Verification in Real Computation, KAML seminar talk slides

Sewon Park

November 4, 2016

1. Real Computation
   - Model of Computation
   - iRRAM
2. Verification
   - Semantics
   - Examples: ilog, gaussian elimination and simple root finding
3. Conclusion and Future work

A real number is commonly approximated with a fixed precision floating point number: `double`, `float`, `long double` $\cdots$. Consider an iterative computation (logistic map):

$$x_n \mapsto x_{n+1} := rx_n(1 - x_n)$$

with $1 < r < 4$ and $x \in [0, 1]$. In C++,

- `float` has 23 mantissa bits,
- `double` has 52 mantissa bits,
- and `long double` has 80 mantissa bits (depending on compilers)

with $r = 3.75$ and $x = 0.5$ we can get

# of iterations	float	double	long double
30	0.715239	0.718096	0.718096
40	0.847816	0.416349	0.416349
85	0.6427	0.550527	0.816883
100	0.936589	0.936749	0.816798
200	0.578099	0.220493	0.513209
500	0.833184	0.786115	0.829312
1000	0.914432	0.340803	0.662663
10000	0.222593	0.276689	0.223618
100000	0.860895	0.239181	0.521347
500000	0.728425	0.274556	0.925446

How can we describe or design a computation over real numbers which contain infinite data?

1. Develop a model of computation which runs over an arbitrary ring R: Extended version of a Turing machine from a finite binary string to finite string of elements of R [BSCC89].
2. Develop a representation of real numbers and their computation so that it can be run by a Turing machine [Wei00].

Definition

A real number x is computable

1. if there exists a computable sequence of rational numbers $(q_n)_{n\in\mathbb{N}}$ that converges rapidly to x; that is $|x - q_i| < 2^{-i}$ for all i. (Cauchy-like sequence)
2. And equivalently, x is computable if there exists a computable sequence of nested open intervals with rational endpoints where x is the only value contained in infinitely many intervals.

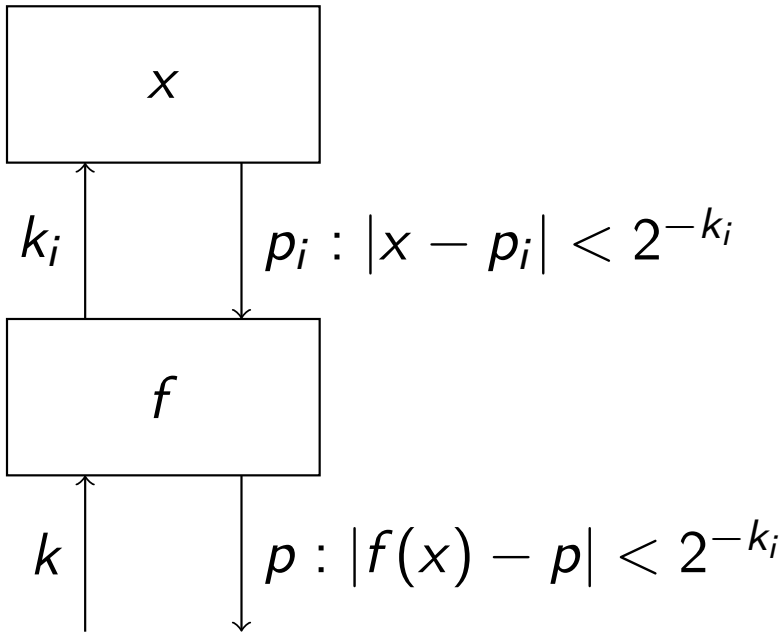
For example, think of π as a function $\mathtt{pi}(k : \mathbb{N}) : \mathbb{Q}$ that computes 2^{-k} approximation of π. Similarly, view a computable real number x as a function $\mathtt{x}(k : \mathbb{N}) : \mathbb{Q}$

Definition

A function $f :\subseteq \mathbb{R} \to \mathbb{R}$ is computable if there exists an oracle Turing machine with given $k \in \mathbb{N}$ which has finitely many oracle accesses to its input $x \in \mathrm{dom}(f)$ asking for its arbitrary close rational approximations and print $q \in \mathbb{Q}$ on the output tape such that $|f(x) - q| < 2^{-k}$ in finitely many steps.

