Congruence Closure in Intensional Type Theory

Daniel Selsam[1] and Leonardo de Moura[2](✉)

[1] Stanford University, Stanford, USA
dselsam@stanford.edu
[2] Microsoft Research, Redmond, USA
leonardo@microsoft.com

Abstract. Congruence closure procedures are used extensively in automated reasoning and are a core component of most satisfiability modulo theories solvers. However, no known congruence closure algorithms can support any of the expressive logics based on intensional type theory (ITT), which form the basis of many interactive theorem provers. The main source of expressiveness in these logics is dependent types, and yet existing congruence closure procedures found in interactive theorem provers based on ITT do not handle dependent types at all and only work on the simply-typed subsets of the logics. Here we present an efficient and proof-producing congruence closure procedure that applies to every function in ITT no matter how many dependencies exist among its arguments, and that only relies on the commonly assumed *uniqueness of identity proofs* axiom. We demonstrate its usefulness by solving interesting verification problems involving functions with dependent types.

1 Introduction

Congruence closure procedures are used extensively in automated reasoning, since almost all proofs in both program verification and formalized mathematics require reasoning about equalities [23]. The algorithm constitutes a fundamental component of most satisfiability modulo theories (SMT) solvers [4,20]; it is often distinguished as the "core theory solver", and is responsible for communicating literal assignments to the underlying SAT solver and equalities to the other "satellite solvers" [10,20]. However, no known congruence closure algorithms can support any of the expressive logics based on intensional type theory (ITT). Yet despite the lack of an algorithm for congruence closure, the benefits that ITTs confer in terms of expressiveness, elegance, and trustworthiness have proved substantial enough that different flavors of ITT form the basis of many interactive theorem provers, such as Coq [8], Lean [21], and Matita [2], and also several emerging programming languages, such as Agda [5], Epigram [16], and Idris [6]. Many of the most striking successes in both certified programming and formalized mathematics have been in variants of ITT, such as the development of a fully-certified compiler for most of the C language [14] and the formalization of the odd-order theorem [11].

There are currently two main workarounds for the lack of a congruence closure algorithm for ITT, and for the lack of robust theorem proving tools for ITT

N. Olivetti and A. Tiwari (Eds.): IJCAR 2016, LNAI 9706, pp. 99–115, 2016.
DOI: 10.1007/978-3-319-40229-1_8

more generally. One option is to rely much more on manual proving. Although many impressive projects have been formalized with little to no automation, this approach is not very attractive since the cost of manual proving can be tremendous. We believe that as long as extensive manual proving is a central part of writing certified software or formalizing mathematics, these will remain niche activities for the rare expert. The other option is to relinquish the use of dependent types whenever manual reasoning becomes too burdensome so that more traditional automation can be used. Note that the Coq system even has a tactic `congruence` that performs congruence closure, but it does not handle dependent types at all and only works on the simply-typed subset of the language. This sacrifice may be appropriate in certain contexts, but losing all the benefits of dependent types makes this an unsatisfactory solution in general.

Given the limitations of these two workarounds, it would be preferable to perform congruence closure and other types of automated reasoning directly in the richer language of ITT. Unfortunately, equality and congruence are both surprisingly subtle in ITT, and as we will see, the theorem that could justify using the standard congruence closure procedure for functions with dependent types is not provable in the core logic, nor does it follow from any of the axioms commonly assumed in existing systems. In this paper, we introduce a new notion of congruence that applies to every function in ITT no matter how many dependencies exist among its arguments, along with a simple and efficient extension of the standard congruence closure procedure to fully automate reasoning about this more general notion of congruence. Our procedure is applicable to a wide variety of projects since it only relies on the *uniqueness of identity proofs* axiom, which is built into the logic of many systems including Agda, Idris, and Lean, and which is commonly assumed in the others. We hope our procedure helps make it possible for users to have the best of both worlds: to reap all the benefits of dependent types while still enjoying all the power of traditional automation.

2 Preliminaries

We assume the term language is a dependent λ-calculus in which terms are described by the following grammar:

$$t,s ::= x \mid c \mid \mathtt{Type} \mid t\ s \mid \lambda x : s, t \mid \Pi x : s, t$$

where x is a variable and c is a constant. To simplify the presentation, we omit type universes at sort `Type`. It is not relevant to this paper whether the universe hierarchy is cumulative or not, nor whether there is a distinguished sort `Prop` (the sort of all propositions). The term Π`x:A, B` denotes the type of functions `f` that map any element `a:A` to an element of `B[a/x]`. When `x` appears in `B` we say that `f` is *dependently-typed*; otherwise we write Π`x:A, B` as `A` $\rightarrow$ `B` to denote the usual non-dependent function space. When `B` is a proposition, Π`x:A, B` can be read as the universally quantified formula $\forall$`x:A, B`, or as the logical implication `A` $\Rightarrow$ `B` if `x` does not appear in `B`. The term `f a` denotes a function application, and the lambda abstraction λ`x:A, t` denotes a function that given an element `a` of type

$\texttt{A}$ produces $\texttt{t[a/x]}$. As usual in Type Theory, a *context* Γ is a sequence of *typing assumptions* $\texttt{a:A}$ and (local) definitions $\texttt{c:A} := \texttt{t}$, where $\texttt{t}$ has type $\texttt{A}$ and $\texttt{c}$ does not occur in $\texttt{t}$. We often omit the type $\texttt{A}$ and simply write $\texttt{c} := \texttt{t}$ to save space when no confusion arises. Similarly, an *environment* Δ is a sequence of (global) definitions $\texttt{f:A} := \texttt{t}$. We use $type(\Delta, \Gamma, t)$ to denote the type of t with respect to Δ and Γ, and $type(t)$ when no confusion arises. Given an environment Δ and a context Γ, every term reduces to a normal form by the standard $\beta\delta\eta\iota\zeta$-reduction rules. For this paper we will assume a fixed environment Δ that contains all definitions and theorems that we present. As usual, we write $\Pi(\texttt{a:A})(\texttt{b:B}),\texttt{C}$ as a shorthand for $\Pi\texttt{a:A,}(\Pi\texttt{b:B,C})$. We use a similar shorthand for λ-terms.

2.1 Equality

One of the reasons that congruence is subtle in ITT is that equality itself is subtle in ITT. The single notion of equality in most other logics splits into at least three different yet related notions in ITT.

