Une procédure de décision pour relations d'équivalence

Sébastien Michelland

sous la supervision de

Pierre Corbineau, Lionel Rieg, et Karine Altisen

Stage de recherche de M2

Janvier – Juin 2020

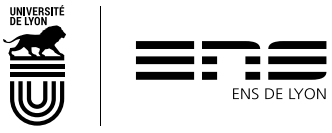


Table des matières

1 Introduction

Décider l'égalité dans un assistant de preuve

Pour assurer la fiabilité des théories et systèmes critiques, il est désormais courant de prouver formellement leurs propriétés dans un assistant de preuve. La vérification par ordinateur des théorèmes permet en effet de traquer systématiquement toute erreur de raisonnement.

Dans ce processus rigoureux de preuve, on souhaite autant que possible automatiser les étapes de raisonnement élémentaires pour concentrer l'effort de l'utilisateur sur le cœur de l'argument.

Un problème commun qui se prête bien à l'automatisation est la déduction d'égalités entre des objets. Dans ce problème, on dispose comme hypothèses d'égalités connues $a_1 = b_1, ... \, a_n = b_n$, et on souhaite en déduire une nouvelle égalité $a = b$. On trouvera par exemple que l'égalité

$$f(e, x) = x, \; g(e) = x \; \vdash \; f(e, g(e)) = x$$

est déductible par "réécriture" de $g(e) = x$ dans la première hypothèse.

Formellement, il s'agit du problème de décision de l'égalité dans une algèbre de termes. Lorsque les fonctions ne sont pas interprétées, il est possible de décrire simplement toutes les égalités déductibles à partir des hypothèses : elles sont obtenues par réflexivité, symétrie et transitivité de l'égalité, plus l'utilisation de l'égalité extensionnelle entre fonctions (définie pour deux fonctions f et g par $\forall x, f(x) = g(x)$).

Cette dernière règle est parfois appelée *congruence*, et a donné son nom aux algorithmes de *clôture par congruence* qui décident l'égalité dans une algèbre de termes en saturant l'ensemble des égalités connues. Le plus célèbre peut-être est celui de Downey, Sethi et Tarjan [DST80], qui décide en temps quasi-linéaire.

Ce sujet est naturellement d'intérêt pour les équipes PACSS (*Proofs and Code analysis for Safety and Security*, où travaille Pierre Corbineau) et Synchrone (où travaillent Lionel Rieg et Karine Altisen) du laboratoire Verimag à l'Université Grenoble-Alpes. Ces équipes abordent plusieurs thématiques liées à la vérification et la sécurité, comme la preuve formelle d'algorithmes distribués, la compilation certifiée, la conception d'un microcontrôleur sécurisé, et la certification d'un système d'exploitation temps réel.

L'algorithme de clôture par congruence a été implémenté dans Coq par Pierre Corbineau sous la forme de la tactique `congruence` [Cor01], puis étendu pour supporter les constructions inductives [Cor06].

Cependant `congruence` s'arrête à l'égalité, même sur les objets où elle est moins naturelle comme les fonctions ou les propositions. De plus, il est courant en Coq moderne d'équiper les objets de relations d'équivalence personnalisées, et de les déclarer dans le système de *typeclasses* du langage. Ce système s'apparente à un annuaire, et peut être consulté à tout moment pour récupérer les relations déclarées. La tactique `congruence` ignore ce méchanisme, ce qui réduit davantage sa portée alors que l'algorithme reste pertinent dans ce nouveau contexte.

Contribution de ce stage

Dans ce stage, on se propose dans un premier temps de généraliser l'algorithme de clôture par congruence aux relations d'équivalence quelconques. Pour capturer les équivalences extensionnelles entre fonctions qui ne sont pas toujours des relations d'équivalence, je définis une notion de *relation complétée*, et fournis une implémentation de référence pour un langage imitant le style de Coq (réduit aux besoins du problème).

On s'intéresse ensuite à augmenter l'expressivité de la méthode par des extensions qui préservent la (semi-)décidabilité du problème. On peut par exemple exploiter des propriétés d'inclusion entre les relations du problème. J'élabore en particulier une *technique d'étiquetage* proposée par Pierre Corbineau pour instancier automatiquement des égalités quantifiées. Cela permet d'imaginer un support pour des théories plus élaborées comme des théories du premier ordre.

Ces développements s'accompagnent d'un suivi de la chaîne de raisonnement, nécessaire à la production de preuves vérifiables par Coq dans les réponses de l'algorithme. J'utilise une technique présente dans la littérature [NO05] pour l'algorithme original de clôture par congruence, adaptée pour supporter les extensions proposées.

Ce travail fait suite aux stages de Licence de Gabrielle Pauvert (ENS Lyon) et Nicolas Chataing (ENS Paris). Ils ont notamment considéré une interprétation de la logique propositionnelle dans l'algorithme et une première généralisation aux relations d'équivalence [Pau18, Cha18].

Note sur le contexte de la crise sanitaire

J'ai pu travailler au laboratoire Verimag à Grenoble de fin Janvier à mi-Mars, avant de rentrer à Lyon du fait de l'ordre de confinement. Ayant toutes les ressources sur ma machine personnelle, je n'ai pas eu de difficulté à poursuivre mon travail à distance, et les conférences web ont convenablement remplacé les réunions physiques.

Après avoir échangé avec d'autres élèves en stage de la promotion, je pense avoir été dans une situation très favorable pour poursuivre le stage en télétravail.

Plan de ce rapport

La section 2 présente l'algorithme de clôture par congruence original et sa généralisation aux relations d'équivalences quelconques. La section 3 porte sur la génération de preuves et leur vérification par Coq. La section 4 décrit la problématique de gestion des hypothèses quantifiées, et les techniques déployées pour les instancier automatiquement. Enfin, la section 5 donne une vue d'ensemble des problématiques d'intégration à Coq de cette procédure de décision.

L'annexe A présente en détail les preuves des résultats importants de ce rapport, avec un formalisme plus précis que dans le corps du texte.

2 La clôture par congruence

2.1 L'algorithme de Downey, Sethi et Tarjan

Le problème de déduction d'égalités peut se voir sous plusieurs formes, notamment comme un problème de décision logique ou plus algébriquement comme un cas restreint de problème du mot. De ce fait, plusieurs algorithmes ont été proposés autour de 1980 pour le résoudre, par Downey, Sethi et Tarjan [DST80], Nelson et Oppen [NO80], et Shostak [Sho82].

Ces algorithmes se recoupent significativement ; on peut même les voir comme différentes stratégies de saturation dans un même système abstrait de déduction d'égalités [BT00, BTV03]. Pour cette raison, je ne présente ici que le premier.

Spécification du problème On se donne un ensemble d'égalités $a_1 = b_1, \ldots\ a_n = b_n$ entre des termes, et un but $a = b$ à décider. Les termes sont des applications curryfiées, de la forme

$$term \ ::= \ \langle variable \rangle \ | \ \langle term \rangle \ \langle term \rangle \, .$$

L'objectif est de décider si le but est dérivable depuis les hypothèses dans le système de déduction comprenant les propriétés d'une relation d'équivalence et la règle de congruence :

$$\frac{}{x = x}\ \textsc{Refl} \qquad \frac{x = y}{y = x}\ \textsc{Sym} \qquad \frac{x = y \quad y = z}{x = z}\ \textsc{Trans} \qquad \frac{f = g \quad x = y}{f(x) = g(y)}\ \textsc{Congruence}$$

Vue d'ensemble de la méthode L'algorithme sature ses connaissances en appliquant toutes les règles possibles jusqu'à convergence. Il utilise pour cela deux structures de données :

- Une structure d'union-find [Tar75] qui représente les classes d'équivalence de l'égalité connues par l'algorithme. L'union-find est une structure qui maintient une partition, et offres deux opérations élémentaires : fusionner deux classes de la partition *(union)* et rechercher la classe à laquelle un élément appartient *(find)*.
  Ici, chaque classe est un ensemble d'éléments que l'algorithme a prouvé égaux. Le traitement d'une égalité $a = b$ consistera à fusionner les classes de a et b, ce qui appliquera automatiquement les règles Refl, Sym et Trans car les termes impliqués partageront alors une seule classe donc une seule identité.
- Une *table de signatures* de termes, décrite en détail ci-dessous, dont le rôle est de localiser efficacement les instances de la règle Congruence pour accélérer le calcul.

Ces structures ont chacune une file de *tâches en attente*, que l'algorithme traite en alternance jusqu'à saturation.

- La file *combine* liste les paires de termes (a, b) dont les classes attendent d'être fusionnées. Chaque fusion provoque la mise à jour d'un certain nombre de signatures.
- La file *pending* liste les termes dont la signature doit être recalculée. Chaque calcul de signature peut provoquer la fusion de deux classes.

La table de signatures Tout comme l'union-find permet de tester si deux termes sont égaux en testant s'ils sont dans la même classe, la table de signatures sert à tester si deux appels de fonctions $f(x)$ et $g(y)$ satisfont les prémisses de la règle CONGRUENCE.

Definition 2.1 (Signature)**.** *On définit la signature d'un terme $f(x)$ comme étant la paire $(\langle$classe de $f\rangle, \langle$classe de $x\rangle)$, ce qui nous donne la propriété*

$$f(x) \text{ et } g(y) \text{ ont la même signature} \iff \text{CONGRUENCE s'applique pour } f(x) \text{ et } g(y).$$

Dans ce rapport, je parle souvent des classes de l'union-find comme des objets. En pratique, chaque classe est identifiée par un quelconque de ses éléments (son *représentant*), que l'on encode de plus par un entier par commodité. Les opérations sur les termes et les classes sont donc élémentaires malgré leur apparente complexité.