For example, think of a computable function $f(x)$ as a function that receives a function that encodes a real number input in previous sense and a desired precision; $f(x)$ can be viewed as $\mathtt{f}(\mathtt{x} : \mathbb{N} \to \mathbb{Q}, k : \mathbb{N}) : \mathbb{Q}$.

A oracle Turing machine 'f' with an oracle 'x' that computes f. Oracle gives an approximation $p_i \in \mathbb{Q}$ which approximates a real number x up to a query k_i and the Turing machine computes an approximation $q \in \mathbb{Q}$ of the real number $f(x)$ up to desired k.



Now we can generalize the idea a little further.

Definition

A surjective function $\rho :\subseteq \Sigma^\omega \to X$ is a representation of X where Σ is alphabet (a finite set). $q \in \Sigma^\omega$ is a ρ-name of $x \in X$ if $\rho(q) = x$. $x \in X$ is ρ-computable if the ρ-name of x is computable.

For example, ρ_{dec} with $\rho_{\text{dec}}(3.141592\dots) = \pi$ is a decimal representation of $\mathbb{R}$ and ρ_{Cauchy} with $\rho_{\text{Cauchy}}(q_1\#q_2\#\cdots) = x$ where $|x - \rho_{\mathbb{Q}}(q_i)| < 2^{-i}$ is a Cauchy representation of $\mathbb{R}$, where $\rho_{\mathbb{Q}}$ is a representation of $\mathbb{Q}$.

Definition

A partial string function $f :\subseteq \Sigma^\omega \to \Sigma^\omega$ is computable if there exists a Turing machine with an infinite input tape and an infinite one-way output tape where $p \in \Sigma^\omega$ is given in the input tape and the Turing machine writes correct output $f(p)$ without stopping if p is in the domain.

Definition

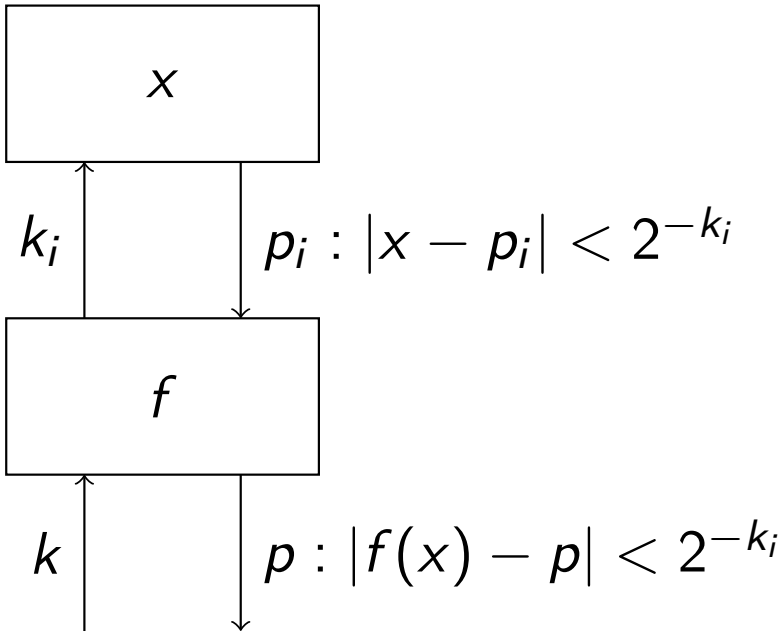
For representations of X and Y, ρ_X and ρ_Y respectively, $f :\subseteq X \to Y$ is (ρ_X, ρ_Y)-computable if there exists a string function F such that

