Interval Analysis Without Intervals

Paul Taylor

Department of Computer Science
University of Manchester
UK EPSRC GR/S58522

Real Numbers and Computers 7
Nancy, Monday, 10 July 2006

www.cs.man.ac.uk/~pt/ASD

A theorist amongst programmers

I am offering you

- a logic that is complete for computably continuous functions $\mathbb{R}^{\mathbf{n}} \to \mathbb{R}$
- and some vague ideas for programming with it.

A theorist amongst programmers

I am offering you

- a logic that is complete for computably continuous functions $\mathbb{R}^n \to \mathbb{R}$
- and some vague ideas for programming with it.

I want you to tell me

- how you could use my ideas to extend your exact real arithmetic systems,
- what other theoretical issues (such as backtracking) emerge from your programming,
- and can you implement my language?

Where am I coming from?

Where am I coming from?

Category theory.

Category theory is a distillation of decades of mathematical experience into a form in which it can be used in other subjects (algebraic topology, logic, computer science, physics...).

Used skillfully, it can often tell us how to do mathematics, though not necessarily why.

Where am I coming from?

Category theory.

Category theory is a distillation of decades of mathematical experience into a form in which it can be used in other subjects (algebraic topology, logic, computer science, physics...).

Used skillfully, it can often tell us how to do mathematics, though not necessarily why.

But it is a strong drug — it becomes more effective when it is diluted.

To cut a long story short

I had a "what if" idea from category theory in 1993.

(It's called Abstract Stone Duality.)

To cut a long story short

I had a "what if" idea from category theory in 1993.

(It's called Abstract Stone Duality.)

I have been diluting it ever since.

It gives a new account of
computably based locally compact spaces.

To cut a long story short

I had a "what if" idea from category theory in 1993.

(It's called Abstract Stone Duality.)

I have been diluting it ever since.

It gives a new account of
computably based locally compact spaces.

In 2004 (with Andrej Bauer) I began to apply it to the real line.

To cut a long story short

I had a "what if" idea from category theory in 1993.

(It's called Abstract Stone Duality.)

I have been diluting it ever since.

It gives a new account of
computably based locally compact spaces.

In 2004 (with Andrej Bauer) I began to apply it to the real line.

It worked very nicely.

Keeping to the original idea, it says that
the real line is Dedekind complete (NB!)
and has the Heine–Borel property ($[0, 1]$ is compact).

The language that I shall discuss today
is the fragment of the main ASD calculus for the type $\mathbb{R}$.

I have been impressed by

Intellectual diversity — many different skills applied to $\mathbb{R}$.

Theoretical issues that emerge from programming —
e.g. when and how to back-track to improve precision.

The logical content of crude arithmetic —
e.g. the Interval Newton algorithm.

I am not impressed by

Timing benchmarks.

Excessive attention to representations of real numbers.
Heavy dependency on dyadic rationals or Cauchy sequences.

Theory without insight.
Naïve and dogmatic application of naïve set theory.
This applies especially to the “theoretical foundations” of Interval Analysis.

What's in it for you?

A theoretical framework
on which to structure your programming.

Not just exact real arithmetic, but also analysis.

How to generalise interval computations
to $\mathbb{R}^n$, $\mathbb{C}$ and other (locally compact) spaces from geometry.

All functions are continuous and computable

This is not a Theorem (*à la* Brouwer) but a design principle.

The language only introduces continuous computable functions.

All functions are continuous and computable

This is not a Theorem (*à la* Brouwer) but a design principle.

The language only introduces continuous computable functions.

For $\mathbb{R}$, we understand "continuity" in the familiar ϵ–δ sense of Weierstrass.

Therefore, step functions, *etc.* are not definable as functions $\mathbb{R} \to \mathbb{R}$.

All functions are continuous and computable

This is not a Theorem (*à la* Brouwer) but a design principle.

The language only introduces continuous computable functions.

For $\mathbb{R}$, we understand "continuity" in the familiar ϵ–δ sense of Weierstrass.

Therefore, step functions, *etc.* are not definable as functions $\mathbb{R} \to \mathbb{R}$.

The full language of Abstract Stone Duality (currently) describes all (not necessarily Hausdorff) locally compact spaces.

All functions are continuous and computable

This is not a Theorem (*à la* Brouwer) but a design principle.

The language only introduces continuous computable functions.

For $\mathbb{R}$, we understand "continuity" in the familiar ϵ–δ sense of Weierstrass.

Therefore, step functions, *etc.* are not definable as functions $\mathbb{R} \to \mathbb{R}$.

The full language of Abstract Stone Duality (currently) describes all (not necessarily Hausdorff) locally compact spaces.

Step functions and lots of other things are definable as functions to other spaces besides $\mathbb{R}$, such as the interval domain.

A very important non-Hausdorff space

Besides $\mathbb{R}$ and $\mathbb{N}$, we also use the Sierpiński space Σ.

Topologically, Σ looks like $\begin{pmatrix}\odot \\ \bullet\end{pmatrix}$.

A very important non-Hausdorff space

Besides $\mathbb{R}$ and $\mathbb{N}$, we also use the Sierpiński space Σ.

Topologically, Σ looks like $\begin{pmatrix} \odot \\ \bullet \end{pmatrix}$.

In programming languages, Σ is called `void` or `unit`.

A very important non-Hausdorff space

Besides $\mathbb{R}$ and $\mathbb{N}$, we also use the Sierpiński space Σ.

Topologically, Σ looks like $\left(\begin{smallmatrix}\odot\\ \bullet\end{smallmatrix}\right)$.

In programming languages, Σ is called `void` or `unit`.

ASD exploits the analogy amongst

- (continuous) functions $X \to \Sigma$,
- programs $X \to \Sigma$,
- open subspaces $U \subset X$,
- recursively enumerable subspaces $U \subset X$,
- and observable properties of $x \in X$.

In fact, it makes this correspondence exact.

A very important non-Hausdorff space

Besides $\mathbb{R}$ and $\mathbb{N}$, we also use the Sierpiński space Σ.

Topologically, Σ looks like $\begin{pmatrix}\odot\\ \bullet\end{pmatrix}$.

In programming languages, Σ is called `void` or `unit`.

ASD exploits the analogy amongst

- (continuous) functions $X \to \Sigma$,
- programs $X \to \Sigma$,
- open subspaces $U \subset X$,
- recursively enumerable subspaces $U \subset X$,
- and observable properties of $x \in X$.

In fact, it makes this correspondence exact.

In particular, the exponential $X \to \Sigma$ is the topology on X.

It is a lattice that is itself equipped with the Scott topology (which is also used in domain theory).

A very important non-Hausdorff space

Besides $\mathbb{R}$ and $\mathbb{N}$, we also use the Sierpiński space Σ.

Topologically, Σ looks like $\begin{pmatrix}\odot \\ \bullet\end{pmatrix}$.

In programming languages, Σ is called `void` or `unit`.

ASD exploits the analogy amongst

- (continuous) functions $X \to \Sigma$,
- programs $X \to \Sigma$,
- open subspaces $U \subset X$,
- recursively enumerable subspaces $U \subset X$,
- and observable properties of $x \in X$.

In fact, it makes this correspondence exact.

In particular, the exponential $X \to \Sigma$ is the topology on X.

It is a lattice that is itself equipped with the Scott topology (which is also used in domain theory).

Similar methods have been used in compiler design, where $X \to \Sigma$ is the type of continuations from X.