Definitional equality. The first notion of equality in ITT is *definitional equality.* We write $\texttt{a} \equiv \texttt{b}$ to mean that $\texttt{a}$ and $\texttt{b}$ are equal by definition, which is the case whenever $\texttt{a}$ and $\texttt{b}$ reduce to the same normal form. For example, if we define a function $\texttt{f} : \mathbb{N} \to \mathbb{N} := \lambda\ \texttt{n} : \mathbb{N}, 0$ in the environment Δ, then the terms 0 and $\texttt{f}\ 0$ both reduce to the same normal form 0 and so are equal by definition. On the other hand, $(\lambda\ \texttt{n m}\colon \mathbb{N}, \texttt{n} + \texttt{m})$ is not definitionally equal to $(\lambda\ \texttt{n m}\colon \mathbb{N}, \texttt{m} + \texttt{n})$, since they are both in normal form and these normal forms are not the same. Note that definitional equality is a judgment at the meta-level, and the theory itself cannot refer to it; in particular, it is not possible to assume or negate a definitional equality.

Homogeneous propositional equality. The second notion of equality in ITT is *homogeneous propositional equality*, which we will usually shorten to *homogeneous equality* since "propositional" is implied. Unlike definitional equality which is a judgment at the meta-level, homogeneous equality can be assumed, negated, and proved inside the logic itself. There is a constant $\texttt{eq} : \Pi\ (\texttt{A} : \texttt{Type}),\ \texttt{A} \to \texttt{A} \to \texttt{Type}$ in Δ such that, for any type $\texttt{A}$ and elements $\texttt{a b : A}$, the expression $\texttt{eq A a b}$ represents the proposition that $\texttt{a}$ and $\texttt{b}$ are "equal". Note that we call this homogeneous equality because the types of $\texttt{a}$ and $\texttt{b}$ must be definitionally equal to even *state* the proposition that $\texttt{a}$ and $\texttt{b}$ are equal. We write $\texttt{a} =_{\texttt{A}} \texttt{b}$ as shorthand for $\texttt{eq A a b}$, or $\texttt{a} = \texttt{b}$ if the type $\texttt{A}$ is clear from context. We say a term $\texttt{t}$ of type $\texttt{a} = \texttt{b}$ is a *proof* for $\texttt{a} = \texttt{b}$.

The meaning of homogeneous equality is given by the introduction and elimination rules for $\texttt{eq}$, which state how to prove that two elements are equal and what one can do with such a proof respectively. The introduction rule for $\texttt{eq}$ is the dependent function $\texttt{refl} : \Pi\ (\texttt{A} : \texttt{Type})\ (\texttt{a} : \texttt{A}),\ \texttt{a} = \texttt{a}$, which says that every element of type $\texttt{A}$ is equal to itself. We call $\texttt{refl}$ the reflexivity axiom, and write $\texttt{refl a}$ whenever the type $\texttt{A}$ is clear from context. Note that if $\texttt{a b : A}$ are definitionally equal, then $\texttt{refl a}$ is a proof for $\texttt{a} = \texttt{b}$. The elimination principle (also known as the recursor) for the type $\texttt{eq}$ is the dependent function $\texttt{erec}$:

$$\mathtt{erec} : \Pi\ (\mathtt{A} : \mathtt{Type})\ (\mathtt{a} : \mathtt{A})\ (\mathtt{C} : \mathtt{A} \to \mathtt{Type}),\ \mathtt{C\ a} \to \Pi\ (\mathtt{b} : \mathtt{A}),\ \mathtt{a} = \mathtt{b} \to \mathtt{C\ b}$$

This principle states that if a property C holds for an element a, and $\mathtt{a} = \mathtt{b}$ for some b, then we can conclude that C must hold of b as well. We say C is the *motive*, and we write (erec C p e) instead of (erec A a C p b e) since A, a and b can be inferred easily from $\mathtt{e} : \mathtt{a} = \mathtt{b}$. Note that by setting C to be the identity function $\mathtt{id} : \mathtt{Type} \to \mathtt{Type}$, erec can be used to change the type of a term to an equal type; that is, given a term $\mathtt{a} : \mathtt{A}$ and a proof $\mathtt{e} : \mathtt{A} = \mathtt{B}$, the term (erec id a e) has type B. We call this a *cast*, and say that we *cast* a to have type B. Note that it is straightforward to use erec and refl to prove that eq is symmetric and transitive and hence an equivalence relation.

Heterogeneous propositional equality. As we saw above, homogeneous equality suffers from a peculiar limitation: it is not even possible to form the proposition $\mathtt{a} = \mathtt{b}$ unless the types of a and b are definitionally equal. The further one strays from the familiar confines of simple type theory, the more severe this handicap becomes. For example, a common use of dependent types is to include the length of a list inside its type in order to make out-of-bounds errors impossible. The resulting type is often called a *vector* and has type $\mathtt{vector} : \Pi\ (\mathtt{A} : \mathtt{Type}),\ \mathbb{N} \to \mathtt{Type}$. It is easy to define an append function on vectors:

$$\mathtt{app} : \Pi\ (\mathtt{A} : \mathtt{Type})\ (\mathtt{n\ m} : \mathbb{N}),\ \mathtt{vector\ A\ n} \to \mathtt{vector\ A\ m} \to \mathtt{vector\ A}\ (\mathtt{n} + \mathtt{m})$$

However, we cannot even state the proposition that app is associative using homogeneous equality, since the type $\mathtt{vector\ A}\ (\mathtt{n} + (\mathtt{m} + \mathtt{k}))$ is not definitionally equal to the type $\mathtt{vector\ A}\ ((\mathtt{n} + \mathtt{m}) + \mathtt{k})$, only propositionally equal. The same issue arises when reasoning about vectors in mathematics. For example, we cannot even state the proposition that concatenating zero-vectors of different lengths m and n over the real numbers $\mathbb{R}$ is commutative, since the type $\mathbb{R}^{m+n}$ is not definitionally equal to the type $\mathbb{R}^{n+m}$. In both cases, we could use erec to cast one of the two terms to have the type of the other, but this approach would quickly become unwieldy as the number of dependencies increased, and moreover every procedure that reasoned about equality would need to do so modulo casts.

Thus there is a need for a third notion of equality in ITT, *heterogeneous propositional equality*, which we will usually shorten to *heterogeneous equality* since "propositional" is implied. There is a constant $\mathtt{heq} : \Pi\ (\mathtt{A} : \mathtt{Type})\ (\mathtt{B} : \mathtt{Type}),\ \mathtt{A} \to \mathtt{B} \to \mathtt{Type}$ that behaves like eq except that its arguments may have different types.[1] We write $\mathtt{a} == \mathtt{b}$ as shorthand for heq A B a b. Heterogeneous equality has an introduction rule $\mathtt{hrefl} : \Pi\ (\mathtt{A} : \mathtt{Type})\ (\mathtt{a} : \mathtt{A}),\ \mathtt{a} == \mathtt{a}$ analogous to refl, and it is straightforward to show that heq is an equivalence relation by proving the following theorems:

$$\mathtt{hsymm} : \Pi\ (\mathtt{A\ B} : \mathtt{Type})\ (\mathtt{a} : \mathtt{A})\ (\mathtt{b} : \mathtt{B}),\ \mathtt{a} == \mathtt{b} \to \mathtt{b} == \mathtt{a}$$
$$\mathtt{htrans} : \Pi\ (\mathtt{A\ B\ C} : \mathtt{Type})\ (\mathtt{a} : \mathtt{A})\ (\mathtt{b} : \mathtt{B})\ (\mathtt{c} : \mathtt{C}),\ \mathtt{a} == \mathtt{b} \to \mathtt{b} == \mathtt{c} \to \mathtt{a} == \mathtt{c}$$

[1] There are many equivalent ways of defining heq. One popular way is "John Major equality" [15]. Additional formulations and formal proofs of equivalence can be found at http://leanprover.github.io/ijcar16/congr.lean.