Comme l'union-find applique structurellement les trois premières règles, l'algorithme n'a besoin pour appliquer toutes les règles possibles que de chercher manuellement les applications de CONGRUENCE. Pour cela, il calcule la signature de tous les appels de fonctions, et fusionne les classes de tous les appels de fonctions ayant la même signature. Notez que la signature d'un terme $f(x)$ change quand la classe de f ou x change de représentant, donc il faut recalculer les signatures et les collisions lorsque des fusions ont lieu.

Pour détecter les collisions (deux appels de fonction ayant la même signature), on construit la table de signatures, qui associe à chaque signature connue un terme qui la possède.

Algorithme détaillé L'algorithme complet est détaillé ci-dessous. Il est important de noter que chaque traitement de tâche prend un temps sensiblement constant ; tous les aspects de propagation sont traités par la boucle extérieure, qui fait office d'ordonnanceur. C'est une idée naturelle car la saturation fonctionne peu importe l'ordre de traitement.

Traitement des hypothèses :
- **Pour chaque** terme t du problème, créer une classe $\{t\}$ dans l'union-find
- **Pour chaque** hypothèse $a = b$, ajouter la paire (a, b) à *combine*
- **Pour chaque** appel de fonction $f(x)$, ajouter $f(x)$ à *pending*

Saturation :
- **Tant que** les files *combine* et *pending* ne sont pas vides
  - **Pour chaque** terme $f(x)$ dans *pending*
    - **Si** la signature de $f(x)$ est présente dans la table et associée à un terme $g(y)$, **alors**
      - Ajouter la paire $(f(x), g(y))$ à la file *combine*
    - **Sinon**, l'y ajouter et l'associer à $f(x)$
  - **Pour chaque** paire (a, b) dans *combine*
    - Fusionner les classes de a et de b
    - **Pour chaque** terme $f(x)$ pour lequel soit f soit x vient de changer de classe
      - Retirer $f(x)$ de la table de signatures (s'il y était)
      - Ajouter $f(x)$ à la file *pending*

Décision d'un but :
- Décider que le but $x = y$ est déductible si x et y sont dans la même classe

---

Considérons un exemple pour montrer un peu mieux le fonctionnement de cet algorithme. On se donne deux objets $x, y : A$ et une fonction $f : A \to A$; on veut prouver $x = y \vdash f(x) = f(y)$. La table ci-dessous montre l'état de l'algorithme de l'initialisation jusqu'à convergence.

Temps	**Partition**	**Table de signatures**	***pending***	***combine***
Initialisation	$\{x\}\ \{y\}\ \{f\}$ $\{f(x)\}\ \{f(y)\}$	—	$f(x), f(y)$	(x, y)
Après *pending*	(idem)	$(\{f\}, \{x\}) \to f(x)$ $(\{f\}, \{y\}) \to f(y)$	—	(x, y)
Après *combine*	$\{x, y\}\ \{f\}$ $\{f(x)\}\ \{f(y)\}$	$(\{f\}, \{x, y\}) \to f(x)$	$f(y)$	—
Après *pending*	(idem)	(idem)	—	$(f(x), f(y))$
Après *combine*	$\{x, y\}\ \{f\}$ $\{f(x), f(y)\}$	(idem)	—	—

Lors de la fusion de x et y dans la première passe de *combine*, je suppose ici que la classe de y est fusionnée dans celle de x donc c'est $f(y)$ dont un enfant change et qui se voit sortir de la table de signatures. Au prochain calcul, sa signature est déjà associée à $f(x)$, donc une tâche de fusion est ajoutée à *combine*.

2.2 Généralisation aux relations d'équivalence

La première extension que l'on souhaite faire à cet algorithme est de supporter des relations d'équivalence quelconques au lieu de l'égalité syntaxique. On se donne pour chaque type de données (de termes) une ou plusieurs relations d'équivalence, et on généralise les hypothèses et le but, qui étaient de la forme $x = y$, en relations bien typées $x\ R\ y$.

Les règles de déduction REFL, SYM et TRANS se transposent directement à ce nouveau contexte. Pour adapter la règle CONGRUENCE, on utilise la généralisation naturelle à l'égalité extensionnelle : la *relation extensionnelle* (*respectful* en Coq).

Definition 2.2 (Relation extensionnelle)**.** *Étant données deux relations R_1 et R_2 sur des types A et B, la relation extensionnelle $R_1 \Rightarrow R_2$ est la relation sur $A \to B$ définie par*

$$f\ (R_1 \Rightarrow R_2)\ g\ \equiv\ \forall (x, y : A),\ x\ R_1\ y \Longrightarrow f(x)\ R_2\ g(y).$$

Cette définition nous donne directement la règle CONGRUENCE modifiée.

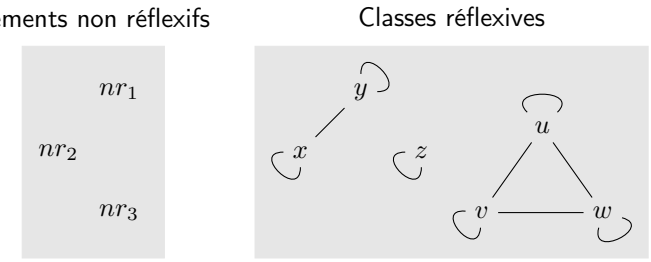
$$\frac{f\ (R_1 \Rightarrow R_2)\ g \quad x\ R_1\ y}{f(x)\ R_2\ g(y)}\ \textsc{Congruence}$$

Mais il y a un problème : $R_1 \Rightarrow R_2$ n'est pas une relation d'équivalence dans le cas général à cause d'un défaut de réflexivité ! En effet, *toute* fonction ne préserve pas *toute* relation... Comme contre-exemple, on peut prendre la congruence modulo 2 pour R_1 et l'égalité pour R_2. La fonction identité n'est alors pas réflexive pour $R_1 \Rightarrow R_2$ car $0 \equiv_2 2$ mais $\mathrm{id}(0) \neq \mathrm{id}(2)$.

Pour pallier ce manque, il faut supporter les relations d'équivalence non réflexives, que l'on appelle *relations d'équivalence partielles* ou *PER*.

Definition 2.3 (PER)**.** *Une PER sur un ensemble A est une relation symétrique et transitive. De façon équivalente, c'est une relation d'équivalence sur un sous-ensemble de A.*

La deuxième caractérisation est celle qui nous intéresse ; Il faut voir une PER comme une relation d'équivalence avec des éléments isolés non réflexifs. Le graphe d'une PER a la forme suivante :



Seuls des éléments isolés peuvent être non réflexifs. En effet, si on a deux éléments en relation $x\ R\ y$, on a aussi $x\ R\ y\ R\ x$ donc $x\ R\ x$ (par SYM et TRANS).

Les PER sont une réponse satisfaisante au défaut de reflexivité, car cette fois on a bien :

Proposition 2.1 (La relation extensionnelle préserve les PER)**.** *Si R_1 et R_2 sont des PER, alors $R_1 \Rightarrow R_2$ est une PER.*

Démonstration. En annexe A.1. □

Pour faire fonctionner l'algorithme de Downey, Sethi et Tarjan avec des relations d'équivalence et des PER, il reste à adapter la structure d'union-find qui ne supporte que les relations d'équivalence. Pour cela, j'associe à toute PER une relation d'équivalence « canonique » qui la remplacera dans l'union-find.

Definition 2.4 (Relation complétée)**.** *La* relation complétée $\hat{R}$ *d'une PER R est définie par*

$$x\ \hat{R}\ y\ \equiv\ (x\ R\ x \vee y\ R\ y) \Longrightarrow x\ R\ y.$$

La relation complétée $\hat{R}$ formalise l'idée d'être en relation par R « modulo réflexivité ». C'est une relation d'équivalence qui étend le graphe de R avec une classe pour tous les éléments non réflexifs pour R. On va alors pouvoir manipuler R dans l'algorithme en représentant dans l'union-find notre connaissance de $\hat{R}$.

Bien que les éléments non réflexifs pour R soient tous en relation dans $\hat{R}$, l'algorithme ne peut pas prouver cette non-réflexivité car il calcule toujours un sous-ensemble des relations présentes dans $\hat{R}$. En revanche, lorsqu'une preuve de $x\ \hat{R}\ y$ sera disponible (notamment par compatibilité, voir la section 2.3), l'algorithme pourra relier x et y. De cette façon, si un devient réflexif, l'autre le devient aussi et on obtient $x\ R\ y$.

Résumé de la généralisation L'algorithme supportant les relations d'équivalence et les PER procède avec le même principe de saturation et de files que l'original, mais :

- On utilise un union-find par relation manipulée et on ajoute la relation considérée dans toutes les tâches de *pending* et *combine*.
- Les union-finds de PER ont un booléen sur chaque classe, indiquant si elle est réflexive.

- La règle CONGRUENCE est modifiée pour utiliser des relations extensionnelles. Comme ce sont généralement des relations non réflexives, elle peut ne plus s'appliquer même quand $f = g$ (c'est-à-dire qu'on n'a pas forcément $(R_1 \Rightarrow R_2)\ f\ f$).

