# Using the IC Reals Library

Lindsay Errington and Reinhold Heckmann

January 22, 2002

## 1 Introduction

The Imperial College Exact Real Arithmetic Library is a collection of C types and functions which implement exact real arithmetic. The functions allow one to construct objects representing real numbers and then to demand information, for example some number of decimal digits, from those objects. The user need not specify a precision in advance. All the digits retrieved are correct and further digits can be demanded.

The library includes arithmetic operations as well as a suite of analytic functions on reals. In addition to a real type, the library also includes a lazy boolean type and a collection of predicates on reals, boolean operations and a conditional construct.

The representation of reals is based on *linear fractional transformations* (LFTs). The underlying theory was developed by Abbas Edalat, Martin Escardo, Reinhold Heckmann, Peter Potts, Philipp Sünderhauf and Lindsay Errington (lazy booleans).

The library was written by Lindsay Errington with contributions from Marko Krznaric and Reinhold Heckmann.

This document describes the types and functions provided by the library. It is not an introduction to exact real arithmetic nor does it describe the details of the LFT approach.

# 2 Copyright

All software in this distribution comes under the following copyright notice:



# 3 Installation

To install the library it is necessary to edit the Makefile and change `REALDIR` to point to the root of the Real Library source tree.

Also, the library requires the GNU Multiple Precision Arithmetic Library (GMP) (Version 3.1). The Makefile defines a variable `GMPDIR` which is assumed to be the root directory of the GMP installation.

Assuming the Makefile has been edited such that `REALDIR` and `GMPDIR` point to the respective directories, then the command

```
make
```

will create the real library. To remove the library and all binaries, type:

```
make clean
```

# 4 Using the library

All applications which call functions in the library must include the file `real.h`. This file defines all the types and prototypes for all functions exported from the library. In particular the file defines the types `Real` and `Bool` corresponding to lazy reals and lazy booleans. The file `real.h` includes `gmp.h`.

The user must call the function

```
initReals();
```

before invoking any other functions in the library. This function should be called only once.

# 5 Types

The main purpose of the library is to provide a type `Real` of real numbers. Since it requires the GNU Multiple Precision Arithmetic Library (GMP) as well, it also provides GMP's big integer type `mpz_t` as a byproduct. Most of the functions defined by the library come in a number of instances for different types. These instances are named using suffixes abbreviating type names; e.g. the suffix `R` indicates the type `Real`. The full list is given by the following table.

| Abbreviation | Prototype | Denotation |
|---|---|---|
| `R` | `Real` $x$ | $x : \mathbb{R}$ |
| `Int` | `int` $x$ | $x : \mathbb{Z}$, $x$ machine integer |
| `Z` | `mpz_t` $x$ | $x : \mathbb{Z}$, $x$ GMP integer |
| `QInt` | `int` $a$, `int` $b$ | $\frac{a}{b} : \mathbb{Q}$, $a$, $b$ machine integers |
| `QZ` | `mpz_t` $a$, `mpz_t` $b$ | $\frac{a}{b} : \mathbb{Q}$, $a$, $b$ GMP integers |

Usually, the specialised functions are more efficient than the general ones. The possibility of using machine integers is particularly useful since these integers are readily available and need not be set up specially.

# 6 Arithmetic

The basic functions for addition, subtraction, multiplication and division occur in a general form operating on two reals, and in various specialised forms involving integers. The available functions are listed in Table 1.