Unfortunately, the flexibility of `heq` does not come without a cost: as we discuss in Sect. 3, `heq` turns out to be weaker than `eq` in subtle ways and does not permit as simple a notion of congruence.

Converting from heterogeneous equality to homogeneous equality. It is straightforward to convert a proof of homogeneous equality `p : a = b` into one of heterogeneous equality using the lemma

```
lemma ofeq (A : Type) (a b : A) : a = b → a == b
```

However, we must assume an axiom in order to prove the reverse direction

```
ofheq (A : Type) (a b : A) : a == b → a = b
```

The statement is equivalent to the *uniqueness of identity proofs* (UIP) principle [26], to Streicher's *Axiom K* [26], and to a few other variants as well. Although these axioms are not part of the core logic of ITT, they have been found to be consistent with ITT by means of a meta-theoretic argument [18], and are built into the logic of many systems including Agda, Idris, and Lean. They also follow from various stronger axioms that are commonly assumed, such as *proof irrelevance* and *excluded middle*. In Coq, UIP or an axiom that implies it is often assumed when heterogeneous equality is used, including in the CompCert project [14]. Our approach is built upon being able to recover homogeneous equalities from heterogeneous equalities between two terms of the same type and so makes heavy use of `ofheq`.

3 Congruence

Congruence over homogeneous equality. It is straightforward to prove the following lemma using `erec`:

```
lemma congr : Π (A B : Type) (f g : A → B) (a b : A), f = g → a = b → f a = g b
```

and thus prove that `eq` is indeed a congruence relation for simply-typed functions. Thus the standard congruence closure algorithm can be applied to the simply-typed subset of ITT without much complication. In particular, we have the familiar property that `f a` and `g b` can be proved equal if and only if either an equality `f a = g b` has been asserted, or if `f` can be proved equal to `g` and `a` can be proved equal to `b`.

Congruence over heterogeneous equality. Unfortunately, once we introduce functions with dependent types, we must switch to `heq` and lose the familiar property discussed above that `eq` satisfies for simply-typed functions. Ideally we would like the following congruence lemma for heterogeneous equality:

```
hcongr_ideal : Π (A A′ : Type) (B : A → Type) (B′ : A′ → Type)
   (f : Π (a : A), B a) (f′ : Π (a′ : A′), B′ a′) (a : A) (a′ : A′),
   f == f′ → a == a′ → f a == f′ a′
```

Unfortunately, this theorem is not provable in ITT [1], even when we assume UIP. The issue is that we need to establish that B = B′ as well, and this fact does not follow from (Π (a : A), B a) = (Π (a′ : A′), B′ a′). Assuming hcongr_ideal as an axiom is not a satisfactory solution because it would limit the applicability of our approach, since as far as we know it is not assumed in any existing interactive theorem provers based on ITT.

However, for any given n, it is straightforward to prove the following congruence lemma using only erec, ofheq and hrefl[2]:

```
lemma hcongr_n
  (A_1: Type)
  (A_2: A → Type)
  ...
  (A_n: Π a_1 ... a_{n-2}, A_{n-1} a_1 ... a_{n-2} → Type)
  (B: Π a_1 ... a_{n-1}, A_n a_1 ... a_{n-1} → Type) :
  Π (f g: Π a_1 ... a_n, B a_1 ... a_n), f = g →
  Π (a_1 b_1: A_1), a_1 == b_1 →
  Π (a_2: A_2 a_1) (b_2: A_2 b_1), a_2 == b_2 →
  ...
  Π (a_n: A_n a_1 ... a_{n-1}) (b_n : A_n b_1 ... b_{n-1}), a_n == b_n →
  f a_1 ... a_n == g b_1 ... b_n
```

The lemmas $\texttt{hcongr}_n$ are weaker than hcongr_ideal because they require the outermost functions f and g to have the same type. Although we no longer have the property that f == g and a == b implies f a == g b, we show in the next section how to extend the congruence closure algorithm to deal with the additional restriction imposed by $\texttt{hcongr}_n$.

When using $\texttt{hcongr}_n$ lemmas, we omit the parameters $\texttt{A}_i$, B, $\texttt{a}_i$ and $\texttt{b}_i$ since they can be inferred from the parameters with types f = g and $\texttt{a}_i == \texttt{b}_i$. Note that even if some arguments of an n-ary function f do not depend on all previous ones, it is still straightforward to find parameters $\texttt{A}_i$ and B that do depend on all previous arguments and so fit the theorem, and yet become definitionally equal to the types of the actual arguments of f once applied to the preceding arguments. We remark that we avoid this issue in our implementation by synthesizing custom congruence theorems for every function we encounter.

4 Congruence Closure

We now have all the necessary ingredients to describe a very general congruence closure procedure for ITT. Our procedure is based on the one proposed by Nieuwenhuis and Oliveras [24] for first-order logic, which is efficient, is proof producing, and is used by many SMT solvers. We assume the input to our congruence closure procedure is of the form $\Gamma \vdash$ a == b, where Γ is a context and a == b is the goal. Note that a goal of the form a = b can be converted into

[2] The formal statements and proofs for small values of n can be found at http://leanprover.github.io/ijcar16/congr.lean, along with formal proofs of all other lemmas described in this paper.

