Implementing Logic and Real Arithmetic

Pieter Collins

Department of Knowledge Engineering

Maastricht University

pieter.collins@maastrichtuniversity.nl

Michal Konečný

Aston University

Maik Brschkens

Maastricht University

Effective Analysis: Foundations, Implementation and Certification, Marseille, 13 January 2016

Introduction

Aims

- Develop a C++ library to support numerical operations on real numbers, Euclidean functions and logical types.
  - Safe: Strong typing preventing e.g. approximate information accidentally being used in place of exact information.
  - Clean: Solid theoretical foundation and implementation.
  - Usable: Accessible to non-experts e.g. applied mathematicians
  - Efficient: Should support use of builtin double-precision floating-point, or multiple-precision floating-point libraries.
  - Broad: Handle a wide range of problems in analysis.
  - Extensible: Allow for new user-defined data-types and algorithms.
- Here, focus on logic and numbers with a view to extending to functions.
- Problems more in design decisions than mathematics...

Example code

- High-level usage working with `Real` numbers.

```
x=RealVariable("x"), v=RealVariable("v"); t=TimeVariable();
phi = flow(dot(x,v)={v,-v+sin(x)+cos(t)},init(x,v)={0,1});
xf=phi[x](t=2);
get_bounds(xf,precision=96);
check(xf<1,effort=3);
```

- Low-level usage working with `Float Bounds` for `Real` numbers.

```
RealFunction f=sin(x);
Float64Bounds pi = interval_newton.solve(f,Interval(3,4));
Real e;
Float64Bounds xkcd = pow(e,pi) - pi;
```

Motivation

- Motivated by work on ARIADNE, a tool for reachability analysis and verification of hybrid systems.
  `http://ariadne.parades.rm.cnr.it/`
- Stable version of ARIADNE has simple numerical module with classes `Rational`, symbolic `Real` numbers, `Intervals` with floating-point endpoints and `Float`.
- Above approach not sufficiently strongly typed, requiring much explicit rounding.
  - Need to distinguish `Exact` and `Approximate` `Float` objects.
  - Reserve `Interval` for geometric sets.
  - ...
- Working version fixes these problems but is overly complex.

Logic and Numbers

- Support mathematical operations on a `Real` number type $\mathbb{R}$.
  - Exact number types `Dyadic` $\mathbb{Q}_2$ and `Rational` $\mathbb{Q}$.
  - Distinguish `LowerReal` $\mathbb{R}_<$ and `UpperReal` $\mathbb{R}_>$ (useful for probabilities and metrics).

- For concrete computations use floating-point numbers.
  - Raw `Float` types $\mathbb{F}_{64}$, $\mathbb{F}_{\{}\text{MP}\}$.
  - Intermediate answers given by `FloatBounds` $[\mathbb{F},\mathbb{F}]$ or `FloatBall` $\mathbb{F}\pm\mathbb{F}$.

- To support comparisons, need `Kleenean` logical type with values $\mathbb{K}=\{\mathsf{T},\mathsf{F},\bot\}\equiv\mathbb{B}_\bot$.
  - Exact `Boolean` supertype $\mathbb{B}=\{\mathsf{T},\mathsf{F}\}$ for decidable predicates.
  - `Sierpinskian` subtype $\mathbb{S}=\{\mathsf{T},\bot\}\equiv\mathbb{K}_>$ for verifyable predicates.

Kinds of Information

- Qualitatively different kinds of information:
- `Abstract`: Exact symbolic information, without computational algorithm:
  - e.g. $\cos(1)$ is a real number (which may be computed in many ways).
  - Useful for problem specification.
- `Effective`: Exact information, with algorithm to compute arbitrarily accurately:
  - e.g. $\cos(1)$ is given by $\sum_{n=0}^{N-1}(-1)^n/(2n)! \pm 1/(2N)!$.
  - Main type from computable analysis.
- `Validated`: Finite precision information with bounds
  - e.g. $\cos(1)$ is in ball $4357/8064 \pm 1/3628800$.
  - Useful in concrete rigorous numerics.
- `Approximate`: No information about accuracy
  - e.g. $\cos(1)$ is roughly 0.541.
  - Useful for scratch computations and preconditioning.
- All levels should support the operations of the mathematical type.

Generic, concrete and numeric data

- Countable sets can be represented by *concrete* data types
  - Natural mathematical types, such as `Rational`.
  - Efficient computational types, such as `Floats`.

- Polymorphic C++ classes can represent data in arbitrary ways.
  - Specified by supported operations

    ```
    EffectiveReal::get(Accuracy) -> ValidatedReal.
    ```
  - Uncountable types, such as `EffectiveReal` and `ValidatedReal`.

Numerical types

- Provide raw double-precision floating-point numbers $\mathbb{F}_{64}$ as class Float64, and multiple-precision numbers $\mathbb{F}_{\mathrm{MP}}$ as class FloatMP.
  - To create a $\mathbb{F}_{64}$ or $\mathbb{F}_{\mathrm{MP}}$ from a $\mathbb{R}$ need a *rounding* mode.
  - To create a $\mathbb{F}_{\mathrm{MP}}$ from a $\mathbb{R}$ also need a *precision*.
  - For uniform creation, use a Precision64 tag to create $\mathbb{F}_{64}$.