Observable arithmetic relations

In particular, functions $\mathbb{R} \times \mathbb{R} \to \Sigma$ correspond to open binary relations.

Observable arithmetic relations

In particular, functions $\mathbb{R} \times \mathbb{R} \to \Sigma$ correspond to open binary relations.

Hence $a < b$, $a > b$ and $a \neq b$ are definable,

but $a \leqslant b$, $a \geqslant b$ and $a = b$ are not definable.

Observable arithmetic relations

In particular, functions $\mathbb{R} \times \mathbb{R} \to \Sigma$ correspond to open binary relations.

Hence $a < b$, $a > b$ and $a \neq b$ are definable,

but $a \leqslant b$, $a \geqslant b$ and $a = b$ are not definable.

This agrees with programming experience (even in classical numerical analysis).

Observable arithmetic relations

In particular, functions $\mathbb{R} \times \mathbb{R} \to \Sigma$
correspond to open binary relations.

Hence $a < b$, $a > b$ and $a \neq b$ are definable,

but $a \leqslant b$, $a \geqslant b$ and $a = b$ are not definable.

This agrees with programming experience
(even in classical numerical analysis).

Topologically, it is because $\mathbb{R}$ is Hausdorff but not discrete.

On the other hand $\mathbb{N}$ and $\mathbb{Q}$ are discrete and Hausdorff,
so we have all six relations for them.

The logic of observable properties

A term $\sigma : \Sigma$ is called a proposition.

A term $\phi : \Sigma^X$ is called a predicate.
Recall that it represents an open subspace or observable predicate.

The logic of observable properties

A term $\sigma : \Sigma$ is called a proposition.

A term $\phi : \Sigma^X$ is called a predicate.
Recall that it represents an open subspace or observable predicate.

We can form $\phi \wedge \psi$ and $\phi \vee \psi$,
by running programs in series or parallel.

The logic of observable properties

A term $\sigma : \Sigma$ is called a proposition.

A term $\phi : \Sigma^X$ is called a predicate.
Recall that it represents an open subspace or observable predicate.

We can form $\phi \wedge \psi$ and $\phi \vee \psi$,
by running programs in series or parallel.

Also $\exists n : \mathbb{N}.\ \phi x$, $\exists q : \mathbb{Q}.\ \phi x$, $\exists x : \mathbb{R}.\ \phi x$ and $\exists x : [0,1].\ \phi x$.

(But not $\exists x : X.\ \phi x$ for arbitrary X — it must be overt.)

The logic of observable properties

A term $\sigma : \Sigma$ is called a proposition.

A term $\phi : \Sigma^X$ is called a predicate.
Recall that it represents an open subspace or observable predicate.

We can form $\phi \wedge \psi$ and $\phi \vee \psi$,
by running programs in series or parallel.

Also $\exists n : \mathbb{N}.\ \phi x$, $\exists q : \mathbb{Q}.\ \phi x$, $\exists x : \mathbb{R}.\ \phi x$ and $\exists x : [0,1].\ \phi x$.

(But not $\exists x : X.\ \phi x$ for arbitrary X — it must be overt.)

Negation and implication are not allowed.

Because:

- this is the logic of open subspaces;
- the function $\odot \leftrightarrows \bullet$ on $\binom{\odot}{\bullet}$ is not continuous;
- the Halting Problem is not solvable.

Universal quantification

When $K \subset X$ is compact (*e.g.* $[0,1] \subset \mathbb{R}$), we can form $\forall x\colon K.\ \phi x$.

$$\frac{\ldots,\ x : K \ \vdash \ \phi x}{\ldots \ \vdash \ \forall x\colon K.\ \phi x}$$

Universal quantification

When $K \subset X$ is compact (*e.g.* $[0,1] \subset \mathbb{R}$), we can form $\forall x : K.\ \phi x$.

$$\frac{\ldots,\ x : K \ \vdash \ \phi x}{\ldots \ \vdash \ \forall x : K.\ \phi x}$$

The quantifier is a (higher-type) function $\forall_K : \Sigma^K \to \Sigma$.

Like everything else, it's Scott continuous.

Universal quantification

When $K \subset X$ is compact (*e.g.* $[0,1] \subset \mathbb{R}$), we can form $\forall x : K.\ \phi x$.

$$\frac{\ldots,\ x : K \ \vdash \ \phi x}{\ldots \ \vdash \ \forall x : K.\ \phi x}$$

The quantifier is a (higher-type) function $\forall_K : \Sigma^K \to \Sigma$.

Like everything else, it's Scott continuous.

The useful cases of this in real analysis are

$$\begin{array}{lcl} \forall x : K. \exists \delta > 0. \phi(x, \delta) & \Leftrightarrow & \exists \delta > 0. \forall x : K. \phi(x, \delta) \\ \forall x : K. \exists n. \phi(x, n) & \Leftrightarrow & \exists n. \forall x : K. \phi(x, n) \end{array}$$

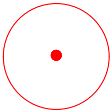
in the case where $(\delta_1 < \delta_2) \quad \wedge \quad \phi(x, \delta_2) \ \Rightarrow \ \phi(x, \delta_1)$

or $(n_1 > n_2) \quad \wedge \quad \phi(x, n_2) \ \Rightarrow \ \phi(x, n_1)$.

Recall that uniform convergence, continuity, *etc.*
involve commuting quantifiers like this.

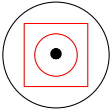
Local compactness

Wherever a point a lies in the open subspace represented by ϕ, so ϕa in my logical notation,



Local compactness

Wherever a point a lies in the open subspace represented by ϕ, so ϕa in my logical notation,

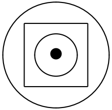


there are a compact subspace K and an open one representing β such that a is in the open set, *i.e.* βa and the open set is contained in the compact one, $\forall x \in K.\ \beta x$.

Altogether, $\phi a \iff \beta a \wedge \forall x \in K.\ \beta x$.

Local compactness

Wherever a point a lies in the open subspace represented by ϕ, so ϕa in my logical notation,



there are a compact subspace K and an open one representing β such that a is in the open set, *i.e.* βa and the open set is contained in the compact one, $\forall x \in K.\ \beta x$.

Altogether, $\phi a \iff \beta a \wedge \forall x \in K.\ \beta x$.

In fact β and K come from a basis that is encoded in some way.

For example, for $\mathbb{R}$, β and K may be the open and closed intervals with dyadic rational endpoints p, q.

Then $\phi a \iff \exists p, q : \mathbb{Q}.\ a \in (p, q) \wedge \forall x \in [p, q].\ \phi x$.

Alternatively, $\phi a \iff \exists \delta > 0.\ \forall x \in [a \pm \delta].\ \phi x$.

Examples: continuity and uniform continuity

Theorem: Every definable function $f : \mathbb{R} \to \mathbb{R}$ is continuous:

$$\epsilon > 0 \;\Rightarrow\; \exists\delta > 0.\, \forall y : [x \pm \delta].\, \left(\left|fy - fx\right| < \epsilon\right)$$

Proof: Put $\phi_{x,\epsilon} y \equiv \left(\left|fy - fx\right| < \epsilon\right)$, with parameters $x, \epsilon : \mathbb{R}$.

Theorem: Every function f is uniformly continuous on any compact subspace $K \subset \mathbb{R}$:

$$\epsilon > 0 \;\Rightarrow\; \exists\delta > 0.\, \forall x : K.\, \forall y : [x \pm \delta].\, \left(\left|fy - fx\right| < \epsilon\right)$$

Proof: $\exists\delta > 0$ and $\forall x : K$ commute.