Quelques exemples Grâce à ce nouveau système, on peut exprimer des théories plus élaborées. Par exemple, il est courant en Coq d'utiliser des listes pour représenter des multi-ensembles, auquel cas l'ordre des éléments ne compte pas. On souhaite donc définir l'égalité multi-ensemble

$$l_1 =_{\mathsf{MS}} l_2 \ \equiv\ l_1 \text{ et } l_2 \text{ ont les mêmes éléments avec la même multiplicité.}$$

En plus de cela, on sait que la concaténation $(+)$ est un morphisme pour cette relation :

$$\forall (l_1, l_2, l_3, l_4),\ l_1 =_{\mathsf{MS}} l_3\ \rightarrow\ l_2 =_{\mathsf{MS}} l_4\ \rightarrow\ l_1 + l_2 =_{\mathsf{MS}} l_3 + l_4.$$

Cette propriété s'exprime aisément grâce à relation extensionnelle (sous forme curryfiée)

$$(+)\ (=_{\mathsf{MS}} \Rightarrow =_{\mathsf{MS}} \Rightarrow =_{\mathsf{MS}})\ (+).$$

La clôture par congruence modifiée peut travailler nativement avec ces structures. Cela n'empêche pas d'utiliser l'égalité ni l'égalité extensionnelle (qui revient à $(=) \Rightarrow (=)$), que l'on déclare ou détecte comme étant des relations d'équivalence.

2.3 Extensions semi-décidables de la méthode

Inclusion et compatibilité On trouve couramment dans les problèmes des relations liées par inclusion. On aimerait les gérer, de sorte par exemple que des listes égales soient automatiquement reconnues égales comme multi-ensembles. Une notion similaire à l'inclusion et appelée *compatibilité* existe pour les PER.

Definition 2.5 (Compatibilité de PER)**.** *Une PER R_1 est compatible avec une PER R_2, noté $R_1 \sqsubseteq R_2$, si*

$$\forall (x, y, z, t), (x\ R_1\ y \wedge z\ R_1\ t) \Longrightarrow (x\ R_2\ z\ \Longleftrightarrow\ y\ R_2\ t).$$

Cette définition formalise l'idée que l'on peut remplacer les opérandes de R_2 par des éléments qui leur sont équivalents pour R_1. Mais en fait il n'est pas nécessaire de s'attarder sur cette vision car la relation complétée permet de la reformuler comme une inclusion.

Proposition 2.2 (Compatibilité et relation complétée)**.**

$$R_1 \sqsubseteq R_2 \iff R_1 \subseteq \hat{R}_2.$$

Démonstration. En annexe A.2. □

On peut alors construire à l'initialisation du problème un graphe d'inclusions et compatibilités entre relations. Pour saturer par rapport aux règles de déduction correspondantes, il suffit de propager les fusions. Par exemple, si on fusionne les classes de x et y pour $R_1 \subseteq R_2$, on enfile dans *combine* une tâche de fusion des classes de x et y pour R_2 également.

D'une façon similaire à l'inclusion, la compatibilité nous fournit un lien universel avec l'égalité, puisqu'on a $= \sqsubseteq R$ pour toute PER R.

Cette extension bien pratique ne cache aucune autre subtilité ; une fois implémentée, elle permet par exemple à la clôture par congruence de prouver

$$(=) \subseteq =_{\mathsf{MS}},\ l_1 = l_3,\ l_2 =_{\mathsf{MS}} l_4,\ (+)\ (=_{\mathsf{MS}} \Rightarrow =_{\mathsf{MS}} \Rightarrow =_{\mathsf{MS}})\ (+)\ \vdash\ l_1 + l_2 =_{\mathsf{MS}} l_3 + l_4.$$

Hypothèses quantifiées L'extension la plus importante de ce stage est celle des hypothèses quantifiées. Ce sont les hypothèses qui ne sont pas de la forme $x\ R\ y$ mais

$$\forall (v_1 \ldots v_n), x\ R\ y$$

et qui modélisent beaucoup de propriétés utiles comme des théories du premier ordre. La section 4 explore leur utilisation en détail.

Constructeurs inductifs Le système peut être étendu avec des constructeurs inductifs. Cela ajoute l'injectivité sur tous les constructeurs et la disjonction des images (deux constructeurs différents ne renvoient jamais la même valeur). Cette extension est présente dans `congruence` [Cor06], mais je ne l'ai pas étudiée en détail.

Logique propositionnelle L'ajout de relations d'équivalences permet d'étudier les propositions du point de vue de l'équivalence logique ($\iff$). Cela ouvre une toute nouvelle perspective car les propositions étudiées peuvent encoder des informations du problème ! Par exemple :

- Obtenir la relation $P \iff \text{True}$ permet effectiver de prouver P ; on peut donc accepter en entrée des buts propositionnels quelconques et les prouver par équivalence logique.
- Obtenir la relation $\text{False} \iff \text{True}$ permet de prouver l'absurde, donc le but.
- La relation entre la proposition $(x\ R\ y)$ et le terme True pour $\iff$ est liée à la relation entre x et y pour R : prouver l'un permet d'obtenir directement l'autre.

C'est une direction très intéressante à explorer pour tester les limites de ce qui reste décidable. Je n'ai cependant pas eu l'opportunité de l'étudier non plus.

2.4 Implémentation et validation

J'ai implémenté l'algorithme étendu de clôture par congruence en OCaml en-dehors de Coq : essentiellement un outil en ligne de commande qui lit ses entrées dans une syntaxe imitant celle de Coq. L'outil cherche automatiquement les relations dans l'entrée et repère les PER extensionnelles et égalités extensionnelles pour imiter ce qu'une tactique Coq ferait avec les *typeclasses*.

À ce stade, trois méthodes sont employées pour valider la correction de l'implémentation.

- Des tests unitaires sous la forme de scripts dans une syntaxe type Coq. Chaque script définit des hypothèses et annonce des buts (pas tous déductibles). La réponse du programme est comparée à une solution fournie avec le test.
- Le programme génère également un arbre de preuve textuel qui peut être inspecté manuellement. La génération des preuves et une méthode de vérification plus élaborée sont abordées dans la section 3.
- Des tests de couverture fournis par le module `bisect_ppx` [CRB] sont utilisés en complément pour mesurer l'exhaustivité des tests unitaires par rapport au code du programme.

L'intégration à Coq de ce programme est discutée rapidement dans la section 5.

3 Génération de preuves

L'algorithme de clôture par congruence décide l'égalité en annonçant simplement si le but est déductible des hypothèses, mais il en faut plus pour convaincre un assistant de preuve. Il faut générer un arbre de preuve complet, ce qui nécessite de garder une trace des règles de déduction que l'algorithme utilise pour justifier chaque fusion de classes dans l'union-find.

Avant de détailler la structure de données utilisée pour cela, il est intéressant de noter que Coq nous permet d'automatiser les arguments simples, ce qui nous évite de les suivre. Voyons rapidement quels arguments cela couvre.

3.1 Preuves nécessaires à la vérification machine

En comptant les extensions, le système utilise tous les arguments ci-dessous :

Preuve de...	**Types d'arguments** (« Hypothèse du problème » implicite partout)
$x\ R\ y$	REFL, SYM, TRANS, CONGRUENCE, Inclusion, Promotion de $\hat{R}$
$x\ \hat{R}\ y$	Compatibilité
PER R	Proposition 2.1, Affaiblissement de Equivalence
Equivalence R	Propriété de l'égalité
$R_1 \subseteq R_2$	Inclusions covariantes, Inclusion de $=$ dans $(=) \Rightarrow (=)$, $= \subseteq R$ pour toute relation d'équivalence R
$R_1 \sqsubseteq R_2$	$= \sqsubseteq R$ pour toute PER R

Coq est largement capable d'inférer une grande partie de ces arguments ; il peut par exemple appliquer la Proposition 2.1 qui dit « PER $R_1 \to$ PER $R_2 \to$ PER $(R_1 \Rightarrow R_2)$ » récursivement où nécessaire. Dans ce cas, il suffit d'annoncer que la relation ciblée est une PER et la preuve est générée automatiquement, dans mon cas par la tactique `auto`. Un exemple plus détaillé est présenté dans la section 3.3.

J'ai programmé en Coq l'automatisation de toutes les preuves de la forme PER R, Equivalence R, inclusion et compatibilité avec les arguments présentés ci-dessus, ce qui nous laisse à suivre uniquement le cœur du raisonnement : les relations entre les éléments.

3.2 Forêts de preuve

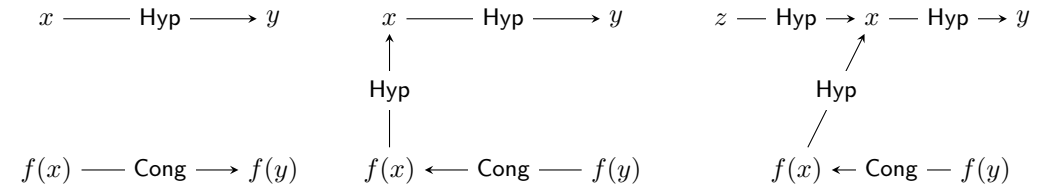
La structure utilisée pour représenter cette information est une forêt. Pour une relation R fixée, on se donne un graphe dont les nœuds sont les termes du problème et les arêtes sont des preuves de relation. Par exemple, une arête reliant x à y dans le graphe de R est étiquetée par une preuve de $x\ R\ y$. On se limite pour l'instant aux relations d'équivalence ; le cas des PER est un peu technique et traité à la fin de cette partie, mais se conforme aux mêmes principes.