| | |
|---|---|
| `Real neg_R(Real` $x$`)`<br>`Real abs_R(Real` $x$`)` | $-x$<br>$\lvert x\rvert$ |
| `Real add_R_R(Real` $x$, `Real` $y$`)`<br>`Real add_R_Int(Real` $x$, `int` $y$`)`<br>`Real add_R_Z(Real` $x$, `mpz_t` $y$`)` | $x+y$ |
| `Real add_R_QInt(Real` $x$, `int` $a$, `int` $b$`)`<br>`Real add_R_QZ(Real` $x$, `mpz_t` $a$, `mpz_t` $b$`)` | $x+\frac{a}{b}$ |
| `Real sub_R_R(Real` $x$, `Real` $y$`)`<br>`Real sub_R_Int(Real` $x$, `int` $y$`)`<br>`Real sub_Int_R(int` $x$, `Real` $y$`)` | $x-y$ |
| `Real sub_R_QInt(Real` $x$, `int` $a$, `int` $b$`)`<br>`Real sub_QInt_R(int` $a$, `int` $b$, `Real` $x$`)` | $x-\frac{a}{b}$<br>$\frac{a}{b}-x$ |
| `Real mul_R_R(Real` $x$, `Real` $y$`)`<br>`Real mul_R_Int(Real` $x$, `int` $y$`)`<br>`Real mul_R_Z(Real` $x$, `mpz_t` $y$`)` | $x\times y$ |
| `Real mul_R_QInt(Real` $x$, `int` $a$, `int` $b$`)`<br>`Real mul_R_QZ(Real` $x$, `mpz_t` $a$, `mpz_t` $b$`)` | $x\times\frac{a}{b}$ |
| `Real div_R_R(Real` $x$, `Real` $y$`)`<br>`Real div_R_Int(Real` $x$, `int` $y$`)`<br>`Real div_Int_R(int` $x$, `Real` $y$`)` | $\frac{x}{y}$ |
| `Real div_R_QInt(Real` $x$, `int` $a$, `int` $b$`)`<br>`Real div_QInt_R(int` $a$, `int` $b$, `Real` $x$`)` | $x/\frac{a}{b}=\frac{bx}{a}$<br>$\frac{a}{b}/x=\frac{a}{bx}$ |
| `Real pow_R_R(Real` $x$, `Real` $y$`)` | $x^y$ |

Table 1: Primitive arithmetic functions

The following two functions define the rational number $\frac{a}{b}$, considered as a real:

`Real real_QInt(int` $a$, `int` $b$)
`Real real_QZ(mpz_t` $a$, `mpz_t` $b$)

Real numbers are implemented using *linear fractional transformations* (LFTs). Users can construct LFT functions explicitly. The following two functions compute the expression $\frac{ax+b}{cx+d}$:

`Real lft_R_Int(Real` $x$, `int` $a$, `int` $b$, `int` $c$, `int` $d$)
`Real lft_R_Z(Real` $x$, `mpz_t` $a$, `mpz_t` $b$, `mpz_t` $c$, `mpz_t` $d$)

The next two functions yield the even more complicated expression $\frac{axy+bx+cy+d}{exy+fx+gy+h}$:

`Real lft_R_R_Int(Real` $x$, `Real` $y$, `int` $a$, `int` $b$, $\ldots$, `int` $h$)
`Real lft_R_R_Z(Real` $x$, `Real` $y$, `mpz_t` $a$, `mpz_t` $b$, $\ldots$, `mpz_t` $h$)

**Examples:**

- To compute $y = x + 1$ ($x$ real), use `y = add_R_Int (x, 1);`
  This is both simpler and more efficient than to use
  ```
  one = real_QInt (1, 1);  y = add_R_R (x, one);
  ```
- To compute $y = \frac{x+1}{x-1}$, use
  ```
  y = lft_R_Int (x, 1, 1, 1, -1);
  ```
  This is both simpler and more efficient than to use
  ```
  y = div_R_R (add_R_Int (x, 1), sub_R_Int (x, 1));
  ```
- To compute $z = \frac{2x+y}{xy-1}$, use
  ```
  z = lft_R_R_Int (x, y, 0, 2, 1,  0,
                         1, 0, 0, -1);
  ```
  This is both shorter and more efficient than to use
  ```
  num = add_R_R (mul_R_Int (x, 2), y);
  den = sub_R_Int (mul_R_R (x, y), 1);
  z = div_R_R (num, den);
  ```
  (We admit that both versions are not quite readable.)