Dedekind completeness

A real number a is specified
by saying whether (real or rational) numbers d, u
are bounds for it: $d < a < u$.

Historically first example: Archimedes calculated π (the area of a circle)
using regular $3 \cdot 2^n$-gons inside and outside it.

Dedekind completeness

A real number a is specified
by saying whether (real or rational) numbers d, u
are bounds for it: $d < a < u$.

Historically first example: Archimedes calculated π (the area of a circle)
using regular $3 \cdot 2^n$-gons inside and outside it.

The question whether d is a lower bound
is an observable predicate, so is expressed in our language.

These two predicates define a Dedekind cut — they have to satisfy certain axioms.

Dedekind completeness

A real number a is specified
by saying whether (real or rational) numbers d, u
are bounds for it: $d < a < u$.

Historically first example: Archimedes calculated π (the area of a circle)
using regular $3 \cdot 2^n$-gons inside and outside it.

The question whether d is a lower bound
is an observable predicate, so is expressed in our language.

These two predicates define a Dedekind cut — they have to satisfy certain axioms.

In practice, most of these axioms are easy to verify.
The one that isn't is called locatedness:
there are some bounds d, u that are arbitrarily close together.

Dedekind completeness

A real number a is specified
by saying whether (real or rational) numbers d, u
are bounds for it: $d < a < u$.

Historically first example: Archimedes calculated π (the area of a circle)
using regular $3 \cdot 2^n$-gons inside and outside it.

The question whether d is a lower bound
is an observable predicate, so is expressed in our language.

These two predicates define a Dedekind cut — they have to satisfy certain axioms.

In practice, most of these axioms are easy to verify.
The one that isn't is called locatedness:
there are some bounds d, u that are arbitrarily close together.

Pseudo-cuts that are not (necessarily) located are called intervals.

A lambda-calculus for Dedekind cuts

Our formulation of Dedekind cuts does not use set theory, or type-theoretic predicates of arbitrary logical strength.
It's based on a simple adaptation of λ-calculus and proof theory.

A lambda-calculus for Dedekind cuts

Our formulation of Dedekind cuts does not use set theory, or type-theoretic predicates of arbitrary logical strength.
It's based on a simple adaptation of λ-calculus and proof theory.

Given any pair $[\delta, \upsilon]$ of predicates for which the axioms of a Dedekind cut are provable,
we may introduce a real number:

$$\dfrac{\begin{matrix}[d : \mathbb{R}] \\ \vdots \\ \delta d : \Sigma\end{matrix} \qquad \begin{matrix}[u : \mathbb{R}] \\ \vdots \\ \upsilon u : \Sigma\end{matrix} \qquad \text{axioms for Dedekind cut}}{(\mathsf{cut}\, du.\, \delta d \wedge \upsilon u) : \mathbb{R}}$$

A λ-calculus for Dedekind cuts

The elimination rules recover the axioms.

The β-rule says that $(\mathsf{cut}\, du.\, \delta d \wedge \upsilon u)$ obeys the order relations that δ and υ specify:

$$e < (\mathsf{cut}\, du.\, \delta d \wedge \upsilon u) < t \qquad \Longleftrightarrow \qquad \delta e \wedge \upsilon t.$$

As in the λ-calculus, this simply substitutes part of the context for the bound variables.

The η-rule says that any real number a defines a Dedekind cut in the obvious way:

$$\delta d \;\equiv\; (d < a), \qquad \text{and} \qquad \upsilon u \;\equiv\; (a < u).$$

Summary of the syntax

		$\mathbb{N}$	$\mathbb{R}$	$\mathbb{N}\&\Sigma$	$\mathbb{R}\&\Sigma$	$\mathbb{N}\&?$	Σ
$\mathbb{N}$	0	succ				rec	the
$\mathbb{R}$	$0, 1$	n	$+, -, \times, \div$			rec	cut
Σ	$\top, \bot$	$=, \leqslant, \geqslant$ $<, >, \neq$	$<, >, \neq$	$\exists n$	$\exists x : \mathbb{R}$ $\forall x : [a, b]$	rec	$\wedge, \vee$

the: definition by description.
cut: Dedekind completeness.

A valuable exercise

Make a habit of trying to formulate statements in analysis according to (the restrictions of) the ASD language.

This may be easy — it may not be possible

The exercise of doing so may be 95% of solving your problem!

Real numbers and representable intervals

The language that we have described

- has continuous variables and terms

$$a, \quad b, \quad c, \quad x, \quad y, \quad z \qquad \text{(in \textit{italic})}$$

  that denote real numbers, or maybe vectors,
- about which we reason using pure mathematics, using ideas of real analysis.

Real numbers and representable intervals

The language that we have described

- has continuous variables and terms

$$a, \quad b, \quad c, \quad x, \quad y, \quad z \qquad \text{(in \textit{italic})}$$

  that denote real numbers, or maybe vectors,
- about which we reason using pure mathematics, using ideas of real analysis.

We need another language

- with discrete variables and terms

$$\mathsf{a}, \quad \mathsf{b}, \quad \mathsf{c}, \quad \mathsf{x}, \quad \mathsf{y}, \quad \mathsf{z} \qquad \text{(in \textsf{sans serif})}$$

  that denote machine-representable intervals or cells,
- with which we compute directly.

Cells for locally compact spaces

For computation on the real line, the interval $\mathbf{x}$ has machine representable endpoints $\underline{\mathbf{x}} \equiv d$ and $\overline{\mathbf{x}} \equiv u$.

Cells for locally compact spaces

For computation on the real line, the interval $\mathbf{x}$ has machine representable endpoints $\underline{\mathbf{x}} \equiv d$ and $\overline{\mathbf{x}} \equiv u$.

For $\mathbb{R}^{\mathbf{n}}$ the cells need not be cubes.

The theory of locally compact spaces tells us what to do.

Cells for locally compact spaces

For computation on the real line, the interval $\mathbf{x}$ has machine representable endpoints $\underline{\mathbf{x}} \equiv d$ and $\overline{\mathbf{x}} \equiv u$.

For $\mathbb{R}^{\mathbf{n}}$ the cells need not be cubes.

The theory of locally compact spaces tells us what to do.

A basis for a locally compact space is a family of cells.

A cell $\mathbf{x}$ is a pair $U \subset K$ of spaces
with $(\mathbf{x}) \equiv U$ open and $[\mathbf{x}] \equiv K$ compact.
For example, $U \equiv (p, q)$ and $K \equiv [p, q]$ in $\mathbb{R}^{\mathbf{1}}$.

The cell $\mathbf{x}$ is encoded in some machine-representable way.
For example, p and q are dyadic rationals.

Theory and practice

You already know how to program interval arithmetic.

The theory tells how to structure its generalisations.

Theory and practice

You already know how to program interval arithmetic.

The theory tells how to structure its generalisations.

Suppose that you want to generalise interval computations to $\mathbb{R}^{\mathbf{2}}$, $\mathbb{R}^{\mathbf{n}}$, $\mathbb{C}$, the sphere S^2 or some other space.

Its natural cells may be respectively
hexagons, close-packed spheres or circular discs.

The geometry and computation of sphere packing in many dimensions is well known amongst group theorists.