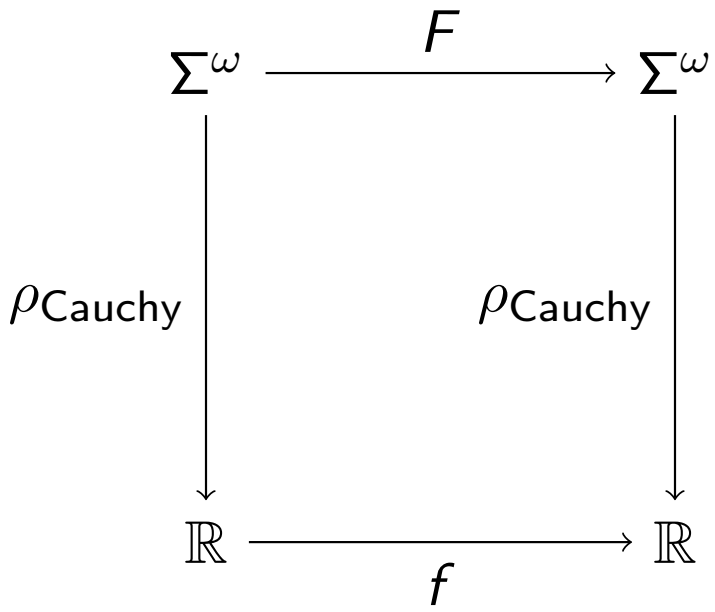
$$\rho_Y(F(s)) = f(\rho_X(s))$$

for all s in the domain of $f\rho_X$.

Observe that $f(x) = 3x$ is not $(\rho_{\text{dec}}, \rho_{\text{dec}})$-computable. A real number function is computable if and only if it is computable in sense of the Cauchy representation. Therefore, when we argue a computability of a real number function, it is in this sense.

We can view a notion of computability of a real function f as below diagrams.





We can verify computability of some basic operations:

1. field operations $+, -, \times, \div$ are computable functions of $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$
2. evaluation of a polynomial with computable coefficients is computable
3. min, max, abs are computable
4. cos, sin, e are computable

More importantly, composition of computable functions is computable.

Omitting a proof here, we address an important theorem:

Theorem

Computability implies continuity.

We can think of some trivial functions in mathematics that is not computable due to their discontinuity:

1. $\text{sign} : \mathbb{R} \to \{-1, 0, 1\}$ is not computable while $\text{sign} : \mathbb{R} \setminus \{0\} \to \{-1, 1\}$ is computable.
2. $\lfloor \cdot \rfloor : \mathbb{R} \to \mathbb{N}$ is not computable while $\lfloor \cdot \rfloor : \mathbb{R} \setminus \mathbb{N} \to \mathbb{N}$ is computable.
3. $=, >, \geq$ are not computable.

Real Computation

iRRAM

So far, we have discovered how to represent a real number, a real function and their computability. Now we introduce the theory implemented: `iRRAM`

Real Computation

iRRAM

- Turing machine model: operating on sequences of approximations from a fixed countable dense subset: TTE
  e.g. A real number x with Cauchy representation $\rho :\subseteq \Sigma^\omega \to \mathbb{R}$:

$$\rho(w_0 \# w_1 \# \cdots) = x$$

  where $\{w_i\}_{i\in\mathbb{N}}$ is an infinite sequence of binary representations of rational numbers converging rapidly to x
- Intuitive algebraic model: machine defined over a Ring or a Field R: BSS-machine
  e.g. Ring operation and in-equality test in unit cost:

$$\text{computation node: } \phi(\eta, \boldsymbol{z}) = (\beta(\eta), g_\eta(\boldsymbol{z}))$$

$$\text{branch node: } \phi(\eta, \boldsymbol{z}) = \begin{cases} (\beta_+(\eta), \boldsymbol{z}) & \text{if } z_1 \geq 0 \\ (\beta_-(\eta), \boldsymbol{z}) & \text{else} \end{cases}$$

  with endomorphism $\phi : \mathbb{N} \times \mathbb{R}^\infty \to \mathbb{N} \times \mathbb{R}^\infty$ where $R = \mathbb{R}$

