Updated Partial Report for Deliverable Nr. 4.2

Simple ODE Solver: Picard and Newton Method

Responsible: Bas Spitters, `spitters@cs.ru.nl`
Site: Radboud Universiteit Nijmegen, The Netherlands

Deliverable Date: February 2011
Forecast Date: February 2012

We report on the organization of our library and our work on the development of numerical integration. We conclude with the route to the ODE-solvers.

1 Maintaining and developing a library for analysis

By its very nature, this task will depend on an extensive library for real analysis. We have explained in task 4.1 how we improved the performance of exact reals in Coq. Here we will also comment on maintaining legacy code and reorganizing the library.

Maintaining legacy code We dependent on a huge base of legacy code: the CoRN library [5]. Some of this code is as much as 12 years old and requires several hours to compile. We are trying to unravel the dependencies, but this is a slow process. As a first step, we replaced a long and too complicated Make-file by a short python-based SCons build system. This avoids quite a lot of recompilation. We have experimented with rebasing the discrete parts of the library on the SSReflect library. We are happy with the features of the SSReflect language. However, we are also dependent on Coq's (trunk) development branch and the two do not always compile together. This became too big an effort. We have (temporarily?) reverted this dependency.

We are maintaining and modernizing the current library. Regarding the former we now use Coq's standard's `ring` and `setoidrewrite` tactics. As to the latter, we have started to use type classes and the `Program` machinery. In fact, our math-classes library [15] is now a separate Coq contribution and is used independently by a number of Coq users. We will now turn to some aspects of this new library.

Type classes. Type classes have been a great success story in the Haskell functional programming language, as a means of organizing interfaces of abstract structures. Coq's type classes provide a superset of their functionality, but are implemented in a different way.

In Haskell and Isabelle, type classes and their instances are second class. They are handled as specialized syntactic constructs whose semantics are given specifically by the type class apparatus. By contrast, the expressivity of dependent types and inductive families as supported in Coq, combined with the use of pre-existing technology in the system (namely proof search and implicit arguments) enable a *first class* type class implementation: classes are ordinary record types ('dictionaries'), instances are ordinary constants of these record types (registered as *hints* with the proof search machinery), class constraints are ordinary implicit parameters, and instance resolution is achieved by augmenting the unification algorithm to invoke ordinary proof search for implicit arguments of class type. Thus, type classes in Coq are realized by relatively minor syntactic aids that bring together existing facilities of the theory and the system into a coherent idiom, rather than by introduction of a new category of qualitatively different definitions with their own dedicated semantics.

We use the algebraic hierarchy based on type classes the abstract specification of $\mathbb{N}, \mathbb{Z}$ and $\mathbb{Q}$ [15]. Surprisingly, this proved to have only a very minor effect on the computation time; see [9].

Abstract interfaces using type classes Type classes offer a convenient layer of abstraction making both proof engineering and programming more flexible: it avoids duplication of code, it introduces a canonical way to refer to operations and properties, both by names and notations, and it allows to easily swap different implementations of number representations and their operations. First we will briefly recap the design decisions made in [15].

Algebraic structures are expressed in terms of a number of carrier sets, a number of relations and operations, and a number of laws that the operations satisfy. One way of describing such a structure is by a *bundled representation*: one uses a dependently typed record that contains the carrier, operations and laws. For example a semigroup can be represented as follows. (The fields sg_car and sg_proper support our explicit handling of naive set theory in type theory.)

```
Record SemiGroup : Type := {
  sg_car :> Setoid ;
  sg_op : sg_car → sg_car → sg_car ;
  sg_proper : Proper ((=) ⟹ (=) ⟹ (=)) sg_op ;
  sg_ass : ∀ x y z . st_car x (st_car y z) = st_car (st_car x y) z) }
```

However, this approach has some serious limitations, the most important one being a lack of support for *sharing* components. For example, suppose we group together two CommutativeMonoids in order to create a SemiRing. Now awkward hacks are necessary to establish equality between the carriers. A second problem is that if we stack up these records to represent higher structures the projection paths become increasingly large and thus inefficient.

Historically these problems have been an acceptable trade-off because *unbundled representations*, in which the carrier and operations are parameterized, introduce even more problems. Type class machinery resolves many of the problems of unbundled representations. However, we should also note that we have clearly met the efficiency problems in the current implementation of type class instance resolution.

An *operational type class* is defined for each operation and relation. For example, we

would have:

```
Class Equiv A := equiv: relation A.
Infix "=" := equiv: type_scope.
Class RingPlus A := ring_plus: A → A → A.
Infix "+" := ring_plus.
```

Now an algebraic structure is just a type class living in Prop that is parametrized by its carrier, relations and operations. This class contains all laws that the operations should satisfy. Since the operations are unbundled we can easily support sharing. For example let us consider the SemiRing interface.

```
Class SemiRing A {e plus mult zero one} : Prop := {
  semiring_mult_monoid :> @CommutativeMonoid A e mult one ;
  semiring_plus_monoid :> @CommutativeMonoid A e plus zero ;
  semiring_distr :> Distribute (.*.) (+) ;
  semiring_left_absorb :> LeftAbsorb (.*.) 0 }.
```

Without type classes it would be a burden to manually carry around the carrier, relations and operations. However, because these parameters are just type class instances, the type class machinery will perform that job for us. For example, an obvious lemma would look like:

```
Lemma double `{SemiRing R} x : 2 * x = x + x.
```

The backtick instructs Coq to automatically insert implicit declarations, namely e plus mult zero one. Furthermore, instance resolution will automatically find instances of the operational type classes for the written notations.