$\mathtt{a} == \mathtt{b}$ before we start our procedure, since when a and b have the same type, any proof for $\mathtt{a} == \mathtt{b}$ can be converted into a proof for $\mathtt{a} = \mathtt{b}$ using ofheq. Similarly, any hypothesis of the form $\mathtt{e}\colon \mathtt{a} = \mathtt{b}$ can be replaced with $\mathtt{e}\colon \mathtt{a} == \mathtt{b}$ using ofeq. As in abstract congruence closure [3,13], we introduce new variables c to name all proper subterms of every term appearing on either side of an equality, both to simplify the presentation and to obtain the efficiency of DAG-based implementations.[3] For example, we encode $\mathtt{f\ N\ a} == \mathtt{f\ N\ b}$ using the local definitions $(\mathtt{c}_1 := \mathtt{f\ N})$ $(\mathtt{c}_2 := \mathtt{c}_1\ \mathtt{a})$ $(\mathtt{c}_3 := \mathtt{c}_1\ \mathtt{b})$ and the equality $\mathtt{c}_2 == \mathtt{c}_3$. We remark that $\mathtt{c}_2 == \mathtt{c}_3$ is definitionally equal to $\mathtt{f\ N\ a} == \mathtt{f\ N\ b}$ by ζ-reduction. Here is an example problem instance for our procedure:

$$(\mathtt{N}\colon \mathtt{Type})\ (\mathtt{a\ b}\colon \mathtt{N})\ (\mathtt{f}\colon \Pi\ \mathtt{A}\colon \mathtt{Type},\ \mathtt{A} \to \mathtt{A})\ (\mathtt{c}_1 := \mathtt{f\ N})$$
$$(\mathtt{c}_2 := \mathtt{c}_1\ \mathtt{a})\ (\mathtt{c}_3 := \mathtt{c}_1\ \mathtt{b})\ (\mathtt{e}\colon \mathtt{a} == \mathtt{b}) \vdash \mathtt{c}_2 == \mathtt{c}_3$$

The term $(\mathtt{hcongr}_2\ (\mathtt{refl\,f})\ (\mathtt{hrefl\,N})\ \mathtt{e})$ is a proof for the goal $\mathtt{c}_2 == \mathtt{c}_3$.

As in most congruence closure procedures, ours maintains a union-find data structure that partitions the set of terms into a number of disjoint subsets such that if a and b are in the same subset (denoted $\mathtt{a} \approx \mathtt{b}$) then the procedure can generate a proof that $\mathtt{a} == \mathtt{b}$. Each subset is an *equivalence class*. The union-find data structure computes the equivalence closure of the relation $==$ by merging the equivalence classes of a and b whenever $\mathtt{e}\colon \mathtt{a} == \mathtt{b}$ is asserted. However, the union-find data structure alone does not know anything about congruence, and in particular it will not automatically propagate the assertion $\mathtt{a} == \mathtt{b}$ to other terms that contain a or b; for example, it would not merge the equivalence classes of $\mathtt{c} := \mathtt{f\ a}$ and $\mathtt{d} := \mathtt{f\ b}$. Thus, additional machinery is required to find and propagate new equivalences implied by the rules of congruence.

We say that two terms are *congruent* if they can be proved to be equivalent using a congruence rule given the current partition of the union-find data structure. We also say two local definitions $\mathtt{c} := \mathtt{f\ a}$ and $\mathtt{d} := \mathtt{g\ b}$ are congruent whenever f a and g b are congruent. We remark that congruence closure algorithms can be parameterized by the structure of the congruence rules they propagate. In our case, we use the family of $\mathtt{hcongr}_n$ lemmas as congruence rules.

We now describe our congruence closure procedure in full, although the overall structure is similar to the one presented in [24]. The key differences are in how we determine whether two terms are congruent, how we build formal proofs of congruence using $\mathtt{hcongr}_n$, and what local definitions we need to visit after merging two equivalence classes to ensure that all new congruences are detected. The basic data structures in our procedure are

- *repr*: a mapping from variables to variables, where $\mathit{repr}[x]$ is the representative for the equivalence class x is in. We say variable x is a *representative* if and only if $\mathit{repr}[x]$ is x.
- *next*: a mapping from variables to variables that induces a circular list for each equivalence class, where $\mathit{next}[x]$ is the next element in the equivalence class x is in.

[3] To simplify the presentation further, we ignore the possibility that any of these subterms themselves include partial applications of equality.

- *pr*: a mapping from variables to pairs consisting of a variable and a proof, where if $pr[x]$ is (y, p), then p is a proof for $x == y$ or $y == x$. We use $target[x]$ to denote $pr[x].1$. This structure implements the *proof forests* described in [24].
- *size*: a mapping from representatives to natural numbers, where for each representative x, $size[x]$ is the number of elements in the equivalence class represented by x.
- *pending*: a list of local definitions and typing assumptions to be processed. It is initialized with the context Γ.
- *congrtable*: a set of local definitions such that given a local definition E, the function $lookup(E)$ returns a local definition in *congrtable* congruent to E if one exists.
- *uselists*: a mapping from representatives to sets of local definitions, such that local definition D is in $uselists[x]$ if D might become congruent to another definition if the equivalence class of x were merged with another equivalence class.

Our procedure maintains the following invariants for the data structures described above.

1. $repr[next[x]] \equiv repr[repr[x]] \equiv repr[x]$
2. If $repr[x] \equiv repr[y]$, then $next^k[x] \equiv y$ for some k.
3. $target^k[x] \equiv repr[x]$ for some k. That is, we can view $target^k[x]$ as a "path" from x to $repr[x]$. Moreover, the proofs in pr can be used to build a proof from x to any element along this path.
4. Let s be $size[repr[x]]$, then $next^s[x] \equiv x$. That is, *next* does indeed induce a set of disjoint circular lists, one for each equivalence class.

Whenever a new congruence proof for `c == d` is inferred by our procedure, we add the auxiliary local definition `e: c == d := p` to *pending*, where `e` is a fresh variable, and `p` is a proof for `c == d`. The proof `p` is always an application of the lemma $\texttt{hcongr}_n$ for some n. We say `e : c == d` and `e: c == d := p` are *equality proofs* for `c == d`. Given an equality proof E, the functions $lhs(E)$ and $rhs(E)$ return the left and right hand sides of the proved equality. Given a local definition E of the form `c := f a`, the function $var(E)$ returns `c`, and $app(E)$ the pair `(f, a)`. We say a variable `c` is a local definition when Γ contains the definition `c := f a`, and the auxiliary partial function $def(\texttt{c})$ returns this local definition.

Implementing congrtable. In order to implement the congruence closure procedure efficiently, the congruence rules must admit a data structure *congrtable* that takes a local definition and quickly returns a local definition in the table that it is congruent to if one exists. It is easy to implement such a data structure with a Boolean procedure CONGRUENT(D, E) that determines if two local definitions are congruent, along with a congruence-respecting hash function. Although the family of $\texttt{hcongr}_n$ lemmas does not satisfy the property that `f a` and `g b` are congruent whenever `f` $\approx$ `g` and `a` $\approx$ `b`, we still have a straightforward criterion for determining whether two terms are congruent.