# 7 Special functions

There are the usual standard functions, each existing in three versions, one for a real argument, one for a rational argument made from machine integers, and one for a rational argument made from GMP integers. Often, the rational version will be more efficient (sometimes, it is just mapped to the real version, but it is offered anyway for uniformity and convenience). The general pattern is illustrated at the square root function:

| | |
|---|---|
| `Real sqrt_R(Real` $x$`)` | to compute $\sqrt{x}$; |
| `Real sqrt_QInt(int` $a$, `int` $b$`)` | to compute $\sqrt{\frac{a}{b}}$; |
| `Real sqrt_QZ(mpz_t` $a$, `mpz_t` $b$`)` | to compute $\sqrt{\frac{a}{b}}$. |

In the following list of functions, we enumerate only the 'R' versions.

Basic: `sqrt_R` for $\sqrt{x}$, `exp_R` for $e^x$, `log_R` for natural logarithm;

| | | | | | | |
|---|---|---|---|---|---|---|
| Trigonometric: | `sin_R`, | `cos_R`, | `tan_R`, | `sec_R`, | `cosec_R`, | `cotan_R` |
| Inverse trigonometric: | `asin_R`, | `acos_R`, | `atan_R`, | `asec_R`, | `acosec_R`, | `acotan_R` |
| Hyperbolic: | `sinh_R`, | `cosh_R`, | `tanh_R`, | `sech_R`, | `cosech_R`, | `cotanh_R` |
| Inverse hyperbolic: | `asinh_R`, | `acosh_R`, | `atanh_R`, | `asech_R`, | `acosech_R`, | `acotanh_R` |

In addition, there are the two predefined constants `Real Pi` and `Real E`.

# 8 Forcing, printing and conversion

When working with the library, it is best to think of real numbers as infinite digit streams (but these digit expansions do not correspond directly to any familiar binary or decimal system). Each finite prefix corresponds to a rational interval (much as the finite prefix 3.14 of 3.14159... corresponds to the interval [3.14, 3.15]). Thus, if more and more digits of the stream are computed, the result is a nested sequence of intervals $[a_1, b_1] \supseteq [a_2, b_2] \supseteq \cdots$ which provide increasingly better approximations to the real number.

If a real number is set up and assigned to a variable, an object is created which records the way the number was constructed, but no digits are actually calculated. It is only when the number is "forced" that a finite prefix of the digit stream is computed.

There are two ways to specify the amount of forcing: the first, `force_R_Digs`, is by indicating the number of digits to be computed. Unfortunately, there is no simple rule telling the size of the resulting interval. This is the reason why there is a second force function, `force_R_Dec`, which forces a real number until an interval is obtained which guarantees a certain decimal precision.

`void force_R_Digs(Real` $x$, `int` $n$)
This computes at least the first $n$ digits of the digit stream describing the value of the argument $x$.

`void force_R_Dec(Real` $x$, `int` $n$)
This computes an approximating interval for $x$ whose size is at most $10^{-n}$.

`void print_R(Real` $x$)
This takes whatever information about the value of $x$ is currently available and prints it as an interval (no forcing).

`void print_R_Digs(Real` $x$, `int` $n$)
This first calls `force_R_Digs(`$x$, $n$) and prints the interval which results from this forcing.

`void print_R_Dec(Real` $x$, `int` $n$)
This first calls `force_R_Dec(`$x$, $n$) and prints the resulting interval.

`double realToDouble(Real` $x$)
This takes whatever information about the value of $x$ is currently available, and converts one of the end-points of this interval to a double precision floating point value.

`void force_B(Bool` $b$, `int` $n$)
This can be called to force the evaluation of a boolean. When $b$ is viewed as a stream, the argument $n$ indicates the maximum depth in the stream to examine to determine the value of the boolean.