Real Computation

iRRAM

- Combining both, model of imperative computation over the reals had been suggested and implemented as abstract data type `REAL` in the object-oriented programming language `C++` : `iRRAM`
  e.g. `REAL` $x :=$ `pi();` `REAL` $y :=$ `exp(1);` **return** $x > y$;
- More precisely, a real number is offered to users as an entity while the suggested theory is adapted internally. In more detail, a real number is dealt as an interval equipped with interval arithmetics and the program reiterates whenever tighter interval is required. Observe that it is equivalent to the definition.
-

$$x \in (a, b), y \in (c, d) \Rightarrow x + y \in (a + c, b + d)$$

Real Computation

iRRAM

- Combining both, model of imperative computation over the reals had been suggested and implemented as abstract data type `REAL` in the object-oriented programming language `C++` : `iRRAM`
  e.g. `REAL` $x :=$ `pi();` `REAL` $y :=$ `exp(1);` **return** $x > y$;
- More precisely, a real number is offered to users as an entity while the suggested theory is adapted internally. In more detail, a real number is dealt as an interval equipped with interval arithmetics and the program reiterates whenever tighter interval is required. Observe that it is equivalent to the definition.
-

$$x \in (a, b), y \in (c, d) \Rightarrow x - y \in (a - d, b - c)$$

Real Computation

iRRAM

- Combining both, model of imperative computation over the reals had been suggested and implemented as abstract data type `REAL` in the object-oriented programming language `C++` : `iRRAM`
  e.g. `REAL` $x :=$ `pi();` `REAL` $y :=$ `exp(1);` **return** $x > y$;
- More precisely, a real number is offered to users as an entity while the suggested theory is adapted internally. In more detail, a real number is dealt as an interval equipped with interval arithmetics and the program reiterates whenever tighter interval is required. Observe that it is equivalent to the definition.
-

$$x \in (a, b), y \in (c, d) \Rightarrow x * y \in (\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)$$

Real Computation

iRRAM

Below is a code of `python2.7` program which shows trivial situation where `double` fails; $a + 1000000b > a + 10000000b$

```
a = 1000000.0
b = 0.00000000001
x, y = 0.0
for i in range(1000000):
    x += b
x += a
y += a
for i in range(10000000):
    y += b
print y > x

>>> False
```

Lets see what happens in `iRRAM`

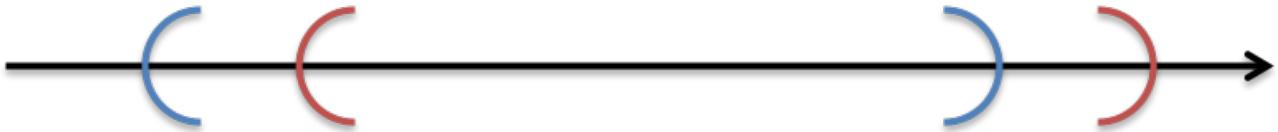
Real Computation

iRRAM

```
REAL a = 1000000;
REAL b = 0.00000000001;
REAL x,y;
for (int i=0; i<1000000; i++)
    x += b;
x += a;
y += a;
for (int i=0; i<10000000; i++)
    y += b;
if (y > x) cout << "True\n";
else cout <<"False\n";
```

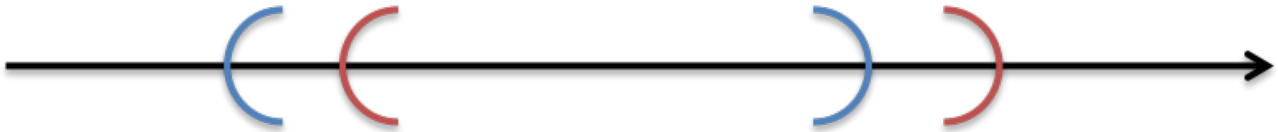
(internal precision step = 1)

- REAL blue, red;
- return blue > red;



(`internal precision step` $= 2$)

- `REAL` blue, red;
- return blue $>$ red;

