Mechanized Mathematics

Jacques Carette

McMaster University

CICM – Thursday July 8th, 2010

Outline

Context

Lessons

Tools

Results

Mechanized Mathematics


OFF
ON
+/−
÷
×
−
+
0
.
7
8
9
C
4
5
6
=
1
2
3

Lessons

What?


Marlboro
Shell
BRIDGESTONE
AMD
vodafone
POTENZA
OLYMPUS
FIAT


AIRFIELD
BAWAG
BAWA
AYS S1H13

The Mathematics Process

In mathematics, we

- define/represent new concepts and new notations
- state, in convenient ways, problems to be solved
- conduct experiments
- make conjectures
- prove theorems
- gain insight through proofs, computations and visualization
- turn theorems into algorithms; compute
- make connections between theories
- communicate results
- reuse previous results

The Mathematics Process

In mathematics, we

- define/represent new concepts and new notations
- state, in convenient ways, problems to be solved
- conduct experiments
- make conjectures
- prove theorems
- gain insight through proofs, computations and visualization
- turn theorems into algorithms; compute
- make connections between theories
- communicate results
- reuse previous results

The Mathematics Process

In mathematics, we

- define/represent new concepts and new notations
- state, in convenient ways, problems to be solved
- conduct experiments
- make conjectures
- prove theorems
- gain insight through proofs, computations and visualization
- turn theorems into algorithms; compute
- make connections between theories
- communicate results
- reuse previous results

The Mathematics Process

In mathematics, we

- define/represent new concepts and new notations
- state, in convenient ways, problems to be solved
- conduct experiments
- make conjectures
- prove theorems
- gain insight through proofs, computations and visualization
- turn theorems into algorithms; compute
- make connections between theories
- communicate results
- reuse previous results

The Mathematics Process

In mathematics, we

- define/represent new concepts and new notations
- state, in convenient ways, problems to be solved
- conduct experiments
- make conjectures
- prove theorems
- gain insight through proofs, computations and visualization
- turn theorems into algorithms; compute
- make connections between theories
- communicate results
- reuse previous results

The Mathematics Process

In mathematics, we

- define/represent new concepts and new notations
- state, in convenient ways, problems to be solved
- conduct experiments
- make conjectures
- prove theorems
- gain insight through proofs, computations and visualization
- turn theorems into algorithms; compute
- make connections between theories
- communicate results
- reuse previous results

Goal: build a tool that helps us do all of that.

Expressions

are

syntax

Some expressions
are
meaningless

$$\begin{tikzcd}
\mathsf{Expression} \arrow[r, "\mathsf{diff}"] \arrow[d, "\llbracket \cdot \rrbracket"'] & \mathsf{Expression} \arrow[d, "\llbracket \cdot \rrbracket"] \\
\mathbb{R} \to \mathbb{R} \arrow[r, "\partial"'] & \mathbb{R} \to \mathbb{R}
\end{tikzcd}$$

Meaningless statements

$\int_{-\infty}^{\infty} f(x)\delta_a(x)dx = f(a)$

$f'(x) =$ principal part $\frac{f(x+\epsilon)-f(x)}{\epsilon}$, ϵ infinitesimal

$\sum n!x^n$ denotes a unique function on $\mathbb{R}^+$

$x^2 + 1$ has exactly 2 roots.

Meaningless statements

Distributions

$$\int_{-\infty}^{\infty} f(x)\delta_a(x)dx = f(a)$$

Non-standard analysis

$$f'(x) = \text{principal part}\,\frac{f(x+\epsilon)-f(x)}{\epsilon},\ \epsilon \text{ infinitesimal}$$

Resummation

$$\sum n!x^n \text{ denotes a unique function on } \mathbb{R}^+$$

Complex numbers

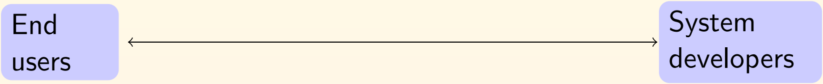
$$x^2 + 1 \text{ has exactly 2 roots.}$$

What is meaningful
changes over time

Experimenting

Who?