Theory and practice

The theory of locally compact spaces tells us what we need to know about the system of cells:

- How are arbitrary open subspaces expressed as unions of basic ones?
- When is the compact subspace $[\mathbf{x}]$ of one cell contained in the open subspace $(\mathbf{y})$ of another?
  We write $\mathbf{x} \Subset \mathbf{y}$ for this observable relation.
- How are any finite intersections of basic compact subspaces
  covered by finite unions of basic open subspaces?

I could give formal axioms, but geometric intuition is enough.

Theory and practice

The theory of locally compact spaces tells us what we need to know about the system of cells:

- How are arbitrary open subspaces expressed as unions of basic ones?
- When is the compact subspace $[\mathbf{x}]$ of one cell contained in the open subspace $(\mathbf{y})$ of another?
  We write $\mathbf{x} \Subset \mathbf{y}$ for this observable relation.
- How are any finite intersections of basic compact subspaces
  covered by finite unions of basic open subspaces?

I could give formal axioms, but geometric intuition is enough.

From the theory we derive a plan for the programming:

- how are (finite unions of) cells to be represented?
- how are the arithmetic operations and relations to be computed?
- how are finite intersections covered by finite unions?

Logic for the representation of cells

Cells are ultimately represented in the machine as integers.

These are finite but arbitrarily large.

In their logic, there is $\exists$ but not $\forall$.

Logic for the representation of cells

Cells are ultimately represented in the machine as integers.

These are finite but arbitrarily large.

In their logic, there is $\exists$ but not $\forall$.

$\exists \mathbf{x}$ in principle involves a search over all possible representations of intervals.

In applications to analysis (*e.g.* solving differential equations), $\exists$ may range over structures such as grids of sample points.

In practice, we find witnesses for $\exists$ by logic programming techniques such as unification.

Logic for the representation of cells

Cells are ultimately represented in the machine as integers. These are finite but arbitrarily large.

In their logic, there is $\exists$ but not $\forall$.

$\exists \mathbf{x}$ in principle involves a search over all possible representations of intervals.

In applications to analysis (*e.g.* solving differential equations), $\exists$ may range over structures such as grids of sample points.

In practice, we find witnesses for $\exists$ by logic programming techniques such as unification.

Programming $\forall x \in [a, b]$ is based on the Heine–Borel theorem.

Some deliberately ambiguous notation

$$x \in \mathbf{a} \qquad \text{means} \qquad x \in (\mathbf{x}) \qquad \text{or} \qquad \underline{\mathbf{x}} < x < \overline{\mathbf{x}}.$$

$$\forall x \in \mathbf{x} \qquad \text{means} \qquad \forall x \in [\mathbf{x}] \qquad \text{or} \qquad \forall x \in [\underline{\mathbf{x}}, \overline{\mathbf{x}}].$$

$$\exists x \in \mathbf{x} \qquad \text{means} \qquad \text{both } \exists x \in (\mathbf{x}) \text{ and } \exists x \in [\mathbf{x}]$$

because these are equivalent, so long as $\mathbf{x}$ is not empty, so $\underline{\mathbf{x}} < \overline{\mathbf{x}}$.

Cells and data flow

The topological duality between compact and open subspaces has a computational meaning.

Cells and data flow

The topological duality between compact and open subspaces has a computational meaning.

Think of $\mathbf{a} \Subset \mathbf{b}$ (which means $[\mathbf{a}] \subset (\mathbf{b})$) as a plug in a socket.

Cells and data flow

The topological duality between compact and open subspaces has a computational meaning.

Think of $\mathbf{a} \Subset \mathbf{b}$ (which means $[\mathbf{a}] \subset (\mathbf{b})$) as a plug in a socket.

The plug or value may be a real number a,
or a compact subspace $[\mathbf{a}]$.

Cells and data flow

The topological duality between compact and open subspaces has a computational meaning.

Think of $\mathbf{a} \Subset \mathbf{b}$ (which means $[\mathbf{a}] \subset (\mathbf{b})$) as a plug in a socket.

The plug or value may be a real number a,
or a compact subspace $[\mathbf{a}]$.

The socket or test may be an open subspace $(\mathbf{b})$,
or a universal quantifier $\forall x \in (-).\ \phi x$.

Cells and data flow

The topological duality between compact and open subspaces has a computational meaning.

Think of $\mathbf{a} \Subset \mathbf{b}$ (which means $[\mathbf{a}] \subset (\mathbf{b})$) as a plug in a socket.

The plug or value may be a real number a,
or a compact subspace $[\mathbf{a}]$.

The socket or test may be an open subspace $(\mathbf{b})$,
or a universal quantifier $\forall x \in (-).\ \phi x$.

These define a natural direction

$$\underset{\longrightarrow}{a \;\in\; \mathbf{b}} \quad \text{and} \quad \underset{\longrightarrow}{\mathbf{a} \;\Subset\; \mathbf{b}} \quad \text{but} \quad \underset{\longleftarrow}{\forall x \in \;\mathbf{a}}$$

which also goes up arithmetic expression trees,
from arguments to results.

Cells and data flow

The topological duality between compact and open subspaces has a computational meaning.

Think of $\mathbf{a} \Subset \mathbf{b}$ (which means $[\mathbf{a}] \subset (\mathbf{b})$) as a plug in a socket.

The plug or value may be a real number a,
or a compact subspace $[\mathbf{a}]$.

The socket or test may be an open subspace $(\mathbf{b})$,
or a universal quantifier $\forall x \in (-).\ \phi x$.

These define a natural direction

$$\underrightarrow{\ a \in \mathbf{b}\ } \quad \text{and} \quad \underrightarrow{\ \mathbf{a} \Subset \mathbf{b}\ } \quad \text{but} \quad \underleftarrow{\ \forall x \in \mathbf{a}\ }$$

which also goes up arithmetic expression trees,
from arguments to results.

$\mathbf{a} \Subset \mathbf{y}$ is like the constraint $\mathbf{y}$ `is` $\mathbf{a}$ in some versions of Prolog.
This transfers the value of $\mathbf{a}$ to $\mathbf{y}$ and (unlike "=" considered as unification) not *vice versa*.

Another constraint, on the output precision

A lazy logic programming interpretation of this would be very lazy.

To make it do anything, we also need a way to specify the precision that we require of the output.

We squeeze the width $\|\mathbf{x}\| \equiv (\overline{\mathbf{x}} - \underline{\mathbf{x}})$ of an interval by the constraint

$$\|\mathbf{x}\| < \epsilon \quad \equiv \quad \forall x, y \in \mathbf{x}.\, \left|x - y\right| < \epsilon.$$

This is syntactic sugar — it is already definable as a predicate in our calculus.

Failure of this constraint (as of others) causes back-tracking. This is one of the cases of back-tracking that has already emerged from programming multiple-precision arithmetic.