Proposition 1. *Consider the terms* `f a` *and* `g b`*. If* `a` $\approx$ `b`*, then* `f a` *and* `g b` *are congruent provided either:*

1. `f` *and* `g` *are homogeneously equal;*
2. `f` *and* `g` *are congruent.*

Proof. First note that in both cases, we can generate a proof that `a == b` since we have assumed that `a` $\approx$ `b`. In the first case, if `f` and `g` are homogeneously equal, then no matter how many partial applications they contain, we can apply $\mathtt{hcongr}_1$ to the proof of homogeneous equality and the proof that `a == b`. In the second case, if `f` and `g` are congruent, it means that we can generate proofs of all the preconditions of $\mathtt{hcongr}_k$ for some k, and the only additional precondition to $\mathtt{hcongr}_{k+1}$ is a proof that `a == b`, which we can generate as well.

```
1: procedure CONGRUENT($D$, $E$)
2:     $(f, a) \leftarrow app(D)$; $(g, b) \leftarrow app(E)$
3:     return $a \approx b$ and
4:            [($f \approx g$ and $type(f) \equiv type(g)$) or
5:             ($f$ and $g$ are local definitions and CONGRUENT($def(f)$, $def(g)$))]
6: procedure CONGRHASH($D$)
7:     given: $h$, a hash function on terms
8:     $(f, a) \leftarrow app(D)$
9:     return $hashcombine(h(repr[f]), h(repr[a]))$
```

Fig. 1. Implementing *congrtable*

Proposition 1 suggests a simple recursive procedure to detect when two terms are congruent, which we present in Fig. 1. The procedure CONGRUENT(D, E), where D and E are local definitions of the form `c := f a` and `d := g b`, returns `true` if a proof for `c == d` can be constructed using an $\mathtt{hcongr}_n$ lemma for some n. Note that although the congruence lemmas $\mathtt{hcongr}_n$ are themselves n-ary, it is not sufficient to view the two terms being compared for congruence as applications of n-ary functions. We must compare each pair of partial applications for homogeneous equality as well (line 4), since two terms with n arguments each might be congruent using $\mathtt{hcongr}_m$ for any m such that $m \leq n$. For example, `f a1 c` and `g b1 c` are congruent by $\mathtt{hcongr}_2$ if `f = g` and `a1 == b1`, and yet are only congruent by $\mathtt{hcongr}_1$ if all we know is `f a1 = g b1`. It is even possible for two terms to be congruent that do not have the same number of arguments. For example, `f = g a` implies that `f b` and `g a b` are congruent by $\mathtt{hcongr}_1$.

Proposition 1 also suggests a simple way to hash local definitions that respects congruence. Given a hash function on terms, the procedure CONGRHASH(D) hashes a local definition of the form `c := f a` by simply combining the hashes of the representatives of `f` and `a`. This hash function respects congruence because if `c := f a` and `d := g b` are congruent, it is a necessary (though not sufficient) condition that `f` $\approx$ `g` and `a` $\approx$ `b`.

```
procedure CC($\Gamma \vdash a == b$)
    $pending \leftarrow \Gamma$
    while $pending$ is not empty do
        remove next $E$ from $pending$
        if $E$ is an equality proof then PROCESSEQ($E$)
        else INITIALIZE($E$)
    if $repr[a] \equiv repr[b]$ then return MKPR(a, b)
    else fail
```

Fig. 2. Congruence closure procedure

The procedure. Fig. 2 contains the main procedure CC. It initializes *pending* with the input context Γ. Variables in typing assumptions and local definitions are processed using INITIALIZE (Fig. 3), and equality proofs are processed using PROCESSEQ (Fig. 4).

```
procedure INITIALIZE($E$)
    $c \leftarrow var(E)$
    $repr[c] \leftarrow c$; $next[c] \leftarrow c$; $size[c] \leftarrow 1$; $uselists[c] \leftarrow \emptyset$
    $pr[c] \leftarrow (c,$‘hrefl $c$’$)$
    if $E$ is a local definition then
        INITUSELIST($E$, $E$)
        if $D = lookup(E)$ then
            $d \leftarrow var(D)$; $e \leftarrow$ make fresh variable
            add ($e : d == c :=$ MKCONGR($D$, $E$, [])) to $pending$ and $\Gamma$
        else add $E$ to $congrtable$
procedure INITUSELIST($E$, $P$)
    $(f, a) \leftarrow app(E)$
    add $P$ to $uselists[f]$ and $uselists[a]$
    if $f$ is a local definition then INITUSELIST($def(f)$, $P$)
```

Fig. 3. Initialization procedure

The INITIALIZE(E) procedure invokes INITUSELIST(E, E) whenever E is a local definition $c := f\ a$. The second argument at INITUSELIST(E, P) represents the *parent* local definition that must be included in the *uselists*. We must ensure that for every local definition D that could be inspected during a call to CONGRUENT(E_1, E_2) for some E_2, we add $var(E_1)$ to the *uselist* of $var(D)$ when initializing E_1. Thus the recursion in INITUSELIST must mirror the recursion in CONGRUENT conservatively, and always recurse whenever CONGRUENT might recurse. For example, assume the input context Γ contains

(A: Type) (a b d: A) (g : A → A → A) (f : A → A) (c_1 := g a) (c_2 := c_1 b) (c_3 := f d).

When INITIALIZE(c_2 := c_1 b) is invoked, c_2 := c_1 d is added to the *uselists* of c_1, b, g and a. By a slight abuse of notation, we write ‘hrefl a’ to represent in the pseudocode the expression that creates the `hrefl`-application using as argument the term stored in the program variable a.

The procedure PROCESSEQ is used to process equality proofs `a == b`. If `a` and `b` are already in the same equivalence class, it does nothing. Otherwise, it first removes every element in *uselists*[*repr*[a]] from *congrtable* (procedure REMOVEUSES). Then, it merges the equivalence classes of a and b so that for every a' in the equivalence class of a, *repr*[a'] is set to *repr*[b]. This operation can be implemented efficiently using the *next* data structure. As in [24], the procedure also reorients the path from a to *repr*[a] induced by *pr* (procedure FLIPPROOFS) to make sure invariant 3 is still satisfied and *locally irredundant transitivity proofs* [22] can be generated. It then reinserts the elements removed by REMOVEUSES into *congrtable* (procedure REINSERTUSES); if any are found to be congruent to an existing term in a different partition, it proves equivalence using the congruence lemma `hcongr`$_n$ (procedure MKCONGR) and puts the new proof onto the queue. Finally, PROCESSEQ updates *next*, *uselists* and *size* data structures.

```
procedure PROCESSEQ($E$)
    $a \leftarrow lhs(E)$;  $b \leftarrow rhs(E)$
    if $repr[a] \equiv repr[b]$ then return
    if $size(repr[a]) > size(repr[b])$ then swap$(a, b)$
    $r_a \leftarrow repr[a]$; $r_b \leftarrow repr[b]$
    REMOVEUSES($r_a$); FLIPPROOFS($a$)
    for all $a'$ s.t. $repr[a'] \equiv r_a$ do $repr[a'] \leftarrow r_b$
    $pr[a] \leftarrow (b, E)$
    REINSERTUSES($r_a$)
    swap$(next[r_a], next[r_b])$
    move $uselists[r_a]$ to $uselists[r_b]$; $size[r_b] \leftarrow size[r_b] + size[r_a]$
procedure FLIPPROOFS($a$)
    if $repr[a] \equiv a$ then return
    $(b, p) \leftarrow pr[a]$; FLIPPROOFS($b$); $pr[b] \leftarrow (a, p)$
procedure REMOVEUSES($a$)
    for all $E$ in $uselists[a]$ do remove $E$ from $congrtable$
procedure REINSERTUSES($a$)
    for all $E$ in $uselists[a]$ do
        if $D = lookup(E)$ then
            $d \leftarrow var(D)$; $e \leftarrow var(E)$; $p \leftarrow$ make fresh variable
            add $(p : d == e :=$ MKCONGR($D$, $E$, $[]$)$)$ to $pending$ and $\Gamma$
        else add $E$ to $congrtable$
```