`Boolean boolValue(Bool` $b$)
This is a macro which returns the value of a boolean. This may be one of three constants: `LAZY_TRUE`, `LAZY_FALSE` or `LAZY_UNKNOWN`

`Real realError(char*` *string*)
This function "computes" and returns a kind of placeholder for a real number which is fine as long as it is not forced. But if this placeholder is forced to produce some digits, then it causes the program to be aborted. The argument string provided in the call of `realError` is printed as an error message.
`Real realDelay(Delay_Fun` $f$, `Delay_Arg` $x$)
This function yields a *closure* for the function $f$ applied to $x$. In other words, it denotes the real $f(x)$ but the function call is not made until the closure is forced. An example of its use is given in the next section.

# 9 Predicates, Boolean operations and conditionals

The library introduces a new type `Bool` for Boolean values which serves as the result type of predicates. Therefore, it must also introduce its own versions of Boolean operations and its own conditional, which is a function reminiscent of Dijkstra's guarded commands. We shall shortly see why this was done, but first we introduce the corresponding functions.

Table 2 shows the available predicates:

| | |
|---|---|
| `Bool lt_R_0(Real` $x$`)` | $x < 0$ |
| `Bool lt_R_QInt(Real` $x$, `int` $a$, `int` $b$`)` | $x < \frac{a}{b}$ |
| `Bool lt_R_R(Real` $x$, `Real` $y$`)` | $x < y$ |
| `Bool ltEq_R_0(Real` $x$`)` | $x \leq 0$ |
| `Bool ltEq_R_QInt(Real` $x$, `int` $a$, `int` $b$`)` | $x \leq \frac{a}{b}$ |
| `Bool ltEq_R_R(Real` $x$, `Real` $y$`)` | $x \leq y$ |
| `Bool gt_R_0(Real` $x$`)` | $x > 0$ |
| `Bool gt_R_QInt(Real` $x$, `int` $a$, `int` $b$`)` | $x > \frac{a}{b}$ |
| `Bool gt_R_R(Real` $x$, `Real` $y$`)` | $x > y$ |
| `Bool gtEq_R_0(Real` $x$`)` | $x \geq 0$ |
| `Bool gtEq_R_QInt(Real` $x$, `int` $a$, `int` $b$`)` | $x \geq \frac{a}{b}$ |
| `Bool gtEq_R_R(Real` $x$, `Real` $y$`)` | $x \geq y$ |

Table 2: Predicates on reals

Boolean values may be combined with the operators presented in Table 3:

| | |
|---|---|
| `Bool and_B_B(Bool` $x$, Bool $y$) | $x \wedge y$ |
| `Bool or_B_B(Bool` $x$, Bool $y$) | $x \vee y$ |
| `Bool not_B(Bool` $x$) | $\neg x$ |

Table 3: Boolean operators

Finally, the conditional is a function with a variable number of arguments:

`Real realIf(int` $n$, `Bool` $b_1$, `Real` $x_1$, ..., `Bool` $b_n$, `Real` $x_n$)

This function takes an integer as its first argument, followed by a variable number of guard/value pairs. The integer argument should tell the number of these pairs. The variable argument list is implemented with stdarg(3).

Roughly spoken, the function evaluates the guards $b_1, \ldots, b_n$, then chooses non-deterministically one of the guards which happened to be true, and returns the corresponding value. Before we can provide a more detailed description, we must say more about the type `Bool` and the behaviour of the predicates.

Recall that an element of `Real` is implemented as a digit stream, whose initial prefixes provide a shrinking sequence of intervals approximating a real number. Correspondingly, the elements of type `Bool` are implemented as sequences of "truth intervals". These sequences usually start out with the interval `Unknown` = [`False`, `True`], which may at some later stage be refined to either `False` or `True`.