Moore arithmetic

Returning specifically to $\mathbb{R}$, we write $\oplus, \ominus, \otimes$ for Moore's arithmetical operations on intervals:

$$
\begin{aligned}
\mathbf{a} \oplus \mathbf{b} &\equiv [\underline{\mathbf{a}} + \underline{\mathbf{b}},\ \overline{\mathbf{a}} + \overline{\mathbf{b}}] \\
\ominus \mathbf{a} &\equiv [-\overline{\mathbf{a}},\ -\underline{\mathbf{a}}] \\
\mathbf{a} \otimes \mathbf{b} &\equiv [\mathsf{min}(\underline{\mathbf{a}} \times \underline{\mathbf{b}}, \underline{\mathbf{a}} \times \overline{\mathbf{b}}, \overline{\mathbf{a}} \times \underline{\mathbf{b}}, \overline{\mathbf{a}} \times \overline{\mathbf{b}}), \\
&\qquad \mathsf{max}(\underline{\mathbf{a}} \times \underline{\mathbf{b}}, \underline{\mathbf{a}} \times \overline{\mathbf{b}}, \overline{\mathbf{a}} \times \underline{\mathbf{b}}, \overline{\mathbf{a}} \times \overline{\mathbf{b}})],
\end{aligned}
$$

and $\lessdot, \gtrdot, \pitchfork, \Subset$ for the computationally observable relations

$$
\begin{aligned}
\mathbf{x} \lessdot \mathbf{y} &\equiv \overline{\mathbf{x}} < \underline{\mathbf{y}} \quad \equiv \quad \mathbf{y} \gtrdot \mathbf{x} \\
\mathbf{x} \pitchfork \mathbf{y} &\equiv [\mathbf{x}] \cap [\mathbf{y}] = \emptyset \quad \text{or} \quad (\overline{\mathbf{x}} < \underline{\mathbf{y}}) \vee (\overline{\mathbf{y}} < \underline{\mathbf{x}}), \\
\mathbf{x} \Subset \mathbf{y} &\equiv \underline{\mathbf{x}} < \underline{\mathbf{y}} < \overline{\mathbf{x}} < \overline{\mathbf{y}}.
\end{aligned}
$$

NB: in $\mathbf{a} \lessdot \mathbf{b}$, $\mathbf{a} \gtrdot \mathbf{b}$ and $\mathbf{a} \pitchfork \mathbf{b}$, the intervals $\mathbf{a}$ and $\mathbf{b}$ are disjoint.

Extending the Moore operations to expressions

By structural recursion on syntax, we may extend the Moore operations from symbols to expressions.

Essentially, we just

$$\begin{array}{llllllllll} \text{replace} & x & + & - & \times & < & > & \neq & \in & \exists x \\ \text{by} & \mathbf{x} & \oplus & \ominus & \otimes & \olessthan & \ogreaterthan & \pitchfork & \Subset & \exists \mathbf{x} \end{array}$$

other variables, constants, $n : \mathbb{N}$, $\wedge$, $\vee$, $\exists n$, rec, the stay the same.

(We can't translate $\forall x \in [a, b]$ — yet.)

Extending the Moore operations to expressions

By structural recursion on syntax, we may extend the Moore operations from symbols to expressions.

Essentially, we just

$$\begin{array}{lccccccccc} \text{replace} & x & + & - & \times & < & > & \neq & \in & \exists x \\ \text{by} & \mathbf{x} & \oplus & \ominus & \otimes & \olessthan & \ogreaterthan & \pitchfork & \Subset & \exists \mathbf{x} \end{array}$$

other variables, constants, $n : \mathbb{N}$, $\wedge$, $\vee$, $\exists n$, rec, the stay the same.

(We can't translate $\forall x \in [a, b]$ — yet.)

This extends the meaning of arithmetic expressions fx and logical formulae ϕx in such a way that

- substituting $\mathbf{x} \equiv [x, x]$ recovers the original value,
- the dependence on the interval argument $\mathbf{x}$ is monotone,
- and substitution is preserved.

Of course, the laws of arithmetic are not preserved.

Extending the Moore operations to expressions

We shall write $\rlap{\forall}\mathsf{M}\, x \in \mathbf{x}.\, fx$ or $\rlap{\forall}\mathsf{M}\, x \in \mathbf{x}.\, \phi x$ for the translation of the arithmetical expression fx or logical formula ϕx.

The symbol $\rlap{\forall}\mathsf{M}$ is a cross between $\forall$ and M (for Moore).

Remember that it is a syntactic translation (like substitution). So the continuous variable x does not occur in $\rlap{\forall}\mathsf{M}\, x \in \mathbf{x}.\, fx$ or $\rlap{\forall}\mathsf{M}\, x \in \mathbf{x}.\, \phi x$.

$\rlap{\forall}\mathsf{M}$ is not a quantifier.

But there is a reason why it looks like one...

The fundamental theorem of interval analysis

Interval computation is reliable in the sense that
it provides upper and lower bounds for all computations in $\mathbb{R}$.

More generally, bounding cells for computations in $\mathbb{R}^{n}$.

The fundamental theorem of interval analysis???

Interval computation is reliable in the sense that
it provides upper and lower bounds for all computations in $\mathbb{R}$.

More generally, bounding cells for computations in $\mathbb{R}^n$.

If this were all that interval computation could do,
it would be useless.

The fundamental theorem of interval analysis

Interval computation is reliable in the sense that
it provides upper and lower bounds for all computations in $\mathbb{R}$.

More generally, bounding cells for computations in $\mathbb{R}^n$.

If this were all that interval computation could do,
it would be useless.

In fact, it is much better than this:
by making the working intervals sufficiently small,
it can compute a compact bounding cell within
any arbitrary open bounding cell that exists mathematically.

The fundamental theorem of interval analysis

Interval computation is reliable in the sense that
it provides upper and lower bounds for all computations in $\mathbb{R}$.

More generally, bounding cells for computations in $\mathbb{R}^n$.

If this were all that interval computation could do,
it would be useless.

In fact, it is much better than this:
by making the working intervals sufficiently small,
it can compute a compact bounding cell within
any arbitrary open bounding cell that exists mathematically.

This is an ϵ–δ statement:
$\forall\epsilon > 0$ (the required output precision),
$\exists\delta > 0$ (the necessary size of the working intervals).

Locally compact spaces again

Recall the fundamental property of locally compact spaces:

$$\phi a \iff \exists \mathbf{x}.\, a \in \mathbf{x} \wedge \forall x \in \mathbf{x}.\, \phi x,$$

which means:

- if a satisfies the observable predicate ϕ
  (or a belongs to the open subspace that corresponds to ϕ),
- then a is in the interior of some cell $\mathbf{x}$
- throughout which ϕ holds
  (or which is contained in the open subspace that corresponds to ϕ).

Here is the fundamental theorem

Using the quantifier $\forall$ we have

$$\phi a \iff \exists \mathbf{x}.\, a \in \mathbf{x} \wedge \forall x \in \mathbf{x}.\, \phi x.$$

Here is the fundamental theorem

Using the quantifier $\forall$ we have

$$\phi a \iff \exists \mathbf{x}.\, a \in \mathbf{x} \wedge \forall x \in \mathbf{x}.\, \phi x.$$

By an easy structural induction on syntax we can prove

$$\phi a \iff \exists \mathbf{x}.\, a \in \mathbf{x} \wedge |\!\forall\!| x \in \mathbf{x}.\, \phi x,$$

for the Moore interpretation $|\!\forall\!|$.

Here is the fundamental theorem

Using the quantifier $\forall$ we have

$$\phi a \iff \exists \mathbf{x}.\, a \in \mathbf{x} \wedge \forall x \in \mathbf{x}.\, \phi x.$$

By an easy structural induction on syntax we can prove

$$\phi a \iff \exists \mathbf{x}.\, a \in \mathbf{x} \wedge |\!\forall\!| x \in \mathbf{x}.\, \phi x,$$

for the Moore interpretation $|\!\forall\!|$. This means:

- if a satisfies the observable predicate ϕ,
- then a is in the interior of some cell $\mathbf{x}$
- which satisfies the translation of ϕ.