Les arbres de la forêt sont donc les classes d'équivalence de R, identiques à celles que l'on trouve dans l'union-find. De plus, l'enchaînement des arêtes d'un chemin forme une preuve par

transitivité.

Cette structure est décrite dans la littérature par Nieuwenhuis et Oliveras [NO05] ; cette section présente principalement leur construction. Une seule modification notable est nécessaire pour y supporter les objets non réflexifs issus des PER.

La figure (a) ci-dessous montre un exemple typique de forêt de preuve pour la relation d'égalité. On y retrouve une classe contenant x et y qui sont égaux par hypothèse, et une classe contenant $f(x)$ et $f(y)$ qui sont égaux par CONGRUENCE. La classe de f pour l'égalité extensionnelle n'est pas représentée.



(a) Forêt de preuve pour $(=)$ (b) Ajout de $f(x) = x$ (c) Ajout de $x = z$

Efficacité des arbres enracinés Comme chaque chemin entre deux objets dans la forêt forme une preuve par transitivité, il suffit que les classes d'équivalence soient connexes pour que l'algorithme puisse justifier toutes ses affirmations. On se contente donc de former des arbres, et on les enracine pour accélérer la recherche de chemin. Cette optimisation est présentée dans le paragraphe « plus petit ancêtre commun ».

On pourrait être tenté de stocker ces informations directement dans l'union-find, mais les optimisations ne s'y prêtent pas :

- La compression de chemin déplace les arêtes alors que les arguments ne changent pas.
- Il y a beaucoup d'effets non locaux car l'ajout d'une relation entre deux éléments se matérialise par une arête entre leurs représentants, qui sont loin dans le graphe.

La structure de forêt de preuve garantit, à l'inverse, que les nouvelles arêtes sont créées exactement entre les éléments dont elle prouvent la relation, et ne changent jamais.

Une fois créée, la forêt fournit une opération d'union utilisée pendant la clôture par congruence, et une recherche de plus petit ancêtre commun utilisée à la fin pour générer les preuves.

Union Initialement, la forêt ne contient aucune arête. L'union de deux termes ajoute une arête qui combine deux arbres ; un exemple est donné avec x et $f(x)$ sur le diagramme (b) ci-dessus. Notez que si les éléments concernés sont déjà en relation, aucune arête n'est ajoutée.

Lorsqu'aucune orientation de la nouvelle relation ne permet de préserver la propriété d'arbre enraciné, un des arbres est modifié pour faire du terme en relation sa racine. Ici, $f(x)$ devient la racine de l'arbre $f(x)$—$f(y)$ pour permettre la fusion. Dans le diagramme (c), on s'en sort en évitant ce problème car l'orientation $z \longrightarrow x$ convient.

On peut construire des cas pathologiques où les changements de racine ont un coût total quadratique. Ce problème pourrait se résoudre en gardant la forêt non orientée pendant la clôture par congruence et en l'orientant en temps linéaire avant la génération de preuve. L'orientation n'est en effet nécessaire que pour la recherche de chemin.

Plus petit ancêtre commun La génération d'une preuve de $x\ R\ y$ consiste à trouver un chemin entre x et y dans leur arbre pour R. Le plus court chemin disponible est celui qui passe par leur plus petit ancêtre commun ; on le trouve en partant de x et y et en remontant en parallèle les arêtes jusqu'à trouver un élément commun. Par exemple, le chemin de preuve naturel entre z et $f(y)$ sur le diagramme (c) passe par x.

Pour faciliter la détection du premier élément commun sur le chemin, on maintient pour chaque noeud un *rang* tel que si $x \longrightarrow y$ alors $\text{rang}(x) < \text{rang}(y)$. On peut alors partir de x et y et remonter celui ayant le rang le plus faible jusqu'à ce que les deux se rencontrent au plus petit ancêtre commun, complétant le chemin de preuve.

Malgré cet effort heuristique pour prendre des chemins courts, il faut noter que les arbres de preuve ne sont pas du tout de taille minimale, ne serait-ce que parce que les relations redondantes sont ignorées alors qu'elles peuvent fournir des preuves courtes. Contrairement à [NO05] qui cherche à produire des explications minimales, on ne se soucie ici que modérément de la longueur du raisonnement car la génération de preuve n'est pas la partie critique du système (qui est composée de la clôture par congruence et de l'instanciation des hypothèses quantifiées décrite dans la section 4).

PER et témoins de réflexivité La gestion des PER requiert un mécanisme supplémentaire en plus des précédents. En effet, pour une PER R il est important de noter les preuves de réflexivité. De plus, la compatibilité prouve des relations pour $\hat{R}$ mais pas R si les éléments concernés ne sont pas réflexifs. Il faut donc étendre la structure d'origine pour supporter ces situations.

Les détails de cette modification sont techniques, mais on peut la résumer à trois principes :

1. Lorsque R est une PER, une arête entre x et y représente une preuve de $x\ \hat{R}\ y$.
2. On maintient le statut réflexif de tous les termes. Un terme x pour une PER peut être prouvé réflexif de deux façons : ou bien on a un terme y tel que $x\ R\ y$ (preuve directe) ; ou bien on a un terme réflexif u tel quel $x\ \hat{R}\ u$ (délégation). Ce mécanisme est utile lorsqu'une classe non réflexive est fusionnée avec un élément réflexif u : on prouve d'un coup la réflexivité de toute la classe par délégation à u. On ne délègue jamais en cascade, u doit être réflexif par preuve directe.
3. Pour prouver $x\ R\ y$, on veut un chemin entre x et y et une preuve de réflexivité pour x. (La réflexivité de x est équivalente à celle de la classe entière.)

Le résumé des arguments présenté à la section 3.1 doit donc en toute rigueur être étendu avec ces preuves de réflexivité.

Preuve de...	Types d'arguments
$x\ R\ x$	Preuve de $x\ R\ y$ pour un y quelconque, Equivalence R, Délégation

3.3 Génération de preuves vérifiables

Comme mentionné dans la section 2.4, les preuves produites par la forêt de preuve étaient initialement affichées au format texte et vérifiées à la main. Ce n'est bien sûr pas une méthode de validation suffisante pour une procédure de décision que l'on souhaite intégrer dans Coq. À ce stade le programme est encore indépendant donc ne peut pas transmettre un arbre de preuve à Coq ; mais on peut simuler cet effet en faisant générer par notre outil un fichier textuel contenant le théorème décidé et un script de preuve que Coq peut vérifier ensuite.

Voici un exemple court pour illustrer le résultat de la génération de preuve. On se donne ici deux hypothèses concernant des objets x, z de type A et une fonction $f : A \to A$:

- `Efxz` : $f\ x = z$
- `Efzx` : $f\ z = x$

Ci-dessous est la preuve Coq générée pour le but $f\ (f\ x) = x$. Chaque but intermédiaire est donné par un `assert` nommé, et l'indentation donne la structure récursive de la preuve : les lignes 4 à 6 sont donc les résultats intermédiaires pour la ligne 7. (Le terme `@eq A` représente l'égalité de Coq sur les objets de type A.)

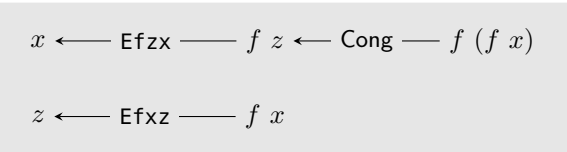
```
1  Lemma L3: f (f x) = x.
2  Proof. intros.
3    assert (E0: Equivalence (@eq A)) by auto.
4      assert (R0t10t3: f x = z) by apply Efxz.
5      assert (E2: Equivalence (@eq A ==> @eq A)) by auto.
6      assert (R2t4t4: (@eq A ==> @eq A) f f) by reflexivity.
7    assert (R0t25t14: f (f x) = f z) by apply (R2t4t4 _ _ R0t10t3).
8    assert (R0t14t1: f z = x) by apply Efzx.
9    assert (R0t25t1: f (f x) = x) by apply (R0t25t14 |> R0t14t1).
10   apply R0t25t1.
11 Qed.
```

On note que les propositions `E0` et `E2` montrant que les égalités sur f sont des relations d'équivalence sont prouvées automatiquement par Coq grâce à une configuration préalable de la tactique `auto`. Il est alors immédiat (ligne 6) que f est réflexif pour l'égalité extensionnelle.

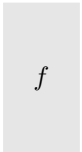
La preuve ligne 7 utilise la règle CONGRUENCE qui est cachée dans la définition de la relation extensionnelle (`R2t4t4` *est* la règle de congruence pour f). Enfin, la dernière étape ligne 9 consiste à appliquer transitivement les deux résultats précédents, ce qui est caché derrière le symbole `|>`.

La forêt de preuve correspondante est la suivante. Tous les objets y sont automatiquement réflexifs puisqu'on travaille uniquement avec l'égalité.

Pour $=_A$:



Pour $=_{A\to A}$:



Pour $=_A \Longrightarrow =_A$:

Ce système a permis de vérifier, en combinaison avec les tests unitaires, toutes les extensions décrites dans la section 2 et implémentées dans le programme. À ce stade, l'algorithme de clôture par congruence supporte les relations d'équivalences et PER, inclusions et compatibilités de relations, et peut générer des preuves vérifiables par Coq.

L'expressivité de ce nouveau problème est satisfaisante par rapport à la clôture par congruence originale. Il reste cependant une piste d'amélioration à étudier, celle des hypothèses quantifiées.