**Example:** Table 4 shows how nested sequences of intervals are mapped to sequences of truth values by the predicate `gtEq_R_0`.

| | | | |
|---|---|---|---|
| $[-3, 2]$ | `Unknown` | $[-2, 3]$ | `Unknown` |
| $[-2, 1]$ | `Unknown` | $[-1, 2]$ | `Unknown` |
| $[-1.5, 0.5]$ | `Unknown` | $[-0.5, 1.5]$ | `Unknown` |
| $[-1, 0]$ | `Unknown` | $[0, 1]$ | `True` |
| $[-0.8, -0.2]$ | `False` | $[0.2, 0.8]$ | `True` |
| $[-0.7, -0.3]$ | `False` | $[0.3, 0.7]$ | `True` |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |

Table 4: Action of the predicate `gtEq_R_0`

Thus, the sequence of truth values computed by `gtEq_R_0`($x$) will eventually reach `True` if $x > 0$, and will eventually reach `False` if $x < 0$. If the exact value of $x$ happens to be 0, it is possible that the sequence of truth values remains `Unknown` for ever—this happens if all the intervals $[a, b]$ approximating $x$ have the property $a < 0 < b$. Yet it is also possible that the sequence switches to `True`—this happens if there is an approximating interval $[a, b]$ with $a = 0$. Which of these two possibilities occur depends on the way how $x$ was set up, and on implementation details. But it should be remembered that `gtEq_R_0` may remain undecided for ever when applied to 0.

The Boolean operations produce their output stream by acting on their input stream(s) element by element, i.e. to produce the $n$th element of the output stream, the $n$th input element(s) of the input stream(s) are combined according to Table 5, where the truth values have been abbreviated by U, F, and T, and the operations by logical symbols.

| $x$ | T | F | U |
|---|---|---|---|
| $\neg x$ | F | T | U |

| $\wedge$ | T | F | U |
|---|---|---|---|
| T | T | F | U |
| F | F | F | F |
| U | U | F | U |

| $\vee$ | T | F | U |
|---|---|---|---|
| T | T | T | T |
| F | T | F | U |
| U | T | U | U |

Table 5: Action of the Boolean operators

Now, we can return to the conditional

`Real realIf(int` $n$, `Bool` $b_1$, `Real` $x_1, \ldots,$ `Bool` $b_n$, `Real` $x_n$)

This function behaves as follows: It constructs a cyclic list of guard/value pairs. Starting with the first pair, `realIf` forces the guard to compute an element of the resulting Boolean stream. If the value of this element is `True`, the real number associated with this guard is returned. If the value is `False`, the pair is removed from the list. If the value remains `Unknown`, then `realIf` tries the next pair. In this way, the function cycles through the list forcing each guard in turn until a guard becomes `True`. If the list becomes empty (all the guards are `False`), then `realIf` issues an error message. If some guards are remaining `Unknown` for ever, `realIf` will not terminate. Some examples will clarify the situation.

```
realIf (2,  lt_R_QInt (x, 1, 1), 0,  gt_R_0(x), 1)
```

means $x < 1 \to 0 \mathbin{[\!]} x > 0 \to 1$. For $x = \frac{1}{2}$, the result is unpredictable; it depends on the actual sequence of intervals which approximate $\frac{1}{2}$. On the other hand, there is no risk of non-termination or error since at least one guard will eventually yield `True` for any $x$.

```
realIf (2,  ltEq_R_0(x), neg_R(x),  gtEq_R_0(x), x)
```

means $x \leq 0 \rightarrow -x \; [\!] \; x \geq 0 \rightarrow x$. As an implementation of $|x|$, this works well for all $x \neq 0$, while there is a considerable risk of non-termination for $x = 0$. (Fortunately, there is the predefined function `abs_R`).

Suppose you have an implementation `sqrt1` for square root which works only for arguments in the interval $(\frac{1}{4}, 4)$, but not for arguments near 0 or very big arguments. With this knowledge, you can set up the following function:

$$\sqrt{x} \;=\; (\begin{array}{lll} x > \frac{1}{4} \wedge x < 4 & \rightarrow \texttt{sqrt1}(x) & [\!] \\ x \geq 0 \wedge x < \frac{1}{2} & \rightarrow \frac{1}{2}\sqrt{4x} & [\!] \\ x > 2 & \rightarrow 2\sqrt{x/4} & [\!] \\ x < 0 & \rightarrow ??? & ) \end{array}$$

Notice how the guards overlap: if the second guard were $x \geq 0 \wedge x \leq \frac{1}{4}$, then there would be a considerable risk of non-termination for $x = \frac{1}{4}$. By the overlap, this non-termination is prevented—without introducing non-determinism since the values following these guards are semantically equal in the overlap region. Similarly, the first and third guard overlap to prevent non-termination for $x = 4$. Yet notice that the function does not terminate for $x = 0 \ldots$

As a C program, the above function appears in figure 1. Note the use of delays to prevent endless eager recursion.