Here is the fundamental theorem

Using the quantifier $\forall$ we have

$$\phi a \iff \exists \mathbf{x}.\, a \in \mathbf{x} \wedge \forall x \in \mathbf{x}.\, \phi x.$$

By an easy structural induction on syntax we can prove

$$\phi a \iff \exists \mathbf{x}.\, a \in \mathbf{x} \wedge |\!\!\forall\!\!| x \in \mathbf{x}.\, \phi x,$$

for the Moore interpretation $|\!\!\forall\!\!|$. This means:

- if a satisfies the observable predicate ϕ,
- then a is in the interior of some cell $\mathbf{x}$
- which satisfies the translation of ϕ.

For example, $fa \in \mathbf{b} \iff \exists \mathbf{x}.\, a \in \mathbf{x} \wedge (|\!\!\forall\!\!| x \in \mathbf{x}.\, fx) \Subset \mathbf{b}$.

So we obtain arbitrary precision $\|\mathbf{b}\|$
by choosing the working interval $\mathbf{x}$ to be sufficiently small.

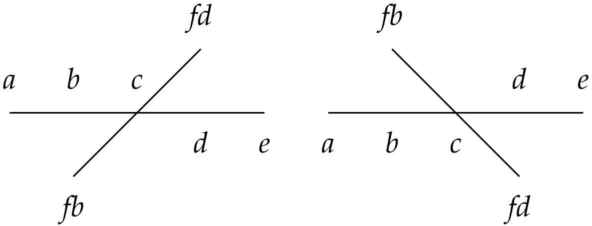
Solving equations

How do we find a zero of a function, x such that $0 = f(x)$?

Solving equations

How do we find a zero of a function, x such that $0 = f(x)$?

Any zero c that we can find numerically
is stable in the sense that,
arbitrarily closely to c, there are b, d with $b < c < d$
and either $f(b) < 0 < f(d)$ or vice versa.



Solving equations

The definition of a stable zero may be written in the calculus for continuous variables, and translated into intervals.

Write $\mathbf{x}$ for the outer interval $[a, e]$.

There are $b \in \mathbf{b}$, $c \in \mathbf{c}$ and $d \in \mathbf{d}$ with
$\mathbf{b} \olessthan \mathbf{c} \olessthan \mathbf{d}$ and $f(\mathbf{b}) \olessthan 0 \olessthan f(\mathbf{d})$.

So if the interval $\mathbf{x}$ contains a stable zero,
$0 \in \mathfrak{f}(\mathbf{x}) \equiv \forall x \in \mathbf{x}.\, f(x)$.
Remember that $\in$ means "in the interior".

This is how $\in f(\mathbf{x})$ and $\Subset f(\mathbf{x})$ arise
with an expression on the right of $\Subset$.

Logic programming with intervals

Remember that the continuous variable x does not occur in the translation $\forall\!\!\!\mid x \in \mathbf{x}.\ \phi x$ of ϕx. Of course, we eliminate the other continuous variables y, z, ... in the same way.

This leaves a predicate involving cellular variables like $\mathbf{x}$.

Logic programming with intervals

Remember that the continuous variable x does not occur in the translation $\forall\!\!\!\forall x \in \mathbf{x}.\ \phi x$ of ϕx. Of course, we eliminate the other continuous variables $y, z, \ldots$ in the same way.

This leaves a predicate involving cellular variables like $\mathbf{x}$.

We build up arithmetical and logical expressions in this order:

- the interval arithmetical operations $\oplus, \ominus, \otimes$;
- more arithmetical operations;
- the relations $\olessthan, \ogreaterthan, \pitchfork, \Subset$;
- conjunction $\wedge$;
- cellular quantification $\exists\mathbf{x}$;
- disjunction $\vee$, integer quantification $\exists n$ and recursion;
- universal quantification $\forall x \in [a, b]$;
- more conjunction, *etc.*

Some logic programming techniques

We can manipulate $\exists \mathbf{x}$ applied to $\wedge$ using various techniques of logic programming.

- Constraint logic programming,
  essentially due to John Cleary.
  This is the closest analogue of unification for intervals.
- Symbolic differentiation, to pass the required precision of outputs back to the inputs.
- The Interval Newton algorithm for solving equations, which are expressed as $0 \in f(\mathbf{x})$.
- (Maybe) classification of semi-algebraic sets.

Some logic programming techniques

We can manipulate $\exists \mathbf{x}$ applied to $\wedge$ using various techniques of logic programming.

- Constraint logic programming, essentially due to John Cleary. This is the closest analogue of unification for intervals.
- Symbolic differentiation, to pass the required precision of outputs back to the inputs.
- The Interval Newton algorithm for solving equations, which are expressed as $0 \in f(\mathbf{x})$.
- (Maybe) classification of semi-algebraic sets.

Surprisingly, this fragment appears to be decidable.

But adding $\exists n$ and recursion makes it Turing complete.

Some logic programming techniques

We can manipulate $\exists\mathbf{x}$ applied to $\wedge$ using various techniques of logic programming.

- Constraint logic programming,
  essentially due to John Cleary.
  This is the closest analogue of unification for intervals.
- Symbolic differentiation, to pass the required precision of outputs back to the inputs.
- The Interval Newton algorithm for solving equations, which are expressed as $0 \in f(\mathbf{x})$.
- (Maybe) classification of semi-algebraic sets.

Surprisingly, this fragment appears to be decidable.

But adding $\exists n$ and recursion makes it Turing complete.

The universal quantifier $\forall x \in [a, b]$ applied to $\vee$ and $\exists n$,
may be turned into a recursive program using the Heine–Borel property, with $\not\forall$ as its base base.

The $\exists\mathbf{x},\wedge$ fragment

We consider the fragment of the language consisting of formulae like

$$\exists \mathbf{y}_1\mathbf{y}_2\mathbf{y}_3.\ \ \mathbf{x}_2 \oplus \mathbf{y}_1 \olessthan \mathbf{x}_3 \otimes \mathbf{x}_1 \ \wedge\ \mathbf{x}_3 \neq \mathbf{y}_3$$

$$\wedge\ \mathbf{y}_1 \otimes \mathbf{x}_3 \Subset \mathbf{z}_2 \ \wedge\ 0 \in \mathbf{z}_1 \otimes \mathbf{z}_1 \ \wedge\ \|\mathbf{z}_1\| < 2^{-40}$$

in which the variables

- $\mathbf{x}_1, \mathbf{x}_2, \ldots$ are free and occur only as plugs (on the left of $\Subset$);
- $\mathbf{y}_1, \mathbf{y}_2, \ldots$ are bound, and may occur as both plugs and sockets;
- $\mathbf{z}_1, \mathbf{z}_2, \ldots$ are free, occurring only as sockets (right of $\Subset$).