4 Instanciation des hypothèses quantifiées

4.1 Les enjeux de l'instanciation

Les hypothèses quantifiées sont probablement l'outil le plus expressif abordé jusqu'à présent. Ce sont toutes les hypothèses de la forme

$$\forall(v_1...v_n),\ x\ R\ y$$

que l'on peut instancier avec n'importe quelles valeurs bien typées de $v_1...v_n$ pour obtenir des égalités. Elles peuvent exprimer de nombreuses propriétés du premier ordre, par exemple...

- Associativité : $\forall(x, y, z), f(f(x, y), z) = f(x, f(y, z))$
- Élément neutre : $\forall x, f(e, x) = x$
- Commutativité pour une relation R : $\forall(x, y), f(x, y)\ R\ f(y, x)$

Attention toutefois, la clôture par congruence n'a aucune notion de calcul et encore moins de forme normale. En saturant, l'algorithme va générer toutes les formes associatives possibles de tous les termes du problème. On a donc un *risque d'explosion combinatoire* à surveiller.

De plus, on a vite fait d'orienter mentalement la règle de neutralité de gauche à droite, mais *a priori* l'algorithme sature dans les deux sens. On a donc aussi un *enjeu de terminaison* car les instanciations peuvent faire apparaître des nouveaux termes à l'infini. En fait cette extension ne peut au mieux que préserver la semi-décidabilité du problème car on peut encoder le calcul des machines de Turing en représentant des configurations par des termes, ce qui n'est pas décidable.

Permettre à l'algorithme d'instancier automatiquement est un avantage certain car les propriétés universelles ne seront pas forcément présentes dans le contexte avec exactement la spécialisation dont on a besoin.

On peut retrouver un exemple d'application avec l'égalité multi-ensembles $=_{\mathsf{MS}}$ sur les listes, présentée à la section 2.2. On a vu que la concaténation est un morphisme pour l'égalité, ce qui s'exprime avec la relation extensionnelle $(+)\ (=_{\mathsf{MS}} \Rightarrow =_{\mathsf{MS}} \Rightarrow =_{\mathsf{MS}})\ (+)$. Mais la concaténation commute également pour cette relation, ce qu'on écrit avec l'hypothèse quantifiée

$$\forall(l_1, l_2), l_1 + l_2 =_{\mathsf{MS}} l_2 + l_1.$$

Cet exemple en particulier est très appréciable pour la clôture par congruence. Le passage de $l_1 + l_2$ à $l_2 + l_1$ est involutif donc la saturation termine toujours. Contrairement aux règles de « normalisation », qui auront moralement tendance à introduire beaucoup de termes intermédiaires inutiles, c'est une règle symétrique facile à manipuler.

L'instanciation est un problème majeur pour les outils SMT, et la littérature qui l'accompagne est riche. Un lien direct est par exemple établi par [BFR17] grâce à un problème d'unification. La portée de ce stage ne permet bien sûr pas d'explorer tous les détails de ces techniques. Ici, je décris un algorithme qui énumère les instances disponibles modulo équivalence parmi les termes déjà présents dans le problème, et constate les enjeux de combinatoire et de terminaison posés par une stratégie d'instanciation naïve sans les attaquer de front.

4.2 Algorithme de haut niveau

Quelles instances sont intéressantes ? La question peut être très compliquée selon le niveau d'exigence de « intéressant ». Mais en toute première approximation, on peut déjà affirmer qu'une instance de $\forall(v_1...v_n), x\ R\ y$ n'est intéressante que si x ou y est dans le problème.

En effet, les termes dont on cherche à décider s'ils sont en relation y sont. Ainsi, même si le chemin de preuve qui les reliera ultimement passe par de nouveaux termes à découvrir, on peut toujours progresser en gardant un pied sur les termes connus, comme un pont que l'on construirait à partir des deux rives au lieu de recouvrir le lac.

Cela nous donne donc une première intuition : on cherche des valeurs à $v_1...v_n$ telles que x ou y soit un terme du problème. Ce sera le sujet de l'algorithme *d'E-matching* (matching modulo équivalence). Une fois ces valeurs trouvées, on procède naïvement : si l'autre terme est nouveau on l'ajoute au problème, et on ajoute la relation.

La méthode Entre chaque génération d'instanciation, on fait une passe de clôture par congruence pour saturer les nouveaux termes et relations. Le schéma général est donc comme ceci :

Initialiser le problème

Tant que l'on souhaite continuer à instancier

  Exécuter une passe de clôture par congruence

  Pour chaque hypothèse quantifiée $\forall(v_1...v_n), x\ R\ y$

    Pour chaque instance σ telle que $\sigma(x)$ ou $\sigma(y)$ est dans le problème

      Ajouter $\sigma(x)$ et $\sigma(y)$ au problème (s'ils n'y sont pas encore)

      Ajouter la paire $(\sigma(x), \sigma(y))$ à la file *combine* pour R

Conclure que le but $x\ R\ y$ est déductible si x et y sont dans la même classe pour R

Algorithme d'E-matching Pour localiser les instances d'un terme quantifié qui sont dans le problème, on commence par extraire un motif en remplaçant les variables quantifiées par des variables existentielles. Par exemple, de $\forall x, (f\ e\ x)\ R\ x$ on extrait les motifs $f\ e\ ?x$ et $?x$ (je note les variables existentielles avec des points d'interrogation).

Le problème d'E-matching consiste alors à trouver dans une classe réflexive C un terme qui est équivalent pour R à une instance du motif m. On note ce problème

$$m \sim=_R C.$$

Une solution du problème est une paire (σ, t) où σ est une substitution qui donne une valeur aux variables existentielles de m, $t \in C$ et $\sigma(m)\ R\ t$. On ne demande pas que $\sigma(m) = t$ car on veut créer une relation pour R (c'est l'hypothèse quantifiée) et pas pour l'égalité. La valeur exacte de t n'est pas importante : deux solutions ayant le même σ sont considérées comme identiques.

Pour supporter correctement les PER et la compatibilité de PER, on a besoin d'une version irréflexive de ce problème, dans laquelle on autorise C à être non réflexive et on demande que $\sigma(m)\ \hat{R}\ t$. Les mêmes principes s'appliquent, donc je mentionnerai rapidement les différences sans rentrer dans les détails.

L'algorithme de recherche d'instance procède récursivement selon la structure du motif. Au cours de la recherche, on maintient la liste des candidats pour chaque $\sigma(?x)$ pour choisir à la fin une valeur compatible avec toutes les ocurrences de la variable (qui sont pour des relations différentes !). Voici comment fonctionnent les deux cas de motifs non récursifs.

- **Termes constants** : $c \sim=_R C$ (pour σ)
  L'ensemble des solutions est $\{\sigma\}$ si $c \in C$ et $\varnothing$ sinon.
- **Variables existentielles** : $?x \sim=_R C$ (pour σ)
  Si $\sigma(?x)$ n'a pas encore de candidats, on initialise l'ensemble des candidats par C. Si $\sigma(?x)$ a des candidats, on intersecte l'ensemble des candidats avec C ; s'il devient vide, il n'y a pas de solution.

À la fin de l'algorithme, on énumère les instances en choisissant pour chaque variable existentielle $?x$ un candidat parmi ceux restants pour $\sigma(?x)$.

Le problème combinatoire dans l'E-matching Le troisième cas des appels de fonctions, quand $m = f\ m_1\ ...\ m_n$, est subtil. En effet, comme on cherche des instances modulo équivalence (une notion sémantique), on ne peut pas trop s'appuyer sur la structure syntaxique des termes pour éliminer des candidats. Par exemple, le terme $g\ y$ peut être une instance du motif $f\ e\ ?x$ si par hasard $g = f\ e$ (et dans ce cas $?x := y$). Ça pose un problème combinatoire rapidement car la recherche des instances finit par tester en détail tous les termes disponibles dans la classe. Une heuristique est nécessaire ici pour trouver rapidement les bons candidats.

4.3 Étiquetage pour accélérer l'E-matching

Pour élaguer cet espace de recherche causé par une question sémantique, on va se placer aussi au niveau sémantique. L'idée de Pierre Corbineau utilisée dans congruence [Cor01] est d'étiqueter chaque classe C avec une paire (f, n) si elle contient un terme équivalent à un appel de f avec n arguments. Ce système fournit une sur-approximation de l'ensemble des solutions de $f\ m_1\ ...\ m_n \sim=_R C$ par celui de $f\ ?x_1\ ...\ ?x_n \sim=_R C$ et réduit donc le nombre de candidats à tester.

Le formalisme de ce système d'étiquetage est assez technique ; les définitions complètes et la preuve du théorème principal sont données dans l'annexe A.3. Dans cette partie, j'en décris seulement les principes.

Étiqueter avec un indice de preuve L'objectif des étiquettes est de trouver les solutions au problème d'E-matching. Mais si on s'intéresse aux moyens d'obtenir $\sigma(m)\ R\ t$, on peut trouver une correspondance rudimentaire entre les décisions prises lors de la recherche des instances et l'arbre de preuve obtenu. Il est donc naturel d'indiquer au travers des étiquettes quelle forme l'arbre de preuve de $\sigma(m)\ R\ t$ doit prendre.

Je définis deux types d'étiquettes (f, n), où f est un symbole de fonction non appliqué, et $n \geq 0$ un entier :

- L'étiquette $\textsc{Refl}(f, n)$ est attribuée à C si elle contient un terme réflexif de la forme $f\ t_1\ ...\ t_n$.

- L'étiquette CONG(f, n, R') est attribuée à C si elle contient un terme de la forme $u\ v$ tel que la classe de u pour $R'{\Rightarrow}R$ a une étiquette $(f, n-1)$ et v est réflexif pour R'.