Using this approach to interfaces we have formalized a standard algebraic hierarchy. Furthermore, combined with category theory and universal algebra $\mathbb{N}$ and $\mathbb{Z}$ are represented as interfaces specifying an initial SemiRing and initial Ring. Such an abstract interface for the naturals makes it easier to change the concrete representation in the future. No such simple specification for $\mathbb{Q}$ seems to exists, so we choose [9] to specify it as the field of fractions of $\mathbb{Z}$. More precisely, $\mathbb{Q}$ is specified as a Field containing $\mathbb{Z}$ that moreover can be embedded into the field of fractions of $\mathbb{Z}$.

```
Inductive Frac R `{e : Equiv R} `{zero : RingZero R} : Type :=
  frac { num : R ; den : R ; den_nonzero : den ≠ 0 }.
Class RationalsToFrac (A : Type) := rationals_to_frac : ∀ B `{Integers B}, A → Frac B.
Class Rationals A {e plus mult zero one opp inv} `{U : !RationalsToFrac A} : Prop := {
  rationals_field :> @Field A e plus mult zero one opp inv ;
  rationals_frac :> ∀ `{Integers Z}, Injective (rationals_to_frac A Z) ;
  rationals_frac_mor :> ∀ `{Integers Z}, SemiRing_Morphism (rationals_to_frac A Z) ;
  rationals_embed_ints :> ∀ `{Integers Z}, Injective (integers_to_ring Z A) }.
```

Metric spaces We now turn to O'Connor's formalization of exact real numbers using the completion monad [11]. Traditionally, a metric space is defined as a set X with a metric function $d : X \times X \to \mathbb{R}_+$ satisfying certain axioms. The usual constructive formulation requires d be a computable function. A more relaxed definition for a metric space that does not require the metric be a function. A similar construction can be found in the work by Richman [14].

Monads. Moggi [10] recognized that many non-standard forms of computation may be modeled by monads. Wadler [17] popularized their use in functional programming. Monads are now an established tool to structure computation with side-effects. Completion is a

monad on the category of metric spaces with uniformly continuous functions. The function $\mathsf{return} : X \to \mathfrak{C}X$ defined by $\lambda x\, \varepsilon\, .\, x$ is the embedding of a metric space in its completion. Moreover, a uniformly continuous function $f : X \to \mathfrak{C}Y$ with modulus μ_f can be lifted to operate on complete prelength spaces as $\mathsf{bind}\, f : \mathfrak{C}X \to \mathfrak{C}Y$ defined by $\lambda x\, \varepsilon\, .\, f\,(x\,(\mu_f \frac{\varepsilon}{2}))\, \frac{\varepsilon}{2}$. The restriction to prelength spaces allows the present efficient definition of bind.

One advantage of this approach is that it helps us to work with simple representations. Let $\mathbb{R} := \mathfrak{C}\mathbb{Q}$. Then to specify a function from $\mathbb{R} \to \mathbb{R}$, one can simply define a uniformly continuous function $f : \mathbb{Q} \to \mathbb{R}$, and then $\check{f} : \mathbb{R} \to \mathbb{R}$ is the required function. Hence, the completion monad allows us to do in a structured way what was already folklore in constructive mathematics: to work with simple, often decidable, approximations to continuous objects.

Having completed our short discussion of the analysis library we now turn to the concrete topic of numerical integration. To see why we want to use a better numerical integration, consider our previous implementation of Riemann integration [13] which computes three decimals of $\int_0^1 \mathsf{sin}$ in 7 seconds. To make this implementation more efficient we would need to port it to the new implementation of real numbers which was developed in task 4.1. It seems better to take this opportunity to also implement a more efficient algorithm, especially since we will iterate integration to obtain a fixed point of the Picard operator.

2 Numerical Integration

The most naive, but very general, algorithm for numerical integration is Riemann integration. The algorithm works for all continuous functions on, say, the real line. A more sophisticated algorithm uses the approximation of a differentiable function by a polynomial. The Newton-Cotes formulas use this method. The correctness of this method depends on the error term for Lagrange interpolation, one of the basic results in numerical integration. The method of proof is illustrative for many proofs in numerical analysis. Hence, it is important to obtain a smooth (constructive) proof of this fact. Here we provide an informative proof using the Hermite-Genocchi formula. We believe these techniques are illustrative for the problems that arise in a constructive treatment of numerical analysis. This constructive treatment is the basis for an integrated implementation and verification of the algorithm in Coq. Our Haskell prototype for integration is efficient. Based on the experience in [9], we expect our algorithm to run efficiently inside the Coq system too.

Rolle's theorem is a key method for error analysis in numerical analysis. To obtain an integrated algorithm and correctness proof we need to understand its constructive status. We recall that the double negation of the axiom of countable choice is not constructively provable. Hence a simple use of the double negation monad in Coq, as proposed by O'Connor [12], does not suffice. As usual, an approximate version is readily obtained, but seems more elaborate than the classical proof. Finally, the use of divided differences gives a satisfying constructive account which is arguably superior to the classical development using Rolle's theorem.