Fig. 4. Process equality procedure

Figure 5 contains a simple recursive procedure MKCONGR to construct the proof that two congruent local definitions are equal. The procedure takes as input two local definitions D and E of the form `c := f a` and `d := g b` such that

CONGRUENT(D, E), along with a possibly empty list of equality proofs *es* for $\mathtt{a}_1$ == $\mathtt{b}_1$, ..., $\mathtt{a}_n$ == $\mathtt{b}_n$, and returns a proof for `f a a`$_1$ ... $\mathtt{a}_n$ == `g b b`$_1$ ... $\mathtt{b}_n$. The two cases in the MKCONGR procedure mirror the two cases of the CONGRUENT procedure. If the types of `f` and `g` are definitionally equal we construct an instance of the lemma $\mathtt{hcongr}_{|es|+1}$. The procedure MKPR(`a`, `b`) (Fig. 5) creates a proof for `a` == `b` if `a` and `b` are in the same equivalence class by finding the common element $target^n[\mathtt{a}] \equiv target^m[\mathtt{b}]$ in the "paths" from `a` and `b` to the equivalence class representative. Note that, if CONGRUENT(D, E) is true, then MKCONGR(D, E, []) is a proof for `c` == `d`.

```
procedure MKCONGR($D$, $E$, $es$)
    assumption: CONGRUENT($D$, $E$)
    $(f, a) \leftarrow app(D)$; $(g, b) \leftarrow app(E)$; $e_{ab} \leftarrow$ MKPR($a$, $b$)
    if $type(f) \equiv type(g)$ then
        $n \leftarrow len(es)$; $e_{fg} \leftarrow$ MKPR($f$, $g$)
        return ‘hcongr$_{n+1}$ (ofheq $e_{fg}$) $e_{ab}$ $es$’
    else return MKCONGR($def(f)$, $def(g)$, [$es$, $e_{ab}$])
procedure MKPR($a$, $b$)
    if $a \equiv b$ then return ‘hrefl $a$’
    let $n$ and $m$ be the smallest values s.t. $target^n[a] \equiv target^m[b]$
    $e_a \leftarrow$ MKTRANS($a$, $n$); $e_b \leftarrow$ MKTRANS($b$, $m$); return ‘htrans $e_a$ (hsymm $e_b$)’
procedure MKTRANS($a$, $n$)
    if $n = 0$ then return ‘hrefl $a$’
    $(b, e_{ab}) \leftarrow pr[a]$; $e \leftarrow$ MKTRANS($b$, $n - 1$)
    if $lhs(e_{ab}) \equiv a$ and $rhs(e_{ab}) \equiv b$ then return ‘htrans $e_{ab}$ $e$’
    else return ‘htrans (hsymm $e_{ab}$) $e$’
```

Fig. 5. Transitive proof generation procedure

Finally, we remark that the main loop of CC maintains the following two invariants.

Theorem 1. *If a and b are in the same equivalence class (i.e., $a \approx b$), then* MKPR(a, b) *returns a correct proof that a == b.*

Theorem 2. *If $type(f) \equiv type(g)$, $f \approx g$, $a_1 \approx b_1$, $\ldots a_n \approx b_n$, $c \equiv f\ a_1 \ldots a_n$ and $d \equiv g\ b_1 \ldots b_n$, then $c \approx d$.*

Extensions. There are many standard extensions to the congruence closure procedure that are straightforward to support in our framework, such as tracking disequalities to find contradictions and propagating injectivity and disjointness for inductive datatype constructors [17]. Here we present a simple extension for propagating equalities among elements of *subsingleton* types that is especially important when proving theorems in ITT. We say a type `A:Type` is a subsingleton if it has at most one element; that is, if for all `(a b:A)`, we have that `a` = `b`. Subsingletons are used extensively in practice, and are especially ubiquitous when *proof irrelevance* is assumed, in which case every proposition is a subsingleton.

One common use of dependent types is to extend functions to take extra arguments that represent proofs that certain preconditions hold. For example, the logarithm function only makes sense for positive real numbers, and we can make it impossible to even call it on a non-positive number by requiring a proof of positivity as a second argument: $c := f\ a$. The second argument is a proposition and hence is a subsingleton when we assume *proof irrelevance*. Consider the following goal: $(\mathtt{a}\ \mathtt{b} : \mathbb{R})\ (\mathtt{Ha} : \mathtt{a} > 0)\ (\mathtt{Hb} : \mathtt{b} > 0)\ (\mathtt{e} : \mathtt{a} = \mathtt{b}) \vdash$ `safe_log a Ha` $=$ `safe_log b Hb`. The core procedure we presented above would not be able to prove this theorem on its own because it would never discover that `Ha` $==$ `Hb`. We show how to extend the procedure to automatically propagate facts of this kind.

We assume we have an oracle $issub(\Gamma, A)$ that returns true for subsingleton types for which we have a proof Π`a b:A`, `a` $=$ `b`. Many proof assistants implement an efficient (and incomplete) *issub* using *type classes* [7,19], but it is beyond the scope of this paper to describe this mechanism. Given a subsingleton type `A` with proof $\mathtt{sse}_A$, we can prove

$\mathtt{hsse}_A$: Π (`C:Type`) (`c:C`) (`a:A`), `C` $==$ `A` $\rightarrow$ `c` $==$ `a`,

which we can use as an additional propagation rule in the congruence closure procedure. The idea is to merge the equivalence classes of `a:A` and `c:C` whenever `A` is a subsingleton and `C` $\approx$ `A`. First, we add a mapping *subrep* from subsingleton types to their representatives. Then, we include the following additional code in INITIALIZE:

```
$C \leftarrow type(c)$; $A \leftarrow repr[C]$
if $issub(\Gamma, A)$ then
    if $a = subrep[A]$ then
        $p \leftarrow$ MKPR$(C, A)$; $e \leftarrow$ make fresh variable
        add $(e : c == a := \mathtt{hsse}_A\ C\ p\ c\ a)$ to $pending$ and $\Gamma$
    else $subrep[A] \leftarrow c$
```

Finally, at PROCESSEQ whenever we merge the equivalence classes of subsingleton types A and C, we also propagate the equality $subrep[A] == subrep[C]$.