Dans le programme, je prends soin de garder la trace des témoins $(f\ t_1\ ...\ t_n)$ et $(u\ v)$ qui ont fourni les étiquettes car ils sont utiles pour obtenir les solutions concrètes.

Essentiellement, l'étiquette REFL(f, n) indique les solutions de $f\ m_1\ ...\ m_n \sim_{=R} C$ pour lesquelles la preuve de $\sigma(f\ m_1\ ...\ m_n)\ R\ t$ se termine par les règles REFL, SYM ou TRANS, et l'étiquette CONG(f, n, R') indique celles pour lesquelles la preuve se termine par une application de CONGRUENCE, avec en prime la relation extensionnelle $R'{\Rightarrow}R$ concernée.

Les étiquettes forment naturellement une structure inductive car les étiquettes CONG(f, n, R) sont définies grâce à des étiquettes $(f, n-1)$; les REFL(f, n) forment les cas de base. Cette structure est utilisée pour prouver la majorité des affirmations dans cette section.

Pour gérer correctement les classes irréflexives de PER, on lève la condition de réflexivité du témoin $f\ t_1\ ...\ t_n$ dans leurs étiquettes REFL(f, n). On parle alors d'étiquettes REFL(f, n) irréflexives. Cela sert dans la variante irréflexive du problème d'E-matching. Notez que la condition de réflexivité n'est pas levée dans CONG(f, n, R) ; comme pour la compatibilité $R_1 \subseteq \hat{R}_2$, où la réflexivité est ignorée sur R_2 mais pas sur R_1, on ne propage pas les étiquettes construites sur des classes non réflexives.

Correction et complétude de l'étiquetage La propriété de sur-approximation annoncée au début de cette section peut être prouvée à l'aide des définitions des étiquettes et est formalisée dans le théorème suivant.

Theorem 4.1 (Correction et complétudes de l'étiquetage)**.** *Une classe C pour une relation R possède une étiquette (f, n) si et seulement si le problème d'E-matching*

$$f\ ?x_1\ ...\ ?x_n \sim_{=R} C$$

a des solutions, et possède une étiquette (f, n) irréflexive si et seulement si le problème irréflexif associé a des solutions.

Démonstration. En annexe A.3. □

Calcul et maintenance des étiquettes Dans l'algorithme de clôture par congruence, les étiquettes sont proches des signatures : il faut les recalculer quand des termes changent de classe. Plus précisément, quand on fusionne deux classes de fonctions C_f et C_g, on recalcule les étiquettes de tous les appels à des fonctions de $C_f \cup C_g$ car ces appels peuvent acquérir une étiquette CONG. De plus, lorsque la classe d'un terme f ou x devient réflexive, on recalcule les étiquettes de tous les appels $f(x)$ car ils peuvent acquérir des étiquettes REFL ou CONG.

Ces tâches sont ajoutées à la file *pending*, mêlées aux tâches de calcul des signatures. On n'a pas besoin de se soucier de l'ordre de traitement, car il n'influence pas le résultat de l'algorithme.

Algorithme d'E-matching complété On peut maintenant décrire une méthode efficace pour la recherche d'instances quand le motif est un appel de fonction !

- **Appels de fonctions** : $f\ m_1\ ...\ m_n \sim_{=R} C$ (pour σ)

Si C n'a aucune étiquette (f, n), il n'y a pas de solution. Sinon, on calcule l'union des solutions obtenues par l'analyse de chaque étiquette à partir de σ.

- Pour chaque étiquette REFL(f, n) avec un témoin $f\ t_1\ ...\ t_n$, on analyse récursivement les problèmes $m_i \sim= t_i$ en utilisant la même copie de σ pour contrôler la cohérence des substitutions.
- Pour chaque étiquette CONG(f, n, R) avec un témoin $u\ v$, on analyse récursivement les deux problèmes

$$f\ m_1\ ...\ m_{n-1} \sim=_{R' \Longrightarrow R} C_{R' \Longrightarrow R}(u) \quad \text{et} \quad m_n \sim=_{R'} C_{R'}(v)$$

avec la même copie de σ pour contrôler la cohérence des substitutions.

Comme illustration simple, on peut considérer l'exemple suivant où une fonction f ayant un élément neutre e est assimilée à l'identité après curryfication.

Classe	$\{f\}$	$\{f\ e, \mathsf{id}\}$	$\{e, \mathsf{id}\ e\}$
Étiquettes	REFL$(f, 0)$	REFL$(f, 1)$ REFL$(\mathsf{id}, 0)$	CONG$(f, 2)$ REFL$(\mathsf{id}, 1)$

Avec les classes ci-dessus pour l'égalité, il y a une solution à $f\ ?x\ ?y \sim= \{e, \mathsf{id}\ e\}$, donnée par $\sigma = (?x\ \rightarrow\ e, ?y\ \rightarrow\ e)$ avec le témoin $t = \mathsf{id}\ e$.

4.4 Résultats pratiques

Comme remarqué au début de la section, cet algorithme trouve les spécialisations disponibles mais n'a aucune stratégie globale d'instanciation. Je n'ai pas été plus loin dans l'implémentation, mais voici quelques aspects qui méritent d'être abordés et pour lesquels j'ai des pistes de réponse.

1. Comment éviter de regénérer les solutions déjà trouvées précédemment ? Ici, chaque génération énumère de nouveau toutes les instances déjà trouvées, ce qui est très inefficace. Comme pour les signatures et étiquettes, on aimerait détecter pendant la clôture par congruence les opérations qui provoquent l'apparition de nouvelles instances.
2. Comment orienter les « règles de simplifications » comme $\forall P, P \vee \mathsf{True} \iff P$? Une heuristique simple consiste à observer qu'un des membres et un sous-terme de l'autre, mais pour les cas plus compliqués il est probablement nécessaire de s'appuyer sur l'utilisateur.
3. Quand arrêter l'instanciation ? Jusqu'à présent on se contente de limiter le nombre de générations, mais ce n'est pas une mesure très pertinente pour l'utilisateur.
4. Il reste enfin toutes les heuristiques concernant la priorité qu'on accorde aux hypothèses quantifiées, ce qui ramène à la question initiale « Quelles instances sont intéressantes ? ».

La fonctionnalité implémentée reste donc une preuve de concept et est mise en défaut sur des cas défavorables comme celui des théories associatives-commutatives. Ce cas est fondamentalement compliqué car le problème du mot associé est EXPSPACE-complet [BTV03], mais on souhaite certainement éviter de générer toutes les décompositions des nombres entiers du problème sous forme de somme (associativité incluse) avec un support dédié pour ces théories.

Je m'attends donc à ce que des améliorations théoriques et pratiques soient apportées à la méthode d'instanciation pour la rendre plus robuste. Il y a certainement des compromis nécessaires à faire sur l'expressivité pour conserver un temps de réponse raisonnable.

5 Intégration à Coq

Mon implémentation en OCaml de l'algorithme étendu de clôture par congruence est indépendante de Coq. Lionel Rieg et Pierre Corbineau travaillent à l'intégrer dans l'assistant de preuve sous la forme d'un plugin, comme `congruence`. Deux changements principaux sur les entrées et sorties du programme sont nécessaires pour cela :

- Il faut détecter les hypothèses et les buts de l'algorithme dans le contexte Coq. C'est presque un sujet entier, avec des questions de *typeclasses*, de normalisation, d'opacité... le plugin détecte actuellement les hypothèses dans le but courant, et l'API de l'algorithme est adaptée pour fonctionner avec les types de Coq.
- Les preuves doivent être produites directement sous la forme de termes.

En parallèle de ce développement, le travail réalisé pendant ce stage a été soumis et accepté au Coq Workshop 2020 [Coq] où il sera présenté à la communauté Coq. Une copie de la courte soumission est incluse à la fin de document pour donner une idée plus précise du contexte Coq.

6 Conclusion

Travail effectué Durant ce stage, j'ai étudié les aspects théoriques et pratiques de plusieurs extensions de l'algorithme de clôture par congruence. Le support des relations d'équivalence et PER en plus de l'égalité permet de décrire beaucoup de problèmes concrets sans modifier les fondements de l'algorithme. La génération de preuves vérifiables permet à la méthode d'être intégrée à un assistant de preuve. Enfin, l'instanciation d'hypothèses quantifiées augmente considérablement l'expressivité de la méthode, ce qui pose des problèmes de décidabilité et d'explosion combinatoire qu'il faut aborder avec précaution.

Ces développements ont été implémentés dans un prototype qui réalise la clôture par congruence et génère des scripts de preuve vérifiables par Coq. Ce prototype est en passe d'être intégré à Coq sous la forme d'une nouvelle tactique pour complémenter la tactique `congruence` existante qui se limite à l'égalité, et sera présenté à la communauté lors du Coq Workshop 2020.

Travail futur Plusieurs pans de la théorie et de la pratique méritent encore d'être explorés. Il y a un lien intéressant à faire avec les SMT pour aborder les heuristiques d'instanciation. L'extension à la logique propositionnelle promet un nouveau gain d'expressivité dont la décidabilité mérite d'être délimitée. Et plus concrètement, l'intégration de l'algorithme à Coq nécessite des tests approfondis pour garantir la robustesse de la tactique.

Remerciements Un grand merci à Pierre et Lionel pour avoir fourni un cadre théorique clair permettant au stage de progresser efficacement, et à Karine pour ses avis éclairés sur la structure et la forme du code et des documents. Merci également à Nicolas Chataing et Gabrielle Pauvert pour m'avoir raconté leurs approches et partagé leurs rapports.