Wide range of requirements and usage patterns.



End users

System developers

Need very different views onto the same system.

Wide range of requirements and usage patterns.

End users

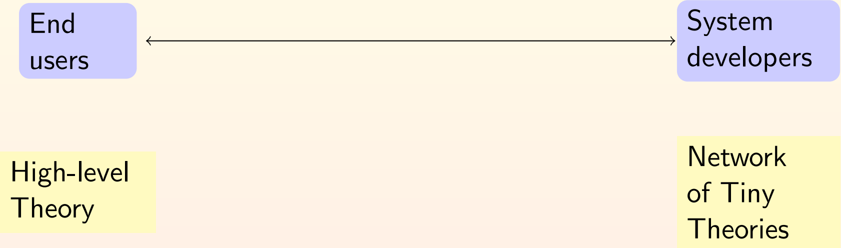
System developers

Context Dependent

Defines Contexts

Need very different views onto the same system.

Wide range of requirements and usage patterns.



End users

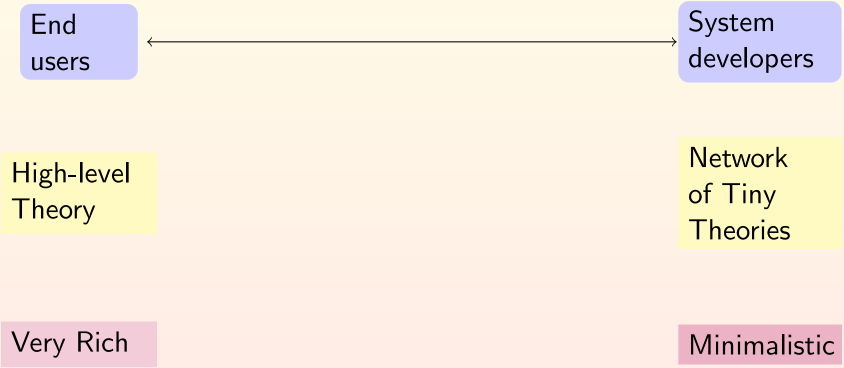
System developers

High-level Theory

Network of Tiny Theories

Need very different views onto the same system.

Wide range of requirements and usage patterns.



End users

System developers

High-level Theory

Network of Tiny Theories

Very Rich

Minimalistic

Need very different views onto the same system.

Don’t

Repeat

Yourself

Duplication

Is

Evil

Non-choices

efficiency
correctness
abstraction
modularity
usability

Tools

Tools

Denotational semantics
Code generation
Polymorphism
First-class syntax
Domain Specific Languages
Universal algebra
Type theory
Biform Theories
High-level theories
Partial evaluation
Abstract interpretation
Genericity
Reflection
Unicode
Category theory
Literate programming
Chiron
Proof generation

Results

Using Structure

```
Empty := Theory {}
Carrier := Empty extended by {U:type}
PointedCarrier := Carrier extended by {e:U}
UnaryOperation := Carrier extended by {prime:U -> U}
BinaryOperation := Carrier extended by {**:(U,U) -> U}
CarrierS := Carrier[U |-> S]
MultiCarrier := combine Carrier,CarrierS over Empty
PointedUnarySystem := combine UnaryOperation,PointedCarrier
                      over Carrier
Magma := BinaryOperation [** |-> *]
AdditiveMagma := BinaryOperation [** |-> +]
IdempotentMagma := Magma extended by {axiom:idempotent((*))}
PointedMagma := combine Magma, PointedCarrier over Carrier
CommutativeMagma := Magma extended by {axiom:commutative((*))}
CommutativeAdditiveMagma := AdditiveMagma extended by
    {axiom:commutative((+))}
```