- Concrete validated classes
  - Ball<FV,FE> and Bounds<FL,FU> for Real.
    - Ball<FV,FE> corresponds to a Cauchy real,
    - Bounds<FL,FU> to a Dedekind real.
  - LowerBound<FL> for LowerReal and UpperBound<FU> for UpperReal.

- Concrete approximate class Approximation<FA> for $\widetilde{\mathbb{R}}$.

- Concrete exact classes Exact<F>.

Design Issues

What is a real number?

- A real number *is*
  - an *equivalence class* of
    - signed-digit (binary) expansions $(z_n \pm 1)/2^n$, or
    - strongly convergent Cauchy sequences of rationals $q_n \pm 1/2^n$, or
    - convergent sequences of nested intervals with dyadic endpoints $[\underline{p}_n, \overline{p}_n]$.
    - $\cdots$

- In applications, usually *specified* by a formula
  - e.g. $x = \sin(\pi/3)$ or $x = \sqrt{3}/2$.
  - It is unknown whether equality of real numbers specified by elementary functions is decidable.

- Agnostically try to define a real number class independently of any one representation.

Polymorphic type interfaces

- Many equivalent definitions of a real number; our code should allow any way of specifying.
- To be *usable*, need *standard* ways to *extract* information.
- Many possible ways to extract bounds from a real number:

```
EffectiveReal::get_bounds(Precision prec) -> Bounds<Float,Float>;
EffectiveReal::get_bounds(Effort eff) -> Bounds<Dyadic,Dyadic>;
EffectiveReal::get_ball(Accuracy acc) -> Ball<Rational,Dyadic>;
EffectiveReal::get(Accuracy acc) -> ValidatedReal;
```

  Which should we use?
- To ensure safety, all approximations should specify their error.
  - Allowing a `Real` to define an fast-converging Cauchy sequence interface

    ```
    Real::get_within(Accuracy acc) -> Rational;
    ```

    is therefore *not* allowed!

Need for Abstract information

- `Abstract` information very similar to `Effective` information.
- `Abstract` information useless without specification of an algorithm.
- Maybe we can simplify framework by eliminating `AbstractReal`...
- Distinguishing `Abstract` and `Effective` allows user specification of algorithms.
- For some operations e.g. `add`, `cos`, *may* be uncontroversial default choice.
  - Unsophisticated users should invisibilly be given a default choice.
- For complex operations (usually functional operators like `flow`), a good algorithm may be problem dependent.
  - In ARIADNE function calculus, use *evaluator* classes e.g. `Flower` for differential equations.

Uninformative reals and approximations

- A `ValidatedReal` $\widehat{\mathbb{R}}$ is a (rational) ball $\check{x} \pm e_x$ or bounds (interval) $[\underline{x}, \overline{x}]$.
- A `ValidatedLowerReal` $\widehat{\mathbb{R}}_<$ is a (rational) lower bound $\underline{x}$, and a `ValidatedUpperReal` $\widehat{\mathbb{R}}_>$ is a (rational) upper bound $\overline{x}$.
- An `ApproximateReal` $\widetilde{\mathbb{R}}$ is a (rational) approximation $\tilde{x}$, semantically the same as `ApproximateLowerReal` and `ApproximateUpperReal`.
  - Should we consider these as the same type??

- General convergent sequences define `UninformativeReal` type $\mathbb{R}_?$.
  - No information about limit can be deduced from any finite subsequence.
  - Related `ValidatedUninformativeReal` type is canonically a `ApproximateReal`!

- Maybe this resolves the undesirable additional approximate real types...

Conversions

- It should be possible to convert to a mathematical subtype with weaker information.
  - An `EffectiveReal` should be usable whenever a `ValidatedUpperReal` is required.

- Conversions need to occur in various situations:
  - When explicitly required by the user.
  - To implicitly downcast arguments to a variable.
  - To implicitly downcast arguments to a (binary) operator.

- Especially important for binary operations e.g. $\widehat{\mathbb{R}} + \mathbb{R}_{<} \to \widehat{\mathbb{R}}_{<}$
  `ValidatedReal + EffectiveLowerReal -> ValidatedLowerReal`.

- Conversion to a *mathematical* subtype e.g. $\mathbb{R} \to \mathbb{R}_{<}$ or $\widehat{\mathbb{R}} \to \widehat{\mathbb{R}}_{<}$ is straightforward.

- There are many ways of moving to weaker information...

Conversions losing information

- `Abstract` to `Effective` conversion requires an *algorithm*.
  - e.g. `EffectiveReal q=sin(2)` requires an algorithm for computing $\sin$.
  - Usually want to provide sensible defaults. This requires a (semi-)global computation environment...

- `Effective` to `Validated` conversion needs some way of determining the accuracy of calculation...
  - Explicit conversions can use `Accuracy`, `Effort` or working `Precision`.