2.1 An axiomatic treatment of integration.

There are two Coq formalizations of the integral. The corn formalization closely follows Bishop's treatment of the Riemann integral. As argued by Dieudonné, it seems better to treat the Cauchy integral (only continuous functions) and develop the full theory of Lebesgue integration when we need to go further. This is roughly the approach taken by Spitters and O'Connor [13] who developed Cauchy integration theory for $C[0,1]$. Here we define the integral for locally uniformly continuous functions $\mathbb{R} \to \mathbb{R}$ with an abstract specification similar to the one by Bridger [3, Ch5.].

```
Class Integrable `{!Integral f}: Prop :=
  { integral_additive:
    forall (a: Q) b c, ∫f a b + ∫f (a+` b) c == ∫f a (b+c)%Qnn
  ; integral_bounded_prim: forall (from: Q) (width: Qpos) (mid: Q) (r: Qpos),
    (forall x, from <= x <= from+width → ball r (f x) ('mid)) →
    ball (width * r) ∫( f from width) (' (width * mid))
  ; integral_wd:> Proper (Qeq ==> QnonNeg.eq ==> @st_eq CRasCSetoid) ∫( f) }.
```

This specification is complete in that it uniquely characterizes the integral. We provide a reference implementation using the technology of type classes.

2.2 An axiomatic treatment of the integral of a function.

We provide a type class for the integral of a partial function defined on an interval. This allows us to apply tailor-made optimizations to numerical integration. E.g. using anti-differentiation whenever possible. By defining new instances of this type class, we can optimize performance. This use of type classes is reminiscent of the selection of the optimal method for an operation in GAP based on logical information that GAP4 has on an object. Specialization in generic Haskell also allows more efficient implementations of generic functions. However, these specializations are bound to a given type. Coq's type classes seem to provide more flexibility in this respect.

2.3 Iterated integrals.

We prepare for the use of iterated integrals.

Definition 2.1 A function is *locally uniformly continuous* (LUC) if it is uniformly continuous on each ball. A function is *locally Lipschitz* (LL) if it is Lipschitz on each ball.

Lemma 2.1 *Locally Lipschitz functions form a category.*

Proof. We prove that $f \circ g$ is LL. Suppose that $x \in B_\epsilon(y)$, choose a Lipschitz constant M for g on this ball. Then $gx \in B_{M\epsilon}(gy)$. Choose a Lipschitz constant L for f on this ball. Then $fgx \in B_{LM\epsilon}(fgy)$. $\square$

It is unclear to us whether LUC functions form a category.

Lemma 2.2 *If $f : \mathbb{R}^m \to X$ and $g : \mathbb{R}^m \to Y$ LL(LUC), then $(f, g) : \mathbb{R}^m \to X \times Y$ LL(LUC).*

Lemma 2.3 *Let $f : \mathbb{R}^{m+1} \to \mathbb{R}$ and $g, h : \mathbb{R}^m \to [0,1]$ be LL. Then $\lambda\vec{y}. \int_{g(\vec{y})}^{h(\vec{y})} f(x, \vec{y})dx$ is LL.*

Proof. It suffices to prove that $\lambda\vec{y}. \int_0^{h(\vec{y})} f(x, \vec{y})dx$ is LL, by linearity.

$\lambda\vec{y}. \int_0^{h(\vec{y})} f(x, \vec{y})$ is $\int_0^- \circ((\mathrm{curry} f)h) \circ \mathrm{diag}$, where the curry of f is $\lambda\vec{y}.\lambda x.f(x, \vec{y}) : \mathbb{R}^m \to C([0,1], \mathbb{R})$ and

$$\int_0^- : [0,1] \times C([0,1], \mathbb{R}) \to \mathbb{R}.$$

This last function is LL. The result follows from Lemma 2.2. □

As a corollary, we can now perform iterated integration, as needed in Theorem 2.17. Defining the iterated integral in Coq turned out to be quite difficult as we need to simultaneously define the new function and the proof that the result is again integrable. Following the proof above works by using vectors defining a reduction function as in the previous lemma.

2.4 Lagrange interpolation

Standard proofs of the Lagrange interpolation formula use the generalized Rolle's theorem.

Theorem 2.4 [Classical Rolle's theorem] *Let f be differentiable and have two zeroes in an interval $[a, b]$. Then f' has a zero in (a, b).*

Theorem 2.5 [Classical generalized Rolle's theorem] *Let f be n times differentiable and have $n + 1$ zeroes in an interval $[a, b]$. Then $f^{(n)}$ has a zero in $[a, b]$.*

Proof. We only treat the case $n = 2$. For concreteness, we assume that $f = 0$ in 0, 1 and 2. By Rolle's theorem $f' = 0$ in $a \in (0, 1)$ and $b \in (1, 2)$. By applying Rolle's theorem to f' we complete the proof. □

We define the Lagrange polynomial.