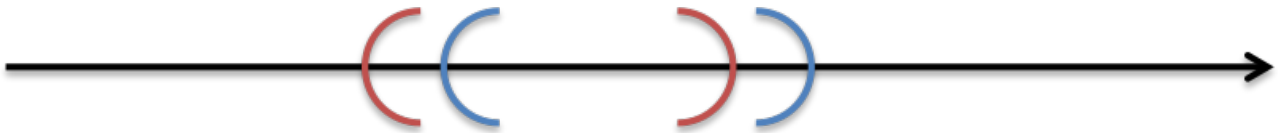


Real Computation

iRRAM

(`internal precision step` $= 3$)

- `REAL` blue, red;
- return blue $>$ red;



(internal precision step = 4)

- REAL blue, red;
- return blue > red;

(`internal precision step` $= 5$)

- `REAL` blue, red;
- return blue $>$ red;

Returning to the logistic map, $x_n \mapsto x_{n+1} := rx_n(1 - x_n)$ with $r = 3.75$ and $x = 0.5$ we can get

# of iterations	float	double	long double	REAL
30	0.715239	0.718096	0.718096	0.718096
40	0.847816	0.416349	0.416349	0.416349
85	0.6427	0.550527	0.816883	0.816938
100	0.936589	0.936749	0.816798	0.888293
200	0.578099	0.220493	0.513209	0.823557
500	0.833184	0.786115	0.829312	0.276753
1000	0.914432	0.340803	0.662663	0.791746
10000	0.222593	0.276689	0.223618	0.824204
100000	0.860895	0.239181	0.521347	0.666946
500000	0.728425	0.274556	0.925446	0.451203

Real Computation

iRRAM

- `iRRAM` possesses an exact computation over real numbers and yields correct output up to any desired precision.
- We can develop many interesting exact algorithms with `iRRAM`!; e.g., `rootFinding(REAL*, INTEGER)`, `eigenvalues(REAL**, INTEGER)`, `gaussianElimination(REAL**, INTEGER)`.
- What happens when the command $x > y$ is evaluated with $x = y$ in a computer state? E.g.
  - `REAL` $x = 3$;
  - `REAL` $y = 3$;
  - return $x > y$;
- verification in real computation is needed.

Definition (>)

Inequality test is defined as a partial function.

$$(x > y) = \begin{cases} 1 & : x > y, \\ 0 & : x < y, \\ \downarrow & : x = y \end{cases}$$

reminder

Equality test is in general equivalent to the Halting Problem.

$$(x > y) = \begin{cases} 1 & : x > y, \\ 0 & : x < y, \\ \downarrow & : x = y \end{cases}$$

Definition $(>_k)$

Not to face a Halting problem, below multivalued test can be introduced [EHS04].

$$(x >_k y) = \begin{cases} \{1\} & : x \geq y + 2^{-k}, \\ \{0\} & : x \leq y - 2^{-k}, \\ \{0,1\} & \text{else} \end{cases}$$

$(x >_k y)$ can be understood as evaluating $x > y - 2^{-k}$ and $x < y + 2^{-k}$ in parallel and return one that has evaluated to be True. Since at least one of two statement is promised to be True.

$$(x >_k y) = \begin{cases} \{1\} & : x \geq y + 2^{-k}, \\ \{0\} & : x \leq y - 2^{-k}, \\ \{0,1\} & \text{else} \end{cases}$$

Definition (choose)

General version of $>_k$ can be devised as a multivalued partial 'function':