A Preuves des résultats principaux

A.1 La relation extensionnelle préserve les PER

Proposition A.1 (La relation extensionnelle préserve les PER)**.** *Si R_1 et R_2 sont des PER, alors $R_1 \Rightarrow R_2$ est une PER.*

Démonstration.

Symétrie Supposons qu'on a $f\ (R_1 \Rightarrow R_2)\ g$ et $x\ R_1\ y$; on veut obtenir $g(x)\ R_2\ f(y)$. En appliquant l'hypothèse avec y et x, on obtient $f(y)\ R_2\ g(x)$, et on conclut par symétrie de R_2.

Transitivité Supposons qu'on a $f\ (R_1 \Rightarrow R_2)\ g\ (R_1 \Rightarrow R_2)\ h$. g est donc réflexive pour $R_1 \Rightarrow R_2$. Soit $x\ R_1\ y$; on applique les deux hypothèses pour avoir $f(x)\ R_2\ g(y)$ et $g(x)\ R_2\ h(y)$. Ensuite, comme g est réflexive, on l'appelle avec y et x pour obtenir $g(y)\ R_2\ g(x)$. Enfin on conclut par transitivité de R_2.

□

A.2 Compatibilité et relation complétée

Proposition A.2 (Compatibilité et relation complétée)**.**

$$R_1 \sqsubseteq R_2 \iff R_1 \subseteq \hat{R}_2.$$

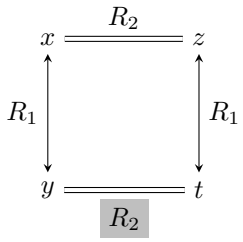
Démonstration. Les trois formes suivantes sont en fait équivalentes :

(i) $R_1 \sqsubseteq R_2$ (Diagramme (a)) ;

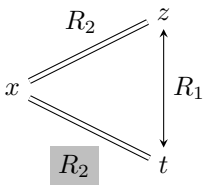
(ii) $\forall (x, z, t), x\ R_2\ z\ R_1\ t \Longrightarrow x\ R_2\ t$ (Diagramme (b)) ;

(iii) $\forall (x, t), x$ réflexif pour R_2 et $x\ R_1\ t \Longrightarrow x\ R_2\ t$ (Diagramme (c)).

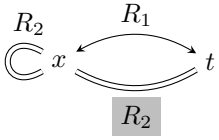
Sur le diagramme ci-dessous, les doubles flèches montrent comment les opérandes des égalités pour R_2 peuvent être réécrites modulo R_1. L'égalité sur fond gris est la conclusion ; les autres relations sont les hypothèses.



(a) Version originale



(b) Réécriture à droite uniquement



(c) Sous forme d'inclusion

On passe d'une forme à l'autre de la façon suivante :

$(ii) \Longrightarrow (i)$ Appliquer (ii) pour (x, z, t) et pour (t, x, y), puis concaténer les diagrammes.

$(iii) \Longrightarrow (ii)$ Supposons $x\ R_2\ z\ R_1\ t$. Alors z est réflexif pour R_2, donc on peut appliquer (iii) avec $x := z$. On obtient $z\ R_2\ t$, et on termine par transitivité.

$(i) \Longrightarrow (iii)$ Supposons $x\ R_2\ x\ R_1\ t$, alors x est réflexif pour R_1. Appliquer (i) pour (x, x, x, t).

□

A.3 Étiquettes pour l'E-matching

On aura besoin de quelques lemmes intermédiaires pour prouver que les étiquettes (f, n) donnent bien les solutions au problème $f\ ?x_1\ ...\ ?x_n \sim=_R C$, en particulier sur leur comportement vis-à-vis des règles de déduction. Je note $C_R(t)$ la classe d'un terme t pour une relation R.

D'abord, les étiquettes irréflexives, comme les classes, sont à un élément réflexif près identiques aux étiquettes normales.

Proposition A.3. *Pour toute classe C_R d'une PER R, il y a une équivalence*

$$C_R \textit{ a une étiquette } (f, n) \iff C_R \textit{ a une étiquette } (f, n) \textit{ irréflexive et } C_R \textit{ est réflexive.}$$

Démonstration. Supposons que C_R a une étiquette REFL(f, n). REFL est plus faible dans sa forme irréflexive, donc C_R a également une étiquette REFL(f, n) irréflexive. De plus, C_R est réflexive puisqu'elle possède une étiquette standard.

Maintenant, si C_R a une étiquette REFL(f, n) et est réflexive, alors le témoin $f\ t_1\ ...\ t_n$ qui est réflexif par hypothèse justifie l'attribution d'une étiquette REFL$(, n)$ standard.

Enfin, les étiquettes CONG sont définies de façon identiques donc se transportent directement. □

Le lemme suivant met en relation les étiquettes de R et $\hat{R}$ vues comme des relations distinctes, et est utilisé pour traduires les compatibilités en inclusions.

Proposition A.4. *Étant donnée la classe $C_R(t)$ d'un terme t pour une PER R, il y a une équivalence*

$$C_R(t) \textit{ a une étiquette } (f, n) \textit{ irréflexive} \iff C_{\hat{R}}(t) \textit{ a une étiquette } (f, n).$$

Démonstration. Si $C_R(t)$ est réflexive, les deux propositions reviennent à l'existence d'une étiquette (f, n) pour C_R, donc elles sont équivalentes. Sinon, $C_{\hat{R}}(t)$ est exactement $C_R(t)$ avec des éléments réflexifs, donc on peut conclure par la Proposition A.3. □

Enfin, ces deux propositions décrivent le comportement des étiquettes vis-à-vis de l'inclusion et de la compatibilité. On retrouve vraiment la même algèbre qu'avec les relations.

Proposition A.5 (Les étiquettes se propagent par inclusion)**.** *Si $R' \subseteq R$ et $C_{R'}(t)$ a une étiquette (f, n), alors $C_R(t)$ a également une étiquette (f, n).*

Démonstration. Par induction sur la structure bien fondée (par décroissance de n) de l'étiquette.

REFL Si $C_{R'}(t)$ a une étiquette REFL(f, n) avec un témoin $f\ t_1\ ...\ t_n$, alors par inclusion ce témoin est dans $C_R(t)$ et aussi réflexif, donc $C_R(t)$ a aussi une étiquette REFL(f, n).

CONG Dans le cas récursif, $C_{R'}(t)$ a une étiquette CONG(f, n) avec un témoin $u\ v$ tel que $C_{R''\Rightarrow R'}(u)$ a une étiquette $(f, n-1)$. Puisque $R''\Rightarrow R' \subseteq R''\Rightarrow R$, par hypothèse de récurrence $C_{R''\Rightarrow R}(u)$ a aussi une étiquette $(f, n-1)$. v est toujours réflexif, donc $C_R(t)$ a une étiquette CONG(f, n) avec le même témoin.

□

Proposition A.6 (Les étiquettes se propagent par compatibilité)**.** *Si $R' \sqsubseteq R$ et $C_{R'}(t)$ a une étiquette (f, n), alors $C_R(t)$ a une étiquette (f, n) irréflexive.*

Démonstration. $R' \sqsubseteq R$ se traduit par $R' \subseteq \hat{R}$, donc d'après la Proposition A.5 $C_{\hat{R}}(t)$ a une étiquette (f, n), et par la Proposition A.4, $C_R(t)$ en a une irréflexive. □

On arrive alors au théorème principal sur les étiquettes.

Theorem A.1 (Correction et complétude de l'étiquetage)**.** *Une classe C_R d'une PER R a une étiquette (f, n) si et seulement si le problème*

$$f\ ?x_1\ ...\ ?x_n \sim_{=R} C_R$$

a des solutions, et a une étiquette (f, n) irréflexive si et seulement si sa variante irréflexive a des solutions.

Démonstration. Notons $m = f\ ?x_1\ ...\ ?x_n$.

Correction Par induction sur la structure bien fondée de l'étiquette.

- Si C_R a une étiquette REFL(f, n), alors elle contient un terme réflexif t de la forme $f\ t_1\ ...\ t_n$. La substitution $\sigma = \{?x_i := t_i : 1 \leq i \leq n\}$ est telle que $\sigma(m) = t$, donc on peut remplacer un t par $\sigma(m)$ dans $t\ R\ t$ et obtenir $\sigma(m)\ R\ t$ du fait que $= \sqsubseteq R$. Donc le problème d'E-matching a une solution.

  De même, si C_R a une étiquette irréflexive REFL(f, n), on a toujours $\sigma(m) = t$; d'après l'argument précédent, $t\ R\ t \Longrightarrow \sigma(m)\ R\ t$, c'est-à-dire la définition de $\sigma(m)\ \hat{R}\ t$. Donc le problème irréflexif a une solution.
- Si C_R a une étiquette CONG(f, n), par hypothèse de récurrence on a une solution (σ, u') (où $u' \in C_{R'\Rightarrow R}(u)$) au sous-problème

  $$f\ ?x_1\ ...\ ?x_{n-1} \sim_{=R'\Rightarrow R} C_{R'\Rightarrow R}(u).$$

  Par définition de la solution, u est en relation avec u' et $\sigma(f\ ?x_1\ ...\ ?x_{n-1})$ par la relation extensionnelle $R'\Rightarrow R$. On étend σ avec $?x_n := v$, qui est une solution au sous-problème

  $$?x_n \sim_{=R'} C_{R'}(v)$$

  puisque v est réflexif. On peut maintenant appliquer la règle CONGRUENCE puisque $C_{R'\Rightarrow R}(u)$ est réflexive, ce qui nous permet de conclure que

  $$\sigma(m)\ R\ u\ v.$$

  Donc les problèmes standard et irréflexif ont tous les deux des solutions.