Definition 2.2 Let $x_1, \ldots, x_n$ be n distinct real numbers and $y_1, \ldots, y_n \in \mathbb{R}$, then a unique polynomial P_n of degree at most $n - 1$ exists with $y_k = P_n(x_k)$, for each $k = 1, ..., n$. This polynomial is called the *Lagrange polynomial*. Explicitly, $P_n(x) := \sum_j y_j \prod_{i,j \neq i} \frac{x - x_i}{x_j - x_i}$. Typically, $\vec{y}$ will be map $f\vec{x}$ for some $f : \mathbb{R} \to \mathbb{R}$.

```
Definition L: cpoly CRasCRing := Σ(map (fun p ⇒ let '((x, y), rest) := p in
  _C_ y ∗ Π(map (fun xy' ⇒ (' (− fst xy') +X∗ 1) ∗ _C_ (' (/ (x − fst xy')))) rest))
  (separates qpoints)).
```

Here _C_ y is the constant polynomial, +X∗ is a constructor for polynomials and ' is the injection of $\mathbb{Q}$ into $\mathbb{R}$. The list $(\vec{x}, \vec{y})$ of points is called qpoints.

Theorem 2.6 (Lagrange error formula) *Let f be n times differentiable. Then*

$$|f(x) - P_n(x)| \leqslant \frac{\prod(x - x_k)}{n!} \sup |f^{(n)}| \quad \textit{for all } x.$$

The proof uses the generalized Rolle's theorem. Rolle's theorem is not constructively provable. However, an ε-version of it *is* provable [2].

Proposition 2.7 *Let f be continuously differentiable and $f(a) = f(b) = 0$. Then there exists $x \in (a, b)$ such that $|f'(x)| \leq \varepsilon$.*

This ε-version can be used to prove an ε-version of the generalized Rolle's theorem by constructing a, b as in the proof of Theorem 2.5, but such that $f(a)$ and $f(b)$ are small and then continuing the proof using the mean value theorem. We will not pursue this direction, but instead develop a smoother presentation using divided differences.

Below is a the outline of a new (constructive) proof the Newton-Cotes formula and a proof with formal statements interspersed. *We are ready to state the Hermite-Genocchi theorem, but we have not finished the proof yet.* See Section 2.8 for a precise description.

2.5 Polynomials

In order to proceed we need a theory about interpolation polynomials. The constructive theory of (interpolation) polynomials is somewhat complicated since $\mathbb{R}[X]$ is not a Euclidean ring: there is no zero test. We could try to extend the Lagrange polynomial to $\mathbb{R}$. Instead, we restrict to a decidable subfield of the reals, such as $\mathbb{Q}$. This allows us to define the reciprocal function as a total function with a dummy variable at 0, thus avoiding a partial function as a function depending on a proof (of apartness from 0). Then we show in Section 2.7 that the divided difference $C^n[a, b] \to C[a, b]$ is continuous for all n. As a consequence, it can be extended to $\mathbb{R}^n$. Alternatively, we use equation 5 as a definition for the divided difference.

2.6 Continuity

As explained above, it is often convenient to for implement a function on the rationals and then lift it to the real numbers. The completion monad facilitates this [11]. The completion monad is a monad on the category of metric spaces with uniformly continuous mappings. However, many functions are only locally uniformly continuous. The exponential function is an example. One can prove, in the meta-logic, that all continuous functions on locally compact metric spaces are locally uniformly continuous. However, for formalization purposes this seems somewhat heavy. Moreover, proving that such locally uniformly continuous functions form a category is again problematic. It is impossible in Bishop-style mathematics. It is unclear to us whether locally uniformly continuous functions on metric spaces compose; i.e. whether they form a category. Fortunately, locally Lipschitz functions form a category which includes all the differentiable functions.

We defined all these classes of continuous functions as type classes and defined instances to derive the inclusions between the classes. Moreover, we automatically prove functions to be continuous as follows. We use type-class quoting [15] to first obtain a point-free version of a λ-term. For instance, $\lambda x.f(gx)$ becomes $f \circ g$. We then use type class instance search to automatically prove that this composition is, in fact, continuous.

2.7 Hermite-Genocchi formula

A constructive and informative substitute for Rolle's Theorem is given by the Hermite-Genocchi formula. We will provide a closed formula for the error $f - P_n f$ in Formula 4.

We consider a set of $k+1$ points $x_0, ..., x_k$ where no two x_j are the same. As stated in Section 2.5, we have chosen to restrict to $\mathbb{Q}$ and hence we can use Coq's nodup lists, lists without a separate proof of no redundancy.

Let $f : \mathbb{Q} \to \mathbb{Q}$. The interpolation polynomial in the Newton form is a linear combination of Newton basis polynomials

$$N(x) := \sum_{j=0}^{k} a_j n_j(x)$$

with the Newton basis polynomials defined as

$$n_j(x) := \prod_{i=0}^{j-1} (x - x_i)$$

and the coefficients defined as $a_j := f[x_j, ..., x_0]$, where $f[x_j, ..., x_0]$ is the notation for *divided differences*, defined recursively by:

$$f[a] = f(a)$$

$$f[a, b] = \frac{f(a) - f(b)}{a - b}$$

$$f[a, b, c] = \frac{f[a, c] - f[b, c]}{a - b}$$

and in general, $f[a : b : l] := \frac{f[a:l] - f[b:l]}{a-b}$. Here $a : l$ denotes the concatenation of an element to a list, the list may be empty.