The $\exists\mathbf{x},\wedge$ fragment

We consider the fragment of the language consisting of formulae like

$$\exists \mathbf{y}_1\mathbf{y}_2\mathbf{y}_3.\ \ \mathbf{x}_2 \oplus \mathbf{y}_1 \oslash \mathbf{x}_3 \otimes \mathbf{x}_1 \ \wedge\ \mathbf{x}_3 \neq \mathbf{y}_3$$

$$\wedge\ \mathbf{y}_1 \otimes \mathbf{x}_3 \Subset \mathbf{z}_2 \ \wedge\ 0 \in \mathbf{z}_1 \otimes \mathbf{z}_1 \ \wedge\ \|\mathbf{z}_1\| < 2^{-40}$$

in which the variables

- $\mathbf{x}_1, \mathbf{x}_2, \ldots$ are free and occur only as plugs (on the left of $\Subset$);
- $\mathbf{y}_1, \mathbf{y}_2, \ldots$ are bound, and may occur as both plugs and sockets;
- $\mathbf{z}_1, \mathbf{z}_2, \ldots$ are free, occurring only as sockets (right of $\Subset$).

Using convex union, each socket contains at most one plug.

The $\exists \mathbf{x}, \wedge$ fragment

We consider the fragment of the language consisting of formulae like

$$\exists \mathbf{y}_1 \mathbf{y}_2 \mathbf{y}_3.\ \ \mathbf{x}_2 \oplus \mathbf{y}_1 \oslash \mathbf{x}_3 \otimes \mathbf{x}_1 \ \wedge\ \mathbf{x}_3 \neq \mathbf{y}_3$$

$$\wedge\ \mathbf{y}_1 \otimes \mathbf{x}_3 \Subset \mathbf{z}_2 \ \wedge\ 0 \in \mathbf{z}_1 \otimes \mathbf{z}_1 \ \wedge\ \|\mathbf{z}_1\| < 2^{-40}$$

in which the variables

- $\mathbf{x}_1, \mathbf{x}_2, \ldots$ are free and occur only as plugs (on the left of $\Subset$);
- $\mathbf{y}_1, \mathbf{y}_2, \ldots$ are bound, and may occur as both plugs and sockets;
- $\mathbf{z}_1, \mathbf{z}_2, \ldots$ are free, occurring only as sockets (right of $\Subset$).

Using convex union, each socket contains at most one plug.

Since the relevant directed graph is acyclic, bound variables that occur as both plugs and sockets may be eliminated.
So wlog bound variables occur only as plugs.

Cleary's algorithm

In the context of the rest of the problem, the free plugs $\mathbf{x}_1, \mathbf{x}_2, \ldots$ have given interval values (the arguments, to their currently known precision). The other free and bound variables are initially assigned the completely undefined value $[-\infty, +\infty]$.

Cleary's algorithm

In the context of the rest of the problem, the free plugs $\mathbf{x}_1, \mathbf{x}_2, \ldots$ have given interval values (the arguments, to their currently known precision). The other free and bound variables are initially assigned the completely undefined value $[-\infty, +\infty]$.

We evaluate the arithmetical (interval) expressions.

Cleary's algorithm

In the context of the rest of the problem, the free plugs $\mathbf{x}_1, \mathbf{x}_2, \ldots$ have given interval values (the arguments, to their currently known precision). The other free and bound variables are initially assigned the completely undefined value $[-\infty, +\infty]$.

We evaluate the arithmetical (interval) expressions.

In any conjunct $\mathbf{a} \Subset \mathbf{z}$, where $\mathbf{z}$ is a (socket) variable (so it doesn't occur elsewhere, and has been assigned the value $[-\infty, +\infty]$), assign the value of $\mathbf{a}$ to $\mathbf{z}$.

Cleary's algorithm

In the context of the rest of the problem, the free plugs $\mathbf{x}_1, \mathbf{x}_2, \ldots$ have given interval values (the arguments, to their currently known precision). The other free and bound variables are initially assigned the completely undefined value $[-\infty, +\infty]$.

We evaluate the arithmetical (interval) expressions.

In any conjunct $\mathbf{a} \Subset \mathbf{z}$, where $\mathbf{z}$ is a (socket) variable (so it doesn't occur elsewhere, and has been assigned the value $[-\infty, +\infty]$), assign the value of $\mathbf{a}$ to $\mathbf{z}$.

If all the constraints are satisfied — return successfully.

Cleary's algorithm

In the context of the rest of the problem, the free plugs $\mathbf{x}_1, \mathbf{x}_2, \ldots$ have given interval values (the arguments, to their currently known precision). The other free and bound variables are initially assigned the completely undefined value $[-\infty, +\infty]$.

We evaluate the arithmetical (interval) expressions.

In any conjunct $\mathbf{a} \Subset \mathbf{z}$, where $\mathbf{z}$ is a (socket) variable (so it doesn't occur elsewhere, and has been assigned the value $[-\infty, +\infty]$), assign the value of $\mathbf{a}$ to $\mathbf{z}$.

If all the constraints are satisfied — return successfully.

If one of them can never be satisfied, even if the variables are assigned narrower intervals —back-track.

Cleary's algorithm

In the context of the rest of the problem, the free plugs $\mathbf{x}_1, \mathbf{x}_2, \ldots$ have given interval values (the arguments, to their currently known precision). The other free and bound variables are initially assigned the completely undefined value $[-\infty, +\infty]$.

We evaluate the arithmetical (interval) expressions.

In any conjunct $\mathbf{a} \Subset \mathbf{z}$, where $\mathbf{z}$ is a (socket) variable (so it doesn't occur elsewhere, and has been assigned the value $[-\infty, +\infty]$), assign the value of $\mathbf{a}$ to $\mathbf{z}$.

If all the constraints are satisfied — return successfully.

If one of them can never be satisfied, even if the variables are assigned narrower intervals —back-track.

If they're not, we update the values assigned to the variables, replacing one interval by a narrower one, using one of the four techniques.

Then repeat the evaluation and test.

Cleary's algorithm

In the context of the rest of the problem, the free plugs $\mathbf{x}_1, \mathbf{x}_2, \ldots$ have given interval values (the arguments, to their currently known precision). The other free and bound variables are initially assigned the completely undefined value $[-\infty, +\infty]$.

We evaluate the arithmetical (interval) expressions.

In any conjunct $\mathbf{a} \Subset \mathbf{z}$, where $\mathbf{z}$ is a (socket) variable (so it doesn't occur elsewhere, and has been assigned the value $[-\infty, +\infty]$), assign the value of $\mathbf{a}$ to $\mathbf{z}$.

If all the constraints are satisfied — return successfully.

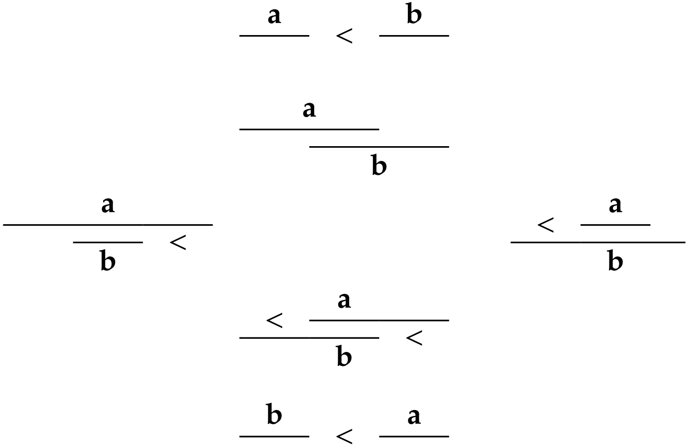
If one of them can never be satisfied, even if the variables are assigned narrower intervals —back-track.

If they're not, we update the values assigned to the variables, replacing one interval by a narrower one, using one of the four techniques.

Then repeat the evaluation and test.
For this fragment, the algorithm terminates.