skipping over Loop, Monoid, Group, ...

```
LeftNearSemiring := (combine Semigroup, AdditiveMonoid
    over Carrier) extended by {
  axiom:leftDistributive((*),(+));
  axiom:leftAnnihilative((*),0) }
LeftNearRing := combine LeftNearSemiring, AdditiveGroup
    over AdditiveMonoid
LeftSemirng := combine LeftNearSemiring, AdditiveCommutativeMonoid
    over AdditiveMonoid
LeftRng := combine LeftNearRing, LeftSemirng over LeftNearSemiring
Monoid1 := Monoid [e |-> 1]
LeftSemiring := combine LeftSemirng, Monoid1 over Semigroup
LeftRing := combine LeftRng, LeftSemiring over LeftSemirng
Semirng := LeftSemirng extended by
  { axiom:leftDistributive((flipuc ((*))),(+)) }
Rng := combine LeftRng, Semirng over LeftSemirng
SemiRing := combine LeftSemiring, Semirng over LeftSemirng
Dioid := SemiRing extended by {axiom:idempotent((+))}
Ring := combine Rng, SemiRing over Semirng
CommutativeRing := Ring extended by {axiom:commutative((*))}
BooleanRing := CommutativeRing extended by {axiom:idempotent((*))}
Domain := Ring extended by {
 axiom:forall x:leftDomain((*)).zeroDivisor((*),x,0) implies (x=0)}
IntegralDomain := Domain extended by {axiom:commutative((*))}
DivisionRing := Ring extended by {
 axiom:forall x:leftDomain((*)).
     (not (x=0)) implies invertible(x,(*),1) }
Field := combine DivisionRing, CommutativeRing over Ring
```

```
LeftRing := Theory {
 U : type;  * : (U, U) -> U;  + : (U, U) -> U; - : (U, U) -> U;
 -- : (U, U) -> U;  0 : U;   1 : U; neg : U -> U;
  neg(x) = (0 - x);
  axiom leftIdentity_*_1 := forall x : U. (1 * x) = x;
  axiom rightIdentity_*_1 := forall x : U. (x * 1) = x;
  axiom left0 := forall x : U. (0 * x) = 0;
  axiom rightIdentity_+_0 := forall x : U. (x + 0) = x;
  axiom leftIdentity_+_0 := forall x : U. (0 + x) = x;
  axiom leftDistributive_*_+ :=
    forall x,y,z:U. (x * (y + z)) = ((x * y) + (x * z));
  axiom rightAbsorb_+_- :=
    forall x, y : U. (((y - x) + x) = y and ((y + x) - x) = y);
  axiom leftAbsorb_+_-- :=
    forall x, y : U. ((x + (x -- y)) = y and (x -- (x + y)) = y);
  axiom associative_+ :=
      forall x:U.forall y:U. forall z:U.((x+y)+z) = (x+(y+z));
  axiom associative_* :=
      forall x,y,z:U. ((x * y) * z) = (x * (y * z));
  axiom commutative_+ := forall x,y:U.(x+y)=(y+x);
  theorem inverse_neg := (
    forall x:U.(x+(neg x))=0 and forall x:U.((neg x)+x)=0)}
```