$$\texttt{choose}(x_1 > y_1, ..., x_L > y_L) = l \text{ such that } x_l > y_l$$

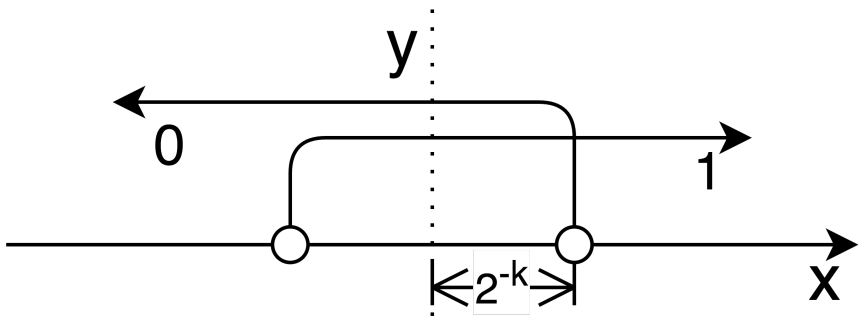


Figure: $(x >_k y) \equiv \texttt{choose}(x < y + 2^{-k}, x > y - 2^{-k}) - 1$

Verification

Semantics

- suppose we have implemented a program $\phi : \texttt{REAL}^n \to \texttt{REAL}^m$, how can we guarantee correctness of the program?
- 4 levels on verification of exact real arithmetic [MüUh12]:
  1. core level: arbitrarily precise floating-point numbers (mainly internal use)
  2. interval level: interval arithmetic (mainly internal use)
  3. basic arithmetic level: basic operations on real numbers (mainly internal use)
  4. application level: non-basic operations and user tools (mainly external use)
- we will generalize and use Floyd-Hoare Logic to formally verify the correctness of three algorithms in real computation:
  1. ilog,
  2. Gaussian Elimination,
  3. simple root finding

- Floyd-Hoare Logic is a deductive proof system built on Hoare triples:
  - $\{P\}\ C\ \{Q\}$: partial correctness
  - $[P]\ C\ [Q]$: total correctness

  with pre-condition P and post-condition Q of a command C.
- examples:
  1. *empty command* ϵ: $\{P\}\ \epsilon\ \{P\}$
  2. *assignment* : $\{P[a/x]\}\ x := a\ \{P\}$
  3. *conditional* : $\{P \wedge b\}\ c_0\ \{Q\},\ \{P \wedge \neg b\}\ c_1\ \{Q\}$
     $\Rightarrow \{P\}$ **if** b **then** c_0 **else** $c_1\ \{Q\}$
  4. *while loop* : $\{P \wedge b\}\ c\{P\}$
     $\Rightarrow \{P\}$ **while** b **do** $c\ \{P\}$

Remark 1

We cannot make division by zero exception [Takayuki Kihara, Arno Pauly *"Dividing by zero - how bad is it, really?"*, 41st International Symposium on Mathematical Foundations of Computer Science, 2016].

$$(x/y) = \begin{cases} x/y & \text{if } y \neq 0, \\ \downarrow & \text{if } y = 0 \end{cases}$$

Remark 2

With a state space Σ, boolean expression space $\mathbf{E}$ and boolean set $\mathbf{B}$, semantic function $[\![-]\!] : \mathbf{E} \to \Sigma \to \mathbf{B}$. For $\sigma \in \Sigma, e \in \mathbf{E}$, $[\![e]\!]\sigma$ is a boolean value of e evaluated in a state σ. With mentioned multivalued tests,

$$([\![x >_k y]\!]\sigma \ni \texttt{True}) \Rightarrow x > y - 2^{-k} \text{ in } \sigma$$
$$([\![x >_k y]\!]\sigma \ni \texttt{False}) \Rightarrow x < y + 2^{-k} \text{ in } \sigma$$

Verification

Example: ilog

- general notion of $\text{ilog}_2 : (0;\infty) \ni x \mapsto k \in \mathbb{Z} : 2^k < x \leq 2^{k+1}$ is not computable over continuous x. Instead we can compute

$$\text{ilog}_2 : (0;\infty) \ni x \mapsto \{k \in \mathbb{Z} : 2^{k-1} < x < 2^{k+1}\} \in \mathbb{N}$$

Verification

Example: ilog

$$\text{ilog}_2 : (0;\infty) \ni x \mapsto \{k \in \mathbb{Z} : 2^{k-1} < x < 2^{k+1}\}$$

Integer-valued logarithm using naïve test semantics

```
 1: function ilog_2(x : ℝ)
 2:     ℕ ∋ l := 1
 3:     if x > 1 then
 4:         ℝ ∋ y := x
 5:         while y > 2 do
 6:             l := l + 1 ; y := y/2
 7:         end while
 8:     else
 9:         repeat
10:             l := l − 1
11:         until x > 2^(l−1)
12:     end if
13:     return l
14: end function
```

