Congruence closure in intensional type theory

Luis Scoccola
lscoccol@uwo.ca

University of Western Ontario

May 26, 2018

Goals

- Convince you that this is an important problem.
  Specifically, that the problem, or some variation of it, must be solved if we want to use a univalent proof assistant to do every-day math.
- Survey solutions that work in other contexts. In particular the solution of (Selsam, de Moura) in Lean.
- Propose an approach for the univalent case (work in progress).

Informal definitions

Given a denumerable set of variables and function symbols, consider relations on the set of words generated by the symbols.

Definition

A relation R satisfies **congruence** *if for all elements $x_1, \cdots, x_n$, $y_1, \cdots, y_n$, and n-ary functions f, we have*

$$x_i \, R \, y_i \text{ for all } i, \text{ implies } f(x_1, \cdots, x_n) \, R \, f(y_1, \cdots, y_n).$$

Definition

A **congruence relation** *is an equivalence relation that satisfies congruence.*

Informal definitions (cont.)

Definition

Given a relation R, its **congruence closure** *R' is the smallest congruence relation containing R.*

Problem

Given a relation R, and words x, y, is $x\, R'\, y$?

Ackermann (1954) notices that (as long as the symbols and relation R are given explicitly enough) the problem is *decidable*. Several other authors[1] worked on efficient solutions.

[1] Downey, Sethy, Tarjan, Kozen, Shostak, Nelson, and Oppen.

Applications

Example (Verification of microprocessor control)

Burch, Dill (1994) show how to automatically verify microprocessor control using the logic of **Equality with Uninterpreted Functions** (a quantifier-free logic where equality is a congruence relation).

The idea is to verify that the implementation of an instruction in a pipelined processor is correct.

Applications (cont.)

Example (Automation in theorem proving)

When doing mathematics informally, we don't justify equalities such as $f(n+1) = f(1+n)$.

Formal proofs require a justification for every step.

We can try to automate as many of them as possible.

A congruence closure procedure automatically produces equalities that follow from the hypothesis.

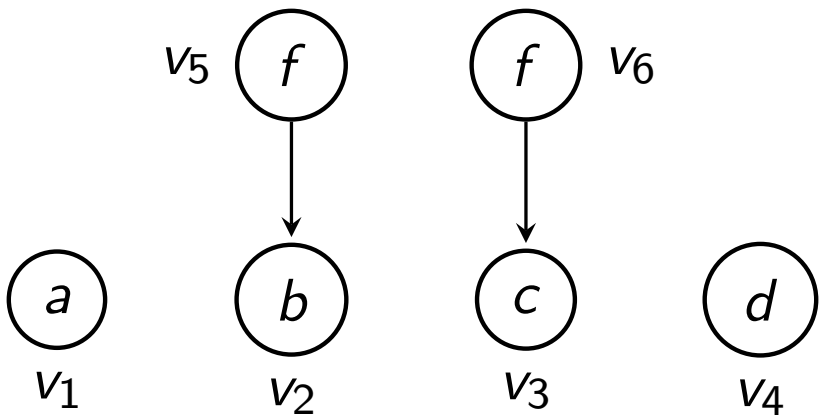
Let us give an idea of how such a procedure works.

Congruence closure of a relation on a graph

Example

Given $b = c$, $a = f(b)$, and $f(c) = d$. Then $a = d$?

Construct a labeled graph using the atomic subexpression in the hypothesis, with arrows indicating function application:



Set $R = \{(v_2, v_3), (v_1, v_5), (v_6, v_4)\}$.

Then $v_1 \, R' \, v_4$, by an inductive argument (algorithm).

Proof producing congruence closure

In many applications (including ours) just knowing that two terms are related is not enough. We need a witness of this fact.

Nieuwenhuis, Oliveras (2005), extend union-find data structures to support an **Explain** operation.

For example, if from $H_1 : a = b$, $H_2 : b = c$, $H_3 : c = d$, we deduce $T : a = c$, $\mathit{explain}(T)$ must mention the hypothesis H_1 and H_2, but *not* H_3.

