application of the (...[...]...) form, this time asymmetrically.

```
addition              A    B
multiplication      ( [A]  [B] )
power              (([[A]] [B] ))

subtraction           A  < B >
division            ( [A] <[B]>)
root               (([[A]]<[B]>))
```

The following forms are spread out to illustrate how each operator is a (...[...]...) elaboration of the previous form. The representation of each operation is the accumulation of new forms and forms above.

```
addition              A    B
multiplication      ( [ ]  [ ] )
power              ( [    ]      )

subtraction           A  < B >
division            ( [ ]  [ ] )
root               ( [    ]      )
```

The placement of containers reflects the properties of each operator. Both forms are free of containment for commutative addition. Both forms are enclosed for commutative multiplication. One form is enclosed for power, it is not commutative. Inversion is generic, the second form is simply inverted in all cases, creating the non-commutative inverse operations.

Note also that

A+B+C	A B C
A*B*C	([A][B][C])
(A*B)/(C*D)	([A][B]<[C][D]>)

Logarithms

The exponent function `exp`, is the inverse of the logarithm function, `log`.

log base e	`ln n`	`[n]`
exp base e	`e^n`	`(n)`
log base b	`logb n`	`([[n]]<[[b]]>)`
exp base b	`b^n`	`(( [n]  [[b]] ))`

Setting the logarithmic base to `e` results in the appropriate reduction:

```
loge n =  ([[n]] <[[(())]]>)                       substitute
          ([[n]] <[   ()  ]>)                      involution
          ([[n]] <        >)                       involution
          ([[n]]           )                       invert zero
            [n]                                    involution
```

Similarly, setting the exponential base to `e` results in the appropriate reduction also.

```
e^n    = (( [n]   [[(())]] ))                      substitute
         (( [n]            ))                      involution
         (   n              )                      involution
```

Log and exp base `b` are inverses:

```
expb[logb n] = n

      (([ ([[n]]<[[b]]>) ][[b]]))                  substitute
      ((   [[n]]<[[b]]>   [[b]]))                  involution
      ((   [[n]]               ))                  inversion
            n                                      involution
```

Using the spread out form, we can see the relationship between logs and other operations. Taking a log violates the `(...[...]...)` involution form, moving instead into a logarithmic space.

```
subtraction             A  <  B  >
division             ( [ ]    [ ]  )
log base B            [   ] [   ]

addition                A     B
multiplication       ( [ ]    [ ]  )
exp base B          (         [   ]   )
```

Finally, in boundary notation, the standard transforms for logarithms translate into an application of Involution.

Conventional notation	*Boundary form*
ln(A*B) = ln A + ln B	[([A][B])] = [A][B]
ln(A/B) = ln A - ln B	[([A]<[B]>)] = [A]<[B]>
ln(A^B) = B ln A	[(([[A]][B]))] = ([[A]][B])
ln(n+1) = ln n + ln (n+1)/n	[n 1] = [n][([n 1]<[n]>)]
log10 A = (ln A)/(ln 10)	([[A]]<[[10]]>) = ([[A]]<[[10]>)

Note that the conversion between bases is explicit in the representation; the form of a logarithm to base `N` specifies the transformations to convert between that base and the natural base.

Generalized Inverse

The *generalized inverse* treats subtraction, division, roots, and logs as the same operation in different contexts. Below, the spacing between characters is used to emphasize the communality of forms.

Subtraction

```
-1                       < ( ) >
-B                       <  B  >
A-B                   A  <  B  >
A+(-B)                A  <  B  >
```

Division

```
1/1                (     <[( )]>)
1/2                (     <[ 2 ]>)
1/B                (     <[ B ]>)
A/B                ( [A] <[ B ]>)
```

Root

```
A^(1/2)           (([[A]]<[ 2 ]>))
A^(1/B)           (([[A]]<[ B ]>))
A^-B              (([[A]] [<B>] ))
```

Log

```
ln A                 [A]
logB A             ([[A]]<[[B]]>)
expB A            (( [A]  [[B]] ))
```

Dominion

An empty square container, `[]`, represents the logarithm of `0`, which is negative infinity. The square basis provides a natural representation of infinity which can be used in the course of computation. The behavior of infinity is specified by the following theorems.