AbelianAdditiveGroup, AbelianGroup, AdditiveCommutativeMonoid, AdditiveGroup, AdditiveMagma, AdditiveMonoid, Band, BiMagma, BinaryOperation, BinaryRelation, BooleanAlgebra, BooleanRing, BoundedDistributiveLattice, BoundedJoinSemilattice, BoundedLattice, BoundedMeetSemilattice, BoundedModularLattice, Carrier, CarrierS, Category, Chain, CommutativeAdditiveMagma, CommutativeBand, CommutativeMagma, CommutativeMonoid, CommutativeRing, CommutativeRingAction, CommutativeSemigroup, ComplementedLattice, Digraph, Dioid, DistributiveLattice, DivisionRing, Domain, DoublyPointed, DualSemilattices, Empty, EquivalenceRelation, Field, FunctionSpace, FunctionalComposition, FuntionalIdentity, GoedelAlgebra, Graph, Group, Heap, HeytingAlgebra, IdempotentMagma, IdempotentSemiheap, IdempotentUpDirectedSet, IntegralDomain, InvolutiveUnarySystem, JoinSemilattice, KleeneAlgebra, KleeneLattice, Lattice, LeftGroup, LeftGroupAction, LeftLoop, LeftMagmaAction, LeftMagmaActionP, LeftMonoidAction, LeftNearRing, LeftNearSemiring, LeftOperation, LeftQuasiGroup, LeftRModule, LeftRing, LeftRingAction, LeftRng, LeftSemigroupAction, LeftSemiring, LeftSemirng, LeftUnital, Loop, Magma, MeetDirectoid, MeetSemilattice, ModalAlgebra, ModularLattice, ModularOrtholattice, Monoid, Monoid1, MoufangLoop, MultiCarrier, NonassociativeRing, OrderRelation, Ortholattice, Orthomodularlattice, PartialOrder, PointedCarrier, PointedCommutativeMagma, PointedMagma, PointedSteiner, PointedUnarySystem, Preorder, PrimeAdditiveGroup, PseudoGraph, Quandle, QuasiGroup, RModule, Rack, ReflexiveOrderRelation, RightGroupAction, RightMagmaAction, RightMagmaActionP, RightMonoid, RightMonoidAction, RightOperation, RightQuasiGroup, RightRModule, RightRingAction, RightSemigroupAction, RightUnital, Ring, Rng, SemiRing, Semigroup, Semiheap, Semirng, SimpleGraph, Sink, Sloop, Squag, StarSemiring, Steiner, SubType, TernaryOperation, TotalOrder, TotalPreorder, TraceMonoid, TransitiveOrderRelation, UnaryOperation, UnaryRelation, Unital, UpDirectedSet, VectorSpace.

137 purely axiomatic theories, 82 properties, using 320 lines of definitions. Should be ≈ 280? (Less?) We stopped expanding because that would cause too much duplication.

137 purely axiomatic theories, 82 properties, using 320 lines of definitions. Should be ≈ 280? (Less?) We stopped expanding because that would cause too much duplication.

Expanded: 2877 lines of property and theory definitions. 303 automatically defined theory morphisms.

137 purely axiomatic theories, 82 properties, using 320 lines of definitions. Should be ≈ 280? (Less?) We stopped expanding because that would cause too much duplication.

Expanded: 2877 lines of property and theory definitions. 303 automatically defined theory morphisms.

Can also automatically define (*universal algebra, category theory*):

- type, sub-structure, homomorphism, free structure, etc,
- type of 'term algebra' over structure, and related morphism(s),
- various transformers (including printing to text, latex, MathML), ...

Also have *structures* (Bit, Peano Naturals) and constructors (Maybe, Either, List, ...)

Using Structure

Generic and Generative Programming

Code Generation - algorithm families

Problem: Encode “design concepts” present in a “software product line” composed of variants of an algorithm.

Code Generation - algorithm families

Problem: Encode “design concepts” present in a “software product line” composed of variants of an algorithm.

Case study: Gaussian Elimination & LU Decomposition.

Rationale: found 80 different implementations in Maple’s library.

Code Generation - algorithm families

Problem: Encode “design concepts” present in a “software product line” composed of variants of an algorithm.

Case study: Gaussian Elimination & LU Decomposition.

Rationale: found 80 different implementations in Maple’s library.

Method:

1. MetaOCaml gives typed generators for typed programs.
2. Uses Functors, Monads, Continuation-passing style, Phantom types (rows and objects, aka open products and open sums), and abstract interpretation.
3. Mostly conditional-free; purely static dispatch

Code Generation - algorithm families

Problem: Encode “design concepts” present in a “software product line” composed of variants of an algorithm.

Case study: Gaussian Elimination & LU Decomposition.

Rationale: found 80 different implementations in Maple’s library.