Equality in dependent type theory

Definition (Definitional equality)

Two terms are **definitionally equal** *if they have the same normal form.*

Problem: $n + 1$ and $1 + n$ are *not* definitionally equal, since if n is a variable, the terms are in (a different) normal form.

Equality in dependent type theory (cont.)

Definition

The type of **propositional equalities** *is the inductive family*

$$\mathsf{Id} : (A : \mathcal{U}) \to A \to A \to \mathcal{U}$$

with one constructor $\mathsf{refl}_A : (a : A) \to \mathsf{Id}_A(a, a)$.

Only terms with the same type can be compared.

The type $\mathsf{Id}_A(x, y)$ is also denoted by $x =_A y$.

Example

One can prove by induction on n, that

$$(n : \mathbb{N}) \to n + 1 =_{\mathbb{N}} 1 + n.$$

Congruence in dependent type theory

It is easy to construct functions:

$$\mathsf{inv} : x = y \to y = x$$
$$\mathsf{concat} : x = y \to y = z \to x = z$$

So propositional equality is an equivalence relation on terms.

Also:

$$\mathsf{congr} : (f : A \to B) \to (x =_A y) \to f(x) =_B f(y).$$

So for non-dependent functions, equality is a congruence relation.

For example, Coq has a `congruence` tactic that works in the simply-typed fragment of the theory.

Heterogeneous equality

What about dependent types?

Definition

The type of **vectors of elements of A**, $\mathrm{vec}_A : \mathbb{N} \to \mathcal{U}$, *is the inductive family with contructors:*

$$\mathrm{nil} : \mathrm{vec}_A(0)$$
$$\mathrm{cons} : (n : \mathbb{N}) \to A \to \mathrm{vec}_A(n) \to \mathrm{vec}_A(\mathrm{succ}(n))$$

We can define by induction:

$$\mathrm{repeat} : (n : \mathbb{N}) \to A \to \mathrm{vec}_A(n)$$

Given $a : A$ and $e : n =_{\mathbb{N}} m$, we cannot even state

$$\mathrm{repeat}(n, a) =_? \mathrm{repeat}(m, a).$$

Heterogeneous equality (cont.)

We have a similar problem when trying to prove associativity of

$$\_ +\!\!+ \_ : \mathsf{vec}_A(n) \to \mathsf{vec}_A(m) \to \mathsf{vec}_A(n+m).$$

One solution (McBride):

Definition

The type of **heterogeneous equalities** *is the inductive family*

$$\mathsf{heq} : (A, A' : \mathcal{U}) \to A \to A' \to \mathcal{U}$$

generated by the constructor

$$\mathsf{refl} : (A : \mathcal{U}) \to (a : A) \to \mathsf{heq}(a, a).$$

Heterogeneous equality (cont.)

This is Lean's approach (Selsam, de Moura).

Writing $==$ for heq, induction proves:

$$n = m \to \mathsf{repeat}(n, a) == \mathsf{repeat}(m, a),$$
$$(v +\!\!+ w) +\!\!+ x == v +\!\!+ (w +\!\!+ x)$$

But $==$ is not a congruence relation, in general. Given $f : A \to B$, we cannot prove

$$\mathsf{congr}_f : (x, y : A) \to (x == y) \to f(x) == f(y).$$

The problem is that if $x, y : A$, we cannot prove

$$\mathtt{ofheq} : x == y \to x =_A y.$$

Heterogeneous equality (cont.)

In Lean, `ofheq` is an axiom. And using it they prove congruence lemmas such as:

$$\begin{aligned}\texttt{hcongr}_1\ & (A_1 : \mathcal{U})\ (B : A_1 \to \mathcal{U}) \\ & (f, g : (a_1 : A_1) \to B(a_1))\ (f = g) \\ & (a_1, b_1 : A_1)\ (a_1 == b_1) : \\ & \qquad f(a_1) == g(b_1).\end{aligned}$$

and all its higher dimensional analogues.

Using a variation of the proof-producing congruence closure of Nieuwenhuis and Oliveras, they get the full congruence closure procedure.

Heterogeneous equality (cont.)