With this extension, our procedure can prove `safe_log a Ha` $=$ `safe_log b Hb` in the example above, since the terms `a` > 0 and `b` > 0 are both subsingleton types with representative elements `Ha` and `Hb` respectively, and when their equivalence classes are merged, the subsingleton extension propagates the fact that their representative elements are equal, i.e. that `Ha` $==$ `Hb`.

5 Applications

We have implemented our congruence closure procedure for Lean[4] along with many of the standard extensions as part of a long-term effort to build a robust theorem prover for ITT. Although congruence closure can be useful on its own, its

[4] https://github.com/leanprover/lean/blob/master/src/library/blast/congruence_closure.cpp.

power is greatly enhanced when it is combined with a procedure for automatically instantiating lemmas so that the user does not need to manually collect all the ground facts that the congruence closure procedure will need. We use an approach called *e-matching* [10] to instantiate lemmas that makes use of the equivalences represented by the state of the congruence closure procedure when deciding what to instantiate, though the details of e-matching are beyond the scope of this paper. The combination of congruence closure and e-matching is already very powerful, as we demonstrate in the following two examples, the first from software verification and the second from formal mathematics. The complete list of examples we have used to test our procedure can be found at http://leanprover.github.io/ijcar16/examples.

Vectors (indexed lists). As we mentioned in Sect. 2.1, a common use of dependent types is to include the length of a list inside its type in order to make out-of-bounds errors impossible. The constructors of `vector` mirror those of `list`:

$$\begin{aligned}&\mathtt{nil} : \Pi\ \{\mathtt{A} : \mathtt{Type}\},\ \mathtt{vector\ A}\ 0\\&\mathtt{cons} : \Pi\ \{\mathtt{A} : \mathtt{Type}\}\ \{\mathtt{n} : \mathbb{N}\},\ \mathtt{A} \to \mathtt{vector\ A\ n} \to \mathtt{vector\ A}\ (\mathtt{succ\ n})\end{aligned}$$

where `succ` is the successor function on natural numbers, and where curly braces indicate that a parameter should be inferred from context. We use the notation `[x]` to denote the one-element `vector` containing only `x`, i.e. `cons x nil`, and `x::v` to denote `cons x v`. It is easy to define append and reverse on `vector`:

$$\begin{aligned}&\mathtt{app} : \Pi\ \{\mathtt{A} : \mathtt{Type}\}\ \{\mathtt{n}_1\ \mathtt{n}_2 : \mathbb{N}\},\ \mathtt{vector\ A\ n}_1 \to \mathtt{vector\ A\ n}_2 \to \mathtt{vector\ A}\ (\mathtt{n}_1 + \mathtt{n}_2)\\&\mathtt{rev} : \Pi\ \{\mathtt{n} : \mathbb{N}\},\ \mathtt{vector\ A\ n} \to \mathtt{vector\ A\ n}\end{aligned}$$

When trying to prove the basic property $\mathtt{rev}\ (\mathtt{app}\ \mathtt{v}_1\ \mathtt{v}_2) == \mathtt{app}\ (\mathtt{rev}\ \mathtt{v}_2)\ (\mathtt{rev}\ \mathtt{v}_1)$ about these two functions, we reach the following goal:

$$\begin{aligned}&(\mathtt{A} : \mathtt{Type})\ (\mathtt{n}_1\ \mathtt{n}_2 : \mathbb{N})\ (\mathtt{x}_1\ \mathtt{x}_2 : \mathtt{A})\ (\mathtt{v}_1 : \mathtt{vector\ A\ n}_1)\ (\mathtt{v}_2 : \mathtt{vector\ A\ n}_2)\\&(\mathtt{IH} : \mathtt{rev}\ (\mathtt{app}\ \mathtt{v}_1\ (\mathtt{x}_2\mathtt{::}\mathtt{v}_2)) == \mathtt{app}\ (\mathtt{rev}\ (\mathtt{x}_2\mathtt{::}\mathtt{v}_2))\ (\mathtt{rev}\ \mathtt{v}_1))\\&\vdash \mathtt{rev}\ (\mathtt{app}\ (\mathtt{x}_1\mathtt{::}\mathtt{v}_1)\ (\mathtt{x}_2\mathtt{::}\mathtt{v}_2)) == \mathtt{app}\ (\mathtt{rev}\ (\mathtt{x}_2\mathtt{::}\mathtt{v}_2))\ (\mathtt{rev}\ (\mathtt{x}_1\mathtt{::}\mathtt{v}_1))\end{aligned}$$

Given basic lemmas about how to push `app` and `rev` in over `cons`, a lemma stating the associativity of `app`, and a few basic lemmas about natural numbers, our congruence closure procedure together with the e-matcher can solve this goal. Once the e-matcher establishes the following ground facts:

$$\begin{aligned}&\mathtt{H}_1 : \mathtt{rev}\ (\mathtt{x}_1\mathtt{::}\mathtt{v}_1) == \mathtt{app}\ (\mathtt{rev}\ \mathtt{v}_1)\ [\mathtt{x}_1]\\&\mathtt{H}_2 : \mathtt{app}\ (\mathtt{x}_1\mathtt{::}\mathtt{v}_1)\ (\mathtt{x}_2\mathtt{::}\mathtt{v}_2) == \mathtt{x}_1\mathtt{::}(\mathtt{app}\ \mathtt{v}_1\ (\mathtt{x}_2\mathtt{::}\mathtt{v}_2))\\&\mathtt{H}_3 : \mathtt{rev}\ (\mathtt{x}_1\mathtt{::}(\mathtt{app}\ \mathtt{v}_1\ (\mathtt{x}_2\mathtt{::}\mathtt{v}_2))) == \mathtt{app}\ (\mathtt{rev}\ (\mathtt{app}\ \mathtt{v}_1\ (\mathtt{x}_2\mathtt{::}\mathtt{v}_2)))\ [\mathtt{x}_1]\\&\mathtt{H}_4 : \mathtt{app}\ (\mathtt{app}\ (\mathtt{rev}\ (\mathtt{x}_2\mathtt{::}\mathtt{v}_2))\ (\mathtt{rev}\ \mathtt{v}_1))\ [\mathtt{x}_1] == \mathtt{app}\ (\mathtt{rev}\ (\mathtt{x}_2\mathtt{::}\mathtt{v}_2))\ (\mathtt{app}\ (\mathtt{rev}\ \mathtt{v}_1)\ [\mathtt{x}_1])\end{aligned}$$

as well as a few basic facts about the natural numbers, the result follows by congruence.