Verification

Example: ilog

$$\mathrm{ilog}_2 : (0;\infty) \ni x \mapsto \{k \in \mathbb{Z} : 2^{k-1} < x < 2^{k+1}\}$$

Integer-valued logarithm using naïve test semantics

```
 1: function ilog2(x : ℝ)                                   // Require: x > 0
 2:     ℕ ∋ l := 1                                          // {0 < x, l = 1}
 3:     if x > 1 then                                       // {1 < x, l = 1}
 4:         ℝ ∋ y := x                                      // {1 = 2^(l-1) < y = y · 2^(l-1) = x}
 5:         while y > 2 do                                  // {2^l < y · 2^(l-1) = x}
 6:             l := l + 1 ; y := y/2                       // {2^(l-1) < y · 2^(l-1) = x}
 7:         end while                                       // {2^(l-1) < x = y · 2^(l-1) ≤ 2^l}
 8:     else                                                // {0 < x ≤ 1 = 2^(l-1)}
 9:         repeat
10:             l := l − 1                                  // {0 < x ≤ 2^l}
11:         until x > 2^(l-1)                               // {2^(l-1) < x ≤ 2^l}
12:     end if
13:     return l                                            // {2^(l-1) < x ≤ 2^l}
14: end function
```

Verification

Example: ilog

$$\text{ilog}_2 : (0;\infty) \ni x \mapsto \{k \in \mathbb{Z} : 2^{k-1} < x < 2^{k+1}\} \in \mathbb{N}$$

Integer-valued logarithm using multivalued test semantics

```
 1: function ilog_2(x : REAL)
 2:     INTEGER ∋ l := 1
 3:     if x >_1 3/2 then
 4:         ℝ ∋ y := x
 5:         while y >_1 5/2 do
 6:             l := l + 1 ; y := y/2
 7:         end while
 8:     else
 9:         repeat
10:             l := l − 1
11:         until x >_{2−l} 3 ∗ 2^{l−2}
12:     end if
13:     return l
14: end function
```

Verification

Example: ilog

$$\text{ilog}_2 : (0;\infty) \ni x \mapsto \{k \in \mathbb{Z} : 2^{k-1} < x < 2^{k+1}\} \in \mathbb{N}$$

Integer-valued logarithm using multivalued test semantics

```
 1: function ilog_2(x : REAL)                          // Require: x > 0
 2:     INTEGER ∋ l := 1                               // {0 < x, l = 1}
 3:     if x >_1 3/2 then                              // {1 < x, ; l = 1}
 4:         ℝ ∋ y := x                                 // {1 = 2^(l-1) < y = y · 2^(l-1) = x}
 5:         while y >_1 5/2 do                         // {2^l < y · 2^(l-1) = x}
 6:             l := l + 1 ; y := y/2                  // {2^(l-1) < y · 2^(l-1) = x}
 7:         end while                                  // {2^(l-1) < x = y · 2^(l-1) < 2^(l+1)}
 8:     else                                           // {0 < x < 2 = 2^l}
 9:         repeat
10:             l := l − 1                             // {0 < x < 2^(l+1)}
11:         until x >_(2−l) 3 ∗ 2^(l−2)                // {2^(l-1) < x < 2^(l+1)}
12:     end if
13:     return l                                       // {2^(l-1) < x < 2^(l+1)}
14: end function
```

Verification

Example: Gaussian Elimination

Gaussian elimination: Given a $n \times m$ matrix A, returns an row echelon form.