This works well, but `ofheq` implies UIP.

In fact, `ofheq` implies that the universe $\mathcal{U}$ is a set, in the following sense.

Example

Given $e : A =_{\mathcal{U}} B$ and $a : A$, we have $a == \mathrm{transport}^{X \mapsto X}(e, a)$, by path induction. In particular, if $e : A =_{\mathcal{U}} A$, using `ofheq`, we have

$$a = \mathrm{transport}^{X \mapsto X}(e, a).$$

So every coercion $e : A =_{\mathcal{U}} A$ transports trivially.

Inconsistent with Univalence.

Congruence closure in univalent type theory?

What can be done in a univalent type theory?

We need congruence lemmas that are provable without assuming any classicality axioms, and useful for a proof-producing congruence closure procedure.

We used **pathovers** (and their higher dimensional generalizations to arbitrary type families with multiple arguments).

Path over path

Definition

Given a type $B : \mathcal{U}$ and a type family $X : B \to \mathcal{U}$, the type family

$$\text{pathover} : (b, b' : B) \to (b = b') \to X(b) \to X(b') \to \mathcal{U}$$

is defined by path induction.

Notice that we don't need a new inductive type.

We write $x =_{\langle e \rangle} x'$ instead of $\text{pathover}(b, b', e, x, x')$.

Path over path, over path . . .

In general, pathovers are not enough, we need paths over paths, over paths . . .

Example

Given $A_1 : \mathcal{U}$, $A_2 : A_1 \to \mathcal{U}$, and $A_3 : (a_1 : A_1) \to A_2(a_1) \to \mathcal{U}$, we need:

$$a_1, a_1' : A_1, \quad a_2 : A_2(a_1), \quad a_2' : A_2(a_1'),$$
$$a_3 : A_3(a_1, a_2), \quad a_3' : A_3(a_1', a_2'),$$
$$e_1 : a_1 = a_1', \quad e_2 : a_2 =_{\langle e_1 \rangle} a_2' \quad \vdash \quad a_3 =_{\langle e_1, e_2 \rangle} a_3' : \mathcal{U}.$$

Again, this type can be defined by path induction.

The congruence lemma

Contribution (S. - Vajjha)

A pathover type for each dependent family, that avoids unnecessary dependencies. Implemented as a tactic in Lean 3.

We will see what we mean by unnecessary dependencies.

Main contribution (S.)

A congruence lemma for every dependent function. Implemented as a tactic in Lean 3.

This lets us prove, for example, basic properties of vectors, without explicit coercions or path induction. For example, associativity of concatenation.

The congruence lemma (cont.)

Example

The congruence lemma for

$$\mathsf{cons} : (n : \mathbb{N}) \to A \to \mathsf{vec}_A(n) \to \mathsf{vec}_A(\mathsf{succ}(n)),$$

has type

$$\begin{aligned}\mathsf{congr}_{\mathsf{cons}}\ & (n, m : \mathbb{N})\ (x, y : A)\ (xs : \mathsf{vec}_A(n))\ (ys : \mathsf{vec}_A(m)) \\ & (e_1 : n = m)\ (e_2 : x = y)\ (e_3 : xs =_{\langle e_1 \rangle} ys) : \\ & \quad \mathsf{cons}(n, x, xs) =_{\langle \mathsf{congr}_{\mathsf{succ}}(e_1) \rangle} \mathsf{cons}(m, y, ys)\end{aligned}$$

where $\mathsf{congr}_{\mathsf{succ}} : (n, m : \mathbb{N}) \to (n = m) \to \mathsf{succ}(n) = \mathsf{succ}(m)$.

Implementation

We proceed in two steps.

- Interpret the function as a context morphism.
- Characterize the identity type of the domain and codomain of the context morphism.

Example (Interpret the function as a context morphism)

cons is a context morphism living over the context morphism succ:

$$\begin{array}{ccc} (n : \mathbb{N}).(x : A, xs : \mathrm{vec}_A(n)) & \xrightarrow{\text{succ.cons}} & (n : \mathbb{N}).(xs : \mathrm{vec}_A(n)) \\ \downarrow & & \downarrow \\ (n : \mathbb{N}) & \xrightarrow{\text{succ}} & (n : \mathbb{N}) \end{array}$$

Implementation (cont.)