As the last example, suppose you want to iterate a function `f` until the difference between two consecutive values in the iteration is smaller than some threshold *eps*. This can be done by the following recursive function:

```
Real iter (Real x) {
  Real y = f(x);
  Real d = abs_R (sub_R_R (x, y));   /* d = |x - y| */
  return realIf (2,
    lt_R_R (d, eps),
      y,
    gt_R_R (d, eps2),
      realDelay((Delay_Fun) iter, (Delay_Arg) y));
}
```

where *eps2* is *eps*/2. By using *eps2*, the two guards overlap non-trivially, and

```
Real sqrt (Real x) {
  return realIf (4,
    and_B_B (gt_R_QInt (x, 1, 4), lt_R_QInt (x, 4, 1)),
        sqrt1 (x),
    and_B_B (gtEq_R_0, lt_R_QInt (x, 1, 2)),
        div_R_Int (
          realDelay(
            (Delay_Fun) sqrt,
            (Delay_Arg) mul_R_Int (x, 4)),
          2),
    gt_R_QInt (x, 2, 1),
        mul_R_Int (
          realDelay(
            (Delay_Fun) sqrt,
            (Delay_Arg) div_R_Int (x, 4)),
          2),
    lt_R_0 (x),
        realError ("Square root of negative number")
  );
}
```

Figure 1: Example of conditional with delays.

non-termination at $d = eps$ is prevented (of course, the iteration still fails to terminate if $d$ never gets small).

# 10 Extracting digits following forcing

The functions in this section provide access to the internal representation of real numbers. They are not very useful for ordinary users of the library.

`void retrieveInfo(Real` *x*, `Sign` $*$*sign*, `int` $*$*count*, `mpz_t` digits)

This function retrieves the information that is currently available on $x$. The sign of $x$ is stored in *sign*, the number of digits calculated so far is stored in *count*, and a compressed representation of all these digits is deposited in *digits*. The variable *digits* must be initialised with the GMP function `mpz_init` prior to calling `retrieveInfo`. The real argument $x$ is unchanged by the call.

From a compressed digit representation as it is provided by `retrieveInfo`, the individual digits can be extracted by means of

`Digit takeDigit(int` $*$*count*, `mpz_t` *digits*)

This function should only be called if *digits* contains at least one digit. It returns the most significant digit contained in *digits*, removes this digit from *digits*, and decreases the counter *count* by 1. Thus, successive calls yield successive digits.

An example of the use of these two functions is given in Figure 2.

Note the two functions `signToString` and `digitToString` which convert signs and digits to strings for output.

# 11 Environment variables

The library uses three environment variables to control its runtime behaviour. These variables are as follows:

`ICR_STACK_SIZE=`$n$ This sets the runtime stack to $n \times k$ words. The default is $n = 20$ for $20k$ words. It is unlikely that the stack size needs to

```
Real x;
mpz_t digits;
Sign sign;
Digit digit;
int count;
...
x = tan_R(y);
force_R_Digs(x, 20);
mpz_init(digits);
retrieveInfo(x, &sign, &count, digits);
printf("%s ", signToString(sign));
while (count > 0) {
  digit = takeDigit(&count, digits);
  printf("%s ", digitToString(digit));
}
printf("\n");
```

Figure 2: Taking digits one-by-one.

be adjusted. If you get a "stack overflow" at runtime, it is more likely that the algorithm for some function is not sufficiently converging for a given argument. Assuming the default has been set to something other than 1, try setting the environment variable `ICR_DEFAULT_FORCE_COUNT=1` and executing your program again.