Thus the Newton polynomial can be written as

$$N(x) := f[x_0] + f[x_1, x_0](x - x_0) + \cdots + f[x_k, ..., x_0](x - x_0)(x - x_1) \cdots (x - x_{k-1}).$$

In code:

```
Fixpoint divdiff_l (a: QPoint) (xs: list QPoint): CR :=
match xs with
  | nil ⇒ snd a
  | cons b l ⇒ (divdiff_l a l − divdiff_l b l) ∗ ' / (fst a − fst b)
end.
Definition divdiff (l: ne_list QPoint): CR := divdiff_l (ne_list.head l) (ne_list.tail l).
Let an (xs: ne_list QPoint): cpoly CRasCRing :=
 _C_ (divdiff xs) ∗ Π(map (fun x ⇒ ' (− fst x) +X∗ 1) (tl xs)).
Definition N (qpoints: ne_list QPoint): cpoly CRasCRing := Σ(map an (ne_list.tails qpoints)).
```

The Newton polynomial is a polynomial of minimal degree which interpolates the list qpoints.

```
Lemma interpolates_economically: interpolates_economically crpoints (N qpoints).
```

And similarly for the Lagrange polynomial. Since such polynomials are unique,

```
Lemma interpolation_unique (l: ne_list (Q ∗ Q)): NoDup (map fst l) →
 forall p q: cpoly Q,
  interpolates_economically l p → interpolates_economically l q → p [=] q.
```

the Newton polynomial coincides with the Lagrange polynomial.

Corollary 2.8 *The divided difference $f[a_n, \ldots, a_1]$ is the coefficient of x^n in the (Newton) polynomial that interpolates f at $a_1, \ldots, a_n$.*
Lemma N_leading_coefficient: nth_coeff (length (tl qpoints)) (N qpoints) == divdiff qpoints.

An interpolating polynomial for a list l also interpolates its permutations:

Corollary 2.9 (Ampere) *The function $f[l]$ is symmetric in the variables.*
Lemma divdiff_Permutation (x y: ne_list QPoint): QNoDup (map fst x) →
ne_list.Permutation x y → divdiff x [=] divdiff y.

We could have made this into a Proper by packing the QNoDup into a Σ-type, but we have explained our reservations about packing before. Alternatively, we could have defined divdiff to first check for duplications and then return 0, or even, have divdiff return an Option Q.

2.8 Derivative

We follow Bridger and Stolzenberg [4] in defining f to be n-times differentiable if the divided difference in $n+1$ variables is locally-uniformly continuous.

Proposition 2.10 *The divided difference is linear and satisfies the Leibniz rule:*

$$\begin{aligned}(f+g)[x_n, \ldots, x_0] &= f[x_n, \ldots, x_0] + g[x_n, \ldots, x_0];\\ (\lambda \cdot f)[x_n, \ldots, x_0] &= \lambda \cdot f[x_n, \ldots, x_0];\\ (f \cdot g)[x_n, \ldots, x_0] &= f[x_0] \cdot g[x_n, \ldots, x_0] + \ldots + f[x_n, \ldots, x_0] \cdot g[x_n].\end{aligned}$$

The chain rule:

$$f \circ g[x, y] = f[g(x), g(y)]g[x, y]$$

The proofs use a custom induction principle on non-empty lists:

```
Definition divdiff_ind {T} (P: ne_list T → Prop)
 (Pone: ∀ p, P (ne_list.one p))(Ptwo: ∀ p q, P (p ::: ne_list.one q))
 (Pmore: ∀ a b l, P (a ::: l) → P (b ::: l) → P (a ::: b ::: l)):forall l, P l.
```

The following proof of the Law of bounded change seems novel.

Lemma 2.11 *Suppose that $f' \geq 0$ on I. Then for all x, y in I, $f[x, y] \geq 0$.*

Proof. We first observe that for each m,

$$f[x, y] = \frac{1}{m+1}(\sum_{i=0}^{m} f[x + it, x + (i+1)t]),$$

where $t := \frac{y-x}{m}$. Suppose that $f[x, y] < -\epsilon$. Then $f[x', y'] < -\epsilon$ for some x', y' such that $|x' - y'| \leq \mu(\epsilon/2)$. A contradiction with the continuity of the divided difference. □

Corollary 2.12 (Law of Bounded change) *Suppose that $A \leq f' \leq B$ on I. Then $A \leq f[-, -] \leq B$ on I^2.*

Theorem 2.13 (Fundamental Theorem of Calculus) *If g is locally uniformly continuous, then $G(x) = \int_a^x g$ is differentiable with $G' = g$.*

Proof. $G(y)-G(x)$ equals the integral of g from x to y. For each $\epsilon > 0$, if $|y-x| \leq \mu(\epsilon)$, every value of g at a point in $[x, y]$ is within ϵ of $g(x)$. But then, $-\epsilon(y-x) \leq \int_x^y |g-g(x)| \leq \epsilon(y-x)$ by the law of bounded change. □

This proof should be easier to formalize than Bishop's proof, the one used in the corn library.

Corollary 2.14 *If $f' = 0$ on I, then $f[-,-] = 0$ on I^2, so f is constant.*

Corollary 2.15 *Two anti-derivatives of a function differ by a constant.*

Theorem 2.16 (Change of variables) *Let $f : \mathbb{R} \to \mathbb{R}$ be continuous and g differentiable. Then $\int_a^b (f \circ g)g' = \int_{g(a)}^{g(b)} f$.*

Proof. Define $H(x) := \int_{g(a)}^x f$. By the chain rule: $(H \circ g)' = (H' \circ g)g' = (f \circ g)g'$.

$$\begin{aligned}\int_a^b (f \circ g)g' &= \int_a^b (H \circ g)' = H(g(b)) - H(g(a)) = \\ &= \int_{g(a)}^{g(b)} H' = \int_{G(a)}^{G(b)} f\end{aligned}$$

□

Consider the following motivation which will only be made precise in Thm 2.17. We have,

$$f[a, b] = \int_0^1 f'(a + (b-a)t)\,\mathrm{dt}\,.$$

$$f[a, b, c] = \int_0^1 \int_0^u f''(a + (b-a)t + (c-b)u)dtdu \tag{1}$$

note that [6, p.64]

$$a + (b-a)t + (c-b)u = (1-t)a + (t-u)b + uc. \tag{2}$$

Moreover, $(1-t) + (t-u) + u = 1$.