Complétude Supposons maintenant que le problème $m \sim\!=_R C_R$ a une solution (σ, t). Il y a une dérivation finie de $\sigma(m)\ R\ t$ dans le système de déduction utilisant les relations d'équivalence, la règle CONGRUENCE, l'inclusion et la compatibilité. Notez que $\sigma(m)$ peut ne pas être un terme du problème, mais tous les symboles de fonctions et les feuilles de l'arbre qui invoquent des hypothèses du problème le sont.

Si le problème irréflexif $m \sim\!=_{\hat{R}} C_R$ a une solution (σ, t), alors on a de la même façon une dérivation de $\sigma(m)\ \hat{R}\ t$.

Cette preuve procède par induction sur la structure de l'arbre de dérivation, en utilisant les deux formes d'E-matching pendant la descente. Prouvons d'abord les cas de base :

- Les feuilles sont toujours des cas d'E-matching standard car aucune règle terminale ne permet de dériver $\sigma(m)\ \hat{R}\ t$. $\sigma(m) = f\ \sigma(x_1)\ ...\ \sigma(x_n)$ est nécessairement un terme du problème donc il est réflexif du fait de sa relation avec t. Donc la classe de t a une étiquette REFL(f, n).

Voyons maintenant les cas inductifs pour l'E-matching standard :

- Si $\sigma(m)\ R\ t$ est dérivé par symétrie ou transitivité, on peut conclure immédiatement par hypothèse de récurrence car les étiquettes sont préservées d'un élément équivalent à l'autre (puisqu'elles sont attribuées à la classe entière).
- Supposons que $\sigma(m)\ R\ t$ est obtenu par inclusion depuis une autre relation R'. Par hypothèse de récurrence, la classe $C_{R'}(t)$ a une étiquette (f, n), donc par la Proposition A.5 $C_R(t)$ en a une également.
- Enfin, si $\sigma(m)\ R\ t$ est obtenu par promotion de $\hat{R}$, alors on a $\sigma(m)\ \hat{R}\ t$ et $t\ R\ t$. Par hypothèse de récurrence en utilisant la variante irréflexive du problème, la classe $C_R(t)$ a une étiquette (f, n) irréflexive. Mais $C_R(t)$ est réflexive puisque $t\ R\ t$, donc d'après la Proposition A.3 elle a une étiquette (f, n).

Enfin, voyons les cas inductifs pour l'E-matching irréflexif :

- Symétrie et transitivité se traitent immédiatement de la même façon car les tags irréflexifs s'appliquent aux classes entières.
- Si $\sigma(m)\ \hat{R}\ t$ est obtenu par compatibilité, alors on a $\sigma(m)\ R'\ t$ pour une certaine relation $R' \sqsubseteq R$. Par hypothèse de récurrence en utilisant la variante standard du problème, $C_{R'}(t)$ a une étiquette (f, n). Donc, d'après la Proposition A.6, $C_R(t)$ a une étiquette (f, n) irréflexive.

Dans tous les cas, $C_R(t)$ a une étiquette (f, n) (E-matching standard) ou une étiquette (f, n) irréflexive (E-matching irréflexif), donc le système d'étiquetage est complet. □

Bibliographie

[BFR17] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 214–230. Springer, 2017.

[BT00] Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In *International Conference on Automated Deduction*, pages 64–78. Springer, 2000.

[BTV03] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2) :129–168, 2003.

[Cha18] Nicolas Chataing. Procédure de décision pour relations d'équivalences. `www.eleves.ens.fr/home/nchataing/rapport_congruence_tools.pdf`, 2018.

[Coq] The Coq Workshop 2020. `https://coq-workshop.gitlab.io/2020/`.

[Cor01] Pierre Corbineau. Autour de la clôture de congruence avec Coq. 2001.

[Cor06] Pierre Corbineau. Deciding equality in the constructor theory. In *International Workshop on Types for Proofs and Programs*, pages 78–92. Springer, 2006.

[CRB] Xavier Clerc, Leonid Rozenberg, and Anton Bachin. Code coverage for OCaml. `https://opam.ocaml.org/packages/bisect_ppx/`.

[DST80] Peter J Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM (JACM)*, 27(4) :758–771, 1980.

[NO80] Greg Nelson and Derek C Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2) :356–364, 1980.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications*, pages 453–468. Springer, 2005.

[Pau18] Gabrielle Pauvert. Decision Procedure for Equivalence Relations. Copie disponible à l'adresse `http://perso.ens-lyon.fr/sebastien.michelland/rapport_pauvert_congruence.pdf`, 2018.

[Sho82] Robert E Shostak. Deciding combinations of theories. In *International Conference on Automated Deduction*, pages 209–222. Springer, 1982.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2) :215–225, 1975.

A Decision Procedure for Equivalence Relations

Sébastien Michelland[1], Pierre Corbineau[2], Lionel Rieg[2], and Karine Altisen[2]

[1] ENS de Lyon
sebastien.michelland@ens-lyon.fr
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG
<First Name>.<Last Name>@univ-grenoble-alpes.fr

1 Motivation

Equality or equivalence between objects is a typical kind of Coq goal that regularly appears in theories. When it is a consequence of the context, one can simply `rewrite` their way through, but it is more convenient to handle it with a tactic. The `congruence` tactic by Pierre Corbineau [1] is a decision procedure designed to solve these problems.

For instance, given some lists `l1`, `l2` and `l3`, `congruence` will prove the following goal.

```
Goal:  l1 ++ l2 = l3 -> l2 ++ l1 = l3 ->
       (forall x y z, (x ++ y) ++ z = x ++ (y ++ z)) ->
       l1 ++ l3 = l3 ++ l1.
```

The `congruence` tactic uses the congruence closure algorithm by Downey, Sethi and Tarjan [3] to saturate the set of known equalities; this method basically propagates equalities through calls to equal functions. Proofs are tracked along the way in the style of [4]. The tactic is also able to instantiate quantified equalities such as the associativity property of `++`, and use injectivity and discrimination of inductive constructors [2].

One of the limits of `congruence` is its restriction to the Coq equality `eq` even for functions or propositions. This is not structural since the congruence closure method applies equally well to any equivalence relation. Using equivalence relations would also prompt the tactic to work with typeclasses and setoids, which are more common in modern Coq code. As an additional benefit, typeclasses register equivalence relations and let the tactic find them automatically.

2 Generalization to Equivalence Relations

We propose to generalize `congruence` by supporting hierarchies of equivalence relations in the congruence closure algorithm. This new tactic would be used to decide ground relations of the form `R x y` using relations in the context. When universally-quantified relation hypotheses are involved, we can only hope to have a semi-decision procedure.

An instance of relation hierarchy on lists would be

$$\text{List equality } (=) \subseteq \text{Multiset equality } (==) \subseteq \text{Set equality } (===),$$

which is modeled by three instances of the `Equivalence` typeclass and two instances of `subrelation`. These instances can be detected automatically through the typeclass mechanism even when not in the current context. This generalization substantially raises the expressiveness, as the tactic can now use (or even discover) properties like

```
forall l1 l2, l1 ++ l2 == l2 ++ l1,
```

and automatically transpose them to weaker relations (in this case, set equality). Because this method supports arbitrary user-defined relations, we expect it to be useful in a wider variety of contexts than `congruence`.

The congruence closure algorithm and its data structures are extended to support multiple relations and proof trees for each term in the system. Universally-quantified hypotheses are instantiated by matching the quantified variables against equivalence classes instead of individual elements, as this reduces the number of redundant matches.

The tactic also works with arbitrary non-reflexive equivalence relations (called partial equivalence relations or PERs), which allows functions to be treated in the same way as the objects they are manipulating. PERs are especially useful to describe morphisms of relations (typically `R x y -> R' (f x) (f y)`) which correspond to the extensional equality found in `congruence`.

A natural extension of this method is the support for reasoning modulo logical equivalence on `Prop`. A proposition that is found equivalent to `True` or `False` for `<->` is in fact decided, and can thus be proven or disproven. This makes it possible to deal with arbitrary propositions as goals (instead of just `R x y`) and prove them equivalent to `True`.

3 Development of the Tactic

The tactic started as a command-line tool for testing and is currently being integrated into Coq as a plugin. It generates standalone Coq proof scripts which can be checked independently. It implements the extended congruence closure algorithm, along with inclusion hierarchies and PERs. Universally-quantified relation hypotheses can be instantiated by the matching algorithm, which makes it possible to describe simple first-order theories.

The main challenges and upcoming work mainly lie in the following directions:

- Design heuristics to control the instantiation of universally-quantified hypotheses (to avoid combinatorial issues).
- Use the Coq context and typeclass information to limit the amount of input from the user.
- Experiment with reasoning modulo logical equivalence.

References

[1] `congruence` tactic in the Coq documentation. https://coq.inria.fr/distrib/current/refman/proof-engine/tactics.html#coq:tacn.congruence. Accessed 2020-04-20.

[2] Pierre Corbineau. Deciding equality in the constructor theory. In *International Workshop on Types for Proofs and Programs*, pages 78–92. Springer, 2006.

[3] Peter J Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM (JACM)*, 27(4):758–771, 1980.

[4] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications*, pages 453–468. Springer, 2005.