- `Validated` to `Approximate` conversions straightforward:

$$\hat{x} = [\underline{x}, \overline{x}] \mapsto (\underline{x} + \overline{x})/2 = \tilde{x}$$

Implicit conversions in binary operations

- `Effective` to `Validated` tricky for binary operation:
  `add(EffectiveReal r1, ValidatedReal r2) -> ValidatedReal`
- Could use the *accuracy* of the `Validated` argument to give accuracy to extract `Effective` argument.
  - But this doesn't work with non-metric types `LowerReal`
    `add(EffectiveLowerReal, ValidatedLowerReal) -> ValidatedLowerReal`
    since we don't have an accuracy.
- Could use a *precision* of a `Validated` numerical argument to give precision to extract `Effective` argument.
  `add(EffectiveReal, FloatMPBounds) -> FloatMPBounds`
  - But this strategy doesn't work with logical types
    `and(EffectiveKleenean, ValidatedKleenean) -> ValidatedKleenean`
    since we don't have a precision.
- Could use the *effort* used to compute the `Validated` argument.
  - But this means every `Validated` object needs to carry around its own `Effort`.

Generic code

- Aim to implement similar types and operations using generic code
  - Analytic functions on any Banach space e.g. $\mathbb{R}$, $C^1(\mathbb{R}^n \to \mathbb{R})$.
  - Real numbers and continuous functions as completions.
  - Bounds and balls using arbitrary (floating-point) types and arbitrary ordered and metric spaces.

- Real numbers are the completion of $\mathbb{Q}_2$ and $\mathbb{Q}$! Which to use to *define* $\mathbb{R}$?
  - Maybe better to define $\mathbb{R}$ axiomatically, and then say the completion of $\mathbb{Q}_2$ and $\mathbb{Q}$ is $\mathbb{R}$.

Implementation Issues

C++: Language issues

- Conversions to subtypes uses different semantics to conversion constructors/operators.
  - Using subtyping, the inheritance hierarchy is transversed.
  - Using operators, only the number of arguments which must be converted is considered; *many* more ambiguities.
- Template functions have different conversion rules to non-template functions.
- Polymorphic types require references or pointers (memory leaks; different syntax).
- Efficiency concerns mean concrete types cannot be part of class hierarchy; require conversions to generic types.
- Compile times quickly become very long...

C++: Language issues

- Conversions to subtypes uses different semantics to conversion constructors/operators.
  - Using subtyping, the inheritance hierarchy is transversed.
  - Using operators, only the number of arguments which must be converted is considered; *many* more ambiguities.
- Template functions have different conversion rules to non-template functions.
- Polymorphic types require references or pointers (memory leaks; different syntax).
- Efficiency concerns mean concrete types cannot be part of class hierarchy; require conversions to generic types.
- Compile times quickly become very long...
- Why not use Haskell?

Haskell: Language Issues

- No subtypes (except at type class level) limits polymorphism.
- Refactoring type classes requires changing entire code.
- Use Strathclyde Haskell Extension (SHE) to allow subclasses to provide defaults:

```
class (CanNeg a, a ~ NegType a, CanAdd a a, a ~ AddType a a,
        CanMul a a, a ~ MulType a a) => Ring a where
    instance CanNeg a where
        type NegType a=a; neg ::  a -> a
    instance CanAdd a a where
        type AddType a a=a; add ::  a -> a -> a
    instance CanSub a a where
        type SubType a a=a; sub ::  a -> a -> a;
        sub x y = add x (neg y)
    instance CanMul a a where
        type MulType a a=a; mul ::  a -> a -> a
```

- SHE is a rather buggy preprocessor...

Proposal

Proposals

- Provide logical types $\mathbb{B}$, $\mathbb{K}$, $\mathbb{S} \equiv \mathbb{K}_<$
- Provide numerical types
  - exact $\mathbb{N} = \mathbb{Z}^+$, $\mathbb{Z}$, $\mathbb{Q}_2$, $\mathbb{Q}$,
  - generic $\mathbb{R}$, $\mathbb{R}_<$, $\mathbb{R}_>$, and positive versions.
  - concrete raw numerical types $\mathbb{F}_{64}, \mathbb{F}_{\mathsf{MP}}$ and bounds.
- Distinguish between
  - `Abstract` symbolic formulae for specification,
  - `Effective` arbitrarily accurate descriptions,
  - `Validated` numerical bounds, and
  - `Approximate` scratch values.
- Allow implicit conversion to weaker types, with appropriate defaults.
- Allow mixed operations with appropriate conversions.

Questions

- Should we provide an `UniformativeReal` with a `ApproximateReal` being a `ValidatedUninformativeReal`?
- Does it make sense to provide a `ApproximateLowerReal` and `ApproximateUpperReal`?
- Do we really need to distinguish `Abstract` and `Effective` information?
- Which operations should we supply to extract information from `Real` numbers?

- How to implement as generically as possible??
- How to specify additional data required for conversions???