The general formula can be put in a symmetric form

$$f[a_0, ..., a_n] = \int \int_n f^{(n)}(t_0 a_0 + ... + t_n a_n)dt_1 \cdots dt_n$$

with $t_0 + \cdots + t_n = 1$ and $0 \leqslant t_i \leqslant 1$. (Note the extra variable.) We use the more explicit presentation of this iterated integral.

Theorem 2.17

$$f[a_0,\ldots,a_n] \quad = \quad \int_0^1 \int_0^{1-t_1} \int_0^{1-(t_1+t_2)} \int_0^{1-(\sum_{j=1}^{n-1} t_j)} f^{(n)}(\sum t_i a_i) dt_n \cdots dt_1. \tag{3}$$

where $t_0 := 1 - (\sum_{j=1}^n t_j)$. *Let* $\int_n^S$ *denote this symmetric integral.*

Proof. This formula is proved by induction on n. We follow the proof by Alexander [1], but make the order of variables explicit.

For $n = 0$, there is nothing to prove.

For the inductive step consider

$$f[a_0,\ldots,a_{n+1}] = \frac{f[a_1,\ldots,a_{n+1}] - f[a_0,\ldots,a_n]}{a_{n+1} - a_0}.$$

By symmetry,

$$f[a_0,\ldots,a_{n+1}] = \frac{f[a_{n+1},a_1,\ldots,a_n] - f[a_0,\ldots,a_n]}{a_{n+1} - a_0}.$$

By induction, $f[a_0,\ldots,a_{n+1}]$ equals

$$\begin{aligned} \tfrac{1}{a_{n+1}-a_0} \quad & \left(\int_0^1 \int_0^{1-(\sum_{j=1}^{n-1} t_j)} f^{(n)}(t_0 a_{n+1} + \cdots + t_n a_n) dt_n \cdots dt_1 \right. \\ - \quad & \left. \int_0^1 \int_0^{1-(\sum_{j=1}^{n-1} t_j)} f^{(n)}(t_0 a_0 + \cdots + t_n a_n) dt_n \cdots dt_1 \right), \end{aligned}$$

where $t_0 := 1 - (\sum_{j=1}^n t_j)$.

By the fundamental theorem of calculus this is

$$\tfrac{1}{a_{n+1}-a_0} \quad \int_0^1 \int_0^{1-(\sum_{j=1}^{n-1} t_j)} \int_{t_0 a_0 + \cdots + t_n a_n}^{t_0 a_{n+1} + \cdots + t_n a_n} f^{(n+1)}(\xi) d\xi dt_n \cdots dt_1.$$

Make the change of variable $\xi = t_0 a_0 + \cdots t_n a_n + t_{n+1}(a_{n+1} - a_0)$. Then for all g,

$$\int g d\xi = (a_{n+1} - a_0) \int_0^{t_0} g dt_{n+1}.$$

We conclude that

$$\int_0^1 \int_0^{1-(\sum_{j=1}^n t_j)} \int_0^{t_0} f^{(n+1)}(t_0 a_0 + \cdots + t_n a_n + t_{n+1}(a_{n+1} - a_0)) dt_{n+1} dt_n \cdots dt_1.$$

Hence,

$$\int_0^1 \int_0^{1-(\sum_{j=1}^n t_j)} f^{(n+1)}(\widehat{t_0} a_0 + \cdots + t_{n+1} a_{n+1}) dt_{n+1} dt_n \cdots dt_1,$$

where $\widehat{t_0} := 1 - (\sum_{j=1}^{n+1} t_j)$. $\square$

The Hermite-Genocchi formula is an integral formulation of the mean value theorem for divided differences.

Corollary 2.18 $\int_n^S 1dt_1 \cdots dt_{n-1} = \frac{1}{(n-1)!}$.

Proof. Choose $a_1, \ldots, a_n$ arbitrary, but distinct. Let $\vec{a}$ be the list $(0, a_1, \ldots, a_n)$. Define the polynomial $g_n := \frac{1}{(n-1)!}\prod(x - a_i)$. Then $g^{(n-1)} = 1$. The polynomial g_n is the Lagrange polynomial for the list $\vec{a}$, so $g_n[\vec{a}] = \frac{1}{(n-1)!}$. The corollary follows from Hermite-Genocchi. □

Lemma 2.19 *The function $f[a_1, \ldots, a_n]$ is continuous on I^n and $f[a, \ldots, a] = f^{(n-1)}(a)/n!$.*

Proof. We consider I^n with the metric $d((a_i), (b_i)) = \max_i |a_i - b_i|$.