Name	Form	Interpretation
Dominion	`A [ ]   = [ ]`	`-inf + A = -inf`

Negative infinity absorbs all forms sharing its space. A variant of dominion converts the negative infinity to a void:

```
(A [ ] ) = void         e^(A + -inf) = 0
```

Positive infinity is the inversion of negative infinity:

```
<[]> = inf
```

Positive infinity also absorbs all forms with its space, except for two (negative infinity and the imaginary `J`). The reasons for this are discussed in the later section on infinities.

Positive Dominion	`A <[]> = <[]>`	`A + inf = inf`

where `A =/= []` and `A =/= [<()>]`.

Proof:

```
A [ ] = [ ]

(A [ ]) (A [ ]) = (A [ ])                    distribution, B=C=0

Let X = (A [ ])

      X X = X

      X = void      is the only solution

       (A [ ])  = void
      [(A [ ])] = [ ]                        In both sides
        A [ ]   = [ ]                        involution


A <[]> = <[]>

  A  <[]>
<<A>><[]>                                    inverse cancel
<<A>  []>                                    inverse collect
<     []>                                    dominion
```

Inverse Theorems

These theorems permit transformation of the inversion container, <>.

Name	Form	Interpretation
Inverse Collection	<A><B> = <A B>	(-A)+(-B) = -(A+B)
Inverse Cancellation	<<A>> = A	--A = A
Inverse Promotion	(A [<B>]) = <(A [B])> (A <[<B>]>) = <(A <[B]>)>	-B(e^A) = -(Be^A) (e^A)/-B = -(e^A/B)

Proof of theorems:

```
<A><B>
<A><B><A B> A B
      <A B>


<<A>>
<<A>><A> A                                            inversion
         A                                            inversion


(A [<B>])
(A [<B>]) <(A [B])> (A [B])                           inversion
(A [<B> B]) <(A [B])>                                 distribution
(A [     ]) <(A [B])>                                 inversion
            <(A [B])>                                 dominion
```

Examples

Here are some examples of proof of other (unnamed) theorems:

```
-ln(e^A) = -A = ln(e^-A)      <[(A)]>
                              <  A  >                 involution
                              [(<A>)]                 involution


A/A = 1                       ([A] <[A]>)
                              (         )             inversion
```

```
1/(1/A) = A                (<[ (<[A]>) ]>)
                           (<   <[A]>   >)        involution
                           (     [A]     )        inverse cancel
                                  A               involution


e^A * e^-A = 1             ([(A)][(<A>)])
                           (  A    <A>  )         involution
                           (            )         inversion


A*(1/B) = A/B              ( [A][(<[B]>)])
                           ( [A]  <[B]>  )        involution


1/(A^B) = A^-B          (<[(([[A]] [ B ] ))]>)
                        (<  ([[A]] [ B ] )  >)    involution
                        (   ([[A]] [<B>] )   )    promote
```

```
1/A + 1/B = (A + B)/AB

(<[A]>)(<[B]>) =?= ([A B] <[A][B]>)

      (<[A]>) = ([B]<[B]><[A]>) = ([B]<[A][B]>)        inversion
      (<[B]>) = ([A]<[A]><[B]>) = ([A]<[A][B]>)        inversion

      (<[A]>)(<[B]>) = ([B]<[A][B]>)([A]<[A][B]>)      substitute
                     = ([A B]<[A][B]>)                 distribution
```

Generalized Cardinality

Multiple reference can be explicit (a listing) or implicit (a counting). n references to A can be abstracted to n times a single A, in both the additive and the multiplicative contexts. The form of cardinality is:

Form	Interpretation
([A][n])	A*n

Adding A to itself n times is the same as multiplying A by n:

```
A..n..A =  ([A][n])
```

Multiplying A by itself n times is the same as raising A to the power n:

```
([A]..n..[A]) = (([[A]][n]))
```

Negative cardinality cancels or suppresses positive occurrences. The form of negative cardinality is

```
      ([A][<n>])                    A*(-n)
```

Adding A to itself -n times is the same as multiplying A by -n, and is also the same as adding -A to itself n times:

```
A..<n>..A = ([A][<n>]) = <([A][n])> = ([<A>][n]) = <A>..n..<A>
```

Dividing by A n times is the same as multiplying A by itself -n times.

```
(<[A]>..n..<[A]>) = (([<[A]>][n])) = (<([[A]][n])>)
                  = (([[A]][<n>])) = ([A]..<n>..[A])
```

Multiplying -A by itself n times is the same as raising -A to the nth power:

```
([<A>]..n..[<A>]) = (([[<A>]][n]))
```

Here is a proof that negative cardinality cancels positive cardinality:

```
([A][n]) ([A][<n>])                (n*A)+(-n*A) = 0

([A][n <n>])                                   distribution
([A][ ])                                       inversion
void                                           dominion
```

Fractional cardinality constructs fractions and roots. The form of fractional cardinality is:

```
      ([A]<[n]>)                    A*(1/n)
```

Adding the fraction A/n to itself n times yields A. Here is a proof that fractional cardinality accumulates into a single form:

```
([A]<[n]>)..n..([A]<[n]>)          (A/n) +..n..+ (A/n) = A

([([A]<[n]>)][n])                              cardinality
(  [A]<[n]>  [n])                              involution
(  [A]          )                              inversion
    A                                          involution
```

Multiplying the fraction n/A by itself 1/n times yields 1/A:

```
([n]<[A]>)..1/n..([n]<[A]>)        (n/A)*(1/n)= 1/A

([([n]<[A]>)][(<[n]>)])                        cardinality
(  [n]<[A]>    <[n]>  )                        involution
(     <[A]>           )                        inversion
```

Broadening the Distributive Axiom

Addition of complex fractions requires a broader distributive law, here expressed as several new theorems:

```
(A  [B] ) (A  [C] ) = (A [B C])                       wholes
(A  [B] ) (A <[C]>) = (A [B (<[C]>)])                 whole + fraction
(  <[B]>) (  <[C]>) = ([B C] <[B][C]>)                reciprocals
(A <[B]>) (A <[C]>) = (A [B C] <[B][C]>)              reciprocals * A
     B    (  <[C]>) = ([([B][C])( )] <[C]>)           compound fraction
(A  [B] ) (D <[C]>) = ([(A [B][C])(D)] <[C]>)         complex fraction
(A <[B]>) (D <[C]>) = ([(A [B])(D [C])] <[B][C]>)     fractions
```

Some proofs:

```
(<[A]>        ) (<[B]>        )                 reciprocals lhs
(<[A]>[B]<[B]>) (<[B]>[A]<[A]>)                 inversion
( [B] <[A][B]>) ( [A] <[A][B]>)                 inverse collect
([A B]<[A][B]>)                                 distribution

(A   <[B]>  ) (A   <[C]>  )                     whole+fraction rhs
(A [(<[B]>)]) (A [(<[C]>)])                     involution
(A [(<[B]>)       (<[C]>)])                     distribution
(A [([B C]<[B][C]>)])                           reciprocals
(A   [B C]<[B][C]>  )                           involution

(  A <[B]>        ) (  A <[C]>        )         whole+fraction rhs
(  A <[B]>[C]<[C]>) (  A <[C]>[B]<[B]>)         inversion
(  A [C]  <[B][C]>) (  A [B]  <[B][C]>)         inverse collect
([(A [C])]<[B][C]>) ([(A [B])]<[B][C]>)         involution
([(A [C]) (A [B])] <[B][C]>)                    distribution
([(A [B       C])] <[B][C]>)                    distribution
(  A [B       C]   <[B][C]>)                    involution

    B             (    <[C]>)                   compound rhs
(  [B]          ) (    <[C]>)                   involution
(  [B][C]  <[C]>) (    <[C]>)                   inversion
([([B][C])]<[C]>) ([()]<[C]>)                   involution
([([B][C])          ()]<[C]>)                   distribution
```