- Gaussian elimination is not computable over `REAL` if rank(A) is not given.
- Partial pivoting is not computable over `REAL` even if rank(A) is given.
- Gaussian elimination using full pivoting with given rank of the matrix is computable. [ZiBr04]

Full pivot search using multivalued test semantics

1: **procedure** CHOOSEPIVOT(n, k : INTEGER, $B[n, n]$: REAL, pi, pj : INTEGER)
2: **var** i, j : INTEGER ; **var** s, t : REAL ; $t := 0$
Require: $B' := B[k \ldots n, k \ldots n]$ not all zero
3: **for** $i := k$ **to** n **do**
4: **for** $j := k$ **to** n **do** $t := \max\left(t, \text{abs}(B[i, j])\right)$ **end for**
5: **end for**
6: **for** $i := k$ **to** n **do**
7: **for** $j := k$ **to** n **do**
8: $s := \text{abs}(B[i, j])$;
9: **if** choose$(s > t/2, t > s) = 1$ **then** $pi := i$; $pj := j$ **end if**
10: **end for**
11: **end for**
12: **end procedure**

Verification

Example: Simple root finding

- Given continuous $f : [0; 1] \to \mathbb{R}$ with $f(0) < 0 < f(1)$ and a unique root as black box as well as $n \in \mathbb{Z}$, producing some $c \in [0; 1]$ such that $|x - c| \leq 2^n$
- bisection may fail when its testing value is the root exactly.

Bisection for approximating a root of a given continuous function with sign change

1: **function** FINDROOT($n : \mathbb{Z},\ f : \mathbb{R} \to \mathbb{R}$) // **Require:** f continuous, $f(0) \leq 0 \leq f(1)$.
2: **var** $k : \mathbb{Z}$; **var** $a, b, c, u, v, w : \mathbb{R}$
3: $a := 0$; $b := 1$ $u := f(a)$; $v := f(b)$; $k := 0$
4: **if** $\neg(0 > u \ \wedge\ v > 0)$ **then exit** **end if**
5: **while** $b - a > 2^n$ **do** $\{0 \leq a < b \leq 1,\ b - a \leq 2^k,\ u = f(a) \leq 0 \leq f(b) = v\}$
6: $k := k - 1$; $c := (a + b)/2$; $w := f(c)$
7: **if** $w > 0$ **then** $b := c$; $v := w$ **else** $a := c$; $u := w$ **end if**
8: **end while**
9: **return** c $\{\exists x \in [0; 1] : f(x) = 0 \ \wedge\ |x - c| \leq 2^n\}$
10: **end function**

Trisection for approximating unique root of given cont.function with sign change

1: **function** FINDROOT(n : INTEGER, f : REAL $\to$ REAL) // **Require:** f continuous has unique root,
2: **var** k : INTEGER ; **var** a, b, c, d, u, v, w, y : REAL // **Require:** $f(0) < 0 < f(1)$.
3: $a := 0$; $b := 1$ $u := f(a)$; $v := f(b)$; $k := 0$
4: **while** $b - a >_{n-2} 3 * 2^{n-2}$ **do**
$\{0 \le a < b \le 1,\ b - a = (3/2)^k,\ u = f(a) < 0 < f(b) = v\}$
5: $k := k - 1$; $c := b/3 + 2 * a/3$; $d := 2 * b/3 + a/3$; $w := f(c)$; $y := f(d)$
6: **if** choose($0 > u * y$, $0 > w * v$) $= 1$ **then** $b := d$; $v := y$ **else** $a := c$; $u := w$ **end if**
7: **end while**
8: **return** c $\{\exists x \in [0; 1] : f(x) = 0 \ \wedge\ |x - c| \le 2^n\}$
9: **end function**

Conclusion and Future work

Conclusion and Perspectives

- We have given a brief introduction to a computation over real numbers,
- We have given three examples of formal verification using Floyd-Hoare Logic in imperative exact but multivalued algebraic real computation:
  1. integer logarithm
  2. Gaussian Elimination
  3. root finding

Conclusion and Future work

Future work

- extend that to include the matrix diagonalization problem,
- employ proof assistants for automatized verification in an appropriate deductive system,
- investigate the computational complexity of multivalued ALGEBRAIC real verification, and explore some TRANSCENDENTAL computation, such as in [Sung Woo Choi, Sung-il Pae, Hyungju Park, Chee K. Yap 2006/2007: Decidability of Collision between a Helical Motion and an Algebraic Motion]