Then, we must characterize the identity types of contexts in a usable way. That is, avoiding unnecessary dependencies.

Example (Characterization of identity context of a context)

For the identity context of $(n : \mathbb{N}, x : A, xs : \mathrm{vec}_A(n))$ it is better to use

$$(e_1 : n = m, e_2 : x = y, e_3 : xs =_{\langle e_1 \rangle} ys)$$

rather than

$$(e_1 : n = m, e_2 : x =_{\langle e_1 \rangle} y, e_3 : xs =_{\langle e_1, e_2 \rangle} ys).$$

Implementation (cont.)

To generate these identity contexts, we represent contexts as inverse diagrams. For example $(a_1 : A_1, a_2 : A_2(a_1), a_3 : A_3(a_1, a_2))$ is represented by

$$\begin{array}{c} A_3 \\ \downarrow \\ A_2 \\ \downarrow \\ A_1 \end{array}$$

whereas $(a_1 : A_1, a_2 : A_2(a_1), a_3 : A_3(a_1))$ is represented by

$$\begin{array}{ccc} A_2 & & A_3 \\ & \searrow \quad \swarrow & \\ & A_1 & \end{array}$$

Identity types of structures

This characterization of identity types has other uses.

Contribution (S.)

A useful characterization of identity types of structures and iterated sigmas.

Example

The context of pointed magmas is $(G : \mathcal{U}, g : G, o : G \to G \to G)$. Then, the context of equalities between (G, g, o) and (G', g', o') is

$$\left(e_1 : G = G', e_2 : g =_{\langle e_1 \rangle} g', e_3 : o =_{\langle e_1 \rangle} o'\right).$$

Again we are avoiding superfluous transports.

To-do list

Automatically apply Univalence and Function Extensionality.

Example

In the previous example, we would like to have:

$$\Big(e_1 : G \simeq G', e_2 : e_1(g) = g',$$
$$e_3 : \prod_{j,k:G} e_1(o(j,k)) = o'(e_1(j), e_1(k))\Big).$$

To-do list (cont.)

Take advantage of indexing types being sets.

Example

Given a type family $A_3 : (a_1 : A_1) \to (a_2 : A_2(A_1)) \to \mathcal{U}$, let

$$(a_3 ==_{A_3} a_3') :\equiv \sum_{e_1 : a_1 = a_1'} \sum_{e_2 : a_2 =_{\langle e_1 \rangle} a_2'} (a_3 =_{\langle e_1, e_2 \rangle} a_3').$$

Then if A_1 and A_2 are valued in sets

$$(e_1 : a_1 = a_1', e_2 : a_2 =_{\langle e_1 \rangle} a_2', e_3 : a_3 =_{\langle e_1, e_2 \rangle} a_3')$$

is equivalent to

$$(e_1 : a_1 = a_1', e_2 : a_2 ==_{A_2} a_2', e_3 : a_3 ==_{A_3} a_3')$$

To-do list (cont.)

Example

The identity context of the domain of cons is equivalent to

$$(e_1 : n = m, e_2 : x = y, e_3 : xs ==_{\mathsf{vec}_A} ys).$$

We don't have to keep track of the paths "under".

We can recover the congruence lemma of (Selsam, de Moura) without assuming UIP for all types, but just for the indexing types.

To-do list (cont.)

- Implement the **full congruence closure procedure**.
- Get a **characterization of identity types of inductive types** (useful to obtain decidability of their equality).

Other approaches?

In (some) **cubical type theories** the congruence of f, an n-ary dependent function, can be proven simply by:

$$\lambda p_1, \ldots, p_n . \langle i \rangle f(p_1 @ i, \ldots, p_n @ i),$$

avoiding path induction.
Does this help in any way? Might solve some coherence problems with the order in which we do path induction.

If we only care about congruence closure for families depending on sets, can a **two level type theory** simplify things, for example, by avoiding transports along the strict equality?

Thank you for listening!