Method:

1. MetaOCaml gives typed generators for typed programs.
2. Uses Functors, Monads, Continuation-passing style, Phantom types (rows and objects, aka open products and open sums), and abstract interpretation.
3. Mostly conditional-free; purely static dispatch

Result:

1. result code is *identical* to human-written versions for some target cases. No abstractions left at all.
2. over $10,000$ variants
3. generator gives domain-specific error messages

Instantiation Example

```
module GVCI = GenericVectorContainer(IntegerDomainL)
module LA = GenLA(GVCI)

module GenIV5 = GenGE(struct
  module Det =        AbstractDet
  module PivotF =     FullPivot
  module PivotRep =   PermList
  module Update =     FractionFreeUpdate
  module Input =      InpJustMatrix
  module Output =     OutUMatDetRank end)
```

From code

```
module IntegerDomain = struct
  type v = int


  let zero = 0
  let one = 1
  let plus x y = x + y
  let div x y = x / y
  ...
  let better_than = Some (fun x y -> abs x > abs y)
  let normalizerf = None
end
```

to monadic generator

```
module IntegerDomain = struct
  type v = int
  type 'a vc = ('a,v) code

  let zero = .< 0 >.
  let one = .< 1 >.
  let plus x y = ret .< .~x + .~y>.
  let div x y = ret .< .~x / .~y>.
  ...
  let better_than = Some
    (fun x y -> ret .<abs .~x > abs .~y >. )
  let normalizerf = None
end
```

Design Concepts

Design Dim.	**Abstracts**
Domain	Matrix values
Normalization	domain needs it?
ZeroEquivalence	decidability of $= 0$
Representation	Matrix representation
Fraction-free	use of division
Pivoting Strategy	ex:use length?
Pivoting Choice	no/row/column/total
Pivot Rep	list, array, matrix
Full Division	division in domain
Rank	track rank?
Determinant	determinant tracking
Output	choice of output

Design Dim.	**Abstracts**
Packed	L and U as one?
Lower	track lower L ?
Code Rep	codegen options
UserInformation	user-feedback
Augmented	matrix is augmented
Input	choice of input
Logging	trace algorithm
Structure	ex: tri-diagonal
Warning	warn on 0? pivot
In-place	res. stored in input
Error-on-singular	input (near) singular
Conditioning	cond. numb. est.

Design space for LU Decomposition ≥ 24 dimensional!

Abstraction, correctness and efficiency can co-exist

Multiple Interpretations

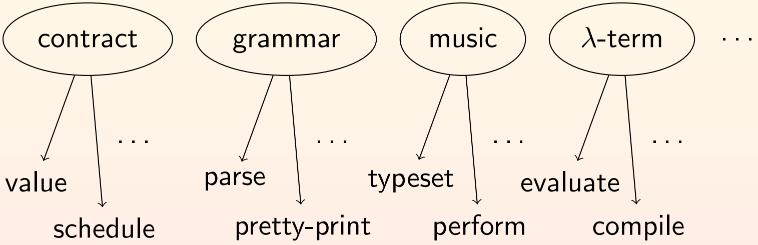
Type-safe interpreters for embedded DSLs

A fold on an inductive data type is an interpreter of a domain-specific language.

contract grammar music λ-term $\cdots$

Type-safe interpreters for embedded DSLs

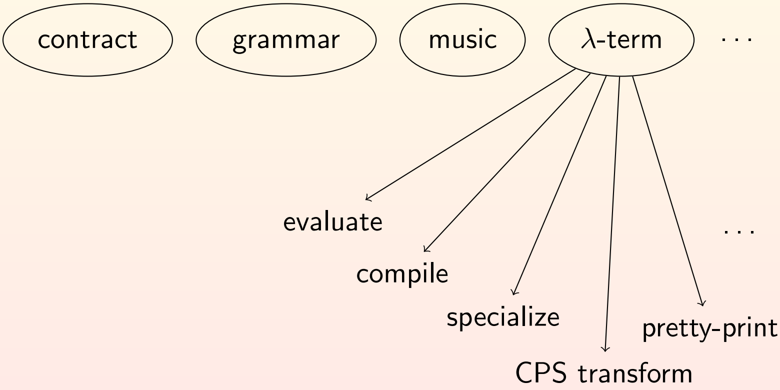
A fold on an inductive data type is an interpreter of a domain-specific language.