Safe arithmetic. As we mentioned in Sect. 4, another common use of dependent types is to extend functions to take extra arguments that represent proofs that certain preconditions hold. For example, we can define safe versions of the logarithm function and the inverse function as follows:

$$\texttt{safe\_log} : \Pi\ (\texttt{x} : \mathbb{R}),\ \texttt{x} > 0 \to \mathbb{R} \qquad \texttt{safe\_inv} : \Pi\ (\texttt{x} : \mathbb{R}),\ \texttt{x} \neq 0 \to \mathbb{R}$$

Although it would be prohibitively cumbersome to prove the preconditions manually at every invocation, we can relegate this task to the theorem prover, so that `log x` means `safe_log x p` and $\texttt{y}^{-1}$ means `safe_inv y q`, where `p` and `q` are proved automatically. Given basic lemmas about arithmetic identities, our congruence closure procedure together with the e-matcher can solve many complex equational goals like the following, despite the presence of embedded proofs:

$$\forall\ (\texttt{x y z w} : \mathbb{R}),\ \texttt{x} > 0 \to \texttt{y} > 0 \to \texttt{z} > 0 \to \texttt{w} > 0 \to \texttt{x} * \texttt{y} = \texttt{exp z} + \texttt{w} \to$$
$$\texttt{log}\ (2 * \texttt{w} * \texttt{exp z} + \texttt{w}^2 + \texttt{exp}\ (2 * \texttt{z}))\ /\ {-2} = \texttt{log y}^{-1} - \texttt{log x}$$

6 Related Work

Corbineau [9] presents a congruence closure procedure for the simply-typed subset of ITT and a corresponding implementation for Coq as the tactic `congruence`. The procedure uses homogeneous equality and does not support dependent types at all. Hur [12] presents a library of tactics for reasoning over a different variant of heterogeneous equality in Coq, for which the user must manually separate the parts of the type that are allowed to vary between heterogeneously equal terms from those that must remain the same. The main tactic provided is `Hrewritec`, which tries to rewrite with a heterogeneous equality by converting it to a cast-equality, rewriting with that, and then generalizing the proof that the types are equal. There does not seem to be any general notion of congruence akin to our family of $\texttt{hcongr}_n$ lemmas.

Sjöberg and Weirich [25] propose using congruence closure during type checking for a new dependent type theory in which definitional equality is determined by the congruence closure relation instead of by the standard forms of reduction. Their type theory is not compatible with any of the standard flavors of ITT such as the calculus of inductive constructions, and so their procedure cannot be used to prove theorems in systems such as Coq and Lean. The congruence rules they use are also not as general as ours, since they require the two functions being applied to be the same, whereas $\texttt{hcongr}_n$ allows them to differ as long as they are homogeneously equal. As a result, given `x = y`, they cannot conclude `f x = g y` from `f = g`, let alone `f a x = g y` from `f a = g`. Moreover, they do not discuss why or whether the natural binary congruence rule (i.e. `hcongr_ideal`) would be unsound in their type theory, nor why their congruence rule needs to be n-ary.

7 Conclusion

We have presented a very general notion of congruence for ITT based on heterogeneous equality that applies to all dependently typed functions. We also presented a congruence closure procedure that can propagate the associated congruence rules efficiently and so automatically prove a large and important set of goals. Just as congruence closure procedures (along with DPLL) form the foundation of modern SMT solvers, we hope that our congruence closure procedure can form the foundation of a robust theorem prover for intensional type theory. We are building such a theorem prover for Lean, and it can already solve many interesting problems.

Acknowledgments. We would like to thank David Dill, Jeremy Avigad, Robert Lewis, Nikhil Swany, Floris van Doorn and Georges Gonthier for providing valuable feedback on early drafts.

References

1. Private communication with Jeremy Avigad and Floris van Doorn
2. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: The Matita interactive theorem prover. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 64–69. Springer, Heidelberg (2011)
3. Bachmair, L., Tiwari, A., Vigneron, L.: Abstract congruence closure. J. Autom. Reason. **31**(2), 129–168 (2003)
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
5. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda – a functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer, Heidelberg (2009)
6. Brady, E.: Idris, a general-purpose dependently typed programming language: design and implementation. J. Funct. Program. **23**(05), 552–593 (2013)
7. Castéran, P., Sozeau, M.: A gentle introduction to type classes and relations in Coq. Technical report. Citeseer (2012)
8. Coq Development Team: The Coq proof assistant reference manual: Version 8.5. INRIA (2015–2016)
9. Corbineau, P.: Autour de la clôture de congruence avec Coq. Master's Thesis, Université Paris-Sud (2001)
10. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
11. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., Théry, L.: A machine-checked proof of the odd order theorem. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 163–179. Springer, Heidelberg (2013)
12. Hur, C.K.: Heq: a Coq library for heterogeneous equality (2010)
13. Kapur, D.: Shostak's congruence closure as completion. In: Comon, H. (ed.) RTA 1997. LNCS, vol. 1232, pp. 23–37. Springer, Heidelberg (1997)
14. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
15. McBride, C.: Elimination with a motive. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 197–216. Springer, Heidelberg (2002)
16. McBride, C.: Epigram: practical programming with dependent types. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 130–170. Springer, Heidelberg (2005)
17. McBride, C., Goguen, H.H., McKinna, J.: A few constructions on constructors. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 186–200. Springer, Heidelberg (2006)
18. Miquel, A., Werner, B.: The not so simple proof-irrelevant model of CC. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 240–258. Springer, Heidelberg (2003)

19. de Moura, L., Avigad, J., Kong, S., Roux, C.: Elaboration in dependent type theory. Technical report (2015). http://arXiv.org/abs/1505.04324
20. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
21. de Moura, L., Kong, S., Avigad, J., Van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) CADE-25. LNAI, vol. 9195, pp. 378–388. Springer, Heidelberg (2015)
22. de Moura, L., Rueß, H., Shankar, N.: Justifying equality. Electron. Notes Theoret. Comput. Sci. **125**(3), 69–85 (2005)
23. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM (JACM) **27**(2), 356–364 (1980)
24. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 453–468. Springer, Heidelberg (2005)
25. Sjöberg, V., Weirich, S.: Programming up to congruence. In: POPL 2015, NY, USA, pp. 369–382. ACM, New York (2015)
26. Streicher, T.: Investigations into Intensional Type Theory, Habilitations-schrift, Ludwig-Maximilians-Universität München (1993)