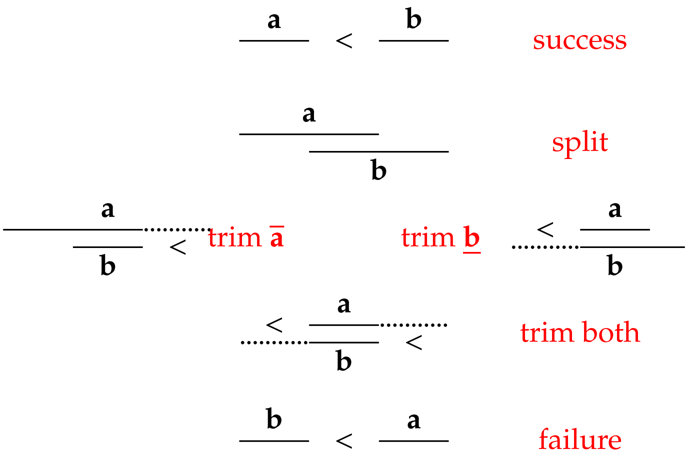
Cleary's "unification" rules for $\mathbf{a} \oslash \mathbf{b}$

There are six possibilities for the existing values of **a** and **b**. Remember that **a** and **b** are our current state of knowledge about certain real numbers $a \in \mathbf{a}$ and $b \in \mathbf{b}$ with $a < b$.



Cleary's "unification" rules for $\mathbf{a} \olessthan \mathbf{b}$

There are six possibilities for the existing values of **a** and **b**. Remember that **a** and **b** are our current state of knowledge about certain real numbers $a \in \mathbf{a}$ and $b \in \mathbf{b}$ with $a < b$.



Cleary's rules for $\mathbf{a} \oplus \mathbf{b}$

Working down the expression tree, the requirement to trim intervals passes from the values to the arguments of arithmetic operators.

Cleary's rules for $\mathbf{a} \oplus \mathbf{b}$

Working down the expression tree, the requirement to trim intervals passes from the values to the arguments of arithmetic operators.

Suppose we want to trim the right endpoint of $\mathbf{a} \oplus \mathbf{b}$ to $\bar{\mathbf{c}}$.

Cleary's rules for $\mathbf{a} \oplus \mathbf{b}$

Working down the expression tree, the requirement to trim intervals passes from the values to the arguments of arithmetic operators.

Suppose we want to trim the right endpoint of $\mathbf{a} \oplus \mathbf{b}$ to $\bar{\mathbf{c}}$.

Think of

- $\mathbf{a}$ as (the range of) the cost of meat and
- $\mathbf{b}$ as (the range of) the cost of vegetables,
- and $\bar{\mathbf{c}}$ as the budget for the whole meal.

Cleary's rules for $\mathbf{a} \oplus \mathbf{b}$

Working down the expression tree, the requirement to trim intervals passes from the values to the arguments of arithmetic operators.

Suppose we want to trim the right endpoint of $\mathbf{a} \oplus \mathbf{b}$ to $\overline{\mathbf{c}}$.

Think of

- $\mathbf{a}$ as (the range of) the cost of meat and
- $\mathbf{b}$ as (the range of) the cost of vegetables,
- and $\overline{\mathbf{c}}$ as the budget for the whole meal.

Then we have to trim

- $\overline{\mathbf{a}}$ to $\overline{\mathbf{c}} - \underline{\mathbf{b}}$, and
- $\overline{\mathbf{b}}$ to $\overline{\mathbf{c}} - \underline{\mathbf{a}}$.

There are similar (but more complicated) rules for $\otimes$.

Moore's Interval Newton algorithm (my version)

Given a function f and and interval $\mathbf{x}$,

Evaluate

- the function f at a point x_0 in the middle of $\mathbf{x}$
- and the derivative f' on the whole interval: $\forall x \in \mathbf{x}.\, f'(x)$.

Moore's Interval Newton algorithm (my version)

Given a function f and and interval $\mathbf{x}$,

Evaluate

- the function f at a point x_0 in the middle of $\mathbf{x}$
- and the derivative f' on the whole interval: $\forall\!\!\!\!|\, x \in \mathbf{x}.\, f'(x)$.

This bounds the values of the function throughout the interval:

$$f(x) \quad \in \quad f(x_0) \oplus (x - x_0) \otimes \forall\!\!\!\!|\, x \in \mathbf{x}.\, f'(x)$$

This is a two-term Taylor series.
It's how we should define derivatives of interval-valued functions.

Moore's Interval Newton algorithm (my version)

Given a function f and and interval $\mathbf{x}$,

Evaluate

- the function f at a point x_0 in the middle of $\mathbf{x}$
- and the derivative f' on the whole interval: $\forall\!\!\!|\, x \in \mathbf{x}.\, f'(x)$.

This bounds the values of the function throughout the interval:

$$f(x) \quad \in \quad f(x_0) \oplus (x - x_0) \otimes \forall\!\!\!|\, x \in \mathbf{x}.\, f'(x)$$

This is a two-term Taylor series.
It's how we should define derivatives of interval-valued functions.

Slogan: Crude arithmetic gives subtle logical information.

Translating the universal quantifier

Applying the translation to ϕx, we need to simplify

$$\forall x \in \mathbf{a}.\, \phi x \;\equiv\; \forall x \in \mathbf{a}.\, \exists \mathbf{x}.\, x \in \mathbf{x} \wedge \forall\!\!\!\mid x' \in \mathbf{x}.\, \phi x'.$$

This says that the compact (closed bounded) interval $\mathbf{a}$
is covered by the open interiors of cells $\mathbf{x}$
each of which satisfies the translation $\forall\!\!\!\mid x' \in \mathbf{x}.\, \phi x'$.

The Heine–Borel property (classical theorem, axiom of ASD)
says that there is a finite sub-cover,
so wlog $\|\mathbf{x}\| = 2^{-k}$ for some k.

Translating $\forall$ with $\vee$ and $\exists n$

It's natural to include ($\vee$ and) $\exists n$ in the Heine–Borel property:

$$\forall x \in [0,1].\, \exists n.\, \phi n x \qquad \Longleftrightarrow$$

$$\exists k.\, \bigwedge_{j=0}^{2^k-1} \exists n.\, \forall\!\!\!\!|\, x \in [j \cdot 2^{-k},\ (j+1)\cdot 2^{-k}].\, \phi_n x.$$

We can read this as a recursive program for

$$\theta[a,b] \ \equiv\ \forall x \in [a,b].\, \exists n.\, \phi_n x$$

that splits $[a,b]$ into subintervals. When these get smaller than $2^{-k}(b-a)$, use $\forall\!\!\!\!|$ instead of deeper recursion.

$$\begin{aligned}\theta[a,b] \quad \Longleftrightarrow \quad & \exists k.\, \left(\exists n.\, \forall\!\!\!\!|\, x \in [a,\ a+2^{-k}(b-a)].\, \phi_n x\right) \\ & \wedge\ \theta[a+2^{-k}(b-a),\ b]\end{aligned}$$

Conclusion: some programming projects

(Logic) programming environment together with multiple precision arithmetic.

Use this to implement:

- Cleary's algorithm, Interval Newton, ...
- Cellular computation for $\mathbb{R}^2$, $\mathbb{R}^3$, $\mathbb{C}$, ...
- Heine–Borel translation of $\forall$.

Syntactic stuff:

- Simple front end to translate the continuous language into the interval methods.
- Proof assistant for the deduction rules of ASD.