(A [B]) (D <[C]>)	complex fraction rhs
(A [B][C] <[C]>) (D <[C]>)	inversion
([(A [B][C])]<[C]>) ([(D)] <[C]>)	involution
([(A [B][C]) (D)] <[C]>)	distribution

James Calculus Unit Combinations

These unit combinations identify stable, irreducible forms in this calculus. Thus, they expose the representational and interpretive basis of numbers.

Form Value Interpretation

Void form

	0	0

The void initializes the system with a zero concept, {0}.

Single unit forms

()	1	e^0
[]	-inf	ln 0
<> = void	0	-0

The single units generate {1, -inf}.

Two unit combinations

(<>) = ()	1	e^-0
(())	e	e^e^0 = e^1
([]) = void	0	e^(ln 0) = e^(-inf)
[<>] = []	-inf	ln -0 = ln 0 = -inf
[()] = void	0	ln e^0 = ln 1 = 0
[[]] = <[]>	inf	lnln 0 = ln -inf = ln-1 + ln inf
<<>> = <>	void	--0 = 0
<()>	-1	-e^0
<[]>	inf	--inf = inf

The two-unit combinations generate {-1, e, inf}.

Three unit combinations

```
<([])> = <[()]> = ([<>]) = [(<>)] = 0

(<[]>) = <[]>                 e^inf = inf

[<()>]                        J, the imaginary ln-1
```

The only three unit combination of all three containers which does not reduce is imaginary. There are 3 additional stable three unit combinations which contain more than one instance of the unit boundary:

```
<(())>          -e

(<()>)          e^-1 = 1/e

((()))          e^e
```

Thus the three unit stable forms generate {-e, 1/e, e^e, ln-1}

Stable Forms

Any representation in a boundary system which is stable, in that no more reductions are possible, must represent a number. The tableau of stable unit forms, independent of the base for logs and exponents, recapitulates the coevolution of form and concept in this system. This analysis is similar to that of examining the void case of the transformation rules.

Let # represent the base of the exp-log forms. Level refers to the number of containers in a form.

Level	Stable forms	Interpretation
0	void	0
1	() []	1 -inf
2	<()> (()) <[]>	-1 # inf
3	<(())> (<()>) ((())) [<()>]	-# 1/# #^# log# -1

The origin is the void, which takes the additive unit value of 0. 1 and -inf are built in as the initial distinctions from the void. The troublesome concepts of infinity and inversion are confounded at level 1, and disambiguated at level 2. The arbitrary base unit # is introduced at level 2 and articulated through all inverse operations at level 3. As defined, # cannot equal any of {0,1,-1,inf,-inf} since each of these has a different stable pattern. These forbidden base values are also anchored by the definition of the exp-log functions, with these relationships:

```
log# 1 = 0                    #^0 = 1
log# 0 = -inf                 #^-inf = 0
log# inf = inf                #^inf = inf
log# # = 1                    #^1 = #
```

These relationships indicate points in the log-exp functions which are independent of base.

Invalid bases can be assigned a meaning by treating them as imaginary. The equation which permits movement between imaginary and real logarithms is

```
#^(log# x) = x
```

We can elect to interpret this equation as valid, again independent of the actual base. Thus # could any form, including the forbidden ones. We will now use this finesse to define logarithms of negative numbers.