Let $0 \leqslant t_i \leqslant 1$ and $\sum t_i = 1$, then

$$\sum t_i a_i - \sum t_i b_i = \sum t_i(a_i - b_i) \leq \sum_i t_i \max_i |a_i - b_i| = \max_i |a_i - b_i|.$$

This proves continuity.

By continuity, we extend $f[\ldots]$ to all, possibly non-distinct, sequences. Then

$$\begin{aligned} f[a, \ldots, a] &= \int\int_{n-1} f^{(n-1)}(a) du_1 \cdots du_{n-1} \\ &= f^{(n-1)}(a) \int\int_{n-1} 1 du_1 \cdots du_{n-1} \\ &= f^{(n-1)}(a) \frac{1}{(n-1)!}. \end{aligned}$$

□

Proposition 2.20 [6, p.63] *We have*

$$f(x) - P_n f(x) = \left(\prod_{i=1}^{n}(x - x_i)\right) f[x_1, \ldots, x_n, x] \tag{4}$$

and hence, by (3),

$$f(x) - P_n f(x) = \prod_{i=1}^{n}(x - x_i) \int_n^S f^{(n)}(t_1 x_1 + \ldots + t_n x_n + t_0 x) du_1 \cdots du_n \tag{5}$$

with $t_0 + \cdots + t_n = 1$ and $0 \leqslant t_i \leqslant 1$.

Proof. We prove by induction that

$$f[x_1, \ldots, x_n] = \sum_{j=1}^{n} \frac{f(x_j)}{\prod_{i, i \neq j}(x_j - x_i)}.$$

Also,

$$\begin{aligned}
\frac{f(x)-P_nf(x)}{\prod_{i=1}^n(x-x_i)} &= \frac{f(x)-\sum_j f(x_j)\prod_{i,j\neq i}\frac{x-x_j}{x_i-x_j}}{\prod_{i=1}^n(x-x_i)} \\
&= \frac{f(x)}{\prod_{i=1}^n(x-x_i)} - \frac{\sum_j f(x_j)\prod_{i,j\neq i}\frac{x-x_j}{x_i-x_j}}{\prod_{i=1}^n(x-x_i)} \\
&= \frac{f(x)}{\prod_{i=1}^n(x-x_i)} - \frac{\sum_j f(x_j)}{\prod_{i,j\neq i}(x_i-x_j)(x-x_i)} \\
&\overset{[x_{n+1}:=x]}{=} \sum_{j=1}^{n+1}\frac{f(x_j)}{\prod_{i,i\neq j}(x_j-x_i)}.
\end{aligned}$$

□

Corollary 2.21 $|f(x)-P_n(x)| \leqslant \frac{\prod(x-x_k)}{n!}\|f^{(n)}\|$

Proof. We write $u_0 := x$. By Corollary 2.18 and Formula 5,

$$|f(x)-P_nf(x)| = \left|\int_{n+1}^S f^{(n)}(u_0x_0+...+u_nx_n)du_0\cdots du_n\right|\prod_{i=1}^n(x-x_i) \leqslant \frac{\|f^{(n)}\|}{n!}\prod_{i=1}^n(x-x_i).$$

□

This allows us to proof the error estimates for the Newton-Cotes formulas. In particular, we can do so for Simpson's rule.

Corollary 2.22 [Simpson's rule]

$$\left|\int_a^b f(x)\,dx - \frac{b-a}{6}\left[f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right]\right| \leqslant \frac{(b-a)^5}{2880}\|F^{(4)}\|.$$

Proof. The right handside is the integral of the lagrange polynomial P_3 at $a, \frac{a+b}{2}, b$; see [7]. For the error we adopt the proof in [16], but replace the use of Rolle's theorem and the Mean Value Theorem by the Hermite-Genocchi formula. Define $F(t) := f(\frac{a+b}{2}+\frac{b-a}{2}t)$. This reduces the problem to showing that $|\int_{-1}^1 F(\tau)d\tau - \frac{1}{3}(F(-1)+4F(0)+F(1)]| \leqslant N/90$, where $N := \|F^{(4)}\|$.

Define

$$G(t) = \int_{-t}^t F(\tau)d\tau - \frac{t}{3}(F(-t)+4F(0)+F(t)]$$

We need to prove that $90G(1) \leqslant N$. To do so, define $H(t) := G(t) - t^5G(1)$. Then

$$H(0) = H(1) = H'(0) = H''(0) = 0.$$

Hence, $H[0,0,0,1] = -(H[0,0,0]-H[0,0,1]) = 0 + (-H[0,0]+H[0,1]) = 0$.

Moreover, $H^{(3)}(t) = -\frac{t}{3}(F^{(3)}(t)-F^{(3)}(-t)) - 60t^2G(1) = -\frac{t}{3}(\int_{-t}^t F^{(4)}) - 60t^2G(1)$.

This shows that

$$\begin{aligned}
0 = H[0,0,0,1] &= \int_0^1 H^{(3)} \\
&= \int_0^1 -\frac{t}{3}(\int_{-t}^{t} F^{(4)}) - 60t^2 G(1) \\
&\geqslant \int_0^1 -\frac{t}{3} 2\,\mathrm{tN} - 60t^2 G(1) \\
&= -\frac{2}{3}(N + 90G(1)) \int_0^1 t^2 \\
&= -\frac{2}{3}(N + 90G(1))\frac{1}{3}.
\end{aligned}$$

Hence, $N \geqslant -90G(1)$. Similarly, $0 \leqslant -\frac{2}{9}(-N + 90G(1))$. Consequently, $90G(1) \leqslant N$. We conclude that $|90G(1)| \leqslant N$. $\square$

Proposition 2.23 [Composite Simpson rule] *For n even,*

$$\int_a^b f(x)\ \mathrm{dx} \approx \frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\right],$$

with error

$$\frac{h^4}{180}(b-a)\max_{\xi\in[a,b]} |f^{(4)}(\xi)|.$$

What has been implemented?