The same language can be interpreted in many useful ways.

Type-safe interpreters for embedded DSLs

A fold on a tagless final type is an interpreter of a domain-specific language.



contract

grammar

music

λ-term

. . .

evaluate

compile

specialize

CPS transform

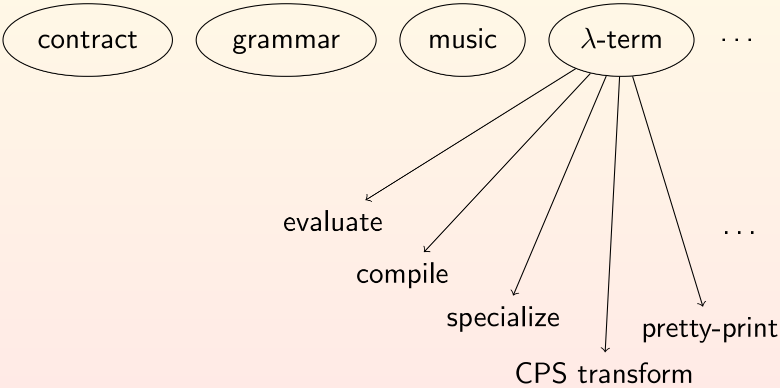
pretty-print

. . .

Church-Scott dual encoding at the constructor level

Type-safe interpreters for embedded DSLs

A fold on a tagless final type is an interpreter of a domain-specific language.



Term typechecked once. Interpretations are compositional.

```
module type Symantics = sig
 type ('c,'sv,'dv) repr
 val int  : int  -> ('c,int,int) repr
 val bool : bool -> ('c,bool,bool) repr
 val add  : ('c,int,int) repr as 'x -> 'x -> 'x
 val mul  : ('c,int,int) repr as 'x -> 'x -> 'x
 val leq  : ('c,int,int) repr as 'x -> 'x -> ('c,bool,bool) repr
 val eql  : ('c,'sa,'da) repr as 'x -> 'x -> ('c,bool,bool) repr
 val if_ : ('c,bool,bool) repr ->
            (unit -> 'x) ->
            (unit -> 'x) -> (('c,'sa,'da) repr as 'x)

 val lam : (('c,'sa,'da) repr -> ('c,'sb,'db) repr as 'x)
             -> ('c,'x,'da->'db) repr
 val app : ('c,'x,'da->'db) repr
             -> (('c,'sa,'da) repr -> ('c,'sb,'db) repr as 'x)
 val fix : ('x -> 'x) ->
    (('c, ('c,'sa,'da) repr -> ('c,'sb,'db) repr, 'da->'db) repr as 'x
end
```

```
module R = struct
  type ('c,'sv,'dv) repr = 'dv
  let int (x:int)   = x
  let bool (b:bool) = b
  let add e1 e2     = e1 + e2
  let mul e1 e2     = e1 * e2
  let leq x y       = x <= y
  let eql x y       = x = y
  let if_ eb et ee =
      if eb then (et ()) else (ee ())

  let lam f = f
  let app e1 e2 = e1 e2
  let fix f = let rec self n = f self n in self
end;;
```

```
let build cast f1 f2 = function
| {st = Some m}, {st = Some n} -> cast (f1 m n)
| e1, e2 -> pdyn (f2 (abstr e1) (abstr e2))

let monoid cast one f1 f2 = function
| {st = Some e'}, e when e' = one -> e
| e, {st = Some e'} when e' = one -> e
| ee -> build cast f1 f2 ee

let ring cast zero one f1 f2 = function
| ({st = Some e'} as e), _ when e' = zero -> e
| _, ({st = Some e'} as e) when e' = zero -> e
| ee -> monoid cast one f1 f2 ee

let add e1 e2 = monoid int 0 R.add C.add (e1,e2)
let mul e1 e2 = ring int 0 1 R.mul C.mul (e1,e2)
let leq e1 e2 = build bool R.leq C.leq (e1,e2)
let eql e1 e2 = build bool R.eql C.eql (e1,e2)
```