`ICR_DEFAULT_FORCE_COUNT=`$n$ Sometimes it is necessary to force an arbitrary number of digits from an LFT. This can happen, for example, when a predicate is forced which in turn must force some number of digits from its real argument. In theory, one would always want to force as few digits as possible (i.e. 1 digit) to avoid unnecessary computation. In practice, it is more efficient to demand more than one digit. The value of `ICR_DEFAULT_FORCE_COUNT` is the number of digits forced in such circumstances. It is also the number of digits forced from an argument to a linear fractional transformation when $\epsilon - \delta$ analysis for the transformation yields a value $\leq 0$. The default is $n = 1$. The maximum reasonable value is $n = 4$.

`ICR_FORCE_DEC_UPPER_BOUND=`$n$ When extracting information from reals, the library works with "digits". Each digit gives a fixed amount of information. Usually, however, a user wishes to extract enough information to ensure some decimal precision. The functions `force_R_Dec` and `print_R_Dec` are provided to retrieve information from a real to a specified decimal precision. Unfortunately, there is not always a direct correspondence between a number of digits and a decimal precision. As a number approaches infinity, more digits are needed to bound it above. This variable sets a bound on the number of digits to retrieve from a real to bound it above before giving up. The default is $n = 10000$.

## 12 The daVinci interface

For debugging and instruction, the library is instrumented to work with the graph visualisation tool daVinci. But be warned that much of this visualisation is only comprehensible to insiders. When a program has been compiled with daVinci enabled, a separate daVinci window will be created when `initReals` is called. Thereafter, any calls to `force_R_Digs` or `force_R_Dec` will lead to control being transferred to the daVinci window.

Once started the daVinci window provides a collection of buttons to control the execution of a real program. The buttons are as follows:

- ■ Stops execution.
- ▶ Enabled when there is work on the stack. This button starts execution.
- **1** Enabled when there is work on the stack. This single steps execution.
- ▼ When enabled, it means the stack is empty. This button returns control to the C program.
- **GC** This button is not implemented. Ultimately it will be used to invoke the garbage collector.

In addition, when the program is stopped, the user can click the right button over any object in the heap. This gives a popup menu of which only the first entry is implemented. It can be used to print (in the main program window) the contents of the selected heap object. This is typically a linear fractional transformation (nearly all functions in the library are implemented as compositions of linear fractional transformations).

## 13 Compilation flags

There are a number of compilation flags in the Makefile. With the exception of those listed below, it is unwise to change these flags.

`-DDAVINCI` When set, the library will connect to the daVinci graph visualisation tool. In this mode, all objects in the heap and all information flow is displayed in a separate daVinci window. Computation is controlled from the daVinci window. The user can run, stop and single-step the activities of the program, which mainly consist of emission and absorption of LFTs, and examine the contents of objects in the heap.

`-DDEFAULT_FORCE_COUNT=`$n$ This sets the default force count to use when it is not given by the environment variable `ICR_DEFAULT_FORCE_COUNT`.

`-DTRACE=`***val*** This enables tracing of force methods. When set to 0, tracing is disabled. When set to 1, tracing is enabled. Finally, when set to `traceOn`, tracing can be enabled and disabled at runtime under software control via the function `debugTrace(int` $b$`)` where $b$ is 1 to enable tracing and 0 to stop tracing.

`-DSTACK_SIZE=`$n$ This sets stack size to use when it is not given by the environment variable `ICR_STACK_SIZE`.

`-DFORCE_DEC_UPPER_BOUND=`$n$ This is the default value used when the environment variable `ICR_FORCE_DEC_UPPER_BOUND` is absent.

## 14 Problems

The library is still under development. Future versions of the library will have specialized versions of the analytic functions for rational arguments. A document describing the implementation is also planned. The most serious omission from the present version of the library is a garbage collector.