We have a definition of the Lagrange and the Newton polynomial and a proof that they are equal as polynomials[1]. We have formalized the Hermite-Genocchi formula[2], but not all proofs are complete yet.

We have implemented Picard iteration based on Riemann integration and are currently finishing the proofs. We test it on the sin function.

```
Definition Picard_raw (t:Q_as_MetricSpace) : CR := Integrate f 0 t.
Definition Picard := (Cbind QPrelengthSpace (Build_UniformlyContinuousFunction Picard_uc)).

Variable f:CR-->CR.

Fixpoint Picard_seq (n : nat) : Q_as_MetricSpace --> CR :=
        match n with
        | O ⇒ f ∘ Cunit
        | S m ⇒ (Picard (Picard_seq m) )∘  Cunit
        end.

Definition do_it (n:nat) := (Picard_seq (Cbind QPrelengthSpace sin_uc) n).
```

[1] https://github.com/spitters/ssrcorn/blob/master/algebra/CPoly_Newton.v https://github.com/spitters/ssrcorn/blob/master/algebra/CPoly_Lagrange.v

[2] https://github.com/spitters/ssrcorn/blob/master/algebra/DivDiff_RepeatedIntegral.v

Here Picard_uc is the proof that the Picard operator is uniformly continuous. We compute one correct decimal of the third iteration in 47 seconds and two correct decimals of the second Picard iteration in 98 seconds.

We have a preliminary implementation of the Simpson integral. It computes ten digits of $\int_0^1 \sin$ in 10 seconds. This should be compared to Riemann integration which gives only three digits in 7 seconds. When we apply the Picard opertion using Simpson integration we obtain two correct decimals of the third iteration in 3 seconds and five correct decimals of the second Picard iteration in 3 seconds. This shows that using Simpson integration is a good choice.

Conclusion

After working on maintainance of the corn library and starting a new library based on type classes, we have provided a proof outline of the formalization of numerical (Newton-Cotes) integration based on Hermite-Genocchi. We are ready to state the Hermite-Genocchi theorem, but we have not finished the proof yet. Once we have the numerical integration in place, the ODE-solver is within reach. The implementation and verification of Picard iteration can be based on earlier work on the geometric series. We already have a Haskell implementation of Newton iteration for exact reals which performs quite well. The correctness proof should be substantially easier than the one by Julien and Pasca [8] as we can avoid dealing explicitly with rounding, the exact reals take care of this for us! This leaves us to recreate a formal proof of about 3000 lines, hopefully reusing parts of Pasca's proof. As a second step we will abstract the Newton iteration to work on Banach spaces. We do not expect much problems here. We already have a general theory of metric spaces and completions. We also have a well-developed algebraic hierarchy. The two can be conveniently combined with our usage of type classes.

References

[1] R.K. Alexander. The Hermite-Genocchi formula.

[2] Errett Bishop and Douglas Bridges. *Constructive analysis*, volume 279 of *Grundlehren der Mathematischen Wissenschaften*. Springer-Verlag, 1985.

[3] Mark Bridger. *Real analysis*. Pure and Applied Mathematics (New York). Wiley-Interscience [John Wiley & Sons], 2007. A constructive approach.

[4] Mark Bridger and Gabriel Stolzenberg. Uniform calculus and the law of bounded change. *Amer. Math. Monthly*, 106(7):628–635, 1999.

[5] L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN, the Constructive Coq Repository at Nijmegen. In *MKM*, pages 88–103, 2004.

[6] Carl de Boor. Divided differences. *Surv. Approx. Theory*, 1:46–69, 2005.

[7] J. Douglas Faires and Richard Burden. *Numerical methods*. Brooks/Cole Publishing Co., Pacific Grove, CA, second edition, 1998.

[8] Nicolas Julien and Ioana Pasca. Formal Verification of Exact Computations Using Newton's Method. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 408–423, 2009.

[9] Robbert Krebbers and Bas Spitters. Computer certified efficient exact reals in Coq. 2011.

[10] E. Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23, 1989.

[11] Russell O'Connor. A Monadic, Functional Implementation of Real Numbers. *MSCS*, 17(1):129–159, 2007.

[12] Russell O'Connor. Classical mathematics for a constructive world. *CoRR*, abs/1008.1213, 2010.

[13] Russell O'Connor and Bas Spitters. A computer verified, monadic, functional implementation of the integral. *TCS*, 411(37):3386–3402, 2010.

[14] Fred Richman. Real numbers and other completions. *Mathematical Logic Quarterly*, 54(1):98–108, 2008.

[15] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *MSCS, special issue on "Interactive theorem proving and the formalization of mathematics"*, 2011.

[16] Endre Süli and David F. Mayers. *An introduction to numerical analysis*. Cambridge University Press, Cambridge, 2003.

[17] Philip Wadler. Comprehending monads. *MSCS*, 2(4):461–493, 1992.