Syntax & Semantics

A Biform Theory, using Chiron

`Theory Derivative-Real1D` $\{$

$\quad \textsc{derivative} : (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$

$\quad$ `axiom` $\forall f : (\mathbb{R} \rightarrow \mathbb{R}).\forall x : \mathbb{R}.$

$$\textsc{derivative}(f)(x) \simeq \lim_{\epsilon \rightarrow 0} \frac{|f(x+\epsilon) - f(x)|}{\epsilon}$$

$\quad \textsc{diff} : E_{(\mathbb{R}\rightarrow\mathbb{R})} \rightarrow E_{(\mathbb{R}\rightarrow\mathbb{R})}$

$\quad$ `meaning` $\forall f : E_{(\mathbb{R}\rightarrow\mathbb{R})}.[\![\textsc{diff}(f)]\!] \simeq \textsc{derivative}([\![f]\!])$

$\}$

A Biform Theory, using Chiron

$$
\begin{array}{l}
\texttt{Theory Derivative-Real1D } \{ \\
\quad \textsc{derivative} : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R}) \\
\quad \texttt{axiom } \forall f : (\mathbb{R} \to \mathbb{R}). \forall x : \mathbb{R}. \\
\qquad\qquad \textsc{derivative}(f)(x) \simeq \lim_{\epsilon \to 0} \dfrac{|f(x+\epsilon) - f(x)|}{\epsilon} \\
\\
\quad \textsc{diff} : E_{(\mathbb{R} \to \mathbb{R})} \to E_{(\mathbb{R} \to \mathbb{R})} \\
\quad \texttt{meaning } \forall f : E_{(\mathbb{R} \to \mathbb{R})}. [\![\textsc{diff}(f)]\!] \simeq \textsc{derivative}([\![f]\!]) \\
\}
\end{array}
$$

But that does not work! Term-rewriting based DIFF is actually

$$
\begin{aligned}
\forall f : E_{(\mathbb{R} \to \mathbb{R})}. (\textsc{total}(f) \wedge \textsc{differentiable}(f)) \Rightarrow \\
([\![\textsc{diff}(f)]\!] \simeq \textsc{derivative}([\![f]\!]))
\end{aligned}
$$

A Biform Theory, using Chiron

`Theory Derivative-Real1D` $\{$

$\quad\textsc{derivative} : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$

$\quad$ `axiom` $\forall f : (\mathbb{R} \to \mathbb{R}). \forall x : \mathbb{R}.$

$$\textsc{derivative}(f)(x) \simeq \lim_{\epsilon \to 0} \frac{|f(x+\epsilon) - f(x)|}{\epsilon}$$

$\quad\textsc{diff} : E_{(\mathbb{R} \to \mathbb{R})} \to E_{(\mathbb{R} \to \mathbb{R})}$

$\quad$ `meaning` $\forall f : E_{(\mathbb{R} \to \mathbb{R})}. [\![\textsc{diff}(f)]\!] \simeq \textsc{derivative}([\![f]\!])$

$\}$

But that does not work! Term-rewriting based $\textsc{diff}$ is actually

$$\forall f : E_{(\mathbb{R} \to \mathbb{R})}. (\textsc{total}(f) \wedge \textsc{differentiable}(f)) \Rightarrow \\ ([\![\textsc{diff}(f)]\!] \simeq \textsc{derivative}([\![f]\!]))$$

1. This is not a silly as it seems. 2. $[\![\cdot]\!]$ is very important. 3. Connections!

But also

- Correct-by-construction software generation
  - generate code (C, Java, Fortran) and proofs (Coq and PVs) in parallel
- Vocabulary and Representation
- On good error messages
- The difference between an indeterminate, a symbol, a variable, a parameter and a generic value